Microsoft WinHEC

Writing Drivers for Reliability, Robustness and Fault Tolerant Systems

Simon Graham, Stratus Technologies International, Sarl

Abstract

This paper provides information about writing drivers for High Availability and Fault Tolerant environments for the Microsoft® Windows® family of operating systems. It provides guidelines for driver writers to ensure that the system keeps running during hardware failures and online configuration changes.

April 2, 2002

Contents

Windows Hardware Engineering Conference

Introduction

This paper provides guidelines for designing, implementing and deploying drivers suitable for use in High Availability and Fault Tolerant systems. These guidelines cover the following main areas:

- Design and coding practices; a Fault Tolerant system has some special characteristics that must be taken into account when designing and implementing drivers.
- Serviceability requirements; again, since a Fault Tolerant system should never stop, it must be possible to diagnose problems with the system whileit is running and without requiring reboots.
- Testing requirements; since a Fault Tolerant system should never stop operating no matter what, it is important to perform a lot of negative testing on drivers that might be used in such a system.

It is also important to note that, although the following sections are aimed specifically at good practices for drivers running in a fault tolerant system, they are also completely applicable to any system and following these guidelines can help improve drivers for all environments.

Fault Tolerant Design and Coding Practices

Handling Error Conditions

Quite often, when OS drivers are designed and implemented, little thought is given to error conditions. A majority of drivers, regardless of operating system, merely halt the system (blue screen) when something unexpected happens. In most environments it is deemed acceptable to handle unexpected, rare events in less-than-graceful ways, because the events are, after all, rare.

In a fault-tolerant system, this sort of driver behavior is not acceptable. For a system to be truly fault-tolerant, both hardware and software must be designed with robustness in mind. It is especially important for software to be robust where the hardware is not fault-tolerant by design. The following guidelines can be effectively applied to OS drivers in any computing environment to yield a hardened, robust subsystem.

Never voluntarily halt the system
This is the simplest and most important rule to follow. It is common for drivers to simply crash the system should something unexpected occur, but in a high-availability environment this sort of thing is often disastrous. Recovering from supposedly "impossible" situations is not usually all that difficult especially when you have redundant hardware paths. Seemingly fatal problems should not be resolved at the driver level, especially not by halting the system. Drivers should do all they can to recover from faults rather than halt the system and if that is not possible, they should pass problems to the operating system.

Recovery should be a matter of resetting the faulty hardware because many hardware failures are transient in nature. Software layers above the driver should not be notified of any problems, as long as they can be successfully resolved by the driver and the I/O operations restarted without loss of data.

Be sure to handle all possible hardware fault codes
Drivers should be coded to deal with ALL possible fault codes a piece of hardware can return, no matter how many there are. Simply ignoring or just reporting unknown or unexpected error codes can lead to I/O hangs or system failure. Driver designers should build the code to specifically handle important error codes in such a way that the I/O thread is guaranteed to move forward. Less important or less specific error codes should be grouped into common error classifications and serviced by common error-handling routines. If there is any doubt about what might cause a particular error, the worst case scenario should be assumed when deciding on the recovery strategy.

Check for device failure often
Drivers should proactively check for their hardware failing or being removed. Typically this requires checking for '-1' return values (i.e. all 1s in read data). The driver should check for all-1s return values and treat them suspiciously. In most normal cases, all-1s is never valid, so the driver can assume the hardware has failed in some way. In the cases where all-1's is a valid return value, the driver should check for this value and if see, check some other location that cannot be all-1s (worst case would be issuing a read to PCI config space, but many times there are other registers in PCI memory or IO space that can be read without side effect and which may not contain all-1s).

Having detected a failure, the driver should immediately fail any IO in progress. It is not a good idea in general to wait for outstanding IO requests to time out as this impedes the ability of higher level drivers to recover by using redundant hardware paths. The exact details are very dependent on the type of driver, but the general rule is that the driver should do whatever is necessary to complete all outstanding IO as quickly as possible with a suitable error that indicates the hardware is no longer present.

Keep error recovery simple, but effective
Error recovery should not be overly complicated. Intricate recovery schemes rarely work well, and usually take longer than they should when they do work. The appropriate recovery strategy is usually the one that brings the bad hardware back into service with minimum effort without affecting other parts of the system.

