Netra™ ft 1800 Developer's Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300   Fax 650 969-9131

Part No. 805-4530-10
February 1999, Revision A

Send comments about this document to: `docfeedback@sun.com`

Contents

Tables

Preface

This guide provides information required by software developers and device driver writers who wish to use OEM devices in the Netra ft 1800.

For an operational view of the Configuration Management System (CMS), and guidance on how to provide CMS support in device drivers, refer to the *Netra ft 1800 CMS Developer's Guide* (Part Number 805-7899-10).

The guide assumes that readers have experience in using the Solaris DDI/DDK interfaces.

How This Book Is Organized

The guide contains the following chapters and appendix:

Chapter 1 describes the steps necessary to integrate an OEM device into a Netra ft 1800 system.

Chapter 2 explains the rules that should be followed in order to harden fully a driver.

Chapter 3 explains the requirements for enabling a device to be added, deleted, moved or replaced without requiring a system reboot or causing a loss of other system services.

Chapter 4 provides a complete example of the topics covered in the first three chapters.

Chapter 5 explains how to use the test harness to simulate hardware faults in order to test the resilience of a hardened device driver.

Chapter 6 contains information concerning the PCI module EEPROM

Glossary contains a list and definitions of words and phrases used in the Netra ft 1800 documentation.

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Software Reference	*Netra ft 1800 Reference Manual*	805-4532-10
CMS API Development	*Netra ft 1800 CMS API Developer's Guide*	805-5870-10
CMS Development	*Netra ft 1800 CMS Developer's Guide*	805-7899-10
EEPROM Configuration	*Netra ft 1800 Module EEPROM v.4 Data File Specifications*	950-3407-10

Sun Documentation on the Web

The `docs.sun.com`sm web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`docfeedback@sun.com`

Please include the part number of your document in the subject line of your email.

CHAPTER **1**

OEM Device Integration

This chapter outlines the steps required to integrate an OEM PCI device into a Netra ft 1800 system.

▼ Module Integration

1. **Create a fault tolerant device driver (see Chapter 2, “Driver Hardening”, Chapter 3, “Handling Faults and Hot Plugging Devices” and Chapter 5, “Test Harness”) .**

2. **Include the CMS interface in the driver (see Chapter 2, “CMS User and System Models” and Chapter 3, “Writing cmsdefs” in the** *Netra ft 1800 CMS Developer’s Guide***, part no. 805-7899-10).**

3. **Assemble the device as a** *module***.**

4. **Locate the module in the system.**

5. **Run the** *abdication* **process.**

CHAPTER 2

Driver Hardening

Driver *hardening* is one of the most important changes required in the Netra ft 1800 system because an inadequately hardened driver compromises the fault tolerance of the system, not just the I/O subsystem being handled by the driver. Total hardening is achieved only by careful analysis of the operation of the driver and I/O device. Different solutions can apply to different devices. This section suggests some guidelines.

Netra ft 1800 device drivers execute within the environment of a virtual fault-free machine (VFFM), and are thus protected from any hardware faults occurring within the trusted core of the actual machine. In order to harden fully a driver (that is, make it resilient to all hardware faults), it is necessary to address those issues arising from faults in I/O space.

The Netra ft 1800 hardware automatically detects gross faults in I/O space, such as bus errors, and the hardware and VFFM code together isolate Solaris software (including drivers) from them, this isolation persisting until the affected hardware is reset. However, drivers still need to be able to cope with the arbitrary (undefined) data values that are returned by reads from I/O space.

The driver must also detect faults generated from beyond the trusted core, such as incorrect data returned by PIO reads or placed in memory by DMA. The driver should not panic or hang or allow the uncontrolled spread of corrupted data as a result of such problems.

To harden a driver fully, follow the rules below. These rules are described in more detail in the remainder of this chapter.

- Each instance of a piece of hardware should be controlled by a separate instance of the device driver.
- Exclusive use of DDI access handle for host read/writes (see `ddi_get(`*n*`)`, `ddi_put(`*n*`)`, `ddi_rep_get(`*n*`)`, `ddi_rep_put(`*n*`)`, where *n* can be 8, 16, 32 or 64).

- Corrupted data detection: the device driver must assume that any data that it might receive from the device could be corrupted. If undesirable consequences might occur from the use or propagation of such data, the data must first be sanity checked.
- Containment of faults: driver-device interactions can be considered as a series of transactions. Each transaction must be validated before its effects are allowed to propagate beyond the driver itself. It is not necessary, however, to validate every step of a transaction. Depending on the consequences of incorrect data at an intermediate step, a single check at the end will often suffice.
- The system must be able to identify whether it is dealing with a pathological interrupt. The driver should return a `DDI_INTR_UNCLAIMED` result unless it detects that the device legitimately asserted interrupt.
- All device writes into system RAM that use DMA must go via the IOMMU into a memory page that is used exclusively by the driver for that PCI slot, thereby preventing a faulty device from corrupting memory not controlled by that driver.
- The driver should ensure that it cannot act as an unlimited drain on the system resources if a command to an I/O card locks up. It should either time out locked requests or impose flow control to limit the number of further requests that can be queued while it is in this state.

Device Driver Instances

The Solaris kernel allows multiple instances of a driver. Each instance has its own data space but shares the text and some global data with the other instances. Each PCI slot in the Netra ft 1800 architecture is independent of all other PCI slots, so a hardware fault in one slot can be completely isolated. The device state is managed on a per instance basis so, unless the driver is designed to handle fail-over internally, it should use a separate instance for each piece of hardware. Note that it is possible to have multiple instances per slot—for example, for multifunction cards.

Exclusive Use of DDI Access Handles

All programmed I/O access by a hardened driver should be via the `ddi_get(`*n*`)`, `ddi_put(`*n*`)`, `ddi_rep_get(`*n*`)` and `ddi_rep_put(`*n*`)` routines and not by accessing the mapped registers directly via the address returned from `ddi_regs_map_setup`. The `ddi_peek` and `ddi_poke` routines should be avoided.

The DDI access mechanism is important because it provides an opportunity to control how data is read into the kernel. DDI access routines enable protection to be provided by constraining the effect of bus timeout traps. They also allow the use of the driver hardening test framework described later in this guide (see Chapter 5, "Test Harness").

Detecting Corrupted Data

This section considers where data corruption can occur and the steps that can be taken to detect it.

Corruption of Device Management and Control Data

The driver should assume that any data received from the device either by PIO or by DMA could potentially be corrupted.

Take extreme care with pointers, array indexes or memory offsets that are read or calculated from data retrieved from the device. Never use such values until they have been checked for being within an expected range and having legal alignment. Such a pointer can be malignant if the device has developed a fault. Malignant pointers are those that will cause the kernel to panic if they are dereferenced.

Even if the pointer is not malignant, it can still be misleading. That is, a pointer can be used directly without causing a panic but might not point to the expected object. The driver should, if possible, further check the pointer against an absolute expected range, or attempt to validate the data obtained through it.

Other data from the board has the potential to disrupt a kernel. For every piece of data read from the board, consider whether unfortunate consequences can result if it is corrupted. Typical examples include data relating to length, channel ids (SAP, VCI, q_ptr, and so forth) and status bits.

If data is used to specify the length of a transfer, ensure that the value is not larger than the buffer itself. Beware of negative lengths; use unsigned variables and comparisons.

If a stream is derived from some form of channel id, ensure that the stream is valid. Remember that, while it might have been valid in the past, it could now have been torn down (`qprocsoff()`). This danger is very real with streams because they are not protected from asynchronous use.

Never loop simply upon a register value. An inadvertent infinite loop can occur if a device breaks and returns *stuck* data. Include a method for breaking out of the loop.

If the retrieved data is expected to remain static, store its value and reference it locally. If recently checked data is re-read from the board, it should be sanity checked once more. Maintain driver state information in main memory, not on the I/O card.

Corruption of Received Data

Device errors can result in corrupted data being placed in receive buffers. Such corruption is indistinguishable from corruption occurring beyond the domain of the device—for example, within a network. Typically, existing software will already be in place to handle such corruption through, for example, integrity checks at the transport layer of a protocol stack or within the application using the device.

If the received data is not going to be subjected to an integrity check at a higher layer—as in the case of a disk driver, for example—it can be integrity checked within the driver itself. Methods of detecting corruption in received data are application-specific issues (typically checksums, CRC).

Faults Detected by the Netra ft 1800 Hardware and VFFM

Access to each PCI bus slot can be enabled or disabled. When access is enabled, read or write cycles occur normally. When disabled, slot accesses return an immediate *transfer-acknowledge* without accessing the bus. Disabled accesses return random data on reads and do nothing on writes. The hardened driver is resilient to any form of corrupted data returned from the device (see “Corruption of Device Management and Control Data” on page 5), so the random data returned poses no additional problems.

When a bus error occurs, the Netra ft 1800 hardware disables the faulty slot. The driver can then continue accessing the module without incurring further bus errors.

In order for the driver to discover that this or some other fault has occurred, the `u4ft_ddi_check_access`(9F) routine is provided.

`u4ft_ddi_check_access`*(9F)*

This function enables a driver to check for faults in the communication path between itself and the device it controls. The check returns a bitmask representing the cumulative set of faults detected. As some bits can be irrelevant to certain drivers, the result should be masked to select those bits that impact the driver. For example, the driver for a simple, non-interrupting, non-DMA device should examine only the PIO bit. All bits not currently used are reserved for future expansion, and can be defined at a future date.

Currently, `u4ft_ddi_check_access`() indicates that the infrastructure has:

- Protected against a Bus timeout or Bus error
- Prevented the device from using DMA to write into main memory
- Disabled a stuck interrupt
- Disabled another device on which this one depends

When a significant access fault occurs, the driver should use u4ft_ddi_report_fault() to report it.

Containment of Faults

Preservation of system integrity requires that faults are detected before they uncontrollably alter the system state. Consequently, steps must be taken to test for faults whenever data returned from the device is going to be used by the system.

- The u4ft_ddi_check_access() call should be made at significant junctures, such as just prior to passing a data block to the upper layers.
- Data must not be forwarded out of the driver if the device has failed.
- The driver must consider other possible impacts of the failure on the integrity of the system. The driver must ensure that kernel resources are not permanently lost through being unable to forward data (that is, free up memory). Threads should not remain blocked waiting for signals that will, as a result of the failure, never be generated.
- The driver should limit its processing while in the failed state (for example, to free messages in wput routines, attempts to disable permanently interrupts from a failed board, and so forth).

Stuck Interrupts

The system needs to be able to identify whether it is dealing with a pathological interrupt. A persistently asserted interrupt will severely affect system performance, almost certainly stalling a single processor machine. The driver should return a DDI_INTR_UNCLAIMED result unless it detects that the device legitimately asserted an interrupt (that is, the device actually requires the driver to do some useful work).

The system maintains a count of consecutive false interrupts, resetting the total when a good interrupt is identified. If the count reaches a predetermined level, the system disables interrupts from the offending PCI slot and subsequent calls to u4ft_ddi_check_access() return U4FT_ACC_NO_INT.

It is sometimes not easy for the driver to identify that a particular interrupt is a hoax. If an Rx interrupt is indicated but no new buffers have been made available, it is obvious that no work was needed. When this is an isolated occurrence, it is not a problem as the actual work might already have been completed by another routine (read service, for example). This is the rationale behind the system allowing a number of apparently hoax interrupts to occur before taking defensive action.

A hung device, while appearing always to have work to do, is simply failing to update its buffer descriptors. The driver should ensure that it is not fooled by such repetitive requests.

The legitimacy of other, more incidental, interrupts is much harder to certify. To this end, an interrupt expected flag is a useful tool for evaluating whether an interrupt is valid. Consider an interrupt such as *descriptor free*. This can be generated if, previously, all the device's descriptors had been allocated. If the driver detects that it has taken the last descriptor from the card, it can set an interrupt expected flag. If this flag is not set when the associated interrupt is delivered, it should be treated with suspicion.

