

[PATCH]: Documentation: Updated PCI Error Recovery

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-02/msg00920.html>

- *From:* linas@xxxxxxxxxxxxxxxx (Linus Vepstas)
 - *Date:* Thu, 2 Feb 2006 18:06:02 -0600
-

I'm not sure who I'm addressing this patch to: Linus, maybe?

Please apply. Fingers crossed, I hope this may make it into 2.6.16.

--linas

This patch is a cleanup/restructuring/clarification of the PCI error handling doc. It should look rather professional at this point.

Signed-off-by: Linus Vepstas <linas@xxxxxxxxxxxxxxxx>

--
pci-error-recovery.txt | 462 +++-----
1 files changed, 306 insertions(+), 156 deletions(-)

Index: linux-2.6.16-rc1-git5/Documentation/pci-error-recovery.txt

=====
--- linux-2.6.16-rc1-git5.orig/Documentation/pci-error-recovery.txt 2006-02-01 17:09:01.000000000
-0600
+++ linux-2.6.16-rc1-git5/Documentation/pci-error-recovery.txt 2006-02-02 18:04:57.714942210 -0600
@@ -1,246 +1,396 @@

PCI Error Recovery

- May 31, 2005
+ February 2, 2006

- Current document maintainer:
- Linus Vepstas <linas@xxxxxxxxxxxxxxxx>
+ Current document maintainer:
+ Linus Vepstas <linas@xxxxxxxxxxxxxxxx>

-Some PCI bus controllers are able to detect certain "hard" PCI errors
-on the bus, such as parity errors on the data and address busses, as
-well as SERR and PERR errors. These chipsets are then able to disable

[PATCH]: Documentation: Updated PCI Error Recovery

-I/O to/from the affected device, so that, for example, a bad DMA
-address doesn't end up corrupting system memory. These same chipsets
-are also able to reset the affected PCI device, and return it to
-working condition. This document describes a generic API for
-performing error recovery.

-
-The core idea is that after a PCI error has been detected, there must
-be a way for the kernel to coordinate with all affected device drivers
-so that the pci card can be made operational again, possibly after
-performing a full electrical #RST of the PCI card. The API below
-provides a generic API for device drivers to be notified of PCI
-errors, and to be notified of, and respond to, a reset sequence.

-
-Preliminary sketch of API, cut-n-pasted-n-modified email from
-Ben Herrenschmidt, circa 5 april 2005
+Many PCI bus controllers are able to detect a variety of hardware
+PCI errors on the bus, such as parity errors on the data and address
+busses, as well as SERR and PERR errors. Some of the more advanced
+chipsets are able to deal with these errors; these include PCI-E chipsets,
+and the PCI-host bridges found on IBM Power4 and Power5-based pSeries
+boxes. A typical action taken is to disconnect the affected device,
+halting all I/O to it. The goal of a disconnection is to avoid system
+corruption; for example, to halt system memory corruption due to DMA's
+to "wild" addresses. Typically, a reconnection mechanism is also
+offered, so that the affected PCI device(s) are reset and put back
+into working condition. The reset phase requires coordination
+between the affected device drivers and the PCI controller chip.
+This document describes a generic API for notifying device drivers
+of a bus disconnection, and then performing error recovery.
+This API is currently implemented in the 2.6.16 and later kernels.

+
+Reporting and recovery is performed in several steps. First, when
+a PCI hardware error has resulted in a bus disconnect, that event
+is reported as soon as possible to all affected device drivers,
+including multiple instances of a device driver on multi-function
+cards. This allows device drivers to avoid deadlocking in spinlocks,
+waiting for some i/o-space register to change, when it never will.
+It also gives the drivers a chance to defer incoming I/O as
+needed.

+
+Next, recovery is performed in several stages. Most of the complexity
+is forced by the need to handle multi-function devices, that is,
+devices that have multiple device drivers associated with them.
+In the first stage, each driver is allowed to indicate what type
+of reset it desires, the choices being a simple re-enabling of I/O
+or requesting a hard reset (a full electrical #RST of the PCI card).
+If any driver requests a full reset, that is what will be done.

