Solaris 8 Software Developer Supplement

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-7503-10
April 2001

Contents

Preface

The *Solaris 8 Software Developer Supplement* describes new features in Solaris™ Update releases. The following information adds to or supersedes information in the previous releases of Solaris 8 documentation sets. Solaris documentation is available on the Solaris 8 Documentation CD.

Note - The Solaris operating environment runs on two types of hardware, or platforms — SPARC™ and IA (Intel Architecture). The Solaris operating environment also runs on both 64-bit and 32-bit address spaces. The information in this document pertains to both platforms and address spaces unless called out in a special chapter, section, note, bullet, figure, table, example, or code example.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

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

CHAPTER 1

What’s New at a Glance

This chapter highlights new features added to the Solaris™ 8 Update releases.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information not found in the *Solaris 8 Reference Manual Collection*.

TABLE 1–1 Solaris 8 Update Features

Feature	Update Release
Writing Device Drivers	
SPARC: The driver hardening test harness is a Solaris device driver development tool. The test harness injects a wide range of simulated hardware faults when the driver under development accesses its hardware. This fault-injection test harness tests the resilience of a SPARC based device driver. For more information, see Chapter 4	4/01
High-availability drivers provides a detailed description of how to design drivers to support high availability through driver hardening and ensuring serviceability. This material extends information provided in the Solaris 8 *Writing Device Drivers.* For more information, see Chapter 3.	10/00

TABLE 1–1 Solaris 8 Update Features *(continued)*

Feature	Update Release
You can use Generic LAN driver (GLD) to implement much of the STREAMS and Data Link Provider Interface (DLPI) functionality for a Solaris network driver. Until the Solaris 8 10/00 release, the GLD module was available only for Solaris *Intel Platform Edition* network drivers. Now GLD is available for Solaris *SPARC Platform Edition* network drivers as well. For the 4/01 release, GLD is updated with bug fixes. For more information, see Chapter 5.	10/00 and updated in 4/01
Language Support	
The File System Safe Universal Transformation Format, or UTF-8, is an encoding defined by X/Open as a multibyte representation of Unicode. UTF-8 encompasses almost all of the characters for traditional single-byte and multibyte locales for European and Asian languages for Solaris locales. For the 10/00 release, Russian and Polish and two new locales for Catalan are added. For the 4/01 release, two additional languages, Turkish UTF-8 Codeset and Russian UTF-8 Codeset are added to a table of already existing Eastern European locales. For more information, see "Additional Partial Locales for European Solaris Software" on page 63.	10/00 and updated in 4/01
The `mp` program accepts international text files of various Solaris locales and produces output that is proper for the specified locale. The output will also contain proper text layout, for instance, bidirectional text rendering, and shaping as the complex text layout (CTL) is supported in `mp`. Depending on each locale's system font configuration for `mp`, the PostScript™ output file can contain glyph images from Solaris system-resident scalable or bitmap fonts. For more information, see Chapter 8.	4/01
Development Tools	
The `appcert` utility verifies an object file's conformance to the Solaris ABI. Conforming to the Solaris ABI greatly increases an application's probability of being compatible with future releases of Solaris software. For more information, see Chapter 10.	4/01
Web-Based Enterprise Management (WBEM) includes standards for web-based management of systems, networks, and devices on multiple platforms. The Sun WBEM Software Developer's Toolkit (SDK) enables software developers to create standards-based applications that manage resources in the Solaris operating environment. Developers can also use this toolkit to write providers, programs that communicate with managed resources to access data. The Sun WBEM SDK includes Client Application Programming Interfaces (APIs) for describing and managing resources in Common Information Model (CIM), and Provider APIs for getting and setting dynamic data on the managed resource. The Sun WBEM SDK also provides CIM WorkShop, a Java application for creating and viewing the managed resources on a system, and a set of example WBEM client and provider programs. For more information, see the *Sun WBEM SDK Developer's Guide*.	4/01

TABLE 1–1 Solaris 8 Update Features *(continued)*

Feature	Update Release
SPARC: *Multithreaded Programming Guide* has been updated with bug fixes: 4308968, 4356675, 4356690. To view the book, see the *Multithreaded Programming Guide*	1/01
The *Linkers and Libraries Guide* has been updated with several new features. For more information, see "Changes to *Linkers and Libraries Guide* " on page 99.	1/01 and 10/00
System Interface Tools	
The *System Interface Guide* is updated to incorporate bug fixes. This release corrects several typographical errors in text and source code examples. To view the book, see *System Interface Guide*.	6/00
Java Releases	
The Java 2 SDK™ Standard Edition v. 1.3.0, also known as J2SE™ 1.3.0, is an upgrade release for Java 2 SDK. The J2SE release includes the following new features and enhancements. ■ Performance Improvements Java HotSpot™ technology- and performance-tuned runtime libraries make J2SE 1.3.0 the fastest Java™ platform to date. ■ Easier Web Deployment New features such as applet caching and automatic installation of optional packages by J2SE 1.3.0's Java™ Plug-in component enhance the speed and flexibility with which you can display programs on the web. ■ Enterprise Interoperability The addition of RMI/IIOP and the Java Naming and Directory Interface™ in J2SE 1.3.0 enhance the interoperability of the Java 2 Platform. ■ Security Advances New support for RSA electronic signing, dynamic trust management, X.509 certificates, and verification of Netscape-signed files mean more ways for developers to protect their electronic data. ■ Java Sound J2SE 1.3.0 includes a powerful new sound API. Previous releases of the platform limited audio support to basic playback of audio clips. With this release, the Java 2 Platform defines a set of standard classes and interfaces for low-level audio support. ■ Enhanced APIs and Improved Ease of Development In response to requests from the development community, J2SE 1.3.0 adds new features to various areas of the Java 2 Platform. These features expand the functionality of the platform to enable development of more powerful applications. In addition, many of the new features make the development process itself faster and more efficient. For more J2SE improvements, see "Java 2 SDK, Standard Edition, version 1.3.0" on page 111.	4/01

TABLE 1–1 Solaris 8 Update Features *(continued)*

Feature	Update Release
The J2SE 1.2.2_07a contains fixes for bugs that were identified in previous releases in the J2SE 1.2.2 series. An important bug fix is a fix for a performance regression that was introduced in J2SE 1.2.2_05. For more information about bug fixes in J2SE 1.2.2_07a, see this web site: `http://java.sun.com/j2se/1.2/ReleaseNotes.html`.	4/01
The Java 2 SDK 1.2.2_06 and JDK 1.1.8_12 are improved with bug fixes since the last release.	1/01
The Java 2 SDK 1.2.2_05a includes the following new features. ■ Scalability improvements to over 20 CPUs ■ Improved just-in time (JIT) compiler optimizations ■ Text-rendering performance improvements ■ `poller` class demo package ■ Swing improvements For more information, see Table 18–1.	10/00
32-bit: With the addition of the `mod_jserv` module and related files, the Apache web server now supports Java servlets. For more information, see “Java Servlet Support in Apache Web Server” on page 122.	10/00
Early Access	
This release includes an Early Access (EA) directory with EA software. For more information, see the Readme on the Solaris Software CD 2 of 2.	

CHAPTER **2**

Writing Device Drivers Topics

This section provides instructions for writing device drivers in the Solaris environment. This section contains these chapters.

Chapter 3	Provides information on driver hardening to prevent driver failures resulting from device failures
Chapter 4	Describes how to configure the test harness, create error injection specifications (referred to as *errdefs*), and execute the tests on your device driver
Chapter 5	Provides the information to use Generic LAN driver (GLD) to implement much of the STREAMS and Data Link Provider Interface (DLPI) functionality for a Solaris network driver

CHAPTER **3**

High-Availability Drivers

Driver hardening is new in the Solaris 8 10/00 release. For more information about how to create a Solaris device driver, see *Writing Device Drivers*.

Availability is a function of both failure rate and speed of repair. In many cases, the failure of an individual device need not result in a total system failure. Redundant hardware components, together with drivers that are designed to support High Availability, can allow a system to continue operation even in the face of individual component failure. In many cases, such drivers can allow the system to be repaired even while it continues to provide service.

The programmatic elimination of driver failures that result from device failures is called *driver hardening*. A hardened driver can tolerate and protect the rest of the system from errors that might otherwise propagate from a faulty device.

Functions within a driver that help isolate faults and assist in more rapid recovery and repair improve the system Serviceability; this improves Availability by reducing time to repair.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Driver Hardening

Hardening is the process of ensuring that a driver works correctly even though faults occur in the I/O device that it controls or other faults originating outside the system core. A hardened driver must not panic, hang the system, or allow the uncontrolled spread of corrupted data as the result of any such faults.

The driver developer must take responsibility for:

- Correct use of the DDI functions
- Detecting and reporting any corruption of device I/O
- Handling devices with deviant interrupt logic

All Solaris drivers should be hardened. Hardened drivers obey these rules:

- Each piece of hardware should be controlled by a separate instance of the device driver.
- Programmed I/O (PIO) must be performed *only* through the DDI access functions, by using the appropriate data access handle.
- The device driver must assume that data it receives from the device could be corrupted. The driver must check the integrity of the data before using it.
- The driver must control the effects of any faults that it detects. Known bad data must not be released to the rest of the system.
- The driver must ensure that all writes by the device into DMA buffers (`DDI_DMA_READ`) are contained within pages of memory that are controlled entirely by the driver. This prevents a DMA fault from corrupting an arbitrary part of the system's main memory.
- The device driver must not be an unlimited drain on system resources if the device locks up. The driver should time-out if a device claims to be continuously busy. The driver should also detect a pathological (stuck) interrupt request and take appropriate action.
- The driver must free resources after a fault. For example, the system must be able to close all minor devices and detach driver instances even after the hardware fails.

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 other instances. The device is managed on a per-instance basis. Hardened drivers should use a separate instance for each piece of hardware unless the driver is designed to handle failover internally. There can be multiple instances of a driver per slot, for example, multi-function cards, which is standard behavior for Solaris device drivers.

Exclusive Use of DDI Access Handles

All programmed I/O (PIO) access by a hardened driver must use Solaris DDI access functions from the `ddi_get`*X*, `ddi_put`*X*, `ddi_rep_get`*X*, and `ddi_rep_put`*X* families of routines. The driver should not directly access the mapped registers by the address returned from `ddi_regs_map_setup`(9F). The use of an access handle ensures that an I/O fault is controlled and its effects confined to the returned value, rather than possibly corrupting other parts of the machine state. (Avoid the `ddi_peek`(9F) and `ddi_poke`(9F) routines because they do not use access handles.)

The DDI access mechanism is important because it provides an opportunity to control how data is read into the kernel. DDI access routines provide protection by constraining the effect of bus time-out traps.

Detecting Corrupted Data

The following sections consider where data corruption can occur and the steps you can take to detect it.

Corruption of Device Management and Control Data

The driver should assume that any data obtained from the device, whether by PIO or DMA, could have been corrupted. In particular, extreme care should be taken with pointers, memory offsets, or array indexes read or calculated from data that is supplied by the device. Such values can be *malignant*, meaning they can cause a kernel panic if dereferenced. All such values should be checked for range and alignment (if required) before use.

Even if a pointer is not malignant, it can still mislead. For example, it can point at a valid instance of an object, but not the correct one. Where possible, the driver should cross-check the pointer with the pointed-to object, or otherwise validate the data obtained through it.

Other types of data can also be misleading, such as packet lengths, status words, or channel IDs. Each type of data should be checked to the extent possible: a packet length can be range-checked to ensure that it is not negative or larger than the containing buffer; a status word can be checked for ”impossible” bits; and a channel ID can be matched against a list of valid IDs.

Where a value is used to identify a Stream, the driver must ensure that the Stream still exists. The asynchronous nature of STREAMS processing means that a Stream can be dismantled while device interrupts are still outstanding.

The driver should not reread data from the device. The data should be read once, validated, and stored in the driver’s local state. This avoids the hazard presented by data that, although correct when initially read and validated, is incorrect when reread later.

The driver should also ensure that all loops are bounded so that a device returning a continuous BUSY status, or claiming that another buffer needs to be processed, does not lock up the entire system.

Corruption of Received Data

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

If the received data will not be checked for integrity 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 typically device-specific (checksums, CRC, and so forth).

Detecting Faults

Any ancestor of a device driver can disable the data path to the device if it detects a fault. When PIO access is disabled, any reads from the device return `undefined` values, while writes are ignored. If DMA access is disabled, the device might be prevented from accessing memory, or it might receive undefined data on reads and have writes discarded.

A device driver can detect that a data path has been disabled by using the following DDI routines:

- `ddi_check_acc_handle`(9F)
- `ddi_check_dma_handle`(9F)

Each function checks whether any faults affecting the data path represented by the supplied *handle* have been detected. If one of these functions returns `DDI_FAILURE`, indicating that the data path has failed, the driver should report the fault by using `ddi_dev_report_fault`(9F), perform any necessary cleanup, and, where possible, return an appropriate error to its caller.

Containment of Faults

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

- The `ddi_check_acc_handle`(9F) and `ddi_check_dma_handle`(9F) calls should be made at significant junctures, such as just before 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, such as memory, are not permanently lost when data cannot be forwarded. Threads should not remain blocked waiting for signals that will never be generated.

- The driver should limit its processing while in the failed state (for example, freeing messages in wput routines, attempting to permanently disable interrupts from a failed board, and so forth).

DMA Isolation

A defective device might initiate an improper DMA transfer over the bus. This data transfer could corrupt good data that was previously delivered. A device that fails might generate a corrupt address that can contaminate memory that does not even belong to its own driver.

In systems with an IOMMU, a device can write only to pages mapped as writable for DMA. Therefore, pages that are to be the target of DMA writes should be owned solely by one driver instance and not shared with any other kernel structure. While the page in question is mapped as writable for DMA, the driver should be suspicious of data in that page. The page must be unmapped from the IOMMU before it is passed beyond the driver, or before any validation of the data.

You can use ddi_umem_alloc(9F) to guarantee that a whole aligned page is allocated, or allocate multiple pages and ignore the memory below the first page boundary. You can find the size of an IOMMU page by using ddi_ptob(9F).

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 by using ddi_dma_sync(9F).

Calls to ddi_dma_sync(9F) should specify SYNC_FOR_DEV before using DMA to transfer data to a device, and SYNC_FOR_CPU after using DMA to transfer data from the device to memory.

On some PCI-based systems with an IOMMU, devices might be able to use PCI dual address cycles (64-bit addresses) to bypass the IOMMU. This gives the device the potential to corrupt any region of main memory. Hardened device drivers must not attempt to use such a mode and should disable it.

Handling Stuck Interrupts

The driver must identify stuck interrupts because a persistently asserted interrupt severely affects system performance, almost certainly stalling a single-processor machine.

Sometimes it is difficult for the driver to identify a particular interrupt as bogus. For network drivers, if a receive interrupt is indicated but no new buffers have been made available, 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).

On the other hand, continuous interrupts with no work for the driver to process can indicate a stuck interrupt line. For this reason, all platforms allow a number of apparently bogus interrupts to occur before taking defensive action.

A hung device, while appearing to have work to do, might be failing to update its buffer descriptors. The driver should defend against such repetitive requests.

In some cases, platform–specific bus drivers might be capable of identifying a persistently unclaimed interrupt and can disable the offending device. However, this relies on the driver's ability to identify the valid interrupts and return the appropriate value. The driver should therefore 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 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*, which can be generated if all the device's descriptors had been previously 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 is suspicious.

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 polling cycle.

If the interrupt occurs again while disabled, this should be considered a false interrupt. Some devices have interrupt status bits that can be read even if the mask register has disabled the associated source and might not be causing the interrupt. Driver designers can devise more appropriate algorithms specific to their devices.

Avoid looping on interrupt status bits indefinitely. Break such loops if none of the status bits set at the start of a pass requires any real work.

Additional Driver Hardening Considerations

In addition to the requirements discussed in the previous sections, the driver developer must consider a few other issues such as:

- Thread interaction
- Threats from top-down requests
- Adaptive strategies

Thread Interaction

Kernel panics in a device driver are often caused by the unexpected interaction of kernel threads after a device failure. When a device fails, threads can interact in ways that the designer had not anticipated.

For example, if processing routines terminate early, they might fail to signal other threads that are waiting on condition variables. Attempting to inform other modules of the failure or handling unanticipated callbacks can result in undesirable thread interactions. Examine the sequence of mutex acquisition and relinquishment that can occur during device failures.

Threads that originate in an upstream STREAMS module can run into unfortunate paradoxes if used to call back into that module unexpectedly. You might use alternative threads to handle exception messages. For instance, a `wput` procedure might use 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 not attempt to use a stale Stream pointer to try to process the message.

Threats From Top-Down Requests

While protecting the system from defective hardware, the driver designer also needs to protect against driver misuse. Although the driver can assume that the kernel infrastructure is always correct (a trusted core), user requests passed to it can be potentially destructive.

For example, a user can request an action to be performed on a user-supplied data block (`M_IOCTL`) that is smaller than that indicated in the control part of the message. The driver should never trust a user application.

The design should consider the construction of each type of `ioctl` that it can receive with a view to the potential harm that it could cause. The driver should make checks to be sure that it does not process malformed `ioctl`s.

Adaptive Strategies

A driver can continue to provide service with faulty hardware and attempt to work around the identified problem by using an alternative strategy for accessing the device. Given that broken hardware is unpredictable and given the risk associated with additional design complexity, adaptive strategies are not always wise. At most, they should be limited to periodic interrupt polling and retry attempts. Periodically retrying the device lets the driver know when a device has recovered. Periodic polling can control the interrupt mechanism after a driver has been forced to disable interrupts.