Some informative interrupts might not be predictable, such as one that indicates that a medium has become disconnected or frame sync has been lost. The easiest method of detecting whether such an interrupt is stuck is to mask this particular source on first occurrence until the next poller cycle. If the interrupt occurs again while disabled, this should be taken as a false interrupt. Beware that some devices have interrupt status bits that can be read even if the mask register has disabled the associated source; they might not be causing the interrupt. The driver designer might be in a position to devise more appropriate algorithms specific to their device.

Particular care should be taken to avoid looping on interrupt status bits indefinitely. Break such loops if none of the status bits set at the start of a pass required any real work.

DMA Isolation

A defective PCI device might be able to initiate improperly a DMA transfer over the bus. This data transfer could corrupt good data that had previously been delivered. As well as a device failing and corrupting its own DMA space, it is possible for a failing device to generate a corrupt address that can contaminate memory not even belonging to this driver.

The IOMMU does not allow malignant DMA to corrupt memory areas other than pages mapped as writable for DMA. It is therefore important that data is written using DMA only into pages that are owned by that driver instance and not shared by any other kernel structure. While the page in question is mapped as writable for DMA, the driver should assume that any data in that page is susceptible to corruption, and the page must be unmapped from the IOMMU first if it is to be passed on beyond the driver, or before any validation of the data is carried out.

Alternatively, the driver can choose to copy the data into a safe part of memory before processing it. If this is done, the data must first be synchronized using `ddi_dma_sync`(9F).

ddi_dma_syncs should be used as defined in the DDI (with a SYNC_FOR_DEV before using DMA to write to a device, and a SYNC_FOR_CPU after using DMA to write from the device). This enables the test harness to inject faults into DMA transfers (see Chapter 5, “Test Harness”).

Note – The undocumented dvma_reserve and dvma_release interfaces are not supported on a Netra ft 1800.

Some PCI cards are able to use dual address cycles (64-bit addresses) to avoid the IOMMU. This gives the device the potential to corrupt any region of main memory. Hardened device drivers must not attempt to make use of such a mode and should disable it. The Netra ft 1800 hardware prevents the use of Dual Address cycles.

Thread Issues

Often, when a device fails, kernel panics are caused by the unexpected interaction of kernel threads. When a device fails, there is opportunity for threads to interact in ways that the designer or implementor has not planned for. It is important to remind driver designers of this potential source of kernel panic and have them ensure that such risks are addressed. Although device driver writers should naturally consider carefully how threads will weave through their code during normal operation, it is important to extend this modeling to cover fault conditions.

Early termination of processing routines can inadvertently leave Condition Variable waiters blocked, because an expected signal is never given. Attempting to inform other modules of the failure or handling unanticipated callbacks can result in undesirable thread interactions. It is important to consider the sequence of mutex acquisition and relinquishing that can occur during device failures.

Threads that originate in an upstream streams module can run into unfortunate paradoxes if used to return to that module unexpectedly. Thought should be given to using alternative threads to handle exception messages. For instance, a wput procedure might consider using a read-side service routine to communicate an M_ERROR rather than doing it directly with a read-side putnext.

A failing streams device that cannot be quiesced during close (because of the fault) can generate an interrupt after the stream has been dismantled. The interrupt handler must ensure that it does not attempt to use a stale stream pointer in an attempt to process the message.

Threats From Above

While putting effort into considering how to protect the system from defective hardware, the driver designer should also take opportunity to protect against driver misuse. The driver must ensure that it does not crash the system while processing top down requests. Although the driver must assume that the kernel infrastructure is always correct (a trusted core), the requests for processing passed to it can be potentially destructive.

For example, a user can request an action to be performed upon a user-supplied data block (M_IOCTL) that is smaller than that indicated in the control part of the message. The driver should not blindly trust that a user application will always be right.

The design should look at the construction of each type of request (`ioctl`) that it can receive with a view to the potential harm that it could cause. The driver should undertake sufficient checks to ensure that it does not attempt to process malignant `ioctls`.

Adaptive Strategies

A driver can continue to provide a service with faulty hardware, attempting to work around the identified problem by adopting an alternative strategy for accessing the device. Given the unpredictability of broken hardware and the risk associated with adding additional complexity to the design, adaptive strategies are not generally recommended. At most, they should be limited to periodic interrupt polling and retry attempts. Periodically retrying the device can allow the driver to identify that a device has recovered. Periodic polling can take on the responsibility of the interrupt mechanism when a driver has been forced to disable interrupts.

Ideally, a system will always have an alternative device to provide a vital system service. Service multiplexors in kernel or user space provide the best possible method of maintaining system services when an individual device fails. Such practices are beyond the scope of this guide.

CHAPTER 3

Handling Faults and Hot Plugging Devices

In addition to hardening the driver, it is essential to ensure that faulty devices can be replaced, and that devices can be added, deleted or moved without requiring a system reboot or loss of unrelated system services

Features

The following additional features must be considered:

- All devices should support hot plugging. As a minimum, this means that the driver should always allow its minor devices to close successfully whatever faults have occurred, and once all minor devices for an instance have closed, the driver should always enable the instance to be detached successfully.

  It must be possible to detach and subsequently re-attach individual instances independently while other instances continue working. If this happens, minor numbers should continue to map onto the same instance number.
- The Netra ft 1800 framework provides a Device State Model (see Chapter 2, "CMS User and System Models" of the *Netra ft 1800 CMS Developer's Guide*, Part Number 805-7899-10) that maintains device state on behalf of the driver on a per instance basis, thus relieving the driver of the task of maintaining this itself. The `u4ft_ddi_dev_is_usable`() command enables the driver to tell if the current state is usable.
- Fault reporting using `u4ft_ddi_report_fault`() should be used to enable the system to deal automatically with a faulty device.

  Although it is permissible to retry a failed command before deciding to report a device as faulty, care should be taken not to allow too long a time to elapse before the fault is reported.

- The driver should ensure that it has a mechanism for making periodic health checks on the device.

Hot Plugging Driver Issues

The following subsections consider how to:

- Bring a device online
- Take a device offline
- Check the current device state
- Report faults

Bringing a Device for Hot Insertion Online

The process of bringing online uses the normal `attach`(9E) entry point of a driver, which is called with the command `DDI_ATTACH` to initialize the device instance.

Any activities required for the device to run, such as downloading firmware, must be scheduled when the device is online and not simply run at boot time. This can be achieved in the kernel as part of the `attach`, or in the CMS through the driver's `cmsdef` (see Chapter 2, "CMS User and System Models" and Chapter 3, "Writing cmsdefs" of the *Netra ft 1800 CMS Developer's Guide*, Part Number 805-7899-10).

Do not assume that the PROM will initialize your device in any way (for example, by resetting the hardware or running fcode). Any properties that would have been created by the fcode must be created by the driver's `cmsdef` instead.

Care should be taken with persistent data (globals and static locals in C). `Attach` code should not assume that such variables have been cleared unless this has been explicitly done in `detach`(9E).

In the standard Solaris driver model, the kernel calls the driver-probe routine at boot time. If this returns `DDI_PROBE_FAILURE`, the driver's `attach` routine is never called. The driver should return `DDI_PROBE_DONTCARE` in the probe call, which results in the `attach` routine always being called irrespective of whether the device actually exists. Alternatively, the driver can specify `nulldev`(9F) for a probe routine, which has the same effect.

Taking a Device for Hot Removal Offline

The process of taking offline uses the normal `detach`(9E) entry point of the driver.

The kernel checks the driver state and, if the driver instance is not open, the kernel invokes the driver's `detach`(9E) entry point with the command `DDI_DETACH`.

Note that the driver's detach entry point can be called with the DDI_DETACH command because memory is low (and the kernel is attempting to unload the driver module) or because the hardware for this instance will be removed. There is no information available to the driver to distinguish between these two cases.

The DDI_DETACH operation is the inverse of DDI_ATTACH. The DDI_DETACH handling should only de-allocate the data structures for the specified instance. Driver global resources must only be de-allocated in _fini(). Since instances are assigned in arbitrary order, the driver must be able to handle instances that are presented out of sequence. In fact, no assumption should be made about the instance number.

Wherever possible, the driver should ensure that the DDI_DETACH will succeed. If the driver fails the DDI_DETACH, the driver should clearly indicate, using console messages, what the user should do to ensure that the next DDI_DETACH succeeds. However, flooding the console with messages should be avoided.

The DDI_DETACH command code should perform the following actions:

- Check whether DDI_DETACH is safe.

  The driver can assume that the device has been closed before DDI_DETACH is issued. However, there might be outstanding callbacks that cannot be canceled, or the device might not currently be in a state that permits it to be reliably shut down and restarted later. While timeouts or callbacks are still active, proper locking must be enforced.

  The driver should not block indefinitely while waiting for callback completion or for the device to become idle.

  Devices that maintain some state after a close operation must be carefully analyzed. When a driver, not currently in use, is automatically unloaded (for example, because the system memory is low) and later automatically reloaded when the user opens the device, this can cause undesirable operation.

  For example, a tape driver that supports non-rewinding tape access can fail the detach operation when the tape head is not at the beginning of the tape. If the drive is powered down, the head position will be lost.

- Shut down the device and disable interrupts.

  A device needs enough hardware information and support to be able to shut off and restart interrupts. This could already be coded in the driver as a function of the existing detach routines.

- Remove any interrupts registered with the system.
- Cancel any outstanding timeouts and callbacks.
- Quiesce or remove any driver threads.
- De-allocate memory resources.

  The driver should be capable of being unloaded without memory leaks.

- Unmap any mapped device registers.

- Execute `ddi_set_driver_private`(dip, NULL) if appropriate.
- Free the `softstate` structure for this instance.

When there is a failure during `detach`, the driver must decide whether to continue the `detach` and return success, or undo the `detach` actions completed to that point and return failure. Undoing the `detach` can be risky and it is usually preferable to continue the `detach` operation.

`DDI_DETACH` can be followed by power interruption; any further references to the device must be preceded by a `DDI_ATTACH`.

Note the following when you use `timeout`() routines:

- Ensure that you do not have multiple instances of the same routine running. For example, a second call to

```
un->un_tid = timeout(XX_to, arg, ticks);
```

  causes `un_tid` to be overwritten, removing the ability to `untimeout`() the first `timeout`() routine.
- Self-rescheduling `timeout`() routines are routines that contain a call to `timeout`() to reschedule themselves. Particular care is required to kill them.

Checking the Current Device State

`u4ft_ddi_dev_is_usable`*(9F)*

This function enables the driver to determine whether the device's current state, as maintained by the framework, is one in which it is usable (INITIALISING, ONLINE, DEGRADED) or not usable (FAILED, OFFLINE). Note that the driver does not normally try to reference a device that is OFFLINE. Generally, the device state will be FAILED in response to a high severity `u4ft_ddi_report_fault`, but could also be set to FAILED as a result of user intervention.

If the device is not usable (OFFLINE or FAILED), the driver should reject all new and outstanding I/O requests, returning (if possible) an appropriate error code (for example, EIO). For a streams driver, `M_ERROR` or `M_HANGUP`, as appropriate, should be put upstream to indicate that the driver is not usable.

The state of the device should be checked at each major entry point, optionally before committing resources to an operation, and after reporting a fault. If, at any stage, the device is found to be unusable, the driver should perform any cleanup actions that are required (for example, releasing resources) and return in a timely fashion. It should *not* attempt any retry or recovery action, nor does it need to report a fault (the state is not a fault, and it is already known to the framework and management agents). It should mark the current request and any other outstanding or queued requests as complete, again with an error indication if possible.