+
+After a full reset and/or a re-enabling of I/O, all drivers are
+again notified, so that they may then perform any device setup/config
+that may be required. After these have all completed, a final

[PATCH]: Documentation: Updated PCI Error Recovery

+ "resume normal operations" event is sent out.
+
+ The biggest reason for choosing a kernel-based implementation rather
+ than a user-space implementation was the need to deal with bus
+ disconnects of PCI devices attached to storage media, and, in particular,
+ disconnects from devices holding the root file system. If the root
+ file system is disconnected, a user-space mechanism would have to go
+ through a large number of contortions to complete recovery. Almost all
+ of the current Linux file systems are not tolerant of disconnection
+ from/reconnection to their underlying block device. By contrast,
+ bus errors are easy to manage in the device driver. Indeed, most
+ device drivers already handle very similar recovery procedures;
+ for example, the SCSI-generic layer already provides significant
+ mechanisms for dealing with SCSI bus errors and SCSI bus resets.
+
+
+ Detailed Design
+-----
+ Design and implementation details below, based on a chain of
+ public email discussions with Ben Herrenschmidt, circa 5 April 2005.

The error recovery API support is exposed to the driver in the form of
a structure of function pointers pointed to by a new field in struct
pci_driver. The absence of this pointer in pci_driver denotes an
"non-aware" driver, behaviour on these is platform dependant.
Platforms like ppc64 can try to simulate pci hotplug remove/add.
-
- The definition of "pci_error_token" is not covered here. It is based on
- Seto's work on the synchronous error detection. We still need to define
- functions for extracting infos out of an opaque error token. This is
- separate from this API.
+ pci_driver. A driver that fails to provide the structure is "non-aware",
+ and the actual recovery steps taken are platform dependent. The
+ arch/powerpc implementation will simulate a PCI hotplug remove/add.

This structure has the form:

```
-  
struct pci_error_handlers  
{  
- int (*error_detected)(struct pci_dev *dev, pci_error_token error);  
+ int (*error_detected)(struct pci_dev *dev, enum pci_channel_state);  
int (*mmio_enabled)(struct pci_dev *dev);  
- int (*resume)(struct pci_dev *dev);  
int (*link_reset)(struct pci_dev *dev);  
int (*slot_reset)(struct pci_dev *dev);  
+ void (*resume)(struct pci_dev *dev);  
+};  
+  
+The possible channel states are:  
+enum pci_channel_state {  
+ pci_channel_io_normal, /* I/O channel is in normal state */
```

[PATCH]: Documentation: Updated PCI Error Recovery

```
+ pci_channel_io_frozen, /* I/O to channel is blocked */
+ pci_channel_io_perm_failure, /* PCI card is dead */
+};
+
+Possible return values are:
+enum pci_ers_result {
+ PCI_ERS_RESULT_NONE, /* no result/none/not supported in device driver */
+ PCI_ERS_RESULT_CAN_RECOVER, /* Device driver can recover without slot reset */
+ PCI_ERS_RESULT_NEED_RESET, /* Device driver wants slot to be reset. */
+ PCI_ERS_RESULT_DISCONNECT, /* Device has completely failed, is unrecoverable */
+ PCI_ERS_RESULT_RECOVERED, /* Device driver is fully recovered and operational */
+};
```

–A driver doesn't have to implement all of these callbacks. The
–only mandatory one is `error_detected()`. If a callback is not
–implemented, the corresponding feature is considered unsupported.
–For example, if `mmio_enabled()` and `resume()` aren't there, then the
–driver is assumed as not doing any direct recovery and requires
+A driver does not have to implement all of these callbacks; however,
+if it implements any, it must implement `error_detected()`. If a callback
+is not implemented, the corresponding feature is considered unsupported.
+For example, if `mmio_enabled()` and `resume()` aren't there, then it
+is assumed that the driver is not doing any direct recovery and requires
a reset. If `link_reset()` is not implemented, the card is assumed as
–not caring about link resets, in which case, if recover is supported,
–the core can try recover (but not `slot_reset()` unless it really did
–reset the slot). If `slot_reset()` is not supported, `link_reset()` can
–be called instead on a slot reset.

–
–At first, the call will always be :

– 1) `error_detected()`

– Error detected. This is sent once after an error has been detected. At
–this point, the device might not be accessible anymore depending on the
–platform (the slot will be isolated on ppc64). The driver may already
–have "noticed" the error because of a failing IO, but this is the proper
–"synchronisation point", that is, it gives a chance to the driver to
–cleanup, waiting for pending stuff (timers, whatever, etc...) to
–complete; it can take semaphores, schedule, etc... everything but touch
–the device. Within this function and after it returns, the driver
+not care about link resets. Typically a driver will want to know about
+a `slot_reset()`.