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

Serviceability

To ensure serviceability, you must enable the driver to do the following:

- Detect faulty devices and report the fault
- Remove a device (as supported by the Solaris hot-plug model)
- Add a new device (as supported by the Solaris hot-plug model)
- Perform periodic health checks to enable the detection of latent faults

Checking the Current Device State

A driver must check its device state at appropriate points in order to avoid needlessly committing resources. The ddi_get_devstate(9F) function enables the driver to determine the device's current state, as maintained by the framework.

```
ddi_devstate_t ddi_get_devstate(dev_info_t *dip);
```

The driver is not normally called on to handle a device that is OFFLINE. Generally, the device state reflects earlier device fault reports, possibly modified by any reconfiguration activities that have occurred.

Correct Behavior When a Device Has Failed

The system must report a fault in terms of the impact it has on the ability of the device to provide service. Typically, loss of service is expected when:

- A PIO or DMA error is detected.
- Data corruption is detected.
- The device is locked or hung (for example, when a command never completes).
- A condition has occurred that the driver does not handle because it was regarded as impossible when the driver was designed.

If the device state, returned by ddi_get_devstate(9F), indicates that the device is not usable, the driver should reject all new and outstanding I/O requests and return (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 way. 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 recovering error status) should continue to work. The state check might therefore need to be on a per-command basis. Alternatively, 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_INTR_UNCLAIMED` for all subsequent interrupts. If interrupts continue to be generated the eventual result is that the interrupt is disabled.

Fault Reporting

This following function notifies the system that your driver has discovered a device fault.

```
void ddi_dev_report_fault(dev_info_t *dip, ddi_fault_impact_t impact,
             ddi_fault_location_t location, const char *message);
```

The *impact* parameter indicates the impact of the fault on the device's ability to provide normal service, 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 service-lost fault causes the device state to be changed to `DOWN` and a service-degraded fault causes the device state to be changed to `DEGRADED`.

A device should be reported as faulty if:

- A PIO error is detected.
- Corrupted data is detected.
- The device has locked up.

Drivers should avoid reporting the same fault repeatedly, if possible. In particular, it is redundant (and undesirable) for drivers to report any errors if the device is already in an unusable state (see `ddi_get_devstate`(9F)).

If a hardware fault is detected during the attach process, the driver must report the fault by using `ddi_dev_report_fault`(9F) as well as by returning `DDI_FAILURE`.

Periodic Health Checks

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

Generally, latent faults that are allowed to remain undetected will eventually cause system failure. Without latent fault checking, the overall availability 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 in which the device can be the secondary or failover device, early detection of a failed secondary device is essential to ensure that it can be repaired or replaced before any failure in the primary device occurs.

Periodic health checks can:

- Run a quick access check on the board (write, read), then check the device with the `ddi_check_acc_handle`(9F) routine.
- Check a register or memory location on the device that has a value the driver expects to have been deterministically altered since the last poll.

  Features of a device that typically exhibit deterministic behavior include heartbeat semaphores, device timers (for example, local `lbolt` that is 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 overaged requests that have not completed.
- Initiate an action on the device that should be completed before the next scheduled check.

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

CHAPTER **4**

SPARC: Driver Hardening Test Harness

The driver hardening test harness is new in the Solaris 8 *SPARC™ Platform Edition* 4/01 release. For information about how to create a Solaris device driver, see *Writing Device Drivers*.

The driver hardening test harness is a Solaris device driver development tool. The test harness injects a wide range of simulated hardware faults when the driver under development accesses its hardware. This chapter describes how to configure the test harness, create error-injection specifications (referred to as *errdefs*), and execute the tests on your device driver.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Test Harness Description

Hardened device drivers are resilient to potential hardware faults. You must test the resilience of device drivers as part of the driver development process. This type of testing requires that the driver handle a wide range of typical hardware faults in a controlled and repeatable way. The driver hardening test harness enables driver developers to simulate such hardware faults in software.

The test harness intercepts calls from the driver to various DDI routines, then corrupts the result of the calls as if the hardware had caused the corruption. In addition, the harness allows for corruption of accesses to specific registers as well as definition of more random types of corruption.

Note - The driver must perform all I/O accesses by using DDI routines to comply with the Solaris DDI/DKI.

The test harness can generate test scripts automatically by tracing all register accesses as well as direct memory access (DMA) and interrupt usage during the running of a specified workload. A script is generated that reruns that workload while injecting a set of faults into each access.

The driver tester must create additional test cases to force the driver down more obscure failure paths. The tester should also remove duplicate test cases from the generated scripts.

The test harness is implemented as a device driver called `bofi`, which stands for `bus_ops` fault injection, and two user-level utilities, `th_define`(1M) and `th_manage`(1M).

The test harness does the following:

- Validates compliant use of Solaris DDI services
- Facilitates controlled corruption of programmed I/O (PIO) and DMA requests and interference with interrupts, thus simulating faults that occur in the hardware managed by the driver
- Facilitates simulation of failures in the data path between the CPU and the device, which are reported from parent nexus drivers
- Monitors a driver's access during a specified workload and generates fault-injection scripts

Fault Injection

The driver hardening test harness intercepts and, when requested, corrupts each access a driver makes to its hardware. This section provides information you should understand to create faults to test the resilience of your driver.

Data Access Functions

Solaris devices are managed inside a tree-like structure called the device (devinfo) tree. Each node of the devinfo tree stores information that relates to a particular instance of a device in the system. Each leaf node corresponds to a device driver, while all other nodes are called *nexus nodes*. Typically, a nexus represents a bus. A bus node isolates leaf drivers from bus dependencies, which enables architecturally independent drivers to be produced.

Many of the DDI functions, particularly the data access functions (DAFs), result in upcalls to the bus nexus drivers. When a leaf driver accesses its hardware, it passes a handle to an access routine. The bus nexus understands how to manipulate the handle and fulfill the request. A DDI-compliant driver only accesses hardware through use of these DDI access routines. The test harness intercepts these upcalls before they reach the specified bus nexus. If the data access matches the criteria that is specified by the driver tester, the access will be corrupted. If the data access does not match the criteria, it is given to the bus nexus to handle in the usual way.

A driver obtains an access handle by using the `ddi_map_regs_setup(`*dip*, *rset*, *ma*, *offset*, *size*, *handle*`)` function. The arguments specify which ''offboard'' memory is to be mapped. The driver must use the returned handle when it references the mapped I/O addresses, as handles are meant to isolate drivers from the details of bus hierarchies. Therefore, do not directly use the returned mapped address, *ma*. Direct use of the mapped address destroys the current and future uses of the DAF mechanism.

For programmed I/O, the suite of DAFs are:

- I/O to Host: `ddi_get`*X*`(`*handle*, *ma*`)` and `ddi_rep_get`*X*`(`*handle*, *buf*, *ma*, *repcnt*, *flag*`)`
- Host to I/O: `ddi_put`*X*`(`*handle*, *ma*, *value*`)` and `ddi_rep_put`*X*`()`

X and *repcnt* are the number of bytes to be transferred. *X* is the bus transfer size of 8, 16, 32, or 64 bytes.

DMA has a similar, yet richer, set of DAFs.

Setting Up the Test Harness

The driver hardening test harness is part of the Solaris Developer Cluster and the Entire Distribution Cluster. If you have not installed either of these Solaris clusters, you must manually install the test harness packages appropriate for your platform.

Installing the Test Harness

To install the test harness packages (`SUNWftduu`, `SUNWftdur`, and `SUNWftdux`), use `pkgadd`(1M).

As superuser, go to the directory in which the packages are located and type:

```
# pkgadd -d . SUNWftduu SUNWftdur SUNWftdux
```

Configuring the Test Harness

After the test harness is installed, edit the `/kernel/drv/bofi.conf` file to configure the harness to interact with your driver. See the following section for descriptions of the test harness properties.

When the harness configuration is complete, reboot the system to load the harness driver.

Test Harness Properties

The test harness behavior is controlled by boot-time properties that were set in the `/kernel/drv/bofi.conf` configuration file.

When the harness is first installed, enable the harness to intercept the DDI accesses to your driver by setting these properties:

`bofi-nexus` Bus nexus type, such as the PCI bus

`bofi-to-test` Name of the driver under test

For example, to test a PCI bus network driver called `xyznetdrv`, set the following property values:

```
bofi-nexus="pci"
bofi-to-test="xyznetdrv"
```

Other properties relate to the use and harness checking of the Solaris DDI data access mechanisms for reading and writing from peripherals that use PIO and transferring data to and from peripherals that use DMA.

`bofi-range-check` When this property is set, the test harness checks the consistency of the arguments that are passed to PIO DAFs.

`bofi-ddi-check` When this property is set, the test harness verifies that the mapped address that is returned by `ddi_map_regs_setup()` is not used outside of the context of the DAFs.

`bofi-sync-check` When this property is set, the test harness verifies correct usage of DMA functions and ensures that the driver makes compliant use of `ddi_dma_sync()`.

Testing the Driver

This section describes how to create and inject faults by using the th_define(1M) and th_manage(1M) commands.

Creating Faults

The th_define(1M) utility provides an interface to the bofi device driver for defining errdefs. An errdef corresponds to a specification for how to corrupt a device driver's accesses to its hardware. The th_define command-line arguments determine the precise nature of the fault to be injected. If the supplied arguments define a consistent errdef, the th_define process stores the errdef with the bofi driver. The process suspends itself until the criteria given by the errdef becomes satisfied. In practice, the suspension ends when the access counts go to zero (0).

Injecting Faults

The test harness operates at the level of data accesses. The characteristics of a data access include the:

- Type of hardware being accessed (driver name)
- Instance of the hardware being accessed (driver instance)
- Register set being tested
- Subset of the register set that is targeted
- Direction of the transfer (read or write)
- Type of access (PIO or DMA)

The test harness intercepts data accesses and injects appropriate faults into the driver. An errdef, specified by the th_define(1M) command, encodes the following information:

- The driver instance and register set being tested (–n *name*, –i *instance*, and –r *reg_number*)
- The subset of the register set eligible for corruption. This subset is indicated by providing an offset into the register set and a length from that offset (–l *offset* [*len*])
- The kind of access to be intercepted: log, pio, dma, pio_r, pio_w, dma_r, dma_w, intr (–a *acc_types*)
- How many accesses should be faulted (–c *count* [*failcount*])

- The kind of corruption that should be applied to a qualifying access (–o *operator* [*operand*]):
  - Replace datum with a fixed value (EQUAL)
  - Perform a bitwise operation on the datum (AND, OR, XOR)
  - Ignore the transfer (for host to I/O accesses NO_TRANSFER)
  - Lose, delay, or inject spurious interrupts (LOSE, DELAY, EXTRA)

Use the –a *acc_chk* option to simulate framework faults in an errdef.

Fault-Injection Process

The process of injecting a fault involves two phases:

1. **Create errdefs by using the** `th_define` **command.**

   Create errdefs by passing test definitions to the `bofi` driver, which stores the definitions so they can be accessed by using `th_manage`(1M).

2. **Create a workload, then use** `th_manage` **to activate and manage the errdef.**

   The `th_manage`(1M) command is a user interface to the various `ioctl`s that are recognized by the `bofi` harness driver. `th_manage` operates at the level of driver names and instances and includes these commands: `get_handles` to list access handles, `start` to activate errdefs, and `stop` to deactivate errdefs.

   The activation of an errdef results in qualifying data accesses to be faulted. The `th_manage` utility supports these commands: `broadcast` to provide the current state of the errdef and `clear_errors` to clear the errdef.

   See `th_define`(1M) and `th_manage`(1M) for more information.

Test Harness Warnings

You can configure the test harness to handle warning messages in the following ways:

- Write warning messages to the console
- Write warning messages to the console and then panic the system

Use the second method to help pinpoint the root cause of a problem.

When the `bofi-range-check` property value is set to `warn`, the harness prints the following messages (or panics if set to `panic`) when it detects a range violation of a DDI function by your driver:

```
ddi_getX() out of range addr %x not in %x
ddi_putX() out of range addr %x not in %x
ddi_rep_getX() out of range addr %x not in %x
```

(Continuation)

```
ddi_rep_putX() out of range addr %x not in %x
```

X is 8, 16, 32, or 64.

When the harness has been requested to insert over 1000 extra interrupts, the following message is printed if the driver does not detect interrupt jabber:

```
undetected interrupt jabber - %s %d
```

Using Scripts to Automate the Test Process

You can create fault-injection test scripts by using the logging access type of the `th_define` utility:

```
# th_define –n name –i instance –a log [–e fixup_script]
```

`th_define` takes the instance offline and brings it back online. Then `th_define` runs the workload that is described by the fixup script and logs I/O accesses that are made by the driver instance.

The fixup script is called twice with the set of optional arguments—once just before the instance is taken offline and again after the instance has been brought online. The following variables are passed into the environment of the called executable:

`DRIVER_PATH`	Device path of the instance
`DRIVER_INSTANCE`	Instance number of the driver
`DRIVER_UNCONFIGURE`	Set to 1 when the instance is about to be taken offline
`DRIVER_CONFIGURE`	Set to 1 when the instance has just been brought online

Typically, the fixup script ensures that the device under test is in a suitable state to be taken offline (unconfigured) or in a suitable state for error injection (for example, configured, error free, and servicing a workload). A minimal script for a network driver could be:

```
#!/bin/ksh
driver=xyznetdrv
ifnum=$driver$DRIVER_INSTANCE

if [[ $DRIVER_CONFIGURE = 1 ]]; then
   ifconfig $ifnum plumb
   ifconfig $ifnum ...
   ifworkload start $ifnum
elif [[ $DRIVER_UNCONFIGURE = 1 ]]; then
   ifworkload stop $ifnum
   ifconfig $ifnum down
   ifconfig $ifnum unplumb
fi
exit $?
```

Note - `ifworkload` should initiate the workload as a background task. The fault injection occurs after the fixup script configures the driver under test and brings it online (`DRIVER_CONFIGURE` is set to 1).

If the `–e` *fixup_script* option is present, it must be the last option on the command line. However, if that option is not present, a default script is used. The default script repeatedly attempts to bring the device under test offline and online. Thus the workload consists of the driver's attach and detach paths.

The resulting log is converted into a set of executable scripts that are suitable for running unassisted fault-injection tests. These scripts are created in a subdirectory of the current directory with the name *driver*`.test.`*id*. The scripts inject faults, one at a time, into the driver while running the workload that is described by the fixup script.

The driver tester has substantial control over the errdefs that are produced by the test automation process. See `th_define`(1M).

If the tester chooses a suitable range of workloads for the test scripts, the harness gives good coverage of the hardening aspects of the driver. However, to achieve full coverage, the tester might need to create additional test cases manually. Add these cases to the test scripts. To ensure that testing completes in a timely manner, the tester might need to manually delete duplicate test cases.

Automated Test Process

The process for automated testing follows.

1. **Identify the aspects of the driver to be tested.**

   Test all aspects of the driver that interact with the hardware:

   - Attach and detach
   - Plumb and unplumb under a stack

- Normal data transfer
- Documented debug modes

A separate workload script (*fixup_script*) must be generated for each mode of use.

2. **For each mode of use, prepare an executable program (*fixup_script*) that configures and unconfigures the device, and creates and terminates a workload.**

3. **Run** `th_define` **with the errdefs, together with an access type of** `–a log`**.**

4. **Wait for the logs to fill.**

   The logs contain a dump of the `bofi` driver's internal buffers. This data is included at the front of the script.

   Because it can take from a few seconds to several minutes to create the logs, use the `th_manage broadcast` command to check the progress.

5. **Change to the created test directory and run the master test script.**

   The master script runs each generated test script in sequence. Separate test scripts are generated per register set.

6. **Store the results for analysis.**

   Successful test results, such as `success (corruption reported)` and `success (corruption undetected)`, show that the driver under test is behaving properly.

   It is fine for a few `test not triggered` failures to appear in the output. However, several such failures indicate that the test is not working properly. These failures can appear when the driver does not access the same registers as when the test scripts were generated.

7. **Run the test on multiple instances of the driver concurrently to test the multithreading of error paths.**

   For example, each `th_define` command creates a separate directory that contains test scripts and a master script:

```
# th_define -n xyznetdrv -i 0 -a log -e script
# th_define -n xyznetdrv -i 1 -a log -e script
```

Once created, run the master scripts in parallel.

Note - The generated scripts produce only simulated fault injections that are based on what was logged during the time the logging errdef was active. When you define a workload, ensure that the required results are logged. Also analyze the resulting logs and fault-injection specifications. Verify that the hardware access coverage that the resulting test scripts created is what is required.

CHAPTER **5**

Drivers for Network Devices

Generic LAN driver is new for the Solaris 8 10/00 release.

The Generic LAN driver (GLD) implements much of the STREAMS and Data Link Provider Interface (DLPI) functionality for a Solaris™ network driver.

Until Solaris 8 10/00, the GLD module was only available for Solaris *Intel Platform Edition* network drivers. Now GLD is available for Solaris *SPARC™ Platform Edition* network drivers, as well.

For more information, see `gld`(7D), `dlpi`(7P), `gld`(9E), `gld`(9F), `gld_mac_info`(9S), `gld_stats`(9S).

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Generic LAN Driver Overview

GLD is a multi-threaded, clonable, loadable kernel module providing support for Solaris local area network device drivers. Local area network (LAN) device drivers in Solaris are STREAMS-based drivers that use DLPI to communicate with network protocol stacks. These protocol stacks use the network drivers to send and receive packets on a local area network. A network device driver must implement and conform to the requirements imposed by the DDI/DKI specification, STREAMS specification, DLPI specification, and programmatic interface of the device itself.

GLD implements most STREAMS and DLPI functionality required of a Solaris LAN driver. Several Solaris network drivers are implemented using GLD.