The `ioctl`() entry point presents a problem in this respect: `ioctl`() operations that imply I/O to the device (for example, formatting a disk) should fail if the device is unusable, while others (such as recover error status) should continue to work. The state check might therefore need to be on a per-command basis. As an alternative, you can implement those operations that work in any state through another entry point or minor device mode, although this might be constrained by issues of compatibility with existing applications.

Note that `close`() should always complete successfully even if the device is unusable. If the device is unusable, the interrupt handler should return `DDI_INR_UNCLAIMED` for any subsequent interrupts. This will eventually result in the interrupt being disabled.

Fault Reporting

`u4ft_ddi_report_fault`*(9F)*

The severity parameter indicates the impact and potential recoverability of the fault, and is used by the fault-management components of the system to determine the appropriate action to take in response to the fault. This action can cause a change in the device state. A high severity fault will cause the device state to be changed to FAILED and a mid severity fault will cause the device state to be changed to DEGRADED.

A board should be failed if:

- A PIO error is detected.
- Corrupted data is detected.
- It has locked up.
- It gives you a status field that purports to indicate a state that physically cannot happen within the confines of the implementation

Drivers should avoid reporting the same fault repeatedly, if possible. In particular, it is redundant (and undesirable) for them to report any errors if the device is already in an unusable state (see `u4ft_ddi_dev_is_usable`() above).

Periodic Health Checks

A latent fault is one that will not show itself until some other action occurs. For example, a hardware failure occurring in a PCI card that is a cold stand-by could remain undetected until a fault occurs on the master PCI card. At this point it will be discovered that the system now contains two defective PCI cards and might be unable to continue operation.

As a general rule, latent faults that are allowed to remain undetected will eventually cause system failure. Without latent fault checking, the overall reliability of a redundant system is jeopardized.

To avoid this, a device driver must detect latent faults and report them in the same way as other faults.

The driver should ensure that it has a mechanism for making periodic health checks on the device. In a fault tolerant situation where the device can be the secondary or fail-over device, it is essential to identify a failed secondary device so that it can be repaired or replaced before any failure in the primary device occurs. Periodic health checks should be scheduled from a timeout call back at a rate appropriate to the device and the service it provides.

Periodic health checks can:

- Run a quick access check to the board (write, read), then check the device with `u4ft_ddi_check_access`().
- Check a register or memory location on the device whose value the driver expects to have deterministically altered since the last poll.

  Features of a device that typically exhibit deterministic behavior include heartbeat semaphores, device timers (for example, local `lbolt` used by download), and event counters. Reading an updated predictable value from the device gives a degree of confidence that things are proceeding satisfactorily.
- Time-stamp outgoing requests (transmit blocks or commands) when issued by the driver.

  The periodic health check can look for any over-age requests that have not completed.
- Initiate an action on the device that should be completed before the next scheduled check.

  If this action results in an interrupt, this is an ideal way of ensuring that the device’s circuitry is still capable of delivering an interrupt.

CHAPTER **4**

Example Character Device Driver

This chapter provides an example of a character device driver.

Character Device Example: `ftxx.c`

The example code shows how a driver for a simple device might be structured. The example has the following features:

- The `ftxx_dog`() function serves the dual purpose of timing out any command that takes too long and probing the device at regular intervals when no command is in progress.
- The state check exists in `open`(), but not in `close`() – the close should complete even if the device has failed.
- At the end of each sequence of accesses (in `ftxx_reset`() and `ftxx_cmd`()), the driver calls `u4ft_ddi_check_access`() to check for I/O errors and reports a fault if any significant ones have been detected.

CODE EXAMPLE 4-1 Character Device Example ft*xx.c*

```
/*
 *  Copyright (C) 1998 Sun Microsystems, Network Systems.
 *  All Rights Reserved.
 */
#pragma ident  "@(#)ftxx.c 1.4 97/06/10 SMI"
/*
 *  ftxx.c - Sample driver for the Ultra-4FT product
 */
/*
 *  Included files.
 */
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/uio.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/stream.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/modctl.h>
#include <sys/mkdev.h>
#include <sys/errno.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/consdev.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/dditypes.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include "u4ftddi.h"                /* Ultra-4FT extensions to DDI  */
#include "ftxx.h"                   /* our own definitions file     */
#ifndef __CEXTRACT__
#include "ftxx.H"                   /* our own prototype file       */
#endif  /* __CEXTRACT__ */

/*
 *  External references
 */
extern struct mod_ops mod_driverops;

/*
 * Variables defined here and visible internally only
 */
static void *ftxx_statep;

/*
 *  Exported variables
 */
/******************************************************************/
/*
 *  System interface structures
 */
/*
 *  cb_ops structure defining the driver entry points
 */
static struct cb_ops ftxx_cb_ops =
{
    ftxx_open,                      /* open                         */
    ftxx_close,                     /* close                        */
    nodev,                          /* strategy                     */
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
    nodev,                          /* print                  */
    nodev,                          /* dump                   */
    ftxx_read,                      /* read                   */
    ftxx_write,                     /* write                  */
    ftxx_ioctl,                     /* ioctl                  */
    nodev,                          /* devmap                 */
    nodev,                          /* mmap                   */
    nodev,                          /* segmap                 */
    nochpoll,                       /* poll                   */
    ddi_prop_op,                    /* prop op                */
    NULL,                           /* ! STREAMS              */
    D_NEW | D_MP                    /* MT/MP Safe             */
};
/*
 *  dev_ops structure defining driver autoconfiguration routines
 */
static struct dev_ops ftxx_dev_ops =
{
    DEVO_REV,                       /* devo_rev               */
    0,                              /* devo_refcnt            */
    ftxx_getinfo,                   /* devo_getinfo           */
    nulldev,                        /* devo_identify          */
    ftxx_probe,                     /* devo_probe             */
    ftxx_attach,                    /* devo_attach            */
    ftxx_detach,                    /* devo_detach            */
    nodev,                          /* devo_reset             */
    &ftxx_cb_ops,                   /* devo_cb_ops            */
    NULL,                           /* devo_bus_ops           */
    ddi_power,                      /* devo_power             */
};
/*
 * module configuration section
 */
static struct modldrv modldrv =
{
    &mod_driverops,
    "ftxx driver",
    &ftxx_dev_ops,
};
static struct modlinkage modlinkage =
{
    MODREV_1,
    &modldrv,
    NULL
};
/*******************************************************************/
/*
 * function     - _init
 * description  - initializes the driver state structure and installs
 *                the driver module into the kernel
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 * inputs       - none
 * outputs      - success or failure of module installation
 */

int
_init(void)
{
    int err;
    err = ddi_soft_state_init(&ftxx_statep, sizeof(ftxx_state_t), 1);
    if (err == 0)
        if ((err = mod_install(&modlinkage)) != 0)
            ddi_soft_state_fini(&ftxx_statep);
    return err;
}

/*
 * function     - _info
 * description  - provide information about a loaded module
 * inputs       - module information
 * outputs      - success or failure of information request
 */

int
_info(struct modinfo *modinfop)
{
    return mod_info(&modlinkage, modinfop);
}

/*
 * function     - _fini
 * description  - removes a module from the kernel and frees
 *                the driver soft state memory
 * inputs       - none
 * outputs      - success or failure of module removal
 */

int
_fini(void)
{
    int err;
    if ((err = mod_remove(&modlinkage)) == 0)
        ddi_soft_state_fini(&ftxx_statep);
    return err;
}

/*
 * function     - ftxx_probe
 * description  - the probe routine always returns don't care. Hotplug
 *                drivers can't, in general, probe for their hardware
 *                because it may not (yet) be present and powered on.
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 * inputs       - device information structure
 * outputs      - DDI_PROBE_DONTCARE
 */

static int
ftxx_probe(dev_info_t *dip)
{
    return DDI_PROBE_DONTCARE;
}

/*
 * function     - ftxx_getinfo
 * description  - routine used to provide information on the driver
 * inputs       - device information structure, command, command arg
 *              (which is really a dev_t), storage area for the result
 * outputs      - DDI_SUCCESS or DDI_FAILURE
 */

static int
ftxx_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void
**result)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    int dev;
    dev = getminor((dev_t)arg);
    instance = FTXX_MINOR_TO_INST(dev);
    switch (cmd)
    {
    default:
        return DDI_FAILURE;
    case DDI_INFO_DEVT2INSTANCE:
        *result = (void *)instance;
        return DDI_SUCCESS;
    case DDI_INFO_DEVT2DEVINFO:
        ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
        if (ftxx_ssp == NULL)
            return DDI_FAILURE;
        *result = (void *)ftxx_ssp->ftxx_dip;
        return DDI_SUCCESS;
    }
}

/******************************************************************/
/*
 * function     - ftxx_attach
 * description  - this routine is responsible for allocating and
 *                initializing certain driver resources, and
 *                initializing the device.
 * inputs       - device information structure, DDI_ATTACH command
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 * outputs      - DDI_SUCCESS or DDI_FAILURE
 */

static int
ftxx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ftxx_state_t *ftxx_ssp;
    ddi_iblock_cookie_t ibc;
    int instance;
    int rslt;
    int dev;
    switch (cmd)
    {
    default:
        return DDI_FAILURE;
    case DDI_ATTACH:
        break;
    }

    /*
     *  Main-line attach code ...
     *
     *  Start by acquiring the iblock_cookie needed to initialize
     *  mutexes
     */
    if (ddi_get_iblock_cookie(dip, FTXX_INTR_0, &ibc) != DDI_SUCCESS)
        return DDI_FAILURE;

    /*
     *  Allocate soft-state structure
     */
    instance = ddi_get_instance(dip);
    if (ddi_soft_state_zalloc(ftxx_statep, instance) != DDI_SUCCESS)
        return DDI_FAILURE;
    if ((ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance)) == NULL)
    {
        ftxx_unattach(ftxx_ssp, instance);
        return DDI_FAILURE;
    }

    /*
     *  Initialize soft-state
     */
    ftxx_ssp->ftxx_dip = dip;
    ftxx_ssp->ftxx_ibc = ibc;
    mutex_init(ftxx_ssp->ftxx_mutex, "ftxx_mutex", MUTEX_DRIVER, ibc);
    cv_init(ftxx_ssp->ftxx_cv, "ftxx_cv", CV_DRIVER, NULL);

    /*
     * Create device minor node
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
     */
    dev = FTXX_INST_TO_MINOR(instance);
   rslt = ddi_create_minor_node(dip, "ftxx", S_IFCHR, dev, DDI_PSEUDO,
0);
    if (rslt != DDI_SUCCESS)
    {
        ftxx_unattach(ftxx_ssp, instance);
        return DDI_FAILURE;
    }
    ftxx_ssp->ftxx_minor = 1;

    /*
    *  Initialize the watchdog data; the watchdog interval is fixed at
    *  1 second, but the command timeout limit is defined by a property
     */
    ftxx_ssp->ftxx_dogticks = drv_usectohz(1000000);
    ftxx_ssp->ftxx_limit = ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0,
                                 FTXX_LIMIT, FTXX_DEFAULT_LIMIT);
       mutex_enter(ftxx_ssp->ftxx_mutex);
    if(ftxx_do_online(ftxx_ssp) != DDI_SUCCESS) {
        ftxx_do_offline(ftxx_ssp);
        mutex_exit(ftxx_ssp->ftxx_mutex);
        ftxx_unattach(ftxx_ssp, instance);
        return DDI_FAILURE;
    }
    return DDI_SUCCESS;

    /*
     *  Report driver and return success
     */
    ddi_report_dev(dip);
}

/*
 * function     - ftxx_detach
 * description  - routine that prepares a module to be unloaded.  The
 *                framework will not call this routine if the device is
 *                 open.
 * inputs       - device information structure, DDI_DETACH command
 * outputs      - DDI_SUCCESS or DDI_FAILURE
 */

static int
ftxx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    switch (cmd)
    {
    default:
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
        return DDI_FAILURE;
    case DDI_DETACH:
        instance = ddi_get_instance(dip);
        ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
        if (ftxx_ssp == NULL)
          return DDI_FAILURE;        /* this "can't happen"         */
          mutex_enter(ftxx_spp->ftxx_mutex);
          ftxx_do_offline(ftxx_ssp);
          mutex_exit(ftxx_ssp->ftxx_mutex);
        if (ftxx_ssp->ftxx_mapping)
          return DDI_FAILURE;        /* still mapped => not offline  */
        ftxx_unattach(ftxx_ssp, instance);
        return DDI_SUCCESS;
    }
}

/*
 * function     - ftxx_unattach
 * description  - routine that does the necessary tidying up if an
 *                attach request fails or the driver is to be detached.
 * inputs       - soft state pointer (can be NULL) and instance number
 * outputs      - none
 */

static void
ftxx_unattach(ftxx_state_t *ftxx_ssp, int instance)
{
    if (ftxx_ssp)
    {
        /*
         * Destroy device minor node
         */
        if (ftxx_ssp->ftxx_minor)
            ddi_remove_minor_node(ftxx_ssp->ftxx_dip, NULL);
        /*
         *  Destroy condition variable and mutexes
         */
        cv_destroy(ftxx_ssp->ftxx_cv);
        mutex_destroy(ftxx_ssp->ftxx_mutex);
    }
    /*
     *  Destroy soft state structure for this instance
     */
    ddi_soft_state_free(ftxx_statep, instance);
}

/*****************************************************************/
/*
 * function     - ftxx_dog
 * description  - checks for timeouts when a command is in progress.
```