+
+The actual steps taken by a platform to recover from a PCI error
+event will be platform–dependent, but will follow the general
+sequence described below.

+
+STEP 0: Error Event

+-----

+PCI bus error is detect by the PCI hardware. On powerpc, the slot

[PATCH]: Documentation: Updated PCI Error Recovery

- +is isolated, in that all I/O is blocked: all reads return 0xffffffff,
- +all writes are ignored.
- +
+
- +STEP 1: Notification
- +-----
- +Platform calls the error_detected() callback on every instance of
- +every driver affected by the error.
- +
+
- +At this point, the device might not be accessible anymore, depending on
- +the platform (the slot will be isolated on powerpc). The driver may
- +already have "noticed" the error because of a failing I/O, but this
- +is the proper "synchronization point", that is, it gives the driver
- +a chance to cleanup, waiting for pending stuff (timers, whatever, etc...)
- +to complete; it can take semaphores, schedule, etc... everything but
- +touch the device. Within this function and after it returns, the driver
- shouldn't do any new IOs. Called in task context. This is sort of a
- "quiesce" point. See note about interrupts at the end of this doc.

- Result codes:
- PCIERR_RESULT_CAN_RECOVER:
- Driver returns this if it thinks it might be able to recover
- +All drivers participating in this system must implement this call.
- +The driver must return one of the following result codes:
- + PCI_ERS_RESULT_CAN_RECOVER:
- + Driver returns this if it thinks it might be able to recover
- the HW by just banging IOs or if it wants to be given
- a chance to extract some diagnostic informations (see
- below).
- PCIERR_RESULT_NEED_RESET:
- Driver returns this if it thinks it can't recover unless the
- slot is reset.
- PCIERR_RESULT_DISCONNECT:
- Return this if driver thinks it won't recover at all,
- (this will detach the driver ? or just leave it
- dangling ? to be decided)

-
- So at this point, we have called error_detected() for all drivers
- on the segment that had the error. On ppc64, the slot is isolated. What
- happens now typically depends on the result from the drivers. If all
- drivers on the segment/slot return PCIERR_RESULT_CAN_RECOVER, we would
- re-enable IOs on the slot (or do nothing special if the platform doesn't
- isolate slots) and call 2). If not and we can reset slots, we go to 4),
- if neither, we have a dead slot. If it's an hotplug slot, we might
- "simulate" reset by triggering HW unplug/replug though.
- + a chance to extract some diagnostic information (see
- + mmio_enable, below).
- + PCI_ERS_RESULT_NEED_RESET:
- + Driver returns this if it can't recover without a hard
- + slot reset.
- + PCI_ERS_RESULT_DISCONNECT:

[PATCH]: Documentation: Updated PCI Error Recovery

- + Driver returns this if it doesn't want to recover at all.
- +
 - +The next step taken will depend on the result codes returned by the
 - +drivers.
 - +
 - +If all drivers on the segment/slot return PCI_ERS_RESULT_CAN_RECOVER,
 - +then the platform should re-enable IOs on the slot (or do nothing in
 - +particular, if the platform doesn't isolate slots), and recovery
 - +proceeds to STEP 2 (MMIO Enable).
 - +
 - +If any driver requested a slot reset (by returning PCI_ERS_RESULT_NEED_RESET),
 - +then recovery proceeds to STEP 4 (Slot Reset).
 - +
 - +If the platform is unable to recover the slot, the next step
 - +is STEP 6 (Permanent Failure).

- >>> Current ppc64 implementation assumes that a device driver will
- >>> *not* schedule or semaphore in this routine; the current ppc64
- +>>> The current powerpc implementation assumes that a device driver will
- +>>> *not* schedule or semaphore in this routine; the current powerpc

implementation uses one kernel thread to notify all devices;

- >>> thus, of one device sleeps/schedules, all devices are affected.
- +>>> thus, if one device sleeps/schedules, all devices are affected.

Doing better requires complex multi-threaded logic in the error recovery implementation (e.g. waiting for all notification threads to "join" before proceeding with recovery.) This seems excessively complex and not worth implementing.