A Solaris network driver implemented using GLD is made up of two distinct parts: a generic component that deals with STREAMS and DLPI interfaces, and a device-specific component that deals with the particular hardware device. The device-specific module indicates its dependency on the GLD module (which is found at /kernel/misc/gld) and registers itself with GLD from within the driver's attach(9E) function. After it is successfully loaded, the driver is DLPI-compliant. The device-specific part of the driver calls gld(9F) functions when it receives data or needs some service from GLD. GLD makes calls into the gld(9E) entry points of the device-specific driver through pointers provided to GLD by the device-specific driver when it registered itself with GLD. The gld_mac_info(9S) structure is the main data interface between GLD and the device-specific driver.

The GLD facility currently supports devices of type DL_ETHER, DL_TPR, and DL_FDDI. GLD drivers are expected to process fully formed MAC-layer packets and should not perform logical link control (LLC) handling.

In some cases, you might need or want to implement a full DLPI-compliant driver without using the GLD facility. This is true for devices that are not ISO 8802-style (IEEE 802) LAN devices, or where you need a device type or DLPI service not supported by GLD.

Type DL_ETHER: Ethernet V2 and ISO 8802-3 (IEEE 802.3)

For devices designated type DL_ETHER, GLD provides support for both Ethernet V2 and ISO 8802-3 (IEEE 802.3) packet processing. Ethernet V2 enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol. A service access point (SAP) is the point through which the user communicates with the service provider.

Streams bound to SAP values in the range [0-255] are treated as equivalent and denote that the user wants to use 8802-3 mode. If the value of the SAP field of the DL_BIND_REQ is within this range, GLD computes the length (not including the 14-byte media access control (MAC) header) of each subsequent DL_UNITDATA_REQ message on that Stream and transmits 8802-3 frames having those lengths in the MAC frame header type fields. Such lengths never exceed 1500.

All frames received from the media that have a type field in the range [0-1500] are assumed to be 8802-3 frames and are routed up all open Streams that are in 8802-3 mode (those Streams bound to a SAP value in the [0-255] range). If more than one Stream is in 8802-3 mode, the incoming frame is duplicated and routed up each such Stream.

Streams bound to SAP values greater than 1500 (Ethernet V2 mode) receive incoming packets whose Ethernet MAC header type value exactly matches the value of the SAP to which the Stream is bound.

Types DL_TPR and DL_FDDI: SNAP Processing

For media types DL_TPR and DL_FDDI, GLD implements minimal SNAP (Sub-Net Access Protocol) processing for any Stream bound to a SAP value greater than 255. SAP values in the range [0-255] are LLC SAP values and are carried naturally by the media packet format. SAP values greater than 255 require a SNAP header, subordinate to the LLC header, to carry the 16-bit Ethernet V2-style SAP value.

SNAP headers are carried under LLC headers with destination SAP 0xAA. For outgoing packets with SAP values greater than 255, GLD creates an LLC+SNAP header that always looks like:

```
AA AA 03 00 00 00 XX XX
```

where ‘‘XX XX’’ represents the 16-bit SAP, corresponding to the Ethernet V2 style ‘‘type.’’ This is the only class of SNAP header supported—non-zero OUI fields and LLC control fields other than 03 are considered to be LLC packets with SAP 0xAA. Clients wanting to use SNAP formats other than this one must use LLC and bind to SAP 0xAA.

Incoming packets are examined to ascertain whether they conform to the format shown above. Packets that conform to this format are matched to any Streams bound to the packet’s 16-bit SNAP type, as well as being considered to match the LLC SNAP SAP 0xAA.

Packets received for any LLC SAP are passed up all Streams that are bound to an LLC SAP, as described for media type DL_ETHER.

Type DL_TPR: Source Routing

For type DL_TPR devices, GLD implements minimal support for source routing. Source routing enables a station that is sending a packet across a bridged medium to specify (in the packet MAC header) routing information that determines the route that the packet will take through the network.

Functionally, the source routing support provided by GLD learns routes, solicits and responds to requests for information about possible multiple routes, and selects among available routes. It adds *Routing Information Fields* to the MAC headers of outgoing packets and recognizes such fields in incoming packets.

GLD’s source routing support does not implement the full *Route Determination Entity* (RDE) specified in Section 9 of *ISO 8802-2 (IEEE 802.2)*. However, it is designed to interoperate with any such implementations that might exist in the same (or a bridged) network.

Style 1 and Style 2 DLPI Providers

GLD implements both Style 1 and Style 2 DLPI providers. A physical point of attachment (PPA) is the point at which a system attaches itself to a physical communication medium. All communication on that physical medium funnels through the PPA. The Style 1 provider attaches the Stream to a particular PPA based on the major/minor device that has been opened. The Style 2 provider requires the DLS user to explicitly identify the desired PPA using DL_ATTACH_REQ. In this case, open(9E) creates a Stream between the user and GLD, and DL_ATTACH_REQ subsequently associates a particular PPA with that Stream. Style 2 is denoted by a minor number of zero. If a device node whose minor number is not zero is opened, Style 1 is indicated and the associated PPA is the minor number minus 1. In both Style 1 and Style 2 opens, the device is cloned.

Implemented DLPI Primitives

GLD implements several DLPI primitives. The DL_INFO_REQ primitive requests information about the DLPI Stream. The message consists of one M_PROTO message block. GLD returns device-dependent values in the DL_INFO_ACK response to this request, based on information the GLD-based driver specified in the gldm_mac_info(9S) structure passed to gld_register(). However, GLD returns the following values on behalf of all GLD-based drivers:

- Version is DL_VERSION_2.
- Service mode is DL_CLDLS — GLD implements connectionless-mode service.
- Provider style is DL_STYLE1 or DL_STYLE2, depending on how the Stream was opened.
- No optional Quality Of Service (QOS) support is present and the QOS fields are zero.

Note - Contrary to the DLPI specification, GLD returns the device’s correct address length and broadcast address in DL_INFO_ACK even before the Stream has been attached to a PPA.

The DL_ATTACH_REQ primitive is used to associate a PPA with a Stream. This request is needed for Style 2 DLS providers to identify the physical medium over which the communication will transpire. Upon completion, the state changes from DL_UNATTACHED to DL_UNBOUND. The message consists of one M_PROTO message block. This request is not permitted when using the driver in Style 1 mode; Streams opened using Style 1 are already attached to a PPA by the time the open completes.

The DL_DETACH_REQ primitive requests to detach the PPA from the Stream. This is only allowed if the Stream was opened using Style 2.

The DL_BIND_REQ and DL_UNBIND_REQ primitives bind and unbind a DLSAP to the Stream. The PPA associated with a Stream will have completed initialization before completion of the processing of the DL_BIND_REQ on that Stream. Binding multiple Streams to the same SAP is allowed; each such Stream receives a copy of any packets received for that SAP.

The DL_ENABMULTI_REQ and DL_DISABMULTI_REQ primitives enable and disable reception of individual multicast group addresses. An application or other DLS user is permitted to create or modify a set of multicast addresses on a per-Stream basis by iterative use of these primitives. The Stream must be attached to a PPA for these primitives to be accepted.

The DL_PROMISCON_REQ and DL_PROMISCOFF_REQ primitives enable and disable promiscuous mode on a per-Stream basis, either at a physical level or at the SAP level. The DL Provider routes all received messages on the media to the DLS user until either a DL_DETACH_REQ or a DL_PROMISCOFF_REQ is received or the Stream is closed. It is possible to specify physical level promiscuous reception of all packets on the medium or of multicast packets only.

Note - The Stream must be attached to a PPA for these promiscuous mode primitives to be accepted.

The DL_UNITDATA_REQ primitive is used to send data in a connectionless transfer. Because this is an unacknowledged service, there is no guarantee of delivery. The message consists of one M_PROTO message block followed by one or more M_DATA blocks containing at least one byte of data.

The DL_UNITDATA_IND type is used when a packet is received and is to be passed upstream. The packet is put into an M_PROTO message with the primitive set to DL_UNITDATA_IND.

The DL_PHYS_ADDR_REQ primitive requests the MAC address currently associated with the PPA attached to the Stream. The address is returned by the DL_PHYS_ADDR_ACK primitive. When using style 2, this primitive is only valid following a successful DL_ATTACH_REQ.

The DL_SET_PHYS_ADDR_REQ primitive changes the MAC address currently associated with the PPA attached to the Stream. This primitive affects all other current and future Streams attached to this device. Once changed, all Streams currently or subsequently opened and attached to this device will obtain this new physical address. The new physical address remains in effect until this primitive is used to change the physical address again or the driver is reloaded.

Note - The superuser is allowed to change the physical address of a PPA while other Streams are bound to the same PPA.

The DL_GET_STATISTICS_REQ primitive requests a DL_GET_STATISTICS_ACK response containing statistics information associated with the PPA attached to the Stream. Style 2 Streams must be attached to a particular PPA using DL_ATTACH_REQ before this primitive can succeed.

Implemented ioctl Functions

GLD implements the ioctl *ioc_cmd* function described below. If GLD receives an ioctl command that it does not recognize, it passes it to the device-specific driver's gldm_ioctl( ) routine, as described in gld(9E).

The DLIOCRAW ioctl function is used by some DLPI applications, most notably the snoop(1M) command. The DLIOCRAW command puts the Stream into a raw mode, which causes the driver to pass full MAC-level incoming packets upstream in M_DATA messages instead of transforming them into the DL_UNITDATA_IND form that is normally used for reporting incoming packets. Packet SAP filtering is still performed on Streams that are in raw mode. If a Stream user wants to receive all incoming packets, it must also select the appropriate promiscuous mode or modes. After successfully selecting raw mode, the application is also allowed to send fully formatted packets to the driver as M_DATA messages for transmission. DLIOCRAW takes no arguments. Once enabled, the Stream remains in this mode until closed.

GLD Driver Requirements

GLD-based drivers must include the header file <sys/gld.h>.

GLD-based drivers must also include the following declaration:

```
char _depends_on[] = "misc/gld";
```

GLD implements the open(9E) and close(9E) functions and the required STREAMS put(9E) and srv(9E) functions on behalf of the device-specific driver. GLD also implements the getinfo(9E) function for the driver.

The mi_idname element of the module_info(9S) structure is a string specifying the name of the driver. This must exactly match the name of the driver module as it exists in the file system.

The read-side qinit(9S) structure should specify the following elements:

qi_putp	NULL
qi_srvp	gld_rsrv
qi_qopen	gld_open
qi_qclose	gld_close

The write-side qinit(9S) structure should specify these elements:

qi_putp	gld_wput
qi_srvp	gld_wsrv

qi_qopen	NULL
qi_qclose	NULL

The devo_getinfo element of the dev_ops(9S) structure should specify gld_getinfo as the getinfo(9E) routine.

The driver's attach(9E) function does all the work of associating the hardware-specific device driver with the GLD facility and preparing the device and driver for use.

The attach(9E) function allocates a gld_mac_info(9S) (''macinfo'') structure using gld_mac_alloc( ). The driver usually needs to save more information per device than is defined in the macinfo structure. It should allocate the additional required data structure and save a pointer to it in the gldm_private member of the gld_mac_info(9S) structure.

The attach(9E) routine must initialize the macinfo structure as described in gld_mac_info(9S) and then call gld_register( ) to link the driver with the GLD module. The driver should map registers if necessary and be fully initialized and prepared to accept interrupts before calling gld_register( ). The attach(9E) function should add interrupts but not enable the device to generate them. The driver should reset the hardware before calling gld_register( ) to ensure it is quiescent. The device must not be started or put into a state where it might generate an interrupt before gld_register( ) is called. That will be done later when GLD calls the driver's gldm_start( ) entry point, described in gld(9E). After gld_register( ) succeeds, the gld(9E) entry points might be called by GLD at any time.

The attach(9E) routine should return DDI_SUCCESS if gld_register( ) succeeds. If gld_register( ) fails, it returns DDI_FAILURE, and the attach(9E) routine should deallocate any resources it allocated before calling gld_register( ) and then also return DDI_FAILURE. Under no circumstances should a failed macinfo structure be reused; it should be deallocated using gld_mac_free( ).

The detach(9E) function should attempt to unregister the driver from GLD. This is done by calling gld_unregister( ) described in gld(9F). The detach(9E) routine can get a pointer to the needed gld_mac_info(9S) structure from the device's private data using ddi_get_driver_private(9F). gld_unregister( ) checks certain conditions that could require that the driver not be detached. If the checks fail, gld_unregister( ) returns DDI_FAILURE, in which case the driver's detach(9E) routine must leave the device operational and return DDI_FAILURE.

If the checks succeed, gld_unregister( ) ensures that the device interrupts are stopped (calling the driver's gldm_stop( ) routine if necessary), unlinks the driver from the GLD framework, and returns DDI_SUCCESS. In this case, the detach(9E) routine should remove interrupts, deallocate any data structures allocated in the attach(9E) routine (using gld_mac_free( ) to deallocate the macinfo structure), and return DDI_SUCCESS. It is important to remove the interrupt *before* calling gld_mac_free( ).

Network Statistics

Solaris network drivers must implement statistics variables. GLD itself tallies some network statistics, but other statistics must be counted by each GLD-based driver. GLD provides support for GLD-based drivers to report a standard set of network driver statistics. Statistics are reported by GLD using the `kstat`(7D) and `kstat`(9S) mechanisms. The `DL_GET_STATISTICS_REQ` DLPI command can also be used to retrieve the current statistics counters. All statistics are maintained as unsigned, and all are 32 bits unless otherwise noted.

GLD maintains and reports the following statistics.

`rbytes64`	Total bytes successfully received on the interface (64 bits).
`rbytes`	Total bytes successfully received on the interface.
`obytes64`	Total bytes requested to be transmitted on the interface (64 bits).
`obytes`	Total bytes requested to be transmitted on the interface.
`ipackets64`	Total packets successfully received on the interface (64 bits).
`ipackets`	Total packets successfully received on the interface.
`opackets64`	Total packets requested to be transmitted on the interface (64 bits).
`opackets`	Total packets requested to be transmitted on the interface.
`multircv`	Multicast packets successfully received, including group and functional addresses (`long`).
`multixmt`	Multicast packets requested to be transmitted, including group and functional addresses (`long`).
`brdcstrcv`	Broadcast packets successfully received (`long`).
`brdcstxmt`	Broadcast packets requested to be transmitted (`long`).
`unknowns`	Valid received packets not accepted by any Stream (`long`).

`noxmtbuf` Packets discarded on output because transmit buffer was busy, or no buffer could be allocated for transmit (`long`).

`blocked` Number of times a received packet could not be put up a Stream because the queue was flow-controlled (`long`).

`xmtretry` Times transmit was retried after having been delayed due to lack of resources (`long`).

`promisc` Current ‘‘promiscuous’’ state of the interface (string).

The device-dependent driver counts the following statistics, keeping track of them in a private per-instance structure. When GLD is asked to report statistics, it calls the driver’s `gldm_get_stats( )` entry point, as described in `gld`(9E), to update the device-specific statistics in the `gld_stats`(9S) structure. GLD then reports the updated statistics using the named statistics variables shown below.

`ifspeed` Current estimated bandwidth of the interface in bits per second (64 bits).

`media` Current media type in use by the device (string).

`intr` Times interrupt handler was called and claimed the interrupt (`long`).

`norcvbuf` Number of times a valid incoming packet was known to have been discarded because no buffer could be allocated for receive (`long`).

`ierrors` Total packets received that could not be processed because they contained errors (`long`).

`oerrors` Total packets that were not successfully transmitted because of errors (`long`).

`missed` Packets known to have been dropped by the hardware on receive (`long`).

`uflo` Times FIFO underflowed on transmit (`long`).

`oflo` Times receiver overflowed during receive (`long`).

The following group of statistics applies to networks of type `DL_ETHER`. These statistics are maintained by device-specific drivers of that type, as shown previously.

`align_errors`
Packets received with framing errors (not an integral number of octets) (`long`).

`fcs_errors`
Packets received with CRC errors (`long`).

`duplex`
Current duplex mode of the interface (string).

`carrier_errors`
Number of times carrier was lost or never detected on a transmission attempt (`long`).

`collisions`
Ethernet collisions during transmit (`long`).

`ex_collisions`
Frames where excess collisions occurred on transmit, causing transmit failure (`long`).

`tx_late_collisions`
Number of times a transmit collision occurred late (after 512 bit times) (`long`).

`defer_xmts`
Packets without collisions where first transmit attempt was delayed because the medium was busy (`long`).

`first_collisions`
Packets successfully transmitted with exactly one collision.

`multi_collisions`
Packets successfully transmitted with multiple collisions.

`sqe_errors`
Number of times SQE test error was reported.

`macxmt_errors`
Packets encountering transmit MAC failures, except carrier and collision failures.

`macrcv_errors`
Packets received with MAC errors, except `align_errors`, `fcs_errors`, and `toolong_errors`.

`toolong_errors`
Packets received larger than the maximum permitted length.

`runt_errors`
Packets received smaller than the minimum permitted length (`long`).

The following group of statistics applies to networks of type `DL_TPR`; these are maintained by device-specific drivers of that type, as shown above.

`line_errors`
Packets received with non-data bits or FCS errors.

`burst_errors` — Number of times an absence of transitions for five half-bit timers was detected.

`signal_losses` — Number of times loss of signal condition on the ring was detected.

`ace_errors` — Number of times an AMP or SMP frame, in which A is equal to C is equal to 0, was followed by another such SMP frame without an intervening AMP frame.

`internal_errors` — Number of times the station recognized an internal error.

`lost_frame_errors` — Number of times the TRR timer expired during transmit.

`frame_copied_errors` — Number of times a frame addressed to this station was received with the FS field ‘A’ bit set to `1`.

`token_errors` — Number of times the station acting as the active monitor recognized an error condition that needed a token transmitted.