CODE EXAMPLE 4-1 Character Device Example `ftxx.c` *(Continued)*

```
 *                probes the hardware using the "check" command when
 *                no other command is in progress.
 * inputs       - soft state pointer
 * outputs      - none
 */

static void
ftxx_dog(caddr_t arg)
{
    ftxx_state_t *ftxx_ssp = (ftxx_state_t *)arg;
    int working;
    uint8_t tmp1;
    uint8_t tmp2;
    mutex_enter(ftxx_ssp->ftxx_mutex);
   ftxx_ssp->ftxx_dog_pending = 0;    /* no callback pending now    */

    /*
     *  If we have been requested to stop, or the device is no longer
     *  usable, wakeup anyone waiting for the dog to stop running, and
     *  return without scheduling another callback.
     */
    working = u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip,
U4FT_DONT_WAIT);
    if (ftxx_ssp->ftxx_stop_dog || !working)
        cv_broadcast(ftxx_ssp->ftxx_cv);
    else if (ftxx_ssp->ftxx_busy)
    {

        /*
         *  Command in progress ... count how many ticks it takes.
         *  If it exceeds the limit, abort the command by issuing
         *  a warm reset, and report a fault.
         */
        if (++ftxx_ssp->ftxx_busy > ftxx_ssp->ftxx_limit)
        {
            ftxx_ssp->ftxx_aborted = 1;
            ftxx_reset(ftxx_ssp, FTXX_WARM_RESET);
            ftxx_fault(ftxx_ssp, U4FT_SEVERITY_MID, "timeout");
        }
    }
    else
    {

        /*
         *  Probe the hardware ... it should return the one's
         *  complement of the value we feed into it!  If it does,
         *  bump the counter for next time, otherwise report a fault.
         */
        tmp1 = tmp2 = (uint8_t)ftxx_ssp->ftxx_dogval;
        ftxx_cmd(ftxx_ssp, FTXX_CHECK, &tmp2);
```

CODE EXAMPLE 4-1 Character Device Example `ftxx.c` *(Continued)*

```
        if (tmp2 == ~tmp1)
            ftxx_ssp->ftxx_dogval += 1;
        else
            ftxx_fault(ftxx_ssp, U4FT_SEVERITY_MID, "bad dog");

        /*
         *  Reschedule ...
         *  We don't bother rechecking device state here, because
         *  it will be checked on the next callback anyway.
         */
        ftxx_watchdog(ftxx_ssp, 1);
    }
    mutex_exit(ftxx_ssp->ftxx_mutex);

}
/*******************************************************************/
/*
 * function     - ftxx_intr
 * description  - interrupt service routine, called on completion of
 *                a write command.
 * inputs       - soft state pointer
 * outputs      - value indicating whether or not the interrupt was
 *                claimed (DDI_INTR_CLAIMED or DDI_INTR_UNCLAIMED)
 */

static uint_t
ftxx_intr(caddr_t arg)
{
    ftxx_state_t *ftxx_ssp = (ftxx_state_t *)arg;
    int rslt;
    mutex_enter(ftxx_ssp->ftxx_mutex);
    if (ftxx_ssp->ftxx_busy)
    {

        /*
         *  Interrupt expected; warm-reset the device to acknowledge
         *  the interrupt and make the signal go away.  This will also
         *  clear the "busy" indicator and wake anyone waiting on it
         */
        ftxx_reset(ftxx_ssp, FTXX_WARM_RESET);
        rslt = DDI_INTR_CLAIMED;
    }
    else
    {

        /*
         *  Interrupt not expected, return unclaimed
         */
        rslt = DDI_INTR_UNCLAIMED;
    }
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
    mutex_exit(ftxx_ssp->ftxx_mutex);
    return rslt;
}

/*******************************************************************/
/*
 * cb_ops routines
 */
/*

 * function     - ftxx_open
 * description  - routine to control access to the device, called in
 *                response to the user's open(2) call.  The driver
 *               supports only one open at a time (exclusive access).
 * inputs       - device number, open flags (ignored), open type
 *                (must be OTYP_CHR), user credentials (not used)
 * outputs      - 0 (success) or errno.
 */

static int
ftxx_open(dev_t *devp, int flag, int otype, cred_t *cred)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    int err;
    int dev;
    dev = getminor(*devp);
    instance = FTXX_MINOR_TO_INST(dev);
    ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
    if (ftxx_ssp == NULL)
       err = ENXIO;                 /* not attached yet            */
    else if (!u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip,
U4FT_DO_WAIT))
        err = ENXIO;
    else if (otype != OTYP_CHR)
        err = EINVAL;
    else
    {
        mutex_enter(ftxx_ssp->ftxx_mutex);
        err = ftxx_ssp->ftxx_open ? EBUSY : 0;
        ftxx_ssp->ftxx_open = 1;
        mutex_exit(ftxx_ssp->ftxx_mutex);
    }
    return err;
}

/*
 * function     - ftxx_close
 * description  - routine to perform the final close on the device.
 * inputs       - device number, open flags (ignored), open type
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 *                (must be OTYP_CHR), user credentials (not used)
 * outputs      - 0 (success) or errno.
 */

static int
ftxx_close(dev_t dev, int flag, int otype, cred_t *cred)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    int err;
    instance = FTXX_MINOR_TO_INST(getminor(dev));
    ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
    if (ftxx_ssp == NULL)
       err = ENXIO;                 /* this "can't happen"         */
    else if (otype != OTYP_CHR)
       err = EINVAL;                /* nor can this                */
    else
    {
        mutex_enter(ftxx_ssp->ftxx_mutex);
        ftxx_ssp->ftxx_open = 0;
        mutex_exit(ftxx_ssp->ftxx_mutex);
        err = 0;
    }
    return err;
}

/*
 * function     - ftxx_read
 * description  - routine to read data from the device
 * inputs       - device number, IO request descriptor,
 *                user credentials (not used)
 * outputs      - 0 (success) or errno.
 */

static int
ftxx_read(dev_t dev, struct uio *uiop, cred_t *cred)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    int err;
    uint8_t buf[FTXX_DATASIZE];
    instance = FTXX_MINOR_TO_INST(getminor(dev));
    ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
    if (ftxx_ssp == NULL)
       return ENXIO;                /* this "can't happen"         */

    /*
     *  The read will fail if the device is not in a usable state,
     *  if the user specifies the wrong number of bytes, if an IO
     *  error occurs during the data transfer, if the checksum is
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
     *  wrong, or if the data can't be copied back to the caller!
     */
    if (!u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip, U4FT_DO_WAIT))
        err = EIO;
    else if (uiop->uio_resid != FTXX_USERDATA)
        err = EINVAL;
    else if ((err = ftxx_do_read(ftxx_ssp, buf)) == 0)
        err = uiomove((void *)buf, FTXX_USERDATA, UIO_READ, uiop);
    return err;
}

static int
ftxx_do_read(ftxx_state_t *ftxx_ssp, uint8_t buf[FTXX_DATASIZE])
{
    uint8_t tmp;
    int err;
    int i;
    mutex_enter(ftxx_ssp->ftxx_mutex);
    /*
     *  Read the device data into the supplied buffer
     */
    ftxx_cmd(ftxx_ssp, FTXX_READ, buf);

    /*
     *  If a high-severity fault has occurred, or the device has
     *  failed (is no longer usable), the data is considered invalid.
     *  If it passes those checks, we still have to validate the
     *  checksum before returning success.
     */
    if (ftxx_ssp->ftxx_fault_level >= U4FT_SEVERITY_HIGH)
        err = EIO;
    else if (!u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip,
U4FT_DONT_WAIT))
        err = EIO;
    else
    {
        for (tmp = 0, i = 0; i < FTXX_USERDATA; ++i)
            tmp += buf[i];
        err = tmp == buf[i] ? 0 : EIO;
    }
    mutex_exit(ftxx_ssp->ftxx_mutex);
    return err;
}

/*
 * function     - ftxx_write
 * description  - routine to write data from the device
 * inputs       - device number, IO request descriptor,
 *                user credentials (not used)
 * outputs      - 0 (success) or errno.
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 */

static int
ftxx_write(dev_t dev, struct uio *uiop, cred_t *cred)
{
    ftxx_state_t *ftxx_ssp;
    dev_info_t *dip;
    int instance;
    int err;
    int tmp;
    uint8_t buf[FTXX_DATASIZE];
    instance = FTXX_MINOR_TO_INST(getminor(dev));
    ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
    if (ftxx_ssp == NULL)
       return ENXIO;                /* this "can't happen"         */
    /*
     *  The read will fail if the device is not in a usable state,
     *  if the user specifies the wrong number of bytes, if the
     *  data can't be copied from the caller to a local buffer, or
     *  if an IO error occurs during the data transfer.
     */
    if (!u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip, U4FT_DO_WAIT))
        err = EIO;
    else if (uiop->uio_resid != FTXX_USERDATA)
        err = EINVAL;
    else if ((err = uiomove((void *)buf, FTXX_USERDATA, UIO_WRITE,
uiop)) == 0)
        err = ftxx_do_write(ftxx_ssp, buf);
    return err;
}

static int
ftxx_do_write(ftxx_state_t *ftxx_ssp, uint8_t buf[FTXX_DATASIZE])
{
    uint8_t tmp;
    int err;
    int i;
    mutex_enter(ftxx_ssp->ftxx_mutex);
    /*
    *  If any fault has previously occurred, the device is not writable
     */
    if (ftxx_ssp->ftxx_fault_level > U4FT_SEVERITY_NONE)
        err = EIO;
    else
    {

        /*
         *  Generate simple checksum
         */
        for (tmp = 0, i = 0; i < FTXX_USERDATA; ++i)
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
            tmp += buf[i];
        buf[i] = tmp;

        /*
         *  Write the data to the device, then wait for completion
         */
        ftxx_ssp->ftxx_aborted = 0;
        ftxx_cmd(ftxx_ssp, FTXX_WRITE, buf);
        while (ftxx_ssp->ftxx_busy)
            cv_wait(ftxx_ssp->ftxx_cv, ftxx_ssp->ftxx_mutex);

        /*
         *  If the command was aborted by a timeout, or any fault
         *  has occurred during the write, or the device is no
         *  longer usable, the write is presumed to have failed.
         */
        if (ftxx_ssp->ftxx_aborted)
            err = EIO;
        else if (ftxx_ssp->ftxx_fault_level > U4FT_SEVERITY_NONE)
            err = EIO;
        else if (!u4ft_ddi_dev_is_usable(ftxx_ssp->ftxx_dip,
U4FT_DONT_WAIT))
            err = EIO;
        else
            err = 0;
    }
    mutex_exit(ftxx_ssp->ftxx_mutex);
    return err;
}

/*
 * function     - ftxx_ioctl
 * description  - ioctl handler routine, called in response to a user
 *              - ioctl(2) request.
 * inputs       - device number, command, user space arg pointer,
 *                flags, user credentials, pointer to return value
 * outputs      - 0 (success) or errno
 */

static int
ftxx_ioctl(dev_t dev, int cmd, intptr_t arg, int flags,
           cred_t *credp, int *rvalp)
{
    ftxx_state_t *ftxx_ssp;
    int instance;
    int err;
    int tmp;
    instance = FTXX_MINOR_TO_INST(getminor(dev));
    ftxx_ssp = ddi_get_soft_state(ftxx_statep, instance);
    if (ftxx_ssp == NULL)
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
        return ENXIO;                /* this "can't happen"        */
    switch (cmd)
    {
    default:
        err = EINVAL;
    case FTXX_IOCTL_GET_LIMIT:
        tmp = ftxx_ssp->ftxx_limit;
        if (ddi_copyout(&tmp, (void *)arg, sizeof(tmp), flags))
            err = EFAULT;
        else
            err = 0;
        break;
    case FTXX_IOCTL_SET_LIMIT:
        if (ddi_copyin((void *)arg, &tmp, sizeof(tmp), flags))
            err = EFAULT;
        else if (tmp < FTXX_MIN_LIMIT || tmp > FTXX_MAX_LIMIT)
            err = EINVAL;
        else
        {

            /*
             *  Update the command timeout limit property, then, if
             *  that works, update the version in the soft state too.
             */
            err = ddi_prop_update_int(dev, ftxx_ssp->ftxx_dip,
                                      FTXX_LIMIT, tmp);
            if (err != DDI_PROP_SUCCESS)
                err = EINVAL;
            else
            {
                ftxx_ssp->ftxx_limit = tmp;
                err = 0;
            }
        }
        break;
    }
    return err;
}

/******************************************************************/
/*
 *  All functions below here are called with the soft-state mutex held
 */

/*
 * function     - ftxx_do_online
 * description  - this routine completes the remaining initialization
 *                required to bring a device into service
 * inputs       - soft state pointer
 * outputs      - DDI_SUCCESS or DDI_FAILURE
```