It's best to define a small number of different recovery procedures to handle all possible error situations. By keeping the procedures simple, they can handle a wide variety of problems, negating the need for large amounts of special-case recovery code. For example, a good set of error classifications might be fatal errors, recoverable errors, timeout errors, etc. All possible errors should be grouped into the classifications, and serviced by common code that is designed to handle the situation regardless of the fine details.

Report all errors and provide as much useful information as possible
While this point may not directly make a driver more robust, complete reporting of errors is extremely important for diagnosing problems. Error messages should be designed to provide useful information to anyone who reads them, but should also contain technical details that may only be of use to the engineer. The best approach is to always log error messages to the system event log and include any relevant hardware state information, no matter how cryptic.

In addition to reporting errors, it is also useful to report important events pertaining to hardware state changes. This immensely simplifies diagnosis and debugging of problems. It is always better to log too many debug messages than too few.

Hide recoverable faults from the user applications

Drivers should always attempt to resolve problems that are within their scope before reporting problems to the operating system. By recovering from problems without involving the operating system, the driver ensures that the system will continue to function normally. Errors that would normally cause a crash should never be passed back to the upper OS layers unless there is absolutely no other alternative. Only if a driver can't resolve a situation on its own should it report back to the Operating System.

For example, it is tempting to simply return an error to the calling software layer if an I/O request can't be issued because of a possibly transient controller error. However, unless there's a good reason to believe the error is permanent, the driver should attempt to reset the device or controller and re-issue the I/O request.

Additionally, conditions that are not normally considered faults, but which could cause operations to fail, should be handled gracefully. This might include out-of-memory conditions, other resource depletions, etc. Drivers should fail such requests in a manner that allows higher level system software to recover.

Ensure assertions are also tested in free build

The ASSERT() macro is a very useful test tool and it’s behavior of dropping into the debugger in checked builds is useful. However, since the tests in this macro are completely removed in free builds, it is important to perform the same test outside of the ASSERT() macro and perform appropriate error recovery when the test fails. The best way to handle this is to define a new version of ASSERT as follows:

```
#if DBG
#define MDASSERT(expr) \
    ((expr) || \
     (RtlAssert(#expr,__FILE__,__LINE__,NULL),0))
#else
#define MDASSERT(expr) \
    ((expr) || 0)
#endif
```

This can then be used directly in if statements:

```
If (MDASSERT(<some test>)) {
    // normal processing
}
else {
    // handle failure case
}
```

This has the advantage of including the assert in debug code while also providing the ability to test the recovery in checked and free builds. A further variant would be to use DebugPrint in the free build to output the failed assertion to the debug console.

Do Not Monopolize CPU Resources or Execution Context

There are often situations that require that the driver wait on a hardware state change or completion of some event. Drivers should be designed to use Windows XP events and kernel threads in situations that require the driver to wait. It is sometimes difficult to design complex code that doesn't over-utilize the thread of execution. Although it may be tempting to spin or stall execution while waiting for a state change, such practices can have serious availability and performance consequences. Over-utilizing the execution context means nothing else can do anything with the CPU or execution context until the driver is done with it. This can slow or halt normal execution and recovery efforts of other OS modules. It can even affect the driver's own ability to service other things while it's waiting. To avoid this situation, drivers must:

- Be designed so that time consuming operations are never single-threaded.
- Use **KeStallExecutionProcessor()** sparingly, and avoid doing so within a loop.
- Never busy-wait. Instead, state machine transitions should be event driven by interrupts or periodic polling (timed callbacks).
- Never hold locks or block interrupts for long periods.
- Run operations in parallel so that all entities controlled by the driver can be used/recovered simultaneously.
- Make the driver's resource hierarchy as simple as possible and document the required ordering of resource acquisition carefully. Lock inversions are a very common problem in the field that may never cause a problem in the lab. In particular it is important to realize that ANY object that can cause processing to be blocked is part of the hierarchy, including:
    - Explicit synchronization objects (Events, semaphores, mutexes, spinlocks, and so on).
    - Implicit synchronization objects; for example, reference counts in data structures. If any code can block waiting for the reference count to reach zero (for example), then this is an implicit resource in your hierarchy.
    - Worker threads if they are part of a fixed pool. If you acquire a resource then block waiting for a thread to run on, while at the same time code already running on a worker thread attempts to get access to the resource, you have a lock inversion. Chance are, you'll never deadlock until you get into the field and your customers run your code in a way you never imagined!