`freq_errors` — Number of times the frequency of the incoming signal differed from the expected frequency.

The following group of statistics applies to networks of type `DL_FDDI`; these are maintained by device-specific drivers of that type, as shown above.

`mac_errors` — Frames detected in error by this MAC that had not been detected in error by another MAC.

`mac_lost_errors` — Frames received with format errors such that the frame was stripped.

`mac_tokens` — Number of tokens received (total of non-restricted and restricted).

`mac_tvx_expired` — Number of times that TVX has expired.

`mac_late` — Number of TRT expirations since this MAC was reset or a token was received.

mac_ring_ops

Number of times the ring has entered the ‘‘Ring Operational’’ state from the ‘‘Ring Not Operational’’ state.

Declarations and Data Structures

gld_mac_info Structure

The GLD MAC information (gld_mac_info) structure is the main data interface between the device-specific driver and GLD. It contains data required by GLD and a pointer to an optional additional driver-specific information structure.

Allocate the gld_mac_info structure using gld_mac_alloc( ) and deallocate it using gld_mac_free( ). Drivers must not make any assumptions about the length of this structure, which might vary in different releases of Solaris, GLD, or both. Structure members private to GLD, not documented here, should not be set or read by the device-specific driver.

The gld_mac_info(9S) structure contains the following fields.

```
caddr_t              gldm_private;              /* Driver private data */
int                  (*gldm_reset)();           /* Reset device */
int                  (*gldm_start)();           /* Start device */
int                  (*gldm_stop)();            /* Stop device */
int                  (*gldm_set_mac_addr)();    /* Set device phys addr */
int                  (*gldm_set_multicast)();   /* Set/delete multicast addr */
int                  (*gldm_set_promiscuous)(); /* Set/reset promiscuous mode */
int                  (*gldm_send)();            /* Transmit routine */
uint_t               (*gldm_intr)();            /* Interrupt handler */
int                  (*gldm_get_stats)();       /* Get device statistics */
int                  (*gldm_ioctl)();           /* Driver-specific ioctls */
char                 *gldm_ident;               /* Driver identity string */
uint32_t             gldm_type;                 /* Device type */
uint32_t             gldm_minpkt;               /* Minimum packet size */
                                                   /* accepted by driver */
uint32_t             gldm_maxpkt;               /* Maximum packet size */
                                                   /* accepted by driver */
uint32_t             gldm_addrlen;              /* Physical address length */
int32_t              gldm_saplen;               /* SAP length for DL_INFO_ACK */
unsigned char        *gldm_broadcast_addr;      /* Physical broadcast addr */
unsigned char        *gldm_vendor_addr;         /* Factory MAC address */
t_uscalar_t          gldm_ppa;                  /* Physical Point of */
                                                   /* Attachment (PPA) number */
dev_info_t           *gldm_devinfo;             /* Pointer to device’s */
                                                   /* dev_info node */
ddi_iblock_cookie_t  gldm_cookie;               /* Device’s interrupt */
                                                   /* block cookie */
```

These members of the gld_mac_info structure are visible to the device driver.

`gldm_private`

This structure member is private to the device-specific driver and is not used or modified by GLD. Conventionally this is used as a pointer to private data, pointing to a driver-defined and driver-allocated per-instance data structure.

The following group of structure members must be set by the driver before calling `gld_register( )`, and should not thereafter be modified by the driver. Because `gld_register( )` might use or cache the values of some of these structure members, changes made by the driver after calling `gld_register( )` might cause unpredictable results.

`gldm_reset`

Pointer to driver entry point; see `gld`(9E).

`gldm_start`

Pointer to driver entry point; see `gld`(9E).

`gldm_stop`

Pointer to driver entry point; see `gld`(9E).

`gldm_set_mac_addr`

Pointer to driver entry point; see `gld`(9E).

`gldm_set_multicast`

Pointer to driver entry point; see `gld`(9E).

`gldm_set_promiscuous`

Pointer to driver entry point; see `gld`(9E).

`gldm_send`

Pointer to driver entry point; see `gld`(9E).

`gldm_intr`

Pointer to driver entry point; see `gld`(9E).

`gldm_get_stats`

Pointer to driver entry point; see `gld`(9E).

`gldm_ioctl`

Pointer to driver entry point; is allowed to be `NULL`; see `gld`(9E).

`gldm_ident`

Pointer to a string containing a short description of the device. It is used to identify the device in system messages.

`gldm_type`

Type of device the driver handles. The values currently supported by GLD are `DL_ETHER` (ISO 8802-3 (IEEE 802.3) and Ethernet Bus), `DL_TPR` (IEEE 802.5 Token Passing Ring), and `DL_FDDI` (ISO 9314-2 Fibre Distributed Data Interface). This structure member must be correctly set for GLD to function properly.

`gldm_minpkt`

Minimum *Service Data Unit* size—the minimum packet size, not including the MAC header, that the device will transmit. This size is allowed to be zero if the device-specific driver handles any required padding.

`gldm_maxpkt`
Maximum *Service Data Unit* size — the maximum size of packet, not including the MAC header, that can be transmitted by the device. For Ethernet, this number is 1500.

`gldm_addrlen`
The length in bytes of physical addresses handled by the device. For Ethernet, Token Ring, and FDDI, the value of this structure member should be 6.

`gldm_saplen`
The length in bytes of the SAP address used by the driver. For GLD-based drivers, this should always be set to `-2`, to indicate that 2-byte SAP values are supported and that the SAP appears *after* the physical address in a DLSAP address. See ‘‘Message DL_INFO_ACK’’ in the DLPI specification for more details.

`gldm_broadcast_addr`
Pointer to an array of bytes of length `gldm_addrlen` containing the broadcast address to be used for transmit. The driver must provide space to hold the broadcast address, fill it in with the appropriate value, and set `gldm_broadcast_addr` to point to it. For Ethernet, Token Ring, and FDDI, the broadcast address is normally 0xFF-FF-FF-FF-FF-FF.

`gldm_vendor_addr`
Pointer to an array of bytes of length `gldm_addrlen` containing the vendor-provided network physical address of the device. The driver must provide space to hold the address, fill it in with information read from the device, and set `gldm_vendor_addr` to point to it.

`gldm_ppa`
PPA number for this instance of the device. Normally this should be set to the instance number, returned from `ddi_get_instance`(9F).

`gldm_devinfo`
Pointer to the `dev_info` node for this device.

`gldm_cookie`
Interrupt block cookie returned by `ddi_get_iblock_cookie`(9F),

ddi_add_intr(9F), ddi_get_soft_iblock_cookie(9F), or ddi_add_softintr(9F). This must correspond to the device's receive-interrupt, from which gld_recv( ) is called.

gld_stats Structure

The GLD statistics (gld_stats) structure is used to communicate statistics and state information from a GLD-based driver to GLD when returning from a driver's gldm_get_stats( ) routine, as discussed in gld(9E) and gld(7D). The members of this structure, filled in by the GLD-based driver, are used when GLD reports the statistics. In the tables below, the name of the statistics variable reported by GLD is noted in the comments. See gld(7D) for a more detailed description of the meaning of each statistic.

Drivers must not make any assumptions about the length of this structure, which might vary in different releases of Solaris, GLD, or both. Structure members private to GLD, not documented here, should not be set or read by the device-specific driver.

The following structure members are defined for all media types:

```
uint64_t        glds_speed;                         /* ifspeed */
uint32_t        glds_media;                         /* media */
uint32_t        glds_intr;                          /* intr */
uint32_t        glds_norcvbuf;                      /* norcvbuf */
uint32_t        glds_errrcv;                        /* ierrors */
uint32_t        glds_errxmt;                        /* oerrors */
uint32_t        glds_missed;                        /* missed */
uint32_t        glds_underflow;                     /* uflo */
uint32_t        glds_overflow;                      /* oflo */
```

The following structure members are defined for media type DL_ETHER:

```
uint32_t        glds_frame;                         /* align_errors */
uint32_t        glds_crc;                           /* fcs_errors */
uint32_t        glds_duplex;                        /* duplex */
uint32_t        glds_nocarrier;                     /* carrier_errors */
uint32_t        glds_collisions;                    /* collisions */
uint32_t        glds_excoll;                        /* ex_collisions */
uint32_t        glds_xmtlatecoll;                   /* tx_late_collisions */
uint32_t        glds_defer;                         /* defer_xmts */
uint32_t        glds_dot3_first_coll;               /* first_collisions */
uint32_t        glds_dot3_multi_coll;               /* multi_collisions */
uint32_t        glds_dot3_sqe_error;                /* sqe_errors */
uint32_t        glds_dot3_mac_xmt_error;            /* macxmt_errors */
uint32_t        glds_dot3_mac_rcv_error;            /* macrcv_errors */
uint32_t        glds_dot3_frame_too_long;           /* toolong_errors */
uint32_t        glds_short;                         /* runt_errors */
```

The following structure members are defined for media type DL_TPR:

```
uint32_t        glds_dot5_line_error              /* line_errors */
uint32_t        glds_dot5_burst_error             /* burst_errors */
uint32_t        glds_dot5_signal_loss             /* signal_losses */
uint32_t        glds_dot5_ace_error               /* ace_errors */
uint32_t        glds_dot5_internal_error          /* internal_errors */
uint32_t        glds_dot5_lost_frame_error        /* lost_frame_errors */
uint32_t        glds_dot5_frame_copied_error      /* frame_copied_errors */
uint32_t        glds_dot5_token_error             /* token_errors */
uint32_t        glds_dot5_freq_error              /* freq_errors */
```

The following structure members are defined for media type DL_FDDI:

```
uint32_t        glds_fddi_mac_error;              /* mac_errors */
uint32_t        glds_fddi_mac_lost;               /* mac_lost_errors */
uint32_t        glds_fddi_mac_token;              /* mac_tokens */
uint32_t        glds_fddi_mac_tvx_expired;        /* mac_tvx_expired */
uint32_t        glds_fddi_mac_late;               /* mac_late */
uint32_t        glds_fddi_mac_ring_op;            /* mac_ring_ops */
```

Most of the above statistics variables are counters denoting the number of times the particular event was observed. Exceptions are:

glds_speed

Estimate of the interface's current bandwidth in bits per second. For interfaces that do not vary in bandwidth or for those where no accurate estimation can be made, this object should contain the nominal bandwidth.

glds_media

Type of media (wiring) or connector used by the hardware. Currently supported media names include GLDM_AUI, GLDM_BNC, GLDM_TP, GLDM_10BT, GLDM_100BT, GLDM_100BTX, GLDM_100BT4, GLDM_RING4, GLDM_RING16, GLDM_FIBER, and GLDM_PHYMII. GLDM_UNKNOWN is also permitted.

glds_duplex

Current duplex state of the interface. Supported values are GLD_DUPLEX_HALF and GLD_DUPLEX_FULL. GLD_DUPLEX_UNKNOWN is also permitted.

Entry Point and Service Routines

Arguments Used by GLD Routines

macinfo Pointer to a gld_mac_info(9S) structure.

macaddr Pointer to the beginning of a character array containing a valid MAC address. The array will be of the length specified by the driver in the gldm_addrlen element of the gld_mac_info(9S) structure.

multicastaddr Pointer to the beginning of a character array containing a multicast, group, or functional address. The array will be of the length specified by the driver in the gldm_addrlen element of the gld_mac_info(9S) structure.

multiflag Flag indicating whether reception of the multicast address is to be enabled or disabled. This argument is specified as GLD_MULTI_ENABLE or GLD_MULTI_DISABLE.

promiscflag Flag indicating what type of promiscuous mode, if any, is to be enabled. This argument is specified as GLD_MAC_PROMISC_PHYS, GLD_MAC_PROMISC_MULTI, or GLD_MAC_PROMISC_NONE.

mp gld_ioctl( ) uses *mp* as a pointer to a STREAMS message block containing the ioctl to be executed. gld_send( ) uses it as a pointer to a STREAMS message block containing the packet to be transmitted. gld_recv( ) uses it as a pointer to a message block containing a received packet.

stats Pointer to a gld_stats(9S) structure to be filled in with the current values of statistics counters.

q Pointer to the queue(9S) structure to be used in the reply to the ioctl.

dip Pointer to the device's dev_info structure.

name Device interface name.

Entry Points

These entry points must be implemented by a device-specific network driver designed to interface with GLD.

As described in gld(7D), the main data structure for communication between the device-specific driver and the GLD module is the gld_mac_info(9S) structure. Some of the elements in that structure are function pointers to the entry points described here. The device-specific driver must, in its attach(9E) routine, initialize these function pointers before calling gld_register( ).

```
int prefix_reset(gld_mac_info_t * macinfo);
```

gldm_reset( ) resets the hardware to its initial state.

```
int prefix_start(gld_mac_info_t * macinfo);
```

gldm_start( ) enables the device to generate interrupts and prepares the driver to call gld_recv( ) for delivering received data packets to GLD.

```
int prefix_stop(gld_mac_info_t * macinfo);
```

gldm_stop( ) disables the device from generating any interrupts and stops the driver from calling gld_recv( ) for delivering data packets to GLD. GLD depends on the gldm_stop( ) routine to ensure that the device will no longer interrupt, and it must do so without fail. This function should always return GLD_SUCCESS.

```
int prefix_set_mac_addr(gld_mac_info_t * macinfo, unsigned char * macaddr);
```

gldm_set_mac_addr( ) sets the physical address that the hardware is to use for receiving data. This function should program the device to the passed MAC address *macaddr*. If sufficient resources are currently not available to carry out the request, return GLD_NORESOURCES. Return GLD_NOTSUPPORTED to indicate that the requested function is not supported.

```
int prefix_set_multicast(gld_mac_info_t * macinfo, unsigned char * multicastaddr,
    int multiflag);
```

gldm_set_multicast( ) enables and disables device-level reception of specific multicast addresses. If the third argument *multiflag* is set to GLD_MULTI_ENABLE, then the function sets the interface to receive packets with the multicast address pointed to by the second argument. If *multiflag* is set to GLD_MULTI_DISABLE, the driver is allowed to disable reception of the specified multicast address.

This function is called whenever GLD wants to enable or disable reception of a multicast, group, or functional address. GLD makes no assumptions about how the device does multicast support and calls this function to enable or disable a specific multicast address. Some devices might use a hash algorithm and a bitmask to enable collections of multicast addresses; this procedure is allowed, and GLD filters out any superfluous packets. If disabling an address could result in disabling more than one address at the device level, it is the responsibility of the device driver to keep whatever information it needs in order to avoid disabling an address that GLD has enabled but not disabled.

gldm_set_multicast( ) will not be called to enable a particular multicast address that is already enabled, nor will it be called to disable an address that is not currently enabled. GLD keeps track of multiple requests for the same multicast address and only calls the driver's entry point when the first request to enable, or the last request to disable, a particular multicast address is made. If sufficient resources are currently not available to carry out the request, return GLD_NORESOURCES. Return GLD_NOTSUPPORTED to indicate that the requested function is not supported.

```
int prefix_set_promiscuous(gld_mac_info_t * macinfo, int promiscflag);
```

gldm_set_promiscuous( ) enables and disables promiscuous mode. This function is called whenever GLD wants to enable or disable the reception of all packets on the medium, or of all multicast packets on the medium. If the second argument *promiscflag* is set to the value of GLD_MAC_PROMISC_PHYS, then the function enables physical-level promiscuous mode, resulting in the reception of all packets on the medium. If *promiscflag* is set to GLD_MAC_PROMISC_MULTI, then reception of all multicast packets will be enabled. If *promiscflag* is set to GLD_MAC_PROMISC_NONE, then promiscuous mode is disabled.

In the case of a request for promiscuous multicast mode, drivers for devices that have no multicast-only promiscuous mode must set the device to physical promiscuous mode to ensure that all multicast packets are received. In this case the routine should return GLD_SUCCESS. The GLD software filters out any superfluous packets. If sufficient resources are currently not available to carry out the request, return GLD_NORESOURCES. Return GLD_NOTSUPPORTED to indicate that the requested function is not supported.

For forward compatibility, gldm_set_promiscuous( ) routines should treat any unrecognized values for *promiscflag* as though they were GLD_MAC_PROMISC_PHYS.

```
int prefix_send(gld_mac_info_t * macinfo, mblk_t * mp);
```

gldm_send( ) queues a packet to the device for transmission. This routine is passed a STREAMS message containing the packet to be sent. The message might include multiple message blocks, and the send routine must traverse all the message blocks in the message to access the entire packet to be sent. The driver should be prepared to handle and skip over any zero-length message continuation blocks in the chain. The driver should check that the packet does not exceed the maximum allowable packet size, and it must pad the packet, if necessary, to the minimum allowable packet size. If the send routine successfully transmits or queues the packet, it should return GLD_SUCCESS.

The send routine should return GLD_NORESOURCES if it cannot immediately accept the packet for transmission; in this case GLD will retry it later. If gldm_send( ) ever returns GLD_NORESOURCES, the driver must, at a later time when resources have become available, call gld_sched( ). This call to gld_sched( ) informs GLD that it should retry packets that the driver previously failed to queue for transmission. (If the driver's gldm_stop( ) routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send( ) routine. However, extra calls to gld_sched( ) will not cause incorrect operation.)