CODE EXAMPLE 4-1 Character Device Example `ftxx.c` *(Continued)*

```
 */

static int
ftxx_do_online(ftxx_state_t *ftxx_ssp)
{
    ddi_device_acc_attr_t attr;
    ddi_acc_handle_t handle;
    caddr_t base;
    int rslt;
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));

    /*
     * Set up the device access attributes
     */
    attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    attr.devacc_attr_endian_flags = FTXX_ENDIAN_FLAGS;
    attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    /*
     * Map in the device's one and only register set
     */
    rslt = ddi_regs_map_setup(ftxx_ssp->ftxx_dip, FTXX_REGSET_0,
&base, 0,sizeof(ftxx_devregs_t), &attr, &handle);
    if (rslt != DDI_SUCCESS)
        return DDI_FAILURE;
    ftxx_ssp->ftxx_handle = handle;
    ftxx_ssp->ftxx_base = base;
    ftxx_ssp->ftxx_mapping = 1;
    rslt = ddi_add_intr(ftxx_ssp->ftxx_dip, FTXX_INTR_0, &ftxx_ssp-
>ftxx_ibc,&ftxx_ssp->ftxx_idc, ftxx_intr, (void *)ftxx_ssp);
    if (rslt != DDI_SUCCESS)
        return DDI_FAILURE;
    ftxx_ssp->ftxx_intr = 1;

    /*
     *  We have to clear the fault level so that any new faults
     *  get reported, initialize the hardware with a cold reset,
     *  and start the watchdog running.  The device can't be open,
     *  and there can't be any commands in progress at this point.
     */
    ftxx_ssp->ftxx_fault_level = U4FT_SEVERITY_NONE;
    ftxx_reset(ftxx_ssp, FTXX_COLD_RESET);
    ftxx_watchdog(ftxx_ssp, 1);
    return DDI_SUCCESS;
}

/*
 * function     - ftxx_do_offline
 * description  - this routine prepares the device and driver for a
 *                return to the offline state.
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 * inputs       - soft-state pointer
 * outputs      - DDI_SUCCESS
 */

static int
ftxx_do_offline(ftxx_state_t *ftxx_ssp)
{
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));
    /*
     *  Stop the watchdog; the device must already be closed,
     *  so no commands can be in progress and no one can be
     *  relying on the watchdog to time out a failed write.
     */
    ftxx_watchdog(ftxx_ssp, 0);
    if (ftxx_ssp->ftxx_mapping)
    {

        /*
         *  Try to shut down the hardware; it doesn't matter if it's
         *  already failed, in which case this will do nothing ...
         */
        ftxx_reset(ftxx_ssp, FTXX_COLD_RESET);
    }
    if (ftxx_ssp->ftxx_intr)
    {

        /*
        *  Unionist the interrupt handler before unmapping registers!
         */
        ddi_remove_intr(ftxx_ssp->ftxx_dip, FTXX_INTR_0, ftxx_ssp-
>ftxx_ibc);
        ftxx_ssp->ftxx_intr = 0;
    }
    if (ftxx_ssp->ftxx_mapping)
    {

        /*
         * Destroy device register mapping
         */
        ddi_regs_map_free(&ftxx_ssp->ftxx_handle);
        ftxx_ssp->ftxx_mapping = 0;
    }
    return DDI_SUCCESS;
}

/*
 * function     - ftxx_watchdog
 * description  - this routine starts or stops the watchdog
 * inputs       - soft-state pointer, start/stop flag
 * outputs      - none
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
 */

static void
ftxx_watchdog(ftxx_state_t *ftxx_ssp, int start)
{
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));
    if (start)
    {
        /*
         *  Schedule a callback to the watchdog, if there isn't already
         *  one pending.  The call to timeout() is assumed not to fail,
         *  because Solaris panics if it does!
         */
        if (!ftxx_ssp->ftxx_dog_pending)
        {
         ftxx_ssp->ftxx_dog_id = timeout(ftxx_dog, (caddr_t)ftxx_ssp,
                                             ftxx_ssp->ftxx_dogticks);
            ftxx_ssp->ftxx_dog_pending = 1;
        }
    }
    else
    {

        /*
         *  Stop the watchdog, if it's running
         */
        ftxx_ssp->ftxx_stop_dog = 1;
        while (ftxx_ssp->ftxx_dog_pending)
            cv_wait(ftxx_ssp->ftxx_cv, ftxx_ssp->ftxx_mutex);
        ftxx_ssp->ftxx_stop_dog = 0;
    }
}

/*******************************************************************/
/*
 *  Hardware access functions
 */
/*
 * function     - ftxx_reset
 * description  - this routine issues a RESET command to the device
 * inputs       - soft-state pointer, reset parameter (warm/cold)
 * outputs      - none
 */

static void
ftxx_reset(ftxx_state_t *ftxx_ssp, ftxx_reset_t cmd)
{
    ddi_acc_handle_t handle;
    ftxx_devregs_t *base;
    int status;
```

CODE EXAMPLE 4-1 Character Device Example ft*xx.c (Continued)*

```
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));
    handle = ftxx_ssp->ftxx_handle;
    base = ftxx_ssp->ftxx_base;

    /*
     *  Reset command can be issued at any time.  After this, the
     *  device is not busy, so we wake up anyone waiting for completion.
     */
    ddi_put8(handle, &base->ftxx_cmd_reg, FTXX_RESET);
    ddi_put8(handle, &base->ftxx_data_reg, cmd);
    ftxx_ssp->ftxx_busy = 0;
    cv_broadcast(ftxx_ssp->ftxx_cv);
    status = u4ft_ddi_check_access(handle);
    if (status & FTXX_ACCESS_MASK)
        ftxx_fault(ftxx_ssp, U4FT_SEVERITY_HIGH, "access fault");
}

/*
 * function     - ftxx_cmd
 * description  - this routine wait until the device is not busy, then
 *                issues a new command and transfers data as required.
 * inputs       - soft-state pointer, command, pointer to data
 * outputs      - none
 */

static void
ftxx_cmd(ftxx_state_t *ftxx_ssp, ftxx_cmd_t cmd, uint8_t *datap)
{
    ddi_acc_handle_t handle;
    ftxx_devregs_t *base;
    int status;
    int i;
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));
    handle = ftxx_ssp->ftxx_handle;
    base = ftxx_ssp->ftxx_base;

    /*
     *  Commands can only be issued when not busy
     */
    while (ftxx_ssp->ftxx_busy)
        cv_wait(ftxx_ssp->ftxx_cv, ftxx_ssp->ftxx_mutex);
    switch (cmd)
    {
    case FTXX_READ:
        ddi_put8(handle, &base->ftxx_cmd_reg, cmd);
        for (i = 0; i < FTXX_DATASIZE; ++i)
            *datap++ = ddi_get8(handle, &base->ftxx_data_reg);
        break;
    case FTXX_WRITE:
        ddi_put8(handle, &base->ftxx_cmd_reg, cmd);
```

CODE EXAMPLE 4-1 Character Device Example `ftxx.c` *(Continued)*

```
        for (i = 0; i < FTXX_DATASIZE; ++i)
            ddi_put8(handle, &base->ftxx_data_reg, *datap++);
        ftxx_ssp->ftxx_busy = 1;    /* start counting!              */
        break;
    case FTXX_CHECK:
        ddi_put8(handle, &base->ftxx_cmd_reg, cmd);
        ddi_put8(handle, &base->ftxx_data_reg, *datap);
        *datap = ddi_get8(handle, &base->ftxx_check_reg);
        break;
    }
    status = u4ft_ddi_check_access(handle);
    if (status & FTXX_ACCESS_MASK)
        ftxx_fault(ftxx_ssp, U4FT_SEVERITY_HIGH, "access fault");
}
/*******************************************************************/
/*
 *  Fault management, filtering and reporting
 */

/*
 * function     - ftxx_fault
 * description  - this routine tracks fault levels and determines when
 *                 a fault should be reported
 * inputs       - soft-state pointer, fault severity, text message
 * outputs      - none
 */

static void
ftxx_fault(ftxx_state_t *ftxx_ssp, u4ft_severity_t level, char
*message)
{
    int report;
    ASSERT(mutex_held(ftxx_ssp->ftxx_mutex));

    /*
     *  Report this fault if it's more severe than any previous fault or
     *  if its severity is NONE (can be used for recovery reporting).
     */
    if (level > ftxx_ssp->ftxx_fault_level)
    {
        ftxx_ssp->ftxx_fault_level = level;
        report = 1;
    }
    else if (level == U4FT_SEVERITY_NONE)
        report = 1;
    else
        report = 0;
    if (report)
        u4ft_ddi_report_fault(ftxx_ssp->ftxx_dip, level, message);
}
```

CHAPTER 5

Test Harness

Testing the resilience of a hardened device driver is as important as testing any other feature. This requires a suitably wide range of different types of typical hardware faults to be injected, preferably in a controlled and repeatable fashion. This is not normally possible using hardware, so the driver-hardening test harness is provided. The harness works by intercepting calls from the driver to various DDI routines, and then corrupting the result of those DDI routine calls as if the hardware had caused the corruption. Obviously, this only works if the driver performs all its I/O accesses via DDI routines, as required for Solaris 2.6 DDI/DKI compliance.