- >>> The current ppc64 implementation doesn't much care if the device
- >>> attempts i/o at this point, or not. I/O's will fail, returning
- +>>> The current powerpc implementation doesn't much care if the device
- +>>> attempts I/O at this point, or not. I/O's will fail, returning

a value of 0xff on read, and writes will be dropped. If the device driver attempts more than 10K I/O's to a frozen adapter, it will assume that the device driver has gone into an infinite loop, and

- >>> it will panic the the kernel.
- +>>> it will panic the the kernel. There doesn't seem to be any other
- +>>> way of stopping a device driver that insists on spinning on I/O.

[PATCH]: Documentation: Updated PCI Error Recovery

– 2) mmio_enabled()

+STEP 2: MMIO Enabled

+-----

+The platform re-enables MMIO to the device (but typically not the
+DMA), and then calls the mmio_enabled() callback on all affected
+device drivers.

– This is the "early recovery" call. IOs are allowed again, but DMA is

+This is the "early recovery" call. IOs are allowed again, but DMA is
not (hrm... to be discussed, I prefer not), with some restrictions. This
is NOT a callback for the driver to start operations again, only to
peek/poke at the device, extract diagnostic information, if any, and
eventually do things like trigger a device local reset or some such,
–but not restart operations. This is sent if all drivers on a segment
–agree that they can try to recover and no automatic link reset was
–performed by the HW. If the platform can't just re-enable IOs without
–a slot reset or a link reset, it doesn't call this callback and goes
–directly to 3) or 4). All IOs should be done synchronously from
–within this callback, errors triggered by them will be returned via
–the normal pci_check_whatever() api, no new error_detected() callback
–will be issued due to an error happening here. However, such an error
–might cause IOs to be re-blocked for the whole segment, and thus
–invalidate the recovery that other devices on the same segment might
–have done, forcing the whole segment into one of the next states,
–that is link reset or slot reset.

+but not restart operations. This is callback is made if all drivers on
+a segment agree that they can try to recover and if no automatic link reset
+was performed by the HW. If the platform can't just re-enable IOs without
+a slot reset or a link reset, it wont call this callback, and instead
+will have gone directly to STEP 3 (Link Reset) or STEP 4 (Slot Reset)

+

+>>> The following is proposed; no platform implements this yet:

+>>> Proposal: All I/O's should be done synchronously from within

+>>> this callback, errors triggered by them will be returned via

+>>> the normal pci_check_whatever() API, no new error_detected()

+>>> callback will be issued due to an error happening here. However,

+>>> such an error might cause IOs to be re-blocked for the whole

+>>> segment, and thus invalidate the recovery that other devices

+>>> on the same segment might have done, forcing the whole segment

+>>> into one of the next states, that is, link reset or slot reset.

– Result codes:

– – PCIERR_RESULT_RECOVERED

+The driver should return one of the following result codes:

+ – PCI_ERS_RESULT_RECOVERED

Driver returns this if it thinks the device is fully

– functional and thinks it is ready to start

+ functional and thinks it is ready to start

normal driver operations again. There is no

guarantee that the driver will actually be

allowed to proceed, as another driver on the

[PATCH]: Documentation: Updated PCI Error Recovery

same segment might have failed and thus triggered a slot reset on platforms that support it.

- - PCIERR_RESULT_NEED_RESET
- + - PCI_ERS_RESULT_NEED_RESET

Driver returns this if it thinks the device is not recoverable in it's current state and it needs a slot reset to proceed.

- - PCIERR_RESULT_DISCONNECT
- + - PCI_ERS_RESULT_DISCONNECT

Same as above. Total failure, no recovery even after reset driver dead. (To be defined more precisely)

->>> The current ppc64 implementation does not implement this callback.

-

- 3) link_reset()

-

- This is called after the link has been reset. This is typically
- a PCI Express specific state at this point and is done whenever a
- non-fatal error has been detected that can be "solved" by resetting
- the link. This call informs the driver of the reset and the driver
- should check if the device appears to be in working condition.
- This function acts a bit like 2) mmio_enabled(), in that the driver
- is not supposed to restart normal driver I/O operations right away.
- Instead, it should just "probe" the device to check it's recoverability
- status. If all is right, then the core will call resume() once all
- drivers have ack'd link_reset().

+ The next step taken depends on the results returned by the drivers.
+ If all drivers returned PCI_ERS_RESULT_RECOVERED, then the platform
+ proceeds to either STEP3 (Link Reset) or to STEP 5 (Resume Operations).