If the driver's send routine returns GLD_SUCCESS, then the driver is responsible for freeing the message when the driver and the hardware no longer need it. If the send routine copied the message into the device, or into a private buffer, then the send routine is permitted to free the message after the copy is made. If the hardware uses DMA to read the data directly out of the message data blocks, then the driver must not free the message until the hardware has completed reading the data. In this case the driver will probably free the message in the interrupt routine, or in a buffer reclaim operation at the beginning of a future send operation. If the send routine returns anything other than GLD_SUCCESS, then the driver must not free the message. Return GLD_NOLINK if gldm_send( ) is called when there is no physical connection to the network or link partner.

```
int prefix_intr(gld_mac_info_t * macinfo);
```

gldm_intr( ) is called when the device might have interrupted. Because it is possible to share interrupts with other devices, the driver must check the device status to determine whether it actually caused an interrupt. If the device that the driver controls did not cause the interrupt, then this routine must return DDI_INTR_UNCLAIMED. Otherwise, it must service the interrupt and should return DDI_INTR_CLAIMED. If the interrupt was caused by successful receipt of a packet, this routine should put the received packet into a STREAMS message of type M_DATA and pass that message to gld_recv( ).

gld_recv( ) will pass the inbound packet upstream to the appropriate next layer of the network protocol stack. It is important to correctly set the b_rptr and b_wptr members of the STREAMS message before calling gld_recv( ).

The driver should avoid holding mutex or other locks during the call to gld_recv( ). In particular, locks that could be taken by a transmit thread must not be held during a call to gld_recv( ): the interrupt thread that calls gld_recv( ) will in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send( ) routine. If the gldm_send( ) routine were to try to acquire a mutex being held by the gldm_intr( ) routine at the time it calls gld_recv( ), this would result in a panic due to recursive mutex entry. If other driver entry points attempt to acquire a mutex that the driver holds across a call to gld_recv( ), deadlock can result.

The interrupt code should increment statistics counters for any errors. This includes failure to allocate a buffer needed for the received data and any hardware-specific errors, such as CRC errors or framing errors.

```
int prefix_get_stats(gld_mac_info_t * macinfo, struct gld_stats * stats);
```

gldm_get_stats( ) gathers statistics from the hardware, driver private counters, or both, and updates the gld_stats(9S) structure pointed to by *stats*. This routine is called by GLD when it gets a request for statistics, and provides the mechanism by which GLD acquires device-dependent statistics from the driver before composing its reply to the statistics request. See gld_stats(9S) and gld(7D) for a description of the defined statistics counters.

```
int prefix_ioctl(gld_mac_info_t * macinfo, queue_t * q, mblk_t * mp);
```

gldm_ioctl( ) implements any device-specific ioctl commands. This element is allowed to be NULL if the driver does not implement any ioctl functions. The driver is responsible for converting the message block into an ioctl reply message and calling the qreply(9F) function before returning GLD_SUCCESS. This function should always return GLD_SUCCESS; any errors the driver might want to report should be returned by the message passed to qreply(9F). If the gldm_ioctl element is specified as NULL, GLD returns a message of type M_IOCNAK with an error of EINVAL.

Return Values

In addition to the return values described above, and subject to the restrictions above, it is permitted for some of the GLD entry point functions to return these values:

GLD_BADARG — If the function detected an unsuitable argument, for example, a bad multicast address, a bad MAC address, or a bad packet or packet length

GLD_FAILURE — On hardware failure

GLD_SUCCESS — On success

Service Routines

```
gld_mac_info_t * gld_mac_alloc(dev_info_t * dip);
```

gld_mac_alloc( ) allocates a new gld_mac_info(9S) structure and returns a pointer to it. Some of the GLD-private elements of the structure might be initialized before gld_mac_alloc( ) returns; all other elements are initialized to zero. The device driver must initialize some structure members, as described in gld_mac_info(9S), before passing the pointer to the mac_info structure to gld_register( ).

```
void gld_mac_free(gld_mac_info_t * macinfo);
```

gld_mac_free( ) frees a gld_mac_info(9S) structure previously allocated by gld_mac_alloc( ).

```
int gld_register(dev_info_t * dip, char * name, gld_mac_info_t * macinfo);
```

gld_register( ) is called from the device driver's attach(9E) routine and is used to link the GLD-based device driver with the GLD framework. Before calling gld_register( ), the device driver's attach(9E) routine must first use gld_mac_alloc( ) to allocate a gld_mac_info(9S) structure, and initialize several of its structure elements. See gld_mac_info(9S) for more information. A successful call to gld_register( ) performs the following actions:

- Links the device-specific driver with the GLD system
- Sets the device-specific driver's private data pointer (using ddi_set_driver_private(9F)) to point to the macinfo structure
- Creates the minor device node
- Returns DDI_SUCCESS

The device interface name passed to gld_register( ) must exactly match the name of the driver module as it exists in the file system.

The driver's attach(9E) routine should return DDI_SUCCESS if gld_register( ) succeeds. If gld_register( ) does not return DDI_SUCCESS, the attach(9E) routine should deallocate any resources it allocated before calling gld_register( ), and then return DDI_FAILURE.

```
int gld_unregister(gld_mac_info_t * macinfo);
```

gld_unregister( ) is called by the device driver's detach(9E) function, and if successful, performs the following tasks:

- Ensures that the device's interrupts are stopped, calling the driver's gldm_stop( ) routine if necessary
- Removes the minor device node
- Unlinks the device-specific driver from the GLD system
- Returns DDI_SUCCESS

If gld_unregister( ) returns DDI_SUCCESS, the detach(9E) routine should deallocate any data structures allocated in the attach(9E) routine, using gld_mac_free( ) to deallocate the macinfo structure, and return DDI_SUCCESS. If gld_unregister( ) does not return DDI_SUCCESS, the driver's detach(9E) routine must leave the device operational and return DDI_FAILURE.

```
void gld_recv(gld_mac_info_t * macinfo, mblk_t * mp);
```

gld_recv( ) is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv( ) determines which STREAMS queues, if any, should receive a copy of the packet, duplicating it if necessary. It then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate Streams.

The driver should avoid holding mutex or other locks during the call to gld_recv( ). In particular, locks that could be taken by a transmit thread must not be held during a call to gld_recv( ): the interrupt thread that calls gld_recv( ) will in some cases carry out processing that includes sending an outgoing packet, resulting in a call to the driver's gldm_send( ) routine. If the gldm_send( ) routine were to try to acquire a mutex being held by the gldm_intr( ) routine at the time it calls gld_recv( ), this would result in a panic caused by a recursive mutex entry. If other driver entry points attempt to acquire a mutex that the driver holds across a call to gld_recv( ), deadlock can result.

```
void gld_sched(gld_mac_info_t * macinfo);
```

gld_sched( ) is called by the device driver to reschedule stalled outbound packets. Whenever the driver's gldm_send( ) routine has returned GLD_NORESOURCES, the driver must later call gld_sched( ) to inform the GLD framework that it should retry the packets that previously could not be sent. gld_sched( ) should be called as soon as possible after resources are again available, to ensure that GLD resumes passing outbound packets to the driver's gldm_send( ) routine in a timely way. (If the driver's gldm_stop( ) routine is called, the driver is absolved from this obligation until it later again returns GLD_NORESOURCES from its gldm_send( ) routine; however, extra calls to gld_sched( ) will not cause incorrect operation.)

```
uint_t gld_intr(caddr_t);
```

gld_intr( ) is GLD's main interrupt handler. Normally, gld_intr( ) is specified as the interrupt routine in the device driver's call to ddi_add_intr(9F). The argument to the interrupt handler (specified as *int_handler_arg* in the call to ddi_add_intr(9F)) must be a pointer to the gld_mac_info(9S) structure. gld_intr( ) will, when appropriate, call the device driver's gldm_intr( ) function, passing that pointer to the gld_mac_info(9S) structure. However, if the driver uses a high-level interrupt, it must provide its own high-level interrupt handler and trigger a soft interrupt from within that. In this case, gld_intr( ) would normally be specified as the soft interrupt handler in the call to ddi_add_softintr( ). gld_intr( ) will return a value appropriate for an interrupt handler.

CHAPTER **6**

Language Support Topics

This section provides instructions for language support in the Solaris environment. This section contains these chapters.

"Additional Partial Locales for European Solaris Software" on page 63	Lists new partial locales
Chapter 8	Describes how to customize the behavior of `mp` print filter and provides troubleshooting information

CHAPTER 7

Additional Partial Locales

Additional partial locales are new in the Solaris 8 10/00 release and the Eastern European locale is updated for Solaris 8 4/01 release. For more information on language support in Solaris software, see *International Language Environments Guide*.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Additional Partial Locales for European Solaris Software

The 10/00 release features are the addition of UTF-8 locales for Russian and Polish and two new locales for Catalan. The locale names are as follows.

- ru_RU.UTF-8
- pl_PL.UTF-8
- ca_ES.ISO8859–1
- ca_ES.ISO8859–15

The 4/01 release includes the addition of UTF-8 locales for Turkish and other locales.

The additional locales are partial locales because of the lack of language support (translation of messages and GUI).

Localization in the Base and Multilingual Solaris Product

Central Europe

TABLE 7–1 Central Europe

Locale	User Interface	Territory	Codeset	Language Support
`cs_CZ.ISO8859-2`	English	Czech Republic	ISO8859-2	Czech (Czech Republic)
`de_AT.ISO8859-1`	German	Austria	ISO8859-1	German (Austria)
`de_AT.ISO8859-15`	German	Austria	ISO8859-15	German (Austria, ISO8859-15 - Euro)
`de_CH.ISO8859-1`	German	Switzerland	ISO8859-1	German (Switzerland)
`de_DE.UTF-8`	German	Germany	UTF-8	German (Germany, Unicode 3.0)
`de_DE.ISO8859-1`	German	Germany	ISO8859-1	German (Germany)
`de_DE.ISO8859-15`	German	Germany	ISO8859-15	German (Germany, ISO8859-15 - Euro)
`fr_CH.ISO8859-1`	French	Switzerland	ISO8859-1	French (Switzerland)
`hu_HU.ISO8859-2`	English	Hungary	ISO8859-2	Hungarian (Hungary)
`pl_PL.ISO8859-2`	English	Poland	ISO8859-2	Polish (Poland)
`pl_PL.UTF-8`	English	Poland	UTF-8	Polish (Poland, Unicode 3.0)
`sk_SK.ISO8859-2`	English	Slovakia	ISO8859-2	Slovak (Slovakia)

Eastern European Additions

TABLE 7–2 Eastern Europe

Locale	User Interface	Territory	Codeset	Language Support
bg_BG.ISO8859-5	English	Bulgaria	ISO8859-5	Bulgarian (Bulgaria)
et_EE.ISO8859-15	English	Estonia	ISO8859-15	Estonian (Estonia)
hr_HR.ISO8859-2	English	Croatia	ISO8859-2	Croatian (Croatia)
lt_LT.ISO8859-13	English	Lithuania	ISO8859-13	Lithuanian (Lithuania)
lv_LV.ISO8859-13	English	Latvia	ISO8859-13	Latvian (Latvia)
mk_MK.ISO8859-5	English	Macedonia	ISO8859-5	Macedonian (Macedonia)
ro_RO.ISO8859-2	English	Romania	ISO8859-2	Romanian (Romania)
ru_RU.KOI8-R	English	Russia	KOI8-R	Russian (Russia, KOI8-R)
ru_RU.ANSI1251	English	Russia	ansi-1251	Russian (Russia, ANSI 1251)
ru_RU.ISO8859-5	English	Russia	ISO8859-5	Russian (Russia)
ru_RU.UTF-8 (Unicode 3.0)	English	Russia	UTF-8	Russian (Russia, Unicode 3.0)
sh_BA.ISO8859-2@bosnia	English	Bosnia	ISO8859-2	Bosnian (Bosnia)
sl_SI.ISO8859-2	English	Slovenia	ISO8859-2	Slovenian (Slovenia)
sq_AL.ISO8859-2	English	Albania	ISO8859-2	Albanian (Albania)
sr_YU.ISO8859-5	English	Serbia	ISO8859-5	Serbian (Serbia)
tr_TR.ISO8859-9	English	Turkey	ISO8859-9	Turkish (Turkey)
tr_TR.UTF-8	English	Turkey	UTF-8	Turkish (Turkey, Unicode 3.0)

South Europe

TABLE 7–3 South Europe

Locale	User Interface	Territory	Codeset	Language Support
`ca_ES.ISO8859-1`	English	Spain	ISO8859-1	Catalan (Spain)
`ca_ES.ISO8859-15`	English	Spain	ISO8859-15	Catalan (Spain, ISO8859-15 - Euro)
`el_GR.ISO8859-7`	English	Greece	ISO8859-7	Greek (Greece)
`es_ES.ISO8859-1`	Spanish	Spain	ISO8859-1	Spanish (Spain)
`es_ES.ISO8859-15`	Spanish	Spain	ISO8859-15	Spanish (Spain, ISO8859-15 - Euro)
`es_ES.UTF-8`	Spanish	Spain	UTF-8	Spanish (Spain, Unicode 3.0)
`it_IT.ISO8859-1`	Italian	Italy	ISO8859-1	Italian (Italy)
`it_IT.ISO8859-15`	Italian	Italy	ISO8859-15	Italian (Italy, ISO8859-15 - Euro)
`it_IT.UTF-8`	Italian	Italy	UTF-8	Italian (Italy, Unicode 3.0)
`pt_PT.ISO8859-1`	English	Portugal	ISO8859-1	Portuguese (Portugal)
`pt_PT.ISO8859-15`	English	Portugal	ISO8859-15	Portuguese (Portugal, ISO8859-15 - Euro)

European Localization

Solaris 8 software supports the euro currency. Local currency symbols are still available for backward compatibility.

TABLE 7–4 User Locales to Support the Euro Currency

Region	Locale Name	ISO Codeset
Austria	`de_AT.ISO8859-15`	8859-15
Belgium (French)	`fr_BE.ISO8859-15`	8859-15

TABLE 7–4 User Locales to Support the Euro Currency *(continued)*

Region	**Locale Name**	**ISO Codeset**
Belgium (Dutch)	`nl_BE.ISO8859-15`	8859-15
Denmark	`da_DK.ISO8859-15`	8859-15
Finland	`fi_FI.ISO8859-15`	8859-15
France	`fr_FR.ISO8859-15`	8859-15
Germany	`de_DE.ISO8859-15`	8859-15
Ireland	`en_IE.ISO8859-15`	8859-15
Italy	`it_IT.ISO8859-15`	8859-15
Netherlands	`nl_NL.ISO8859-15`	8859-15
Portugal	`pt_PT.ISO8859-15`	8859-15
Spain	`ca_ES.ISO8859-15`	8859–15
Spain	`es_ES.ISO8859-15`	8859-15
Sweden	`sv_SE.ISO8859-15`	8859-15
Great Britain	`en_GB.ISO8859-15`	8859-15
U.S.A.	`en_US.ISO8859-15`	8859-15

CHAPTER **8**

Print Filter Enhancement mp(1)

The mp print filter is new in the Solaris 8 1 4/01 release. For more information on language support in Solaris software, see *International Language Environments Guide*.

Note - For the most current man pages, use the man command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

mp(1) Print Filter Enhancement Overview

The mp(1) print filter is enhanced with the ability to work as an X Print Server client in the 2.5.11 version. This section describes how to customize the behavior of mp(1).

Customization can be divided into

- Changes to enable mp(1) to print the correct output
- Changes to enable the mp(1) output to appear differently

The following information outlines what to do if mp(1) is not printing the correct output. The new enhancements done to the mp version 2.5.11 permit it to work as an X Print Server client.

If a correctly configured X Print Server is running, then mp(1) prints correctly in the target locale without changing any of the configuration files described below. If you do not have a working X Print Server, localize /usr/lib/lp/locale/$LANG/mp/mp.conf as described in the next section. mp.conf file effectively achieves the same functionality as the prolog.ps file available in some locales under /usr/openwin/lib/locale/$LANG/print/. The prolog.ps file can also be customized.

/usr/lib/lp/locale/$LANG/mp/ directory can also contain the prolog.ps file. This file is provided for backward compatibility purposes only, and users are encouraged to use either the direct X Print Server client mode, when giving –P or –D options, or by configuring the mp.conf file. For changing the appearance of the output, customize the existing prolog files.

The following guidelines describe how the choice of a configuration file is made by mp(1)

- If '–D <*target printer name*>' or '– P <*target printer name*>' is given, then .xpr prolong files are read. No other file is read.
- If '–D' or '–P' is not given in the command line, and if /usr/openwin/lib/locale/$LANG/print/prolog.ps exists, then it is prepended to the output. Depending on the print style, one of the .ps prolog page layout files is also prepended to the output. If MP_LANG is defined in the environment, it is used instead of LANG. LANG is a locale environment variable.
- If the prolog.ps file is not in/usr/openwin/lib/locale/$LANG/print/, then that file is searched in /usr/lib/lp/locale/$LANG/mp/. $MP_LANG is substituted for $LANG if it is defined. If prolog.ps file is not in that directory, and the mp.conf file is present, then the mp.conf file is scanned. Depending on the print style, one of the ps prolog layout files is also prepended to the output. If neither the prolog.ps nor the mp.conf file is present, then mp prints ASCII only as in C locale.

The following guidelines describe how a PostScript (.ps) or X Print client page (.xpr) formatting file is selected.

- If –D <*target printer name*> or –P <*target printer name*> is given, then the .xpr prolog files in /usr/lib/lp/locale/C/mp are used. To set up your own directory for output prolog files, set the MP_PROLOGUE variable to point to that directory.
- When printing in the normal output mode, use the ps files in the/usr/lib/lp/locale/C/mp directory. The MP_PROLOGUE variable is also applicable here.

Localization of the Configuration File

The idea behind configuration files is to capture the flexibility they provide when adding or changing font entries, or font group entries, as the need arises.

The system default configuration file that is used is /usr/lib/lp/locale/$LANG/mp/mp.conf where $LANG is a locale environment variable in the locale in which printing occurs. Users can have a personal configuration file that can be specified by the '–u <*config.file path*>' option.

The mp.conf file is used mainly for mapping the intermediate code points in a locale to the presentation forms in the encoding of the font that is used to print that code point.