The expectation is that the writer of a hardened device driver will produce a set of test scripts that use the test harness to demonstrate the resilience of the driver. These scripts can use knowledge of the internals of the driver to generate those types of faults that are most likely to cause problems (for example, register accesses returning values outside the expected range). The test harness enables corruption of accesses to specific registers as well as the definition of more random types of corruption. Once written, the test scripts can be rerun at a later date when new versions of the driver or the platform become available.

The test harness is implemented as a device driver, `harness`(7D), plus two user-level commands, `th_define`(1) and `th_manage`(1). These are provided as the test harness package.

Examples of Test Harness Usage

- To define some errdefs for instance 0 of the foo device, type:

```
# th_define foo 0 ....... &
# th_define foo 0 ....... &
# th_define foo 0 ....... &
# th_define foo 0 ....... &
```

- To start the test, type:

```
# th_manage foo 0 START
```

- To check the status of the errdefs, type:

```
# th_manage foo 0 BROADCAST
```

  This will cause each th_define process to print out its current status.

- If the driver has reported a fatal error, you can take the driver offline using u4ftcmd(), clear the error condition by typing:

```
# th_manage foo 0 CLEAR_ACC_CHK
```

  or

```
# th_manage foo 0 CLEAR_ERRORS
```

  and bring the driver online again using u4ftcmd().

- To terminate testing, type:

```
# th_manage foo 0 CLEAR_ERRDEFS
```

Examples of Error Definitions

```
th_define foo 3 1 0 0 PIO_R 0 1 U4FT_ACC_NO_PIO OR 0x100
```

Causes 0x100 to be ORed into the next physical I/O read access from any register in register set 1 of instance 3 of the foo driver. Subsequent calls in the driver to `u4ft_ddi_check_access()` will have the `U4FT_ACC_NO_PIO` bit set.

```
th_define foo 3 1 0 0 PIO_R 0 1 - OR 0x0 1
```

Causes 0x0 to be ORed into the next physical I/O read access from any register in register set 1 of instance 3 of the foo driver. This, of course, has no effect. However, the additional debug flag is set. This causes console messages to be produced, listing the various handles in use with sequential allocation number for `ddi_dma_handle` and number/offset/length for `ddi_acc_handle`, and notifying when the qualifying PIO read occurs.

```
th_define foo 3 1 0x8100 1 PIO_R 0 10 - EQ 0x70003
```

Causes the next ten next physical I/O reads from the register at offset 0x8100 in register set 1 of instance 3 of the foo driver to return 0x70003.

```
th_define foo 3 1 0x8100 1 PIO_W 100 3 - AND 0xffffffffffffefff
```

The next 100 physical I/O writes to the register at offset 0x8100 in register set 1 of instance 3 of the foo driver take place as normal. However, on each of the three subsequent accesses, the 0x1000 bit will be cleared.

```
th_define foo 3 1 0x8100 0x10 PIO_R 0 1 U4FT_ACC_NO_PIO XOR 7
```

Causes the bottom three bits to have their values toggled for the next physical I/O read access to registers with offsets in the range 0x8100 to 0x8110 in register set 1 of instance 3 of the foo driver. Subsequent calls in the driver to `u4ft_ddi_check_access()` will have the `U4FT_ACC_NO_PIO` bit set.

```
th_define foo 3 -1 0 0 PIO_R 0 1 - NO 0x999
```

Prevents the next physical I/O read access from any register in any register set of instance 3 of the foo driver from going out on the bus and returns, instead, 0x999.

```
th_define foo 3 -1 0 0 PIO_W 0 1 - NO 0
```

Prevents the next physical I/O write access to any register in any register set of instance 3 of the foo driver from going out on the bus.

```
th_define foo 3 -1 0 8192 DMA_R 0 1 - OR 7
```

Causes 0x7 to be ORed into each long long in the first 8192 bytes of the next DMA read, using any DMA handle for instance 3 of the foo driver.

```
th_define foo 3 2 0 8 DMA_R 0 1 - OR 0x7070707070707070
```

Causes 0x70 to be ORed into each byte of the first long long of the next DMA read, using the DMA handle with sequential allocation number 2 for instance 3 of the foo driver.

```
th_define foo 3 -1 256 256 DMA_W 0 1 U4FT_ACC_NO_DMA OR 7
```

Causes 0x7 to be ORed into each long long in the range from offset 256 to offset 512 of the next DMA write, using any DMA handle for instance 3 of the `foo` driver. Subsequent calls in the driver to `u4ft_ddi_check_access()` will have the `U4FT_ACC_NO_DMA` bit set.

```
th_define foo 3 0 0 8 DMA_W 100 3 - AND 0xffffffffffffefff
```

The next 100 DMA writes using the DMA handle with sequential allocation number 0 for instance 3 of the foo driver take place as normal. However, on each of the three subsequent accesses, the 0x1000 bit will be cleared in the first long long of the transfer.

```
th_define foo 3 -1 - - DMA_A 0 1 - EQ 0xffffffffeeeeeeee
```

Causes the `dmac_laddress` field of the next `ddi_dma_cookie` to be returned to the driver for any DMA handle for instance 3 of the foo driver to be set to `0xffffffffeeeeeeee`. Note that this also sets the `dmac_address` field to `0xeeeeeeee`.

```
th_define foo 3 -1 - - DMA_L 0 1 - XOR 1
```

Causes the bottom bit of the `dmac_size` field of the next `ddi_dma_cookie` to be returned to the driver for any DMA handle for instance 3 of the foo driver to be toggled.

```
th_define foo 3 - - - INTR 0 6 - LOSE -
```

Causes the next six interrupts for instance 3 of the foo driver to be lost.

```
th_define foo 3 - - - INTR 30 1 U4FT_ACC_NO_IRQ EXTRA 10
```

When the 31st subsequent interrupt for instance 3 of the foo driver occurs, a further ten interrupts are also generated. Subsequent calls in the driver to `u4ft_ddi_check_access()` will have the `U4FT_ACC_NO_IRQ` bit set.

```
th_define foo 3 - - - INTR 0 1 - DELAY 1024
```

Causes the next interrupt for instance 3 of the foo driver to be delayed by 1024 microseconds.

The `harness` Driver

The test harness driver has the following syntax:

```
harness@0:harness,ctl
```

Conceptually, the test harness driver consists of two parts:

- A driver that supports a number of `ioctl`s which enable error definitions (`errdef`s) to be defined and subsequently managed.

  The driver is a clone driver, so each open creates a separate invocation. Any `errdef`s created by using `ioctl`s to that invocation are automatically deleted when the invocation is closed.

- Intercept routines.

  When the driver is attached, it edits the `bus_ops` structure of the bus nexus specified by the harness-nexus field in the `harness.conf` file (by default, this is `pci`), thus allowing the test harness to intercept various DDI functions. These intercept routines primarily carry out fault injections based on the `errdef`s created for the device.

Faults can be injected into:

- **DMA** – corrupting data and generating bad addresses for DMA to or from memory areas defined by `ddi_dma_setup()`, `ddi_dma_bind_handle()`, and so forth
- **Physical I/O** – corrupting data sent or received via `ddi_get8()`, `ddi_put8()`, and so forth
- **Interrupts** – generating spurious interrupts, losing interrupts, delaying interrupts, and so forth

The following two `u4ft_ddi` routines are also intercepted:

- `u4ft_ddi_check_access()`

  Returns the *access denied* bits based on the `acc_chk` fields of any `errdef`s with the name and instance corresponding to this `ddi_acc_handle`

- `u4ft_ddi_report_fault()`
  - Stores the error message into all `errdef`s associated with this `dev_info`, providing that the `access_count` for that `errdef` has gone to zero, and this is a higher severity than any previously saved message in that `errdef`
  - Awakens any processes sleeping in `HARNESS_CHK_STATE_W` `ioctl`s for `errdef`s in which a new error message has been saved

By default, DDI routines called from all drivers are intercepted and faults potentially injected. However, the harness-to-test field in the `harness.conf` file can be set to a space-separated list of drivers to test (or by preceding each driver name in the list with an '`!`', to a list of drivers to be omitted from the test).

IOCTLS

The following `ioctl`s are supported and are defined in a header file `<sys/harness.h>`.

HARNESS_ADD_DEF

This has as an argument a pointer to the following structure:

```
struct harness_errdef {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
int rnumber;
offset_t offset;
offset_t len;
uint_t access_type;
uint_t access_count;
uint_t fail_count;
uint_t acc_chk;
uint_t operator;
longlong_t operand;
void *errdef_handle;
}
```

The `errdef` is passed to the driver but is not acted upon until a `HARNESS_START` `ioctl` is made. On successful return, the `errdef_handle` field will be filled in.

Multiple concurrent `errdef`s can be supported, referring to the same or different devices. However, if multiple faults are injected into the same access, the result might be undefined.

The fault defined by `operator` is injected into the *n*th qualifying access, where *n* is given by `access_count`, and will continue to be injected for `fail_count` consecutive qualifying accesses.

The value in `acc_chk` can be

- `NULL`
- `U4FT_ACC_NO_PIO`
- `U4FT_ACC_NO_DMA`
- `U4FT_ACC_NO_IRQ`

and is ORed into the value to be returned from `u4ft_ddi_check_access()` after the first fault is injected.

Although errors cease to be injected once fail_count has gone to zero, acc_chk remains set until the errdef is cleared, or a HARNESS_CLEAR_ACC_CHK, HARNESS_CLEAR_ERRORS or HARNESS_CLEAR_ERRDEFS ioctl is called for this name/instance.

Qualifying access are defined by access_type, which can be one of the following:

- HARNESS_PIO_R

  A qualifying access is a physical I/O read where the ddi_acc_handle is allocated using a dev_info with the specified name/instance and using the specified rnumber, and where the requested address falls within the specified offset/len. If len is 0, the remainder of the register set qualifies. If rnumber is -1, all register sets qualify.

  operator can be one of one following:

  - HARNESS_EQUAL

    The data is read from the I/O card but ignored and the contents of the operand are returned to the caller instead.

  - HARNESS_AND

    The data is read from the I/O card and ANDed with the operand before being returned to the caller.

  - HARNESS_OR

    The data is read from the I/O card and ORed with the operand before being returned to the caller.

  - HARNESS_XOR

    The data is read from the I/O card and XORed with the operand before being returned to the caller.

  - HARNESS_NO_TRANSFER

    No data is read from the I/O card and the operand is returned to the caller instead.

- HARNESS_PIO_W

  A qualifying access is a physical I/O write where the ddi_acc_handle was allocated using a dev_info with the specified name/instance and using the specified rnumber, and where the requested address falls within the specified offset/len. If len is 0, the remainder of the register set qualifies. If rnumber is -1, all register sets qualify.

  operator can be one of the following:

  - HARNESS_EQUAL

    The contents of the operand are written to the I/O card instead of the requested data.

- HARNESS_AND

  The operand is ANDed with the requested data before being written to the I/O card.

- HARNESS_OR

  The operand is ORed with the requested data before being written to the I/O card.

- HARNESS_XO

  The operand is XORed with the requested data before being written to the I/O card.

- HARNESS_NO_TRANSFER

  No data is written to the I/O card.

- HARNESS_DMA_R

A qualifying access is an implicit or explicit ddi_dma_sync() with DDI_DMA_SYNC_FORCPU or DDI_dma_SYNC_FORKERNEL, where

- The ddi_dma_handle was allocated using a dev_info with the specified name/instance.
- rnumber is -1 or corresponds to the sequential allocation number of the ddi_dma_handle.
- There are one or more 64-bit-aligned 64-bit words within the range specified by offset/len that lie within the amount of space mapped by the ddi_dma_handle.

The corruption applies to all such qualifying long longs.

operator can be one of the following:

- HARNESS_EQUAL

  The data is read from the I/O card into memory, but then the specified range of memory is overwritten by the contents of the operand.

- HARNESS_AND

  The data is read from the I/O card into memory and then the specified range of memory is ANDed with the operand.

- HARNESS_OR

  The data is read from the I/O card into memory and then the specified range of memory is ORed with the operand.