Drivers Should Be Designed for Non-Modal Operation

It's quite common for drivers to expect events to occur in a certain order, and to be coded accordingly. Inevitably, however, assumptions about such things often prove to be wrong, and the driver fails to function. For robustness, drivers should never be designed to expect anything to happen in a particular sequence or order. Assumptions should never be made with regard to what will happen next.

To avoid such pitfalls, drivers should be designed in a non-modal manner. No assumptions should be made with regard to ordering of events that aren't clearly and unequivocally specified to occur in a particular order. The only assumption that should be made is that anything can happen at any time, and the code should be designed with that expectation.

All driver actions should be viewed as being event-driven. For any given event or class of events, there should be a well-defined action on the part of the driver. Those actions should not depend on any other actions occurring or having occurred at any time. The driver should simply react to events without concern for what might have happened before, or what might happen next. If the driver is in the midst of reacting to a particular event when another event occurs, there are three possible courses of action:

- Ignore the event, if it's of lesser priority than the current event. For example, if the driver is notified of a card failure and is recovering the card, it's probably best to ignore subsequent interrupts from the card since they're either bogus or stale.

- Queue the event notification for later processing, if it's of lesser priority than the current event. It is not suggested that events be queued, since this can quickly add complication to driver design. There may be some events for which this is necessary or convenient, but it should be avoided if at all possible.
- Preempt the current event, if the new event is of greater importance. For example, if the driver is initializing a card and the card suddenly goes offline, there is no point in continuing the initialization process. The card failure event should be serviced instead.

Proactively Detect Hardware Problems

Detecting hardware problems is often difficult and rarely straightforward. There are uncountable symptoms of hardware faults, ranging from outright failure to respond, to "forgetting" outstanding I/O, to generally problematic behavior. Coding specifically to detect particular cases is not realistic, due to the large number of possibilities, and because it's not always clear just what a particular symptom might signify in the first place. However, unless the driver proactively tries to detect faults, many may go unnoticed forever or until it's too late to recover.

Here are some simple guidelines for detecting controller faults:

- Time transactions.

  This is the most obvious way of detecting hardware problems, and probably the most effective. When a piece of hardware stops behaving predictably from a timing standpoint, it is a sure sign that something is wrong. This approach does not help diagnose problems much, but does hint that there is a problem in the first place, which is the most important factor. All driver/hardware transactions should be timed, such as I/O operations, expected card state changes, etc. The driver should never depend on a hardware response of any kind without a timer running against that response. Timer expiry should be considered a hardware failure, and the appropriate recovery steps taken.

  In hierarchical stacks of software, each layer that needs to provide a timeout should do so such that the upper layer timeouts are longer than any lower layer timeouts of the same type. This allows timeouts to be processed from the bottom up so that fault recovery can be accomplished at the lowest level, thus impacting the system the least.
- Test hardware periodically.

  The driver should periodically test the device in some way without interfering with normal operations. A simple example is to exercise some sort of no-op provided by the hardware when it is idle. In the absence of such functionality, some benign hardware operation should be performed. This guarantees that the hardware is at least alive, though it doesn't guarantee much else. Hardware tests should always be timed, as outlined above.
- Sanity-check important hardware responses.

  If at all possible, sanity checking should be performed against all responses from hardware. This can be done a variety of ways. Status data in hardware responses can be checked against known valid possibilities, tags/serial numbers can be placed in status areas for comparison with control data, data patterns can be placed in DMA areas to verify that data transfers have occurred, etc.

Minimize Use of System Resources

There are two reasons why hardware and drivers should be designed to minimize the use of system resources:

- First, a Fault Tolerant System typically has redundant hardware which means that the system has to operate with twice the number of devices installed.
- Second, devices and drivers must support hot-add and hot-remove of devices many times while the system is running. Thus one cannot rely on large amounts of contiguous resources being available when a device is started.