Note - A ligature or variant glyph that has been encoded as a character for compatibility is called presentation forms.

Intermediate code points can either be Wide Characters or output of the Portable Layout Services (PLS) layer. Complex Text Layout printing requires that the intermediate code points be PLS output. The default intermediate code generated by mp(1) is PLS output.

Font formats currently supported are Portable Compiled Format (PCF), TrueType, and Type1 format. Both system-resident and printer-resident Type1 fonts are supported.

mp.conf configuration file localization is for configuring mp to print according to the needs of a specific locale. The following describes the format and contents of the mp.conf configuration file for mp(1).

- Lines must begin with a valid keyword (directive).
- Arguments to a keyword must appear on the same line as the keyword.
- Lines that begin with a "#" character are treated as comments until the end of the line.
- Numeric arguments that begin with 0x are interpreted as a hexadecimal number.

The following list describes different sections in the mp.conf file.

- Font Aliasing
- Font Group definition
- Mapping from the intermediate code ranges to the Font Group in a locale
- Associating each font with the shared object that maps the intermediate code points to the presentation forms in the font encoding

Font Aliasing

This section is used to define alias names for each font used for printing. Each line in this section is of the form

```
keyword   font alias name    font type      font path
```

<keyword> The keyword for this section is FontNameAlias.

<font alias name> The usual convention for aliasing a font name is to specify the encoding/script name of the font followed by a letter that indicates whether the font is Roman, Bold, Italic, or BoldItalic (R, B, I or BI). For example `/usr/openwin/lib/X11/fonts/75dpi/courR18.pcf.Z`, since it is an iso88591 Roman font, can be given the alias name iso88591R.

<font type> Specify PCF for .pcf fonts, Type1 for Adobe Type1 fonts, and TrueType for truetype fonts. Only these three kinds of fonts can be configured in this `config.` file.

<font path> Give the absolute path name for the font files here. For type1 printer-resident fonts, just specify the font name. See the following example.

```
FontNameAlias    prnHelveticaR     Type1       Helvetica
```

Font Group Definition

You can combine same-type fonts to form a font group. The format of the font group is as follows.

- <keyword> for this section is FontGroup.
- <fontgroupname> is the group name for the fonts.
- <GroupType> is the font type. Create font groups for the same type of fonts only (PCF, Type1, TrueType).
- <Roman> is the Roman Font name in the font group.
- <Bold> is the Bold Font name in the font group.
- <Italic> is the Italic Font name in the font group.
- <BoldItalic> is the BoldItalic Font name in the font group.

Note - For creating a group, only a Roman font entry is required. The Bold, Italic, and BoldItalic fonts are optional. The different types of fonts are used to display the header lines for mail/news articles. If only the Roman font is defined, it is used in place of other fonts.

Mapping From the Intermediate Code Ranges to the Font Group in a Locale

- <keyword> for this section is MapCode2. Font for this section is MapCode2Font.
- <range_start> is a 4–byte hex value that starts with 0x, which indicates the start of the code range to map to one or more font group.
- <range_end> indicates the end of the code range to map. It can be a '-' in which case only a single intermediate code point is mapped to the target font.

- <group> is a Type1, PCF, or TrueType font group, with which the presentation forms are to be printed.

Associating Each Font With the Shared Object That Maps the Intermediate Code Points to the Presentation Forms in the Fonts Encoding

- <keyword> is CnvCode2Font.
- <font alias name> is the alias name defined for the font.
- <mapping function> takes in the intermediate code and returns presentation forms in fonts encoding, which is in turn used to get the glyph index, and draw the glyph.
- <file path having mapping function> is the .so file name that contains the mapping function. You can use the utility in dumpcs to find out the intermediate codeset for EUC locales.

Note - The Current TrueType Engine employed by mp(1) is capable of dealing only with format 4 and PlatformID 3 cmap. That is, you can only configure Microsoft .ttf files. Additionally, the character map encoding has to be Unicode or Symbol for the TrueType font engine to work correctly. Because most of the .ttf fonts in the Solaris environment are obeying these restrictions, you can map all TrueType fonts in Solaris software within the mp.conf file.

When you create a shared object for mapping a font that corresponds to an X Logical Fonts Description (XLFD) , consider the following. If you are mapping a pcf/type1 font, then create the shared object that maps from the intermediate code range to the encoding specified by XLFD. For example:

```
-monotype-arial-bold-r-normal-bitmap-10-100-75-75-p-54-iso8859-8
```

The corresponding pcf font is:

```
/usr/openwin/lib/locale/iso_8859_8/X11/fonts/75dpi/ariabd10.pcf.Z
```

This font is encoded in iso8859-8, so shared objects have to map between intermediate code and corresponding iso8859-8 code points.

But if a TrueType font with XLFD:

```
-monotype-arial-medium-r-normal--0-0-0-0-p-0-iso8859-8
```

has the corresponding font:

```
/usr/openwin/lib/locale/iso_8859_8/X11/fonts/TrueType/arial__h.ttf
```

In this situation, map between the intermediate code and Unicode, because the cmap encoding for the previous TrueType font is in Unicode. In the example of this TrueType font, if a sample intermediate code in the en_US.UTF-8 locale that corresponds to a Hebrew character (produced by the PLS layer) is 0xe50000e9, you need to consider the following. Because the font is Unicode encoded, design the function within the corresponding .so module in such a way that when you are passing 0xe50000e9, the output corresponds to presentation form in Unicode. The example here is 0x000005d9.

The function prototype for the <mapping function> should be:

```
unsigned int function(unsigned int inter_code_pt)
```

The following are optional keyword/value pairs that you can use in mp.conf:

```
PresentationForm        WC/PLSOutput
```

The default value is PLSOutput. If the user is specifying "WC", then the intermediate code points that are generated are Wide Characters. For CTL printing, this default value should be used.

If the locale is non-CTL locale and has the value for the keyword is PLSOutput, it is ignored and the mp(1) generates wide-character codes instead.

You can use the following optional keyword/value pairs if the locale supports CTL. These variables can assume any of the possible values given on the right side of the table.

Orientation	ORIENTATION_LTR/ ORIENTATION_RTL/ ORIENTATION_CONTEXTUAL	Default is ORIENTATION_LTR
Numerals	NUMERALS_NOMINAL/ NUMERALS_NATIONAL/ NUMERALS_CONTEXTUAL	Default is NUMERALS_NOMINAL
TextShaping	TEXT_SHAPED/ TEXT_NOMINAL/ TEXT_SHFORM1/ TEXT_SHFORM2/ TEXT_SHFORM3/ TEXT_SHFORM4	Default is TEXT_SHAPED

The following example illustrates the steps that you need to follow when you add a new PCF, TrueType, or Type1 printer-resident font to the configuration file.

Replace the font for displaying characters in the range 0x00000021 - 0x0000007f with a TrueType font instead of the currently configured PCF font.

Before adding a new font, look at various components in the configuration file that correspond to the currently configured font, as shown next.

```
FontNameAlias   iso88591R       PCF     /usr/openwin/lib/X11/fonts/75dpi/courR18PCF.Z
FontNameAlias   iso88591B       PCF     /usr/openwin/lib/X11/fonts/75dpi/courB18PCF.Z
.
.
```

```
.
FontGroup       iso88591         PCF       iso88591R iso88591B
.
.
.
MapCode2Font    0x00000020      0x0000007f      iso88591
.
.
.
CnvCode2Font    iso88591R _xuiso88591 /usr/lib/lp/locale/$LANG/mp/xuiso88591.so
CnvCode2Font    iso88591B _xuiso88591 /usr/lib/lp/locale/$LANG/mp/xuiso88591.so
```

Suppose you selected `/usr/openwin/lib/locale/ja/X11/fonts/TT/HG-MinchoL.ttf` as your candidate for doing the mapping in the `en_US.UTF-8` locale. Because this is a Unicode character-mapped TrueType font file, in the mapping function within the `.so` module you only need to have a function that directly returns the incoming ucs-2 code points.

```
unsigned short _ttfjis0201(unsigned short ucs2) {
                 return(ucs2);
         }
```

Save this in a `ttfjis0201.c` file. Create a shared object as follows.

```
cc -G -Kpic -o ttfjis0201.so ttfjis0201.c
```

But if you are mapping a PCF file, such as `/usr/openwin/lib/locale/ja/X11/fonts/75dpi/gotmrk20.pcf.Z`, then look in the `fonts.dir` file in the `/usr/openwin/lib/locale/ja/X11/fonts/75dpi/` directory. Become familiar with the encoding, corresponding to XLFD, which is:

```
-sun-gothic-medium-r-normal--22-200-75-75-c-100-jisx0201.1976-0
```

If `jisx0201` is the encoding, prepare a shared object that maps from ucs-2 to jisx0201. You need to obtain the mapping table for creating the `.so` module (if one is not already provided). For a Unicode locale, find the mappings from the many charsets to Unicode under `ftp.unicode.org/pub/MAPPINGS/`. Follow these mappings, in order to write a `xu2jis0201.c` file:

```
 unsigned short _xu2jis0201(unsigned short ucs2) {
                         if(ucs2 >= 0x20 && ucs2 <= 0x7d )
                                 return (ucs2);
                         if(ucs2==0x203e)
                                 return (0x7e);
                         if(ucs2 >= 0xff61 && ucs2 <= 0xff9f)
                                 return (ucs2 - 0xff60 + 0xa0);
                        return(0);
                 }
```

When you create a mapping file, include all the UCS-2 to jisx0201 cases.

```
cc  -G -o xu2jis0201.so xu2jis0201.c
```

This example creates a shared object file.

Add this font by adding the following lines to the corresponding sections of mp.conf. The following example shows how to add the TrueType font. The PCF font follows the same pattern except that you change the keyword to PCF instead of TrueType.

```
FontNameAlias   jis0201R TrueType
/home/fn/HG-Minchol.ttf
FontGroup     jus0201        TrueType    jis0201R
MapCode2Font    0x0020  0x007f   jis0201
CnvCode2Font    jis0201R    _ttfjis0201 <.so path>
```

# this line needs to be deleted from mp.conf before adding.

```
                        MapCode2Font    0x0020  0x007f  jis0201 CnvCode2Font
                        jis0201R  _ttfjis0201 <.so path>
```

where the .so path points to the xu2jis0201.so file.

Invoking mp(1) with the changed mp.conf file causes the range 0x0020-0x007f to be printed in the new font. Map the other Japanese character ranges too with the same .so file, for example, the range 0x0000FF61 0x0000FF9F.

To maintain backward compatibility, the /usr/openwin/lib/locale/$LANG/print/prolog.ps file, if it exists, is used to create output in the current locale, where $LANG is one of the locale components. In that situation, no configuration file mechanism is used.

Refer to /usr/lib/lp/locale/en_US.UTF-8/mp/mp.conf, which is a sample mp.conf file.

Customizing Existing prolog Files and Adding New prolog Files

The prolog files can be divided into two main categories:

- PostScript™ prolog files (.ps)
- X print server client prolog files (.xpr).

The customization of PostScript files is discussed first:

- Common prolog file
- Print layout prolog files
- Locale dependent prolog files

Customization of each category of prolog files is discussed next.

Common prolog Files

The common prolog files are:

- mp.pro.ps
- mp.common.ps
- mp.pro.alt.ps
- mp.pro.fp.ps
- mp.pro.ps
- mp.pro.ts.ps
- mp.pro.altl.ps
- mp.pro.ff.ps
- mp.pro.l.ps
- mp.pro.tm.ps

These files are the print layout prolog files, of which mp.common.ps is the common prolog file that is prepended before other prolog files.

The common prolog file, mp.common.ps, which resides in the /usr/lib/lp/locale/C/mp/ directory, contains a PostScript routine to re-encode a font from the StandardEncoding to ISOLatin1 Encoding. The .reencodeISO routine is called from the print layout prolog files to change encoding of the fonts. Usually this prolog file does not need any customization. If the users are creating their own prolog files, set the environment variable MP_PROLOGUE to point to the directory that contains the modified prolog files.

Print Layout prolog *Files*

The print layout prolog files, mp.*.ps files, contain routines for controlling the page layout for printing. In addition to giving a header and a footer for a print page with user name, print date, and page number, these prolog files can provide other information. For example, the prolog files can give effective print area dimensions and landscape and portrait mode of printing to be used.

A set of standard functions needs to be defined in every prolog file. These functions are called when a new print page starts, print page ends, or a new column ends. The implementations of these functions define the print attributes of the printout.

The following PostScript variables are defined at runtime by the mp(1) binary. All the print layout files can use these variables for printing dynamic information such as user name, subject, print time. This information taken from the variables normally appears in the header or footer of the print page.

User The name of the user who is running mp, obtained from the system passwd file.

MailFor Variable used to hold the name of the type of article to print. The possible values for this variable are:

- "Listing for" - When the input is a text file
- "Mail for" - When the input is a mail file
- "Article from" - When the input is an article from a news group

Subject The subject taken from the mail and news headers. You can use the '–s' option to force a subject to the mail and news files as well as to normal text files.

Timenow The time of print that appears in the header and footer. This information is taken from the `localtime()` function.

Following are the functions implemented in print layout `prolog` files. All these functions can use subfunctions.

endpage usage : page_number endpage

Called when the bottom of a printed page is reached. This function restores the graphic context of the page and issues a "showpage." In some `prolog` files the header and footer information is displayed in only a page-by-page mode rather than in a column-by-column mode. You can implement this function to call subfunctions that display the header and footer gray scale lozenges.

newpage usage : page_number newpage

Routines or commands to be executed when a new page begins. Setting landscape print mode, saving the print graphic context, and translating the page coordinates are some of the functions for routine.

endcol usage : page_number col_number endcol

Display header and footer information. Move to the new print position, and so forth.

For adding new print layout `prolog` files, you need to define the following variables explicitly within the print layout `prolog` file.

NumCols	<number of columns in a print page>
PrintWidth	<width of print area in inches>
PrintHeight	<height of print area in inches>

- /NumCols 2 def
- /PrintWidth 6 def
- /PrintHeight 9 def

Locale-Dependent `Prolog` *File*

The locale-dependent `prolog` file is `/usr/openwin/lib/locale/$LANG/print/prolog.ps` and it usually is a PostScript file. This file can contain included Type1 fonts that define PostScript `prolog` information with some additional PostScript routines. One of the main goals of the `prolog` file is to set the locale's fonts in an alias for a set of font names that are pre-defined and used in the `mp(1)`.

Support for this file is provided to conform with the `prolog.ps` file that is used by `/usr/bin/mp`. If this file exists, then it is given preference and `mp.conf` file is not scanned for backward compatibility.

The sections about `mp.conf` file that follow are reprinted from the *OpenWindows™ Localization Guide*.

What Is `prolog.ps`

The purpose of the `prolog.ps` file is to set up non-generic fonts. Applications use these pre-defined PostScript font names for printing. The `prolog` file must define at least the following font names for Desk Set Calendar manager and `mp`.

- LC_Times-Roman
- LC_Times-Bold
- LC_Helvetica
- LC_Helvetica-Bold
- LC_Courier
- LC_Helvetica-BoldOblique
- LC_Times-Italic

These fonts need to be able to print the local character set in the following example usage of those fonts:

- 100 100 moveto
- /LC_Times-Roman findfont 24 scale font setfont
- (Any text string in your locale) show

Example of the `prolog.ps` *File*

The localization kit provides a sample `prolog.ps` for the Japanese environment. Alternatively, this file is found in the `/usr/openwin/lib/locale/ja/print/` directory.

How to Add or Change Composite Fonts in an existing `prolog.ps` *File*

For example, the following defines a composite font called LC_Base-Font:

```
%
(Foo-Fine) makecodeset12
(Base-Font) makeEUCfont
%
```

LC_Base-Font is a composite font of Foo-Fine and a base font called Base-Font. Foo-Fine is a font that contains the local character set. You do not need any in-depth PostScript knowledge to add or change a font.

How to Create a `prolog.ps` *File*

The best way is to study the example version. In the example `prolog.ps`, two routines need to be written, `makecodeset12` and `makeEUCfont`. `Makecodeset12` sets up local font encoding information. This routine might differ from locale to locale. `MakeEUCfont` combines the base font and the locale font to form a composite font. The creator of the `prolog` file should have good knowledge of PostScript in order to write `makecodeset12` and `makeEUCfont`.

`prolog.ps` file support is kept for backward compatibility only. Do not create a new `prolog.ps` file for the printing needs of a locale. Use `mp.conf` instead.

Where Is `prolog.ps`*?*

The path is:

```
/usr/openwin/lib/locale/$LANG/print/prolog.ps
```

`.xpr` File Customization

These files are located, by default, at `/usr/lib/lp/locale/C/mp/`. A `.xpr` file corresponds to each PostScript `prolog` layout file, except for `mp.common.ps.` You can define an alternate `prolog` directory by defining the `MP_PROLOGUE` environment variable.

These files work as keyword/values pairs. Lines that start with "#" are considered comments. Spaces separate different tokens unless explicitly stated in the following description. Three main sections for each `.xpr` file are bound by the following keyword pairs:

- STARTCOMMON / ENDCOMMON
- STARTPAGE / ENDPAGE
- STARTCOLUMN / ENDCOLUMN

Certain keyword / value pairs can be used in these three areas. Each area is described next.

STARTCOMMON/ENDCOMMON Keywords

All the keyword / value pairs that appear after the STARTCOMMON keyword and before the ENDCOMMON keyword define general properties of the print page. Different valid values for a keyword are separated by using "/".

ORIENTATION 0/1

"0" means the printing occurs in portrait and "1" means in landscape.

PAGELENGTH <unsigned integer>

A value that indicates the number of lines per logical page.

LINELENGTH <unsigned integer>

A value that indicates the number of single column characters per line.