- HARNESS_XOR

  The data is read from the I/O card into memory and then the specified range of memory is ORed with the operand.

- HARNESS_DMA_W

A qualifying access is an implicit or explicit ddi_dma_sync() or with DDI_DMA_SYNC_FORDEV, where

  - The ddi_dma_handle is allocated using a dev_info with the specified name/instance.
  - rnumber is -1 or corresponds to the sequential allocation number of the ddi_dma_handle.
  - There are one or more 64-bit-aligned 64-bit words within the range specified by offset/len that lie within the amount of space mapped by the ddi_dma_handle.

The corruption applies to all such qualifying 64-bit words.

operator can be one of the following:

  - HARNESS_EQUAL

    A copy of the data is taken, and the specified range of memory is then overwritten by the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

  - HARNESS_AND

    A copy of the data is taken, and the specified range of memory is then ANDed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

  - HARNESS_OR

    A copy of the data is taken, and the specified range of memory is then ORed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

  - HARNESS_XOR

    A copy of the data is taken, and the specified range of memory is then XORed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

- HARNESS_INTR

  A qualifying access is an interrupt service routine called where the intrspec was allocated using a dev_info with the specified name/instance. The arguments rnumber, offset and len are ignored in this case.

  operator can be one of the following:

  - HARNESS_DELAY_INTR

    This causes the interrupt to be held off for operand microseconds (except for hilevel interrupts).

- HARNESS_LOSE_INTR

  This causes the interrupt to be lost permanently. operand is ignored in this case.

- HARNESS_EXTRA_INTR

  This causes operand additional spurious interrupts to be generated.

HARNESS_DEL_DEF

This has as an argument a pointer to the following item:

```
void *errdef_handle;
```

where errdef_handle should be the value returned in the errdef_handle field of the errdef structure as returned from a HARNESS_ADD_DEF ioctl. This will cancel the specified errdef.

HARNESS_START

This has as an argument a pointer to the following structure:

```
struct harness_errctl {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
}
```

This sets running all errdefs with the specified name/instance. Note that errdefs are not automatically set running by the HARNESS_ADD_DEF ioctl, so the HARNESS_START ioctl must always be called.

HARNESS_STOP

This has as an argument a pointer to the following structure:

```
struct harness_errctl {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
}
```

This suspends all errdefs with the specified name or instance. The errdefs can be restarted again by a subsequent HARNESS_START ioctl.

HARNESS_BROADCAST

This has as an argument a pointer to the following item:

```
struct harness_errctl {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
}
```

Any processes that are sleeping in a HARNESS_CHK_STATE_W ioctl for an errdef with this name/instance are awakened.

HARNESS_CLEAR_ACC_CHK

This has as an argument a pointer to the following structure:

```
struct harness_errctl {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
}
```

For all errdefs with the specified name/instance, if access_count and fail_count are already zero, this sets the acc_chk field to zero. Any processes that are sleeping in a HARNESS_CHK_STATE_W ioctl for an errdef with this name/instance are awakened.

HARNESS_CLEAR_ERRORS

This has as an argument a pointer to the following structure:

```
struct harness_errctl {
uint_t namesize;
char *name;              /* as returned by ddi_get_name() */
int instance;            /* as returned by ddi_get_instance() */
}
```

For all errdefs with the specified name or instance, if access_count is already zero, this sets the acc_chk and fail_count fields to zero. Any processes that are sleeping in a HARNESS_CHK_STATE_W ioctl for an errdef with this name or instance are awakened.

HARNESS_CLEAR_ERRDEFS

This has as an argument a pointer to the following structure:

```
struct harness_errctl {
uint_t namesize;
char *name;             /* as returned by ddi_get_name() */
int instance;           /* as returned by ddi_get_instance() */
}
```

For all errdefs with the specified name/instance, this sets the acc_chk, fail_count and access_count fields to zero. Any processes that are sleeping in a HARNESS_CHK_STATE_W ioctl for an errdef with this name/instance are awakened.

HARNESS_CHK_STATE

This has as an argument a pointer to the following structure:

```
struct harness_errstate {
ulong_t fail_time;       /* time that access_count went to */
                         /* zero */
ulong_t msg_time;        /* time that u4ft_ddi_report_fault */
                         /* was called */
uint_t access_count;     /* current value of access_counter */
uint_t fail_count;       /* current value of fail_counter */
uint_t acc_chk;          /* current value of acc_chk */
uint_t errmsg_count;     /* no of applicable */
                         /* u4ft_ddi_report_faults */
char buffer[ERRMSGSIZE]; /* latest u4ft_ddi_report_fault */
                         /* message */
ddi_severity_t severity; /* latest u4ft_ddi_report_fault */
                         /* severity */
void *errdef_handle;
}
```

where errdef_handle should be the value returned in the errdef_handle field of the errdef structure as returned from a HARNESS_ADD_DEF ioctl. On return from the HARNESS_CHK_STATE ioctl, the other fields will be filled in.

The msg_time, buffer, and severity are for the first occurrence of the highest severity error message reported since access_count went to zero for this errdef.

HARNESS_CHK_STATE_W

This has as an argument a pointer to the following structure:

```
struct harness_errstate {
ulong_t fail_time;       /*time that access_count went to zero */
ulong_t msg_time;        /* time that u4ft_ddi_report_fault */
                         /* was called */
uint_t access_count;     /* current value of access_counter */
uint_t fail_count;       /* current value of fail_counter */
uint_t acc_chk;          /* current value of acc_chk */
uint_t errmsg_count;     /* no of applicable */
                         /* u4ft_ddi_report_faults */
char buffer[ERRMSGSIZE]; /* latest u4ft_ddi_report_fault */
                         /* message */
ddi_severity_t severity; /* latest u4ft_ddi_report_fault */
                         /* severity */
void *errdef_handle;
}
```

where errdef_handle should be the value returned in the errdef_handle field of the errdef structure as returned from a HARNESS_ADD_DEF ioctl. On return from the HARNESS_CHK_STATE_W ioctl, the other fields will be filled in.

The msg_time, buffer and severity are for the first occurrence of the highest severity error message reported since access_count went to zero for this errdef.

If the access_count has gone to zero for this errdef and a u4ft_ddi_report_fault() has occurred since the last time HARNESS_CHK_STATE_W was called, the ioctl returns immediately. Otherwise, the ioctl sleeps until the access_count for this errdef handle has gone to zero and the next subsequent u4ft_ddi_report_fault() occurs, or until one of the following ioctls is called for the same name/instance:

- HARNESS_BROADCAST
- HARNESS_CLEAR_ACC_CHK
- HARNESS_CLEAR_ERRORS
- HARNESS_CLEAR_ERRDEFS.

HARNESS_DEBUG_ON

This has as an argument a pointer to the following item:

```
void *errdef_handle;
```

where errdef_handle should be the value filled in the errdef_handle field of the errdef structure as returned from a HARNESS_ADD_DEF ioctl. This turns on debug information for the specified errdef so that the driver outputs sequential allocation number for ddi_dma_handle and rnumber/offset/len for ddi_acc_handle to the console.

HARNESS_DEBUG_OFF

This has as an argument a pointer to the following item:

```
void *errdef_handle;
```

where errdef_handle should be the value filled in the errdef_handle field of the errdef structure as returned from a HARNESS_ADD_DEF ioctl. This turns off debug information for the specified errdef.

See also th_define(1), th_manage(1).

Defining an Error Definition

An error definition (errdef) can be specified for the test harness using th_define(1), which has the following syntax:

```
th_define name instance rnumber offset len type access_count
fail_count acc_chk operator operand [debug]
```

The errdef is passed to the harness(7D) driver, but is not acted upon until th_manage(1) is called to start testing. th_define(1) sleeps until either a u4ft_ddi_report_fault(9E) occurs for the device in question or until th_manage(1) awakens it.

th_define(1) then outputs the parameters with which it was called and the current state of the errdef to standard output. If access_count or fail_count or acc_chk are still non-zero, th_define(1) goes back to sleep again. When th_define(1) is finally awakened with access_count, fail_count and acc_chk all zero, it will exit, causing the errdef to be canceled.

If the optional debug parameter is a non-zero number, the debug is turned on for this errdef, causing sequential allocation number for ddi_dma_handle and rnumber, offset or len for ddi_acc_handle to be displayed to the console by the driver.

Multiple concurrent errdefs can be supported, referring to the same or different devices. However, if multiple faults end up being injected into the same access, the result might be undefined.

The fault defined by operator is injected into the *n*th qualifying access where *n* is given by access_count, and then continues to be injected for fail_count consecutive qualifying accesses.

The value in acc_chk can be

- NULL
- U4FT_ACC_NO_PIO
- U4FT_ACC_NO_DMA
- U4FT_ACC_NO_IRQ

and will be ORed into the value to be returned from u4ft_ddi_check_access() after the first fault is injected. Although errors cease to be injected once fail_count has gone to zero, acc_chk remains set until th_define(1) exits, or th_manage(1) is called to clear errors etc. for this device.

Qualifying access are defined by access_type, which can be one of the following:

- PIO_R

  A qualifying access is a physical I/O read where the ddi_acc_handle was allocated using a dev_info with the specified name/instance and using the specified rnumber, and where the requested address falls within the specified offset or len. If len is 0, the remainder of the register set qualifies. If rnumber is -1, all register sets qualify.

  operator can be one of:

  - EQ

    The data is read from the I/O card, but ignored and the contents of the operand are returned to the caller instead.

  - AND

    The data is read from the I/O card and ANDed with the operand before being returned to the caller.

- OR

  The data is read from the I/O card and ORed with the operand before being returned to the caller.

- XOR

  The data is read from the I/O card and XORed with the operand before being returned to the caller.

- NO

  No data is read from the I/O card and the operand is returned to the caller.

- PIO_W

  A qualifying access is a physical I/O write where the ddi_acc_handle was allocated using a dev_info with the specified name or instance and using the specified rnumber, and where the requested address falls within the specified offset/len. If len is 0, the remainder of the register set qualifies. If rnumber is -1, all register sets qualify.

  operator can be one of the following:

  - EQ

    The contents of the operand are written to the I/O card instead of the requested data.

  - AND

    The operand is ANDed with the requested data before being written to the I/O card.

  - OR

    The operand is ORed with the requested data before being written to the I/O card.

  - XOR

    The operand is XORed with the requested data before being written to the I/O card.

  - NO

    No data is written to the I/O card.

- DMA_R

  A qualifying access is an implicit or explicit ddi_dma_sync() or with DDI_dma_SYNC_FORCPU or DDI_dma_SYNC_FORKERNEL where:

  - The ddi_dma_handle was allocated using a dev_info with the specified name or instance.

- rnumber is -1 or corresponds to the sequential allocation number of the ddi_dma_handle.
- There are one or more long long aligned long longs within the range specified by offset/len that lie within the amount of space mapped by the ddi_dma_handle.

The corruption applies to all such qualifying long longs.

operator can be one of the following:

- EQ

  The data is read from the I/O card into memory, but then the specified range of memory is overwritten by the contents of the operand.

- AND

  The data is read from the I/O card into memory and then the specified range of memory is ANDed with the operand.

- OR

  The data is read from the I/O card into memory and then the specified range of memory is ORed with the operand.

- XOR

  The data is read from the I/O card into memory and then the specified range of memory is XORed with the operand.

- DMA_W

  A qualifying access is an implicit or explicit ddi_dma_sync() or with DDI_dma_SYNC_FORDEV where:

  - The ddi_dma_handle was allocated using a dev_info with the specified name/instance
  - rnumber is -1 or corresponds to the sequential allocation number of the ddi_dma_handle
  - There are one or more long long aligned long longs within the range specified by offset/len that lie within the amount of space mapped by the ddi_dma_handle.