+

+ If any driver returned PCI_ERS_RESULT_NEED_RESET, then the platform
+ proceeds to STEP 4 (Slot Reset)

+

+>>> The current powerpc implementation does not implement this callback.

+

+

+STEP 3: Link Reset

+-----

+ The platform resets the link, and then calls the link_reset() callback
+ on all affected device drivers. This is a PCI-Express specific state
+ and is done whenever a non-fatal error has been detected that can be
+ "solved" by resetting the link. This call informs the driver of the
+ reset and the driver should check to see if the device appears to be
+ in working condition.

+

+ The driver is not supposed to restart normal driver I/O operations
+ at this point. It should limit itself to "probing" the device to
+ check it's recoverability status. If all is right, then the platform
+ will call resume() once all drivers have ack'd link_reset().

Result codes:

- (identical to mmio_enabled)
- + (identical to STEP 3 (MMIO Enabled))

->>> The current ppc64 implementation does not implement this callback.
+The platform then proceeds to either STEP 4 (Slot Reset) or STEP 5
(Resume Operations).

- 4) slot_reset()
+>>> The current powerpc implementation does not implement this callback.

- This is called after the slot has been soft or hard reset by the
-platform. A soft reset consists of asserting the adapter #RST line
-and then restoring the PCI BARs and PCI configuration header. If the
-platform supports PCI hotplug, then it might instead perform a hard
-reset by toggling power on the slot off/on. This call gives drivers
-the chance to re-initialize the hardware (re-download firmware, etc.),
-but drivers shouldn't restart normal I/O processing operations at
-this point. (See note about interrupts; interrupts aren't guaranteed
-to be delivered until the resume() callback has been called). If all
-device drivers report success on this callback, the platform will call
-resume() to complete the error handling and let the driver restart
-normal I/O processing.

+
+STEP 4: Slot Reset

+-----
+The platform performs a soft or hard reset of the device, and then
+calls the slot_reset() callback.

+
+A soft reset consists of asserting the adapter #RST line and then
+restoring the PCI BAR's and PCI configuration header to a state
+that is equivalent to what it would be after a fresh system
+power-on followed by power-on BIOS/system firmware initialization.
+If the platform supports PCI hotplug, then the reset might be
+performed by toggling the slot electrical power off/on.

+
+It is important for the platform to restore the PCI config space
+to the "fresh poweron" state, rather than the "last state". After
+a slot reset, the device driver will almost always use its standard
+device initialization routines, and an unusual config space setup
+may result in hung devices, kernel panics, or silent data corruption.

+
+This call gives drivers the chance to re-initialize the hardware
+(re-download firmware, etc.). At this point, the driver may assume
+that the card is in a fresh state and is fully functional. In
+particular, interrupt generation should work normally.

+
+Drivers should not yet restart normal I/O processing operations
+at this point. If all device drivers report success on this
+callback, the platform will call resume() to complete the sequence,

+and let the driver restart normal I/O processing.

A driver can still return a critical failure for this function if it can't get the device operational after reset. If the platform -previously tried a soft reset, it might now try a hard reset (power +previously tried a soft reset, it might now try a hard reset (power cycle) and then call slot_reset() again. If the device still can't be recovered, there is nothing more that can be done; the platform will typically report a "permanent failure" in such a case. The device will be considered "dead" in this case.

+Drivers for multi-function cards will need to coordinate among +themselves as to which driver instance will perform any "one-shot" +or global device initialization. For example, the Symbios sym53cxx2 +driver performs device init only from PCI function 0:

```
+  
++ if (PCI_FUNC(pdev->devfn) == 0)  
++ sym_reset_scsi_bus(np, 0);  
+
```

Result codes:

```
- - PCIERR_RESULT_DISCONNECT  
+ - PCI_ERS_RESULT_DISCONNECT
```

Same as above.

```
->>> The current ppc64 implementation does not try a power-cycle reset  
->>> if the driver returned PCIERR_RESULT_DISCONNECT. However, it should.
```

```
-  
- 5) resume()  
+Platform proceeds either to STEP 5 (Resume Operations) or STEP 6 (Permanent  
+Failure).
```

```
- This is called if all drivers on the segment have returned  
-PCIERR_RESULT_RECOVERED from one of the 3 previous callbacks.  
-That basically tells the driver to restart activity, tht everything  
-is back and running. No result code is taken into account here. If  
-a new error happens, it will restart a new error handling process.
```

```
-  
-That's it. I think this covers all the possibilities. The way those  
-callbacks are called is platform policy. A platform with no slot reset  
-capability for example may want to just "ignore" drivers that can't  
+>>> The current powerpc implementation does not currently try a  
+>>> power-cycle reset if the driver returned PCI_ERS_RESULT_DISCONNECT.  
+>>> However, it probably should.
```

```
+  
+  
+STEP 5: Resume Operations
```

```
+-----  
+The platform will call the resume() callback on all affected device  
+drivers if all drivers on the segment have returned  
+PCI_ERS_RESULT_RECOVERED from one of the 3 previous callbacks.  
+The goal of this callback is to tell the driver to restart activity,
```