NUMCOLS <unsigned integer>

The number of logical pages per physical page.

HDNGFONTSIZE <unsigned integer>

The heading font point size in decipoints.

BODYFONTSIZE <unsigned integer>

The body font point size in decipoints.

PROLOGDPI <unsigned integer>

The dots-per-inch scale in which the current `.xpr` file is created.

YTEXTBOUNDARY <unsigned integer>

This y-coordinate establishes the boundary for text printing in a page or logical page (column). This boundary is used as an additional check to see whether text printing is occurring within the expected area. This boundary is needed for Complex Text

Layout and EUC printing, as character height information obtained from corresponding fonts can be wrong.

STARTTEXT <unsigned integer> <unsigned integer>

The decipoint x/y points where the actual text printing starts in the first logical page in a physical page.

PAGESTRING 0/1

The 1 indicates that a "Page" string needs to be appended before the page number in the heading.

0 indicates that only the page number is displayed.

EXTRAHDNGFONT "font string 1, font string 2, ... font string n"

The 'font string 1' to 'font string n' are X Logical Font Descriptions. The Token which separates the keyword EXTRAHDNGFONT from the comma separated font name list is '"', not spaces or tabs. These fonts are given preference over the built-in fonts when printing of the heading occurs. Usually, EXTRABODYFONT is used to assign printer-resident fonts that are configured in `/usr/openwin/server/etc/XpConfig/C/print/models/<model name>/fonts` directory. The `fonts.dir` contains the XLFD of the printer-resident fonts.

Usually a font is specified like

"-monotype-Gill Sans-Regular-r-normal- -*-%d-*-*-p-0-iso8859-2"

in the .xpr file. "%d", if present, is replaced by the `mp(1)` to the point size of the current heading fonts in the `.xpr` file. The x resolution and y resolution are specified "*" and the average width field is set as "0" to indicate selection of scalable font, if possible. You can give more specific font names also.

EXTRABODYFONT "font string 1, font string 2, ... font string n"

This is the same as EXTRAHDNGFONT, except that these fonts are used to print the page body.

XDISPLACEMENT <signed/unsigned int>

Gives the x coordinate displacement to be applied to the page for shifting the contents of the page in the x direction. This displacement can be a +ve or -ve value.

YDISPLACEMENT <signed/unsigned int>

This parameter is the same as x displacement except that the shifting happens in the y direction.

These two keywords are useful when you find that some printers have nonstandard margin widths and you need to shift the printed contents in a page.

STARTPAGE/ENDPAGE

The keyword value pairs in this section are bound by STARTPAGE and ENDPAGE keywords. This section contains drawings and heading information that is to be applied for a physical page. A physical page can contain many logical pages, but all the drawing routines that are contained between these keywords are applied only once to a physical page.

The valid drawing entities are LINE and ARC. XDrawLine and XDrawArc functions are executed on values of these keywords.

The dimensions within this section are mapped in PROLOGDPI units. Angles are in degrees.

LINE x1 y1 x2 y2

The /y unsigned coordinates define a pair of points for connecting a line.

ARC x y width height angle1 angle2

x and y are both unsigned integers that represent the arc origin. Width and height are unsigned ints that represent the width and height of the arc.

USERSTRINGPOS x y

Unsigned coordinates represent the position in which the user information is printed on the heading.

TIMESTRINGPOS x y

Unsigned coordinates represent the position in which the time for printing is printed on the heading.

PAGESTRINGPOS x y

Unsigned coordinates represent the position to print the page string for each printed page.

SUBJECTSTRINGPOS x y

Unsigned coordinates represent the position to print the subject in the page.

STARTCOLUMN/ENDCOLUMN

All keywords are the same as the previous STARTPAGE/ENDPAGE section except that the entries in this section are applied to NUMCOLS times to a physical page.

If NUMCOLS is 3, then the printable area of the physical page is divided into three, and lines, arcs, or heading decorations are three times per page.

Creating a New `.xpr` File

The following are the `mp(1)` program defaults for different keyword values if these values are not specified in the `.xpr` file for the STARTCOMMON/ENDCOMMON section.

- ORIENTATION 0
- PAGELENGTH 60
- LINELENGTH 80
- YTEXTBOUNDARY 3005
- NUMCOLS 01
- HDNGFONTSIZE 120
- BODYFONTSIZE 90
- PROLOGDPI 300
- STARTTEXT 135 280
- PAGESTRING 0

No default values are needed for the other two sections bound by STARTPAGE/ENDPAGE and STARTCOLUMN/ENDCOLUMN.

When you create a new `.xpr prolog` file, you need to specify only the values that differ from the default.

To create a page with no decoration, use four logical pages per physical page, in portrait format.

- STARTCOMMON
- NUMCOLS 04
- LINELENGTH 20
- ENDCOMMON

In this situation, you do not need the other two sections:

– STARTPAGE/ENDPAGE

– STARTCOLUMN/ENDCOLUMN

These parameters are not needed if you are not putting decorations on the printed page. All the coordinates are in 300 dpi default unless you are not specifying the PROLOGDPI keyword. If target printer resolution is different, the `.xpr` file is scaled to fit into that resolution by the program.

When you create a `.xpr` file, you must know the paper dimensions beforehand. For U.S. paper, 8.5x11 inches, for a printer of resolution 300 dpi, 2550X3300 are the total dimensions. Most printers cannot print from the top left corner of the paper. Instead, they put some margin around the physical paper. That means that even if you try to print from 0,0 the printing won't be in the top left corner of the page. You need to consider this limitation when you create a new `.xpr` file.

CHAPTER **9**

Development Tools Topics

This section describes development tools in the Solaris environment. This section contains these chapters.

Chapter 10	Provides instructions for using `appcert`, correcting problems that are reported by `appcert`, and writing `appcert`-compliant code
Chapter 11	Describes Sun WBEM software developer's toolkit (SDK)
Chapter 12	Describes new features in linkers and libraries
Chapter 13	Describes new information in *Solaris Modular Debugger Guide*
Chapter 14	Describes new information in *Multithreaded Programming Guide*

CHAPTER **10**

Using appcert

appcert is new in the Solaris 4/01 release. For more information see, *System Interface Guide*.

This chapter discusses:

- Purpose of the appcert utility
- Running and using appcert
- Correcting problems appcert reports
- Writing appcert-compliant code

Note - For the most current man pages, use the man command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Purpose of the appcert Utility

As new Solaris releases become available, some library interfaces might change their behavior or disappear entirely. Applications that rely on those interfaces would then stop functioning. The Solaris Application Binary Interface (ABI) defines runtime library interfaces that are safe and stable for application use. Applications that are written to conform with the Solaris ABI will remain stable under future releases of Solaris. The appcert utility is designed to help developers verify an application's compliance with the Solaris ABI.

What appcert Checks

The appcert utility examines your applications for:

- Private symbol usage
- Static linking
- Unbound symbols

Private Symbol Usage

Private symbols are functions or data that is used by Solaris libraries to call one another. The semantic behavior of private symbols might change, and symbols may sometimes be removed (such symbols are called *demoted symbols*.) The mutable nature of private symbols introduces the potential for instability in applications that depend on them.

Static Linking

Because the semantics of private symbol calls from one Solaris library to another might change between releases, the creation of static links to archives degrades the binary stability of an application. Dynamic links to the archive's corresponding shared object file avoid this problem.

Unbound Symbols

The appcert utility uses the dynamic linker to resolve the library symbols that are used by the application being examined. Symbols the dynamic linker cannot resolve are called *unbound symbols*. Unbound symbols might be caused by environment problems, such as an incorrectly set LD_LIBRARY_PATH variable, or build problems, such as omitting the definitions of the -l*lib* or -z switches at compile time. While these examples are minor, unbound symbols that are reported by appcert might indicate a more serious problem.

What appcert Does Not Check

If the object file you want appcert to examine depends on libraries, those dependencies must be recorded in the object. To do so, be sure to use the compiler's -l switch when compiling the code. If the object file depends on other shared libraries, those libraries must be accessible through LD_LIBRARY_PATH or RPATH at the time you run appcert.

The appcert application cannot check 64–bit applications unless the machine is running the 64–bit Solaris kernel. Static linking checks are not performed by appcert when it is checking 64–bit applications.

The appcert utility cannot examine:

- Object files that are completely or partially statically linked. A completely statically linked object is reported as unstable.
- Executable files that do not have the execute permission set. The appcert utility skips such executables. Shared objects without the execute permission set are examined normally.
- Object files whose user ID is set to root.
- Non-ELF executables, such as shell scripts.
- Solaris interfaces in languages other than C. The code itself need not be in C, but the call to the Solaris library must be.

Working with appcert

To check your application with appcert, type:

```
appcert object|directory
```

replacing *object|directory* with either:

- The complete list of objects you want appcert to examine and/or
- The complete list of directories that contain such objects.

Note - If appcert is run in an environment different from the one in which the application being checked would be run, the utility may not be able to resolve references to Solaris library interfaces correctly.

The `appcert` utility uses the Solaris runtime linker to construct a profile of interface dependencies for each executable or shared object file. This profile is used to determine the Solaris system interfaces upon which the application depends. The dependencies outlined in the profile are compared to the Solaris ABI to verify conformance (no private interfaces should be found).

The `appcert` utility recursively searches directories for object files, ignoring non-ELF object files. After `appcert` has finished checking the application, it prints a rollup report to the standard output (`stdout`, usually the screen). A copy of this report is written in the working directory, which is usually `/tmp/appcert.`*pid*`/Report`, in a file named `Report`. In the subdirectory name, *pid* represents the 1– to 6–digit process ID of that particular instantiation of `appcert`. See "`appcert` Results" on page 93 for more on the directory structure to which `appcert` writes output files.

`appcert` Options

The following options modify the behavior of the `appcert` utility. You can type any of these options at the command line, after the **`appcert`** command but before the *object* | *directory* operand.

`-B`

Runs `appcert` in batch mode.

In batch mode, the report produced by `appcert` will contain one line for each binary checked.

A line that begins with PASS indicates the binary named in that line did not trigger any `appcert` warnings.

A line that begins with FAIL indicates problems were found in that binary.

A line that begins with INC indicates the binary named in that line could not be completely checked.

`-f` ***infile***

The file *infile* should contain a list of files to check, with one file name per line. These files are added to any files already specified at the command line. If you use this switch, you do not need to specify an object or directory at the command line.

`-h`

Prints usage information for `appcert`.

-L

By default, appcert notes any shared objects in an application and appends the directories they reside in to LD_LIBRARY_PATH. The -L switch disables this behavior.

-n

By default, appcert follows symbolic links when it searches directories for binaries to check. The -n switch disables this behavior.

-S

Appends the Solaris library directories /usr/openwin/lib and /usr/dt/lib to LD_LIBRARY_PATH.

-w ***working_dir***

Specifies a directory in which to run the library components and create temporary files. If this switch is not specified, appcert uses the /tmp directory.

appcert Results

The results of the appcert utility's analysis of an application's object files are written to several files that are located in the appcert utility's working directory (typically /tmp) The main subdirectory under the working directory is appcert.*pid*, where *pid* is the process ID for that instantiation of appcert.

Index

Contains the mapping between checked binaries and the subdirectory in which appcert output specific to that binary is located.

Report

Contains a copy of the rollup report displayed on stdout when appcert is run.

Skipped

Contains a list of binaries that appcert was asked to check but was forced to skip, along with the reason each binary was skipped. These reasons are:

- File is not a binary object
- File cannot be read by the user
- File contains metacharacters
- File does not have the execute bit set

`objects/`***object_name***

A separate subdirectory is under the `objects` subdirectory for each object examined by `appcert`. Each of these subdirectories contains the following files:

`check.demoted.symbols`

Contains a list of symbols `appcert` suspects are demoted Solaris symbols.

`check.dynamic.private`

Contains a list of private Solaris symbols to which the object is directly bound.

`check.dynamic.public`

Contains a list of public Solaris symbols to which the object is directly bound.

`check.dynamic.unbound`

Contains a list of symbols not bound by the dynamic linker when running `ldd -r`. Lines returned by `ldd` containing "`file not found`" are also included.

`summary.dynamic`

Contains a printer-formatted summary of dynamic bindings in the objects `appcert` examined, including tables of public and private symbols used from each Solaris library.

When `appcert` exits, it returns one of four exit values.

0

No potential sources of binary instability were found by `appcert`.

1

The `appcert` utility did not run successfully.

2

Some of the objects checked by `appcert` have potential binary stability problems.

3

The `appcert` utility did not find any binary objects to check.

Correcting Problems Reported by appcert

- *Private Symbol Use.* Because private symbols might change their behavior or disappear from one Solaris release to another, an application that depends on private symbols might not run on a Solaris release different from the one it was developed in. If appcert reports private symbol usage in your application, rewrite the application to avoid the use of private symbols.
- *Demoted Symbols.* Demoted symbols are functions or data variables in a Solaris library that have been removed or scoped locally in a later Solaris release. An application that directly calls such a symbol will fail to run on a release whose libraries do not export that symbol.
- *Unbound Symbols.* Unbound symbols are library symbols that are referenced by the application that the dynamic linker was unable to resolve when called by appcert. While unbound symbols are not always an indicator of poor binary stability, they might indicate a more serious problem, such as dependencies on demoted symbols.
- *Obsolete Library.* An obsolete library might be removed from Solaris in a future release. The appcert utility flags any use of such a library, because applications that depend on them will not function in a future release that does not feature the library. To avoid this problem, do not use interfaces from obsolete libraries.
- *Use of sys_errlist or sys_nerr.* The use of the sys_errlist and sys_nerr symbols might degrade binary stability, as a reference might be made past the end of the sys_errlist array. To avoid this risk, use strerror instead.
- *Use of strong and weak symbols.* The strong symbols that are associated with weak symbols are reserved as private because their behavior might change in future releases of Solaris. Applications should only directly reference weak symbols. An example of a strong symbol is _socket, which is associated with the weak symbol socket.

CHAPTER **11**

WBEM SDK

The WBEM SDK is new in the Solaris 8 4/01 release. For more information on the WBEM SDK, see *Sun WBEM SDK Developer's Guide*.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Web-Based Enterprise Management (WBEM)

Web-Based Enterprise Management (WBEM) includes standards for web-based management of systems, networks, and devices on multiple platforms. The Sun WBEM Software Developer's Toolkit (SDK) allows software developers to create standards-based applications that manage resources in the Solaris operating environment. Developers can also use this toolkit to write providers, programs that communicate with managed resources to access data. The Sun WBEM SDK includes Client Application Programming Interfaces (APIs) for describing and managing resources in Common Information Model (CIM) , and Provider APIs for getting and setting dynamic data on the managed resource. The Sun WBEM SDK also provides CIM WorkShop, a Java application for creating and viewing the managed resources on a system, and a set of example WBEM client and provider programs.

CHAPTER **12**

Linkers and Libraries Guide Revisions

The *Linkers and Libraries Guide* is revised in 1/01 and 10/00 releases. The following tables list the changes. To view the book, see *Linker and Libraries Guide*.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Changes to *Linkers and Libraries Guide*

TABLE 12–1 What's New for *Linkers and Libraries Guide*

For Solaris 8 1/01 release, *Linker and Libraries Guide* has been updated with the following information.	■ The symbolic information available from `dladdr(3DL)` has been enhanced with the introduction of `dladdr1()`. ■ The `$ORIGIN` of a dynamic object can be obtained from `dlinfo(3DL)`. ■ The maintenance of runtime configuration files, created with `crle(1)`, has been simplified with the display of the command-line options that were used to create the configuration file. Also available is an update capability (see the `–u` option). ■ The runtime linker and its debugger interface have been extended to detect procedure-linkage-table entry resolution. This update is identified by a new version number. See the section "Agent Manipulation." This update extends the rd_plt_info_t structure. See the section "Procedure Linkage Table Skipping." ■ An applications stack can be defined non-executable by using the new `mapfile` segment descriptor STACK. See the section "Segment Declarations."
For the Solaris 8 10/00 release, *Linker and Libraries Guide* has been updated with the following information.	■ The environment variable LD_BREADTH is ignored by the runtime linker. See the section, "Initialization and Termination Routines." ■ The runtime linker and its debugger interface have been extended for better runtime and core file analysis. This update is identified by a new version number. See the section "Agent Manipulation." This update expands the `rl_flags, rl_bend`, and `rl_dynamic` fields of the `rd_loadobj_t` structure. See the section "Scanning Loadable Objects." ■ The validation of displacement-relocated data in regard to its use, or possible use, with copy relocations is now provided. See the section "Displacement Relocations." ■ 64-bit filters can be built solely from a mapfile by using the `link-editors -64` option. See the section "Generating a Standard Filter." ■ Some explanatory notes are provided on why `$ORIGIN` dynamic string token expansion is restricted within secure applications. See the section "Security." ■ The search paths that are used to locate the dependencies of dynamic objects can be inspected by using `dlinfo(3DL)`. ■ `dlsym(3DL)` and `dlinfo(3DL)` look-up semantics have been expanded with a new handle, `RTLD_SELF`. ■ The runtime symbol look-up mechanism that is used to relocate dynamic objects can be significantly reduced by establishing direct binding information within each dynamic object. See the sections "External Bindings" or "Direct Binding."

CHAPTER **13**

Solaris Modular Debugger Guide Revisions

The *Solaris Modular Debugger Guide* is revised for the Solaris 8 10/00 release. To view the book, see *Solaris Modular Debugger Guide*.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Changes to *Solaris Modular Debugger Guide*

- The “Arithmetic Expansion” section of Chapter 3 has been updated to include unary operators.
- Minor technical errors have been corrected.