Examples of resource usage that should be minimized are as follows:

- PCI Device BARs; the size of BARs should be kept as small as possible for the following reasons:
  - First, in a fault tolerant system which supports hot-plug, it is likely that PCI devices will actually be installed behind a PCI bridge. This means that the PCI bridge has to be initialized with a bridge window large enough to accommodate all possible PCI devices that can be inserted behind it. Currently, the OS does not support reassigning bridge resources once the system is running so devices with very large PCI Memory requirements can either fail to start or severely limit the ability to use other devices in the system.
  - Second, a contiguous range of system PTEs must be found to map the BAR into virtual address space. Although this practice is usually acceptable during system boot time, over time the system PTE pool becomes fragmented, which can lead to failures when devices are hot-inserted.
- Common buffers used to communicate between the hardware and the driver. As above, these require contiguous ranges of system PTEs to be available so requiring large buffers can result in failures when devices are hot-inserted.
- Don't assume that a driver will get unloaded when hardware is removed; in a fault tolerant system with redundant hardware, removal or failure of one card will not result in the driver being unloaded.

Input Verification and Exiting

Wherever possible, all input to driver routines should be verified and an error returned if invalid data is passed. In addition, the exit paths from a routine should be limited, in order to keep resource cleanup and error handling centralized as much as possible. On the other hand, it is not a good idea to write complex nested if statements because they can introduce subtle bugs. There are a number of ways of structuring the code to minimize the exit paths while keeping the main line straightforward; two that are useful are:

- Structured Exception Handling provides a convenient means to accomplish this task when the inline code might generate exceptions. However, there is a performance and data size penalty associated with SEH and it cannot be used in code that runs at elevated IRQL For more information on Structured Exception Handling and the __try/__finally/__except constructs, refer to http://msdn.microsoft.com/library/en-us/debug/structex_93on.asp.
- When running at elevated IRQL or in environments where exceptions cannot be generated, a pseudo-try/finally implementation can used by defining C preprocessor macros:

```
#define TRY
#define LEAVE goto __tf_Leave
```

```
#define FINALLY __tf_Leave:
```

Having done this, code can be written in a structured manner as follows:

```
TRY {
    Status = op1();
    If (!NT_SUCCESS(Status))
        LEAVE;
    …
}
FINALLY {
    --clean-up-code–
}
```

Serviceability Requirements

When the inevitable problems occur in the field, it is necessary to be able to diagnose the issue without impacting the availability of a fault tolerant system. The specific requirements include:

- It must be possible to enable diagnosis while the system is running; it is not acceptable to reboot the system with a special version of a driver installed. The tracing and other tools must therefore be included in both the checked and free builds of the driver.
- When not in use, the tools should have minimal impact on performance.
- The diagnosis tools must produce clear results that can be interpreted by support engineers not intimately familiar with the source of the driver.
- The tools should be used during development; this leads to tools that are actually useful for diagnosing issues in the field.

These are discussed in more detail in the following sections.

Diagnosis in the Field

Since a fault tolerant system must continue to run in the face of problems, it is necessary to be able to diagnose problems without affecting the availability of the system. Obviously said diagnosis is likely to have an impact on system performance when enabled, but the system must continue to run without interruption. This means that:

- The tools must be included in both checked and free builds of the driver.
- It must be possible to enable and disable diagnosis dynamically via a user interface. It is not sufficient to require data items to be modified via the kernel debugger for example, because this cannot typically be done in the field.
- It must be possible to control the tools remotely. This means that it is typically not possible to hook up extra hardware to perform diagnosis.

Examples of suitable tools for diagnosis are as follows:

- Filter drivers to trace IRP flow between drivers. There are existing tools available that can do this at specific places in the driver stack (file system, disk, etc). There are also a multitude of places in the driver stack where no tool exists; think of it as an opportunity!
- Filter drivers to trace message flow between peer components; for example, the Network Monitor tool. Again there are lots of situations where no standard tool exists in which case one should be developed.

- Debug print statements; there are always times when you need to look inside the processing of a driver. This really requires the use of debug print statements in both checked AND FREE builds and mechanisms for collecting the resulting output. (This can be done via one of the available tools for intercepting debugprints or by writing your own debug package that collects debug output. Don't forget to provide interactive tools for enabling/disabling trace dynamically.)

Performance Requirements

Since the free build will potentially include instrumentation for diagnosing problems, it is crucial that this instrumentation be added in a way that causes minimal impact when not being used. If the diagnosis tools can be written as filter drivers that are dynamically loaded this is easy. However, to support the Debug Print statements, some important requirements need to be adhered to:

- The test for whether or not debug output is enabled must be simple and must be coded to be inline. Once the trace is determined to be active, the actual formatting and output of trace information can be done in library functions. This is typically achieved via #define's or inline functions that test global variables to see if debug output is enabled and then call a function to perform the output.
- You should provide fairly fine granularity of control over the amount of debug output produced. Some suggestions for this include:
  - Categorize trace statements by type of thing being traced; for example, you might distinguish between IRP, Plug and Play, overall driver processing, polling, etc. This allows one to enable very specific trace based on the issue being investigated.
  - Define the verbosity of any given trace statement via levels; the higher the level, the more verbose the output. The lowest level should provide a summary of message flow through the driver (maybe one line per IRP for example). Higher levels would show the details of routines that are called and their parameters. Another example would be to dump the contents of buffers associated with processing at higher levels.

Instrumentation Output Guidelines

It is important to design the instrumentation so that it makes sense to people who are not reading the source. It's also important to provide enough information to make the problem diagnosable without using a kernel debugger or other low level tool. Suggestions here include:

- Use text wherever possible rather than hex values; it's much easier to read a trace that says "IRP_MJ_CLOSE" than to hunt for "0x2".
- Output information to identify data structures. It's very little use to output something like this:
  ```
  DispatchIrp; Device=0x81234567, Irp=0x87654321
  ```
  Much better to output the name of the device and the type of IRP:
  ```
  DispatchIrp; foo12 (Do=0x81234567), IRP_MJ_CREATE
  ```
- Output summary information using a standard format to show the general flow of processing.
- Attempt to show the structure of routine calls; one suggestion is to have special trace call to mark routine entry and exit. On routine entry, output a marker string and arrange to indent future lines. On exit, reduce the indentation level and output a marker string.

- Include standard prefix information on each output line; time, current thread and current IRQL are very useful in debugging multi-thread issues on SMP systems.

Test Plan and Test Tools

In addition to the normal driver testing described in the Windows DDK, drivers should undergo the following tests specific to fault tolerant systems:

- Load and Stress testing
- Failure testing; simulate hardware errors at all points in the path from driver to hardware.
- Power failure testing
- Plug-and-Play testing, especially hot add and hot remove (also called surprise removal) testing. The hardware should be cycled in and out of service as often as possible for an extensive period of time.
- Installation and configuration testing; especially important here is to test maximum configurations; since Fault Tolerant systems will have redundant hardware installed it is important to ensure that multiple devices are fully supported and that failures in one device do not cause outages in others.
- Duration testing - Duration testing should focus on running tests in a typical customer environment with respect to loads, configurations and applications.

Driver Test Suite Guidelines

Testing a device for this environment introduces some special requirements on the test tools themselves, including:

- In order to do heavy stress testing you need to design a test suite in which you can break things and reconstruct test paths quickly to allow for high volume testing of faults. For example, when testing disk faults in a system that uses mirroring, mirror regeneration needs to be automated to allow for heavy testing.
- In general, when testing device failure, the system needs to provide replicated paths so that the OS continues normal operation in the face of failures.
- In the absence of replicated paths, a fast recovery method needs to be designed to allow volume testing.
- Design the test suite to predictive data patterns to facilitate fault analysis. For example, use data that includes time stamps and information on the location of the data (disk address, IP byte count, etc). Automated analysis tools can then be developed to check that the data is written as expected.
- Check for side affects on upper layers of the software for resources not being properly handled after a failure. For example, check that the Plug and Play manager is still fully functional at the end of a test run.

Microsoft HCTs

Your driver must pass the Microsoft HCT test suite. It is also a good test of the baseline functionality of your device. However, this should be considered the starting point for testing and not the goal.

Driver Verifier

This tool should be run throughout the testing process using the “standard” set up (which enables all tests except resource shortage). In addition, specific testing should be done with resource shortage simulation enabled.

Resources

- Windows Driver Development Kit:
  http://www.microsoft.com/ddk
- WHQL Test Specifications, HCTs, and testing notes:
  http://www.microsoft.com/hwtest/