[PATCH]: Documentation: Updated PCI Error Recovery

+that everything is back and running. This callback does not return
+a result code.

+

+At this point, if a new error happens, the platform will restart
+a new error recovery sequence.

+

+STEP 6: Permanent Failure

+-----

+A "permanent failure" has occurred, and the platform cannot recover
+the device. The platform will call error_detected() with a
+pci_channel_state value of pci_channel_io_perm_failure.

+

+The device driver should, at this point, assume the worst. It should
+cancel all pending I/O, refuse all new I/O, returning -EIO to
+higher layers. The device driver should then clean up all of its
+memory and remove itself from kernel operations, much as it would
+during system shutdown.

+

+The platform will typically notify the system operator of the
+permanent failure in some way. If the device is hotplug-capable,
+the operator will probably want to remove and replace the device.
+Note, however, not all failures are truly "permanent". Some are
+caused by over-heating, some by a poorly seated card. Many
+PCI error events are caused by software bugs, e.g. DMA's to
+wild addresses or bogus split transactions due to programming
+errors. See the discussion in powerpc/eeh-pci-error-recovery.txt
+for additional detail on real-life experience of the causes of
+software errors.

+

+

+Conclusion; General Remarks

+-----

+The way those callbacks are called is platform policy. A platform with
+no slot reset capability may want to just "ignore" drivers that can't
recover (disconnect them) and try to let other cards on the same segment
recover. Keep in mind that in most real life cases, though, there will
be only one driver per segment.

-Now, there is a note about interrupts. If you get an interrupt and your
+Now, a note about interrupts. If you get an interrupt and your
device is dead or has been isolated, there is a problem :)

-

-After much thinking, I decided to leave that to the platform. That is,
-the recovery API only precies that:

+The current policy is to turn this into a platform policy.

+That is, the recovery API only requires that:

- There is no guarantee that interrupt delivery can proceed from any
device on the segment starting from the error detection and until the
-restart callback is sent, at which point interrupts are expected to be
+resume callback is sent, at which point interrupts are expected to be

fully operational.

- - There is no guarantee that interrupt delivery is stopped, that is, a driver that gets an interrupt after detecting an error, or that detects an error within the interrupt handler such that it prevents proper
+ - There is no guarantee that interrupt delivery is stopped, that is, a driver that gets an interrupt after detecting an error, or that detects an error within the interrupt handler such that it prevents proper ack'ing of the interrupt (and thus removal of the source) should just
- return IRQ_NOTHANDLED. It's up to the platform to deal with that condition, typically by masking the irq source during the duration of
+ return IRQ_NOTHANDLED. It's up to the platform to deal with that condition, typically by masking the IRQ source during the duration of the error handling. It is expected that the platform "knows" which interrupts are routed to error-management capable slots and can deal
- with temporarily disabling that irq number during error processing (this
+ with temporarily disabling that IRQ number during error processing (this isn't terribly complex). That means some IRQ latency for other devices sharing the interrupt, but there is simply no other way. High end platforms aren't supposed to share interrupts between many devices anyway :)

+>>> Implementation details for the powerpc platform are discussed in
+>>> the file Documentation/powerpc/eeh-pci-error-recovery.txt

+
+>>> As of this writing, there are six device drivers with patches
+>>> implementing error recovery. Not all of these patches are in
+>>> mainline yet. These may be used as "examples":

+>>>
+>>> drivers/scsi/ipr.c
+>>> drivers/scsi/sym53cxx_2
+>>> drivers/net/e100.c
+>>> drivers/net/e1000
+>>> drivers/net/ixgb
+>>> drivers/net/s2io.c

-Revised: 31 May 2005 Linas Vepstas <linas@xxxxxxxxxxxxxxxx>

+The End

+-----

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>