CHAPTER **14**

Multithreaded Programming Guide Revisions

The *Multithreaded Programming Guide* is revised for the Solaris 8 1/01 release. To view the book, see *Multithreaded Programming Guide*.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

SPARC: Changes to *Multithreaded Programming Guide*

The *Multithreaded Programming Guide* has been revised to incorporate bug fixes: 4308968, 4356675, 4356690.

CHAPTER 15

Interface Development Topics

This section provides instructions for interface development in the Solaris operating environment. This section contains these chapters.

Chapter 16	Describes changes to the *System Interface Guide*

CHAPTER **16**

System Interface Guide Revisions

The *System Interface Guide* is revised for the Solaris 8 6/00 release. To view the book, see *System Interface Guide.*

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Changes to *System Interface Guide*

The *System Interface Guide* is updated to incorporate bug fixes. This release corrects several typographical errors in text and source code examples.

CHAPTER **17**

Java 2 Standard Edition and JDK Topics

This section provides Java 2 Standard Edition and Java Developer Kit (JDK) feature descriptions. This section contains this chapter.

Chapter 18	Provides detailed feature descriptions for each Java 2 SDK and JDK release in Update releases

CHAPTER **18**

Java 2 Standard Edition and JDK New Feature Information

The Solaris 8 4/01 release includes the Java 2 SDK Standard Edition v. 1.3.0, also known as J2SE™ 1.3.0. Below are new feature descriptions. Also included are descriptions of J2SE and JDK releases in previous Update releases. The information in this section supplements that found in *The Java 2 SDK for Solaris Developer's Guide*. The following describes new features in each Update release.

Note - For the most current man pages, use the `man` command. The Solaris 8 Update release man pages include new feature information that is not in the *Solaris 8 Reference Manual Collection*.

Java 2 SDK, Standard Edition, version 1.3.0

J2SE 1.3.0 is an upgrade release for Java 2 SDK. The software includes the following new features and enhancements.

- Performance improvements

  Java HotSpot™ technology and performance-tuned runtime libraries make J2SE 1.3.0 faster in many functional areas than previous versions of the Java 2 SDK.

- Easier web deployment

  New features such as applet caching and automatic installation of optional packages by J2SE 1.3.0's Java Plug-in component enhance the speed and flexibility with which programs can be deployed on the web.

- Enterprise interoperability

  The addition of RMI/IIOP and the Java Naming and Directory Interface™ in J2SE 1.3.0 enhance the interoperability of the Java 2 Platform.

- Security advances

  New support for RSA electronic signing, dynamic trust management, X.509 certificates, and verification of Netscape-signed files mean more ways for developers to protect their electronic data.

- Java Sound

  J2SE 1.3.0 includes a powerful new sound API. Previous releases of the platform limited audio support to basic playback of audio clips. With this release, the Java 2 Platform for the first time defines a set of standard classes and interfaces for low-level audio support.

- Enhanced APIs and improved ease of development

  In response to requests from the development community, J2SE 1.3.0 adds new features to various areas of the Java 2 Platform. These features expand the functionality of the platform to enable development of more powerful applications. In addition, many of the new features make the development process itself faster and more efficient.

Each of these features is described more fully below. For detailed information about the new features in J2SE 1.3.0, see the Java 2 Platform documentation at `http://java.sun.com/j2se/1.3/docs`. The Java 2 Platform documentation includes the API specification for J2SE 1.3.0.

Performance Enhancements

Many enhancements have been made to improve the performance of the Java 2 SDK in version 1.3.0 These changes include the addition of the Java HotSpot™ Client Virtual Machine (VM) and the Java HotSpot Server VM, both of which implement high-performance Java HotSpot technology. The Java HotSpot Client VM is tuned to maximize performance on client systems, improving performance in areas of startup time and memory footprint. The Java HotSpot Server VM is tuned to maximize performance of program execution speed and is aimed at server applications that are less concerned with startup and memory footprint.

J2SE 1.3.0 also includes newly tuned class libraries for improved runtime performance.

Easier Web Deployment

Applet Caching

J2SE 1.3.0's new applet-caching feature ensures that often-used applets are always available for rapid loading and fast startup by keeping copies of the applets in a local cache. When an applet has been downloaded more than once, it can be stored in the local applet cache. This storage eliminates the need for a browser to download an applet over the network every subsequent time that the applet is needed. The local, cached copy can be used instead.

This feature is valuable for large, high-use applets. Many enterprise applets, for example, are in the megabyte-size range, and applets that large can take tens of minutes to load over a network. The new applet-caching feature eliminates the download time, enabling businesses to use a larger array of more powerful applets than ever before.

Automatic Deployment of Optional Packages

J2SE 1.3.0 also supports automatic deployment of optional packages. Optional packages are sets of features and APIs that, while not part of the Java 2 Platform, Standard Edition, are available separately for developers to use for specialized programming needs. Examples are Java Media Framework technology and the JavaHelp™ optional packages.

Before J2SE 1.3.0, an applet that used one or more optional packages had to trust that up-to-date versions of the optional packages were installed on every client that might want to run the applet. Without the proper optional package installed on the client, the applet could exhibit unintended behavior or not run at all.

With J2SE 1.3.0, applets can specify version and vendor information for any optional packages that they require. Developers can have their applets specify a URL at which the latest version of a required optional package can be downloaded if any of the following conditions are met.

- The optional package is not already installed locally.
- The optional package is installed but has an out-of-date version number.
- The optional package is installed but is not from a specified vendor.

J2SE 1.3.0 supports any native and Java language installer programs that an optional package might have, automatically launching the installer program when a new version of an optional package is retrieved from the network.

Enterprise Interoperability

Java IDL and RMI-IIOP

J2SE 1.3.0 includes two significant enhancements to support for CORBA technologies: a production CORBA IDL compiler written in the Java language and the RMI over IIOP (RMI-IIOP) API. CORBA Interface Definition Language (IDL) is a language that defines only the interfaces for distributed systems. By using a neutral language to define interfaces, CORBA can support multiple languages. For the first time, the Java 2 SDK, Standard Edition includes an IDL compiler to compile language-neutral CORBA IDL into standard Java language bindings. These language bindings work with the Java IDL Object Request Broker (ORB) to support traditional CORBA programming in the Java programming language.

Since version 1.1 of the Java platform, the Remote Method Interface (RMI) has allowed programmers to write interfaces for distributed computing directly in the Java language. Because RMI used its own wire protocol, programmers had to give up the ability to communicate with objects written in other languages when they used RMI. RMI-IIOP uses the Java IDL ORB to enable the standard CORBA wire protocol, Internet InterORB Protocol (IIOP), to be used with RMI. Since IIOP is used for all communication, objects written in other languages such as C++ can communicate with RMI over IIOP distributed objects. Further, RMI has been accepted as the CORBA standard for mapping interfaces in the Java programming language to CORBA IDL. To facilitate programming in other languages, CORBA standard IDL can be generated from RMI-enabled classes. Existing RMI programs can be converted to use the IIOP protocol, typically with very limited changes.

RMI-IIOP combines the programming ease of RMI with JavaIDL’s CORBA-compliant interaction with software written in other languages. By adhering to a few restrictions, RMI programmers can now use CORBA’s IIOP communications to protocol to communicate with clients of any type, whether written entirely in the Java programming language or made up of components written in other CORBA-compliant languages.

Java Naming and Directory Interface (JNDI) API

J2SE 1.3.0’s new Java Naming and Directory Interface™ (JNDI) API enables developers to add naming and directory functionality to applications written in the Java programming language. JNDI is designed to be independent of any specific naming or directory service implementation to enable seamless connectivity to heterogeneous enterprise naming and directory services. Thus a variety of services — new, emerging, and already deployed ones—can be accessed in a common way. Developers can use J2SE 1.3.0 to build powerful and portable directory-enabled applications by using this industry-standard interface.

The JNDI architecture consists of an API and a Service Provider Interface (SPI). Java applications use this API to access a variety of naming and directory services. The

SPI enables a variety of naming and directory services to be plugged in transparently, allowing the Java application to access their services. JNDI in the J2SE 1.3.0 release comes with service providers for accessing the following services:

- Lightweight Directory Access Protocol (LDAP) — An Internet standard for accessing directory services
- Common Object Services (COS) Name Server — The name server for storing CORBA object references
- The RMI registry service provider — The name server for storing RMI remote objects

Security Advances

With the security enhancements available in J2SE 1.3.0, developers have more tools at their disposal for protecting their technology investments. New support for RSA signatures and J2SE 1.3.0's enhanced dynamic trust management facilities also greatly increase the ease of web-based deployment.

Support for RSA Signatures

J2SE 1.3.0 includes a cryptographic service provider to support the widely used RSA signatures for electronically signing software that is delivered over the Web. Standard RSA certificates are supported, including those from VeriSign and Thawte.

Prior to J2SE 1.3.0, Java platform users who wanted to use RSA certificates needed to write their own RSA service providers or purchase an RSA service provider from a third party. Now an RSA provider is included as a standard part of J2SE 1.3.0.

Dynamic Trust Management

New dynamic trust management facilities in J2SE 1.3.0 provide pop-up dialogs to let users validate applet signers, eliminating the need to deploy security key files to each client that runs the signed applet.

Previously, if a user wanted to give an applet from a trusted source extra security permissions to allow the applet to perform normally forbidden operations, the user needed to preconfigure his or her local cache of trusted signer certificates to recognize the certificate of the applet's trusted source. This would need to be done for every client machine on which the applet might potentially be run.

J2SE 1.3.0 provides a better solution by providing facilities to extract the applet's signers from the applet's codesource and pass them to the browser. The browser then verifies the certificate chain all the way up to its root certificate, and checks if that root certificate is contained in the browser's database of trusted root certificates. If so, the browser displays the chain of the authenticated signer and give the user the option to remove all security restrictions on the applet.

Improved Support for Public-Key Certificates

J2SE 1.3.0 provides enhanced support for X.509 public-key certificates. J2SE 1.3.0 now supports all X.520 attributes that are either mandated or recommended by the most recent proposed standard protocol (RFC 2459). In addition, J2SE 1.3.0 can handle multiple Attribute/Value Assertions within a Relative Distinguished Name.

Java Sound

The Java Sound API enables Java programs to capture, process, and play audio and Musical Instrument Digital Interface (MIDI) data. These new capabilities enable developers to create new types of applications, including:

- Communication frameworks, including conferencing and telephony applications.
- End-user content delivery systems. These systems range from simple desktop media players to streamed music delivery systems or broadcast audio applications for live events.
- Interactive applications, such as games and web sites, that generate sound dynamically in response to user interaction.
- Tools and toolkits for creating and editing original audio or musical content.

The Java Sound API is supported by an efficient sound engine that guarantees high-quality audio mixing and MIDI synthesis capabilities for the platform. More specifically, the implementation that is included with J2SE 1.3.0 supports the following features:

- Audio file formats: AIFF, AU and WAV
- Music file formats: MIDI type 0 and type 1 and Rich Music Format
- Audio codecs: u-law and a-law
- Audio data formats: 8- and 16-bit audio samples, in mono and stereo, with sample rates from 8 kHz to 48 kHz
- MIDI wavetable synthesis and sequencing in software, and access to hardware MIDI devices
- An all-software mixer that can mix and render up to 64 total channels of digital audio and synthesized MIDI music.

Additionally, the API defines a set of service provider interfaces that developers can use to extend the capabilities of the current implementation. Users can install modules that provide support for additional file formats, codecs, and devices. The API includes methods for querying and accessing the resources currently available on the system.

Enhanced APIs and Improved Ease of Development

AWT Enhancements

J2SE 1.3.0 has a new Robot API that is designed to make automated Abstract Window Toolkit (AWT) and Swing testing possible. The Robot API enables code that is written in the Java programming language to generate low-level native mouse and keyboard input events. Because the events are generated at the operating system level, they are indistinguishable from real user input to the rest of the AWT.

Though designed primarily to improve testability, the Robot API also provides other benefits:

- Accessibility-enabled applications can give more feedback. For example, if the user acts on a screen object by using voice commands, the mouse pointer can be moved to indicate the object being manipulated.
- The Robot API enables creation of computer-based training (CBT) and other demo-type applications.

J2SE 1.3.0 also has an improved API for printing. The new printing API gives developers an easy mechanism to print the AWT components by using native platform facilities. By using the new API, developers can control properties of a print job such as destination, number of copies, page ranges, page size, orientation, print quality, and more.

Java 2D Technology Enhancements

J2SE 1.3.0 introduces support for rendering on multiple monitors the GUI Frames and Windows that belong to the same application. The Java 2D™ API supports three multi-screen configurations:

- Two or more independent screens
- Two or more screens where one screen is the primary screen and the other screens display copies of what appears on the primary screen
- Two or more screens that form a virtual desktop

With J2SE 1.3.0's new dynamic font-loading API, a developer can create and load TrueType fonts during runtime. Developers can use the Java 2D API to give their dynamically loaded fonts the desired features such as size, style, transforms, and others.

The Java 2D API in J2SE 1.3.0 now supports the Portable Graphics Network (PGN) format, a flexible, extensive, non-proprietary file format that represents lossless and portable storage of raster images. PGN supports gray scale, indexed-color, and truecolor images, with an optional alpha channel.

Java Platform Debugger Architecture (JPDA)

JPDA technology is a multi-tiered debugging architecture that enables tool developers to easily create debugger applications that run portably across platforms, virtual-machine implementations, and J2SE versions.

JPDA consists of three layers:

- JVMDI - Java Virtual Machine Debug Interface

  Defines the debugging services a VM must provide for debugging.

- JDWP - Java Debug Wire Protocol

  Defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the Java Debug Interface.

- JDI - Java Debug Interface

  Defines a high-level Java programming language interface that tool developers can easily use to write remote debugger applications.

Internationalization

The internationalization enhancements in the J2SE 1.3.0 release give developers even more flexibility in localizing their applications for international users. Two new features are described here.

Input methods are software components that interpret user operations such as typing keys or speaking to generate text input for applications, and they play an important role in enabling entry of text in international locales. Unlike English text which can be entered by directly typing it in from the keyboard, entering text in languages such as Japanese or Chinese requires a more sophisticated input method framework, and J2SE 1.3.0 provides a powerful set of the tools that developers need to handle the job.

Modern text-editing components permit the display of entered text inside the context of the document in which the text will finally appear. This is called the *on-the-spot* input, and it has always been supported by the Java 2 Platform.

J2SE 1.3.0 adds support for a second style of input, called *below-the-spot*, that is popular is such countries such as China. In below-the-spot text editing, composed text is shown in a separate composition window that is automatically positioned close to the insertion point where text will be inserted.

It might be that a developer would want to change and customize the windows that appear as part of his or her input method framework. J2SE 1.3.0 gives developers full flexibility to do so by providing a new API for an input method engine Service Provider Interface (SPI). The SPI enables developers to construct their own custom input method engines to meet the needs their software.

A further example of new international locale support is that J2SE 1.3.0 can render application frames and dialog boxes to have toolbars and menu bars with a right-to-left orientation for locales such as Arabic and Hebrew.

Other Enhancements to Platform Libraries and Tools.

J2SE 1.3.0 contains select new functionality that Sun has added to the platform and Java 2 SDK tools suite in consultation with business partners and in response to input from developers. A sampling of the enhancements include:

- New javac Compiler

  The javac compiler has been re-implemented from scratch in J2SE 1.3 making it faster for many applications than the compilers in previous versions of the Java 2 SDK.

- Dynamic Proxy Classes

  J2SE 1.3 contains a new API for dynamic proxy classes. A dynamic proxy class is a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class is encoded and dispatched to another object through a uniform interface. Thus, you can use a dynamic proxy class can be used to create a type-safe proxy object for a list of interfaces without requiring pre-generation of the proxy class, such as with compile-time tools. Dynamic proxy classes are useful to developers who need to provide the type-safe reflective dispatch of invocations to objects that present interface APIs.

  For example, you can use a dynamic proxy class to create an object that implements multiple arbitrary event listener interfaces to process a variety of events of different types in a uniform fashion, such as by logging all such events to a file.

- Expanded API for Collections

  The J2SE 1.3 version of the popular Collections API has been made even easier for you to use. The 1.3 Collections API includes new convenience methods and copy constructors for Lists and Maps.

- Expanded Java Foundation Classes/Swing Functionality

  A large part of the J2SE 1.3.0 engineering effort has been directed into tuning and enhancing the Swing components of the Java Foundation Classes API. In addition to performance tuning of the Swing libraries, new JFC/Swing functionality has been added to the Swing libraries in several areas. One example is the new support for variable-height rows in lightweight table components.

- Improved Math and Utility Libraries

  J2SE 1.3.0 includes two math-related classes that have the same API: `Math` and `StrictMath`. `StrictMath` is defined to return bit-for-bit reproducible results for numeric operations in all implementations for developers who need that guarantee. Implementations of class `Math`, on the other hand, can vary within specified constraints to enable flexibility for better performance. Developers who want best performance but don't require bit-for-bit reproducible results on different platforms will want to use `Math` rather than `StrictMath` for their numeric code.

  The J2SE API for arbitrary precision math, classes `BigInteger` and `BigDecimal`, enables arithmetic operations that never overflow or lose precision, features necessary for many types of computations such as financial calculations. Class `BigInteger` has been reimplemented in pure Java programming-language code. Previously, the implementation of the `BigInteger` class was based on an underlying C library. The new implementation performs many standard operations faster than the old implementation. The new API also includes new convenience features that make it easier for you to use.

  A new Timer API has been added to the Java 2 Platform to support animations, human interaction timeouts, on-screen clocks and calendars, work-scheduling routines, reminder facilities, and more.

  An API for virtual machine shutdown hooks has been added to class `java.lang.Runtime` that provides a simple, portable interface to the underlying operating system's process-shutdown notification so that an application written in the Java programming language can initiate shutdown actions such as