  The corruption applies to all such qualifying long longs.

  operator can be one of the following:

  - EQ

    A copy of the data is taken, and the specified range of memory is overwritten by the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

- AND

  A copy of the data is taken, and the specified range of memory is then ANDed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

- OR

  A copy of the data is taken, and the specified range of memory is then ORed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

- XOR

  A copy of the data is taken, and the specified range of memory is then XORed with the contents of the operand. The DMA is then pointed at the copy of the data rather than the original.

- `DMA_A`

  A qualifying access is an implicit or explicit `ddi_dma_htoc`() where the `ddi_dma_handle` was allocated using a `dev_info` with the specified name or instance, and where `rnumber` is -1 or corresponds to the sequential allocation number of the `ddi_dma_handle`. The arguments `offset` and `len` are ignored in this case.

  `operator` can be one of the following:

  - EQ

    DDI routines returning a `ddi_dma_cookie_t` return the value given by operand for `dmac_laddress` and `dmac_address`.

  - AND

    DDI routines returning a `ddi_dma_cookie_t` return the intended value ANDed with the operand for `dmac_laddress` and `dmac_address`.

  - OR

    DDI routines returning a `ddi_dma_cookie_t` return the intended value ORed with the operand for `dmac_laddress` and `dmac_address`.

  - XOR

    DDI routines returning a `ddi_dma_cookie_t` return the intended value XORed with the operand for `dmac_laddress` and `dmac_address`.

- DMA_L

  A qualifying access is an implicit or explicit ddi_dma_htoc() where the ddi_dma_handle was allocated using a dev_info with the specified name or instance, and where rnumber is -1 or corresponds to the sequential allocation number of the ddi_dma_handle. The arguments offset and len are ignored in this case.

  operator can be one of the following:

  - EQ

    DDI routines returning a ddi_dma_cookie_t return the value given by operand for dmac_size.

  - AND

    DDI routines returning a ddi_dma_cookie_t return the intended value ANDed with the operand for dmac_size.

  - OR

    DDI routines returning a ddi_dma_cookie_t return the intended value ORed with the operand for dmac_size.

  - XOR

    DDI routines returning a ddi_dma_cookie_t return the intended value XORed with the operand for dmac_size.

- INTR

  A qualifying access is an interrupt service routine being called where the intrspec was allocated using a dev_info with the specified name or instance. The arguments rnumber, offset and len are ignored in this case.

  operator can be one of the following:

  - DELAY

    This causes the interrupt to be held off for operand microseconds (except for high level interrupts).

  - LOSE

    This causes the interrupt to be lost permanently. operand is ignored in this case.

  - EXTRA

    This causes operand additional spurious interrupts to be generated.

  See also th_manage(1), harness(7D).

Managing Test Harness for a Specific Device

th_manage(1) acts on all error definitions (errdefs) for the specified name/instance, and has the following syntax:

```
th_manage name instance command
```

It supports the following command values:

- START

  Set running or resume all errdefs for this name or instance.

- STOP

  Suspend all errdefs for this name or instance.

- BROADCAST

  Awaken all th_define(1) processes for this name or instance, causing them to display their current status and exit if the errdef is now defunct (that is, access_count, fail_count and acc_chk are all zero).

- CLEAR_ACC_CHK

  Awaken all th_define(1) processes for this name or instance. If access_count and fail_count are already zero, then set acc_chk to zero as well so that th_define(1) exits once it has displayed its status.

- CLEAR_ERRORS

  Awaken all th_define(1) processes for this name/instance. If access_count is already zero, set fail_count and acc_chk to zero as well so that th_define(1) exits once it has displayed its status.

- CLEAR_ERRDEFS

  Awaken all th_define(1) processes for this name/instance. access_count, fail_count and acc_chk are all set to zero so that all th_define(1) commands exits once they have displayed their status.

See also th_define(1), harness(7D).

CHAPTER 6

EEPROM Data

Each module contains an EEPROM that holds essential information about the module which is used when the module is integrated into a Netra ft 1800 system. The EEPROM is initialized at the point of manufacture.

The EEPROM contains information concerning the module's

- identification
- configuration
- properties
- history

Programming the EEPROM

For information about how to program the EEPROM, refer to the *Netra ft 1800 Module EEPROM v.4 Data File Specifications* (Part Number 950-3407-10).

Contents of the EEPROM

This section outlines the fields of interest in the module specific part of the EEPROM. Refer also to Table 4, *Fixed Fields – PCI Modules* in the *Netra ft 1800 Module EEPROM v.4 Data File Specifications* (Part Number 950-3407-10).

EE_PCI_DEVICE_ID This field contains the device ID of the PCI card.

EE_PCI_VENDOR_ID This field contains the vendor ID for the PCI card.

EE_PCI_DEVDATA This field contains information related to the devices on the module. A maximum of four devices per module is supported by the Netra ft 1800.

EE_PCI_DEVDATA_*N*_DEVNAME This field is used to generate the node names for the device.

EE_PCI_DEVDATA_*N*_PROPS This field relates to the properties of the PCI card and is available for use by the card. As the Netra ft 1800 does not probe the device to determine the card's properties, they are stored in the EEPROM in the format

property_name1=content1; / *property_name2=content2*;...

property can take the following data types:

TABLE 6-1 Property Data Types

Data Type	Example
string	"text";
string array	"text1",'text2"...;
integer	23;
integer array	23,42,...;
byte array	$xxyyzz;

EE_PCI_DEVDATA_*N*_FUNCTION_NO This field relates to the function of the device and can be in the range 0–7, or 255 for no function.

The following example shows the contents of the module specific part of an EEPROM. In this example, the `DEVNAME` `atm` is a local alias and is used for clarity.

```
E_MSP=
  EE_MSP_MSPVERS=4
  EE_MSP_FRUNAME=PCI
  EE_MSP_KEY=
    EE_MSP_KEY_REQUIREMENTS=1
    EE_MSP_KEY_SETSIZE=1
    EE_MSP_KEY_PRIMARY_CAPABILITIES=1
    EE_MSP_KEY_SECONDARY_CAPABILITIES=1
  EE_MSP_MAX_FAN_SPEED=0
  EE_MSP_VARIANT_TYPE=5
  EE_MSP_LOG_OFFSET=0
  EE_PCI=
    EE_PCI_DEVICE_ID=0
    EE_PCI_VENDOR_ID=0
    EE_PCI_DEVDATA=
      EE_PCI_DEVDATA_0=
        EE_PCI_DEVDATA_0_DEVNAME=atm
        EE_PCI_DEVDATA_0_PROPS=pll="none";\nmedia="multimode-
          fiber";\nmclk449="true";\natm-speed="SUNI-155";\nsahi-
          revision="d";\ncompatible="SUNW,ma";
        EE_PCI_DEVDATA_0_FUNCTION_NO=0
      EE_PCI_DEVDATA_1=
        EE_PCI_DEVDATA_1_DEVNAME=
        EE_PCI_DEVDATA_1_PROPS=
        EE_PCI_DEVDATA_1_FUNCTION_NO=255
      EE_PCI_DEVDATA_2=
        EE_PCI_DEVDATA_2_DEVNAME=
        EE_PCI_DEVDATA_2_PROPS=
        EE_PCI_DEVDATA_2_FUNCTION_NO=255
      EE_PCI_DEVDATA_3=
        EE_PCI_DEVDATA_3_DEVNAME=
        EE_PCI_DEVDATA_3_PROPS=
        EE_PCI_DEVDATA_3_FUNCTION_NO=255
  EE_MSP_CSUM=59698
```

Glossary

ASIC Application-Specific Integrated Circuit.

ASR Automatic System Recovery: reboot on system hang.

BMX+ Crossbar switch *ASIC*.

bridge The interface between the *CPUsets* and the I/O devices.

CAF Console, Alarms and Fans *module*.

CMS Configuration Management System. The software that records and monitors the *modules* in the system. Users access the CMS via a set of utilities which they use to add and remove modules from the system configuration and *enable* and *disable* modules that are in the system configuration.

component An identifiable part of a *module*.

configure (*CMS*) Notify the CMS that a *module* is present in a specified location.

constituent (*CMS*) An object that provides part of the functionality of another object. An object references its constituents.

CPUset A *module* containing the system processors and associated components.

craft-replaceable A *module* which clearly indicates when it is faulty and can be *hot-replaced* by a trained craftsperson.

DIMM Dual Inline Memory Module.

disable (*CMS*). Bring offline and power down a *module*.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

DSK Disk chassis *module*.

DVMA Direct Virtual Memory Access. A mechanism to enable a device on the PCI bus to initiate data transfers between it and the CPUsets.

ECC Error Correcting Code.

EEPROM Electrically Erasable Programmable Read Only Memory.

EMI Electro-magnetic Interference.

enable (*CMS*) Power up and bring online a *module* that is already *configured* into the system.

engineer-replaceable A *module* which may not indicate that it is faulty and which may require special tools for diagnosis and replacement. The Netra ft 1800 does not have any engineer-replacable modules.

ESD ElectroStatic Discharge.

EState Error limitation mode.

fault tolerant A system in which no single hardware failure can disrupt system operation.

fault-free No faults are evident in the operating system, or application software, or in external systems, except in the case of certain high demand real-time uses.

faulty module A *module* one or more of whose devices have gone into the degraded or failed states, as indicated to the *CMS* via the *hot-plug* device driver framework.

FPGA Field Programmable Gate Array.

front-replaceable The ability to replace a *module* from the front of the system.

FRU Field Replacable Unit. Another name for a *module*, used within the *CMS*.

HDD Hard Disk Drive.

hardened Specially engineered to be resistant to hardware and some causes of software failure. Applies to device drivers.

health features Features that can indicate that a fault is about to occur.

hot plug The ability to insert or remove a *module* without causing an interruption of service to the operating platform.

hotPCI An implementation of the PCI bus designed to minimize the probability that a fault on a *module* will corrupt the bus, and so to ensure that the system control mechanism runs without interruption

hot-replaceable A *module* that can be replaced without stopping the system.

I^2C Inter Integrated Circuit

IOMMU Input/Output Memory Management Unit

LED Light-Emitting Diode.

location A slot where a *module* can be inserted. Each location has a unique name and is clearly marked on the chassis.

lockstep The process by which two *CPUsets* work in synchronization.

losing side The side of a *split* system which has a new identity when rebooted.

MBD Motherboard.

Mbus Maintenance bus.

module An assembly that can be replaced without requiring the base machine to be returned to the factory. A module is a physical assembly that has a module number which is stored in the software on the machine, generally in the *EEPROM* of the physical assembly

PCI Peripheral Component Interconnect.

PCIO PCI-to-Ebus2/Ethernet controller *ASIC*.

PRI Processor re-integration. The process by which the two CPUsets come into *lockstep* to function as a fault tolerant system. *Re-integration* is preferred.

PROM Programmable Read Only Memory.

PSU Power Supply Unit.

RAS Reliability, Availability and Serviceability.

RCP Remote Control Processor.

RMM Removable Media Module.

RS232 An EIA specification that defines the interface between DTE and DCE using asynchronous binary data interchange.

SC_UP+ System controller *ASIC*.

side One *CPUset* and its associated *modules*, capable of running as a standalone system. A side is one half of a *fault tolerant* system or one of two systems in a *split* system.

SPF Single Point of Failure.

split system A system whose two *sides* run as separate systems.

stealthy PRI Stealthy processor re-integration. Processor re-integration (*PRI*) which is completed without user intervention.

subsystem (*CMS*) A fault tolerant configuration of *modules* defined in the CMS.

system attribute (*CMS*) An attribute of a CMS object that is written only by the CMS.

TLB Translation Lookaside Buffer. The hardware which handles the mapping of virtual addresses to real addresses.

surviving side The side of a *split* system which retains the identity of the previous *fault tolerant* system.

U2P UPA-to-PCI bridge (U2P) *ASIC*.

UPA UltraSPARC Port Architecture.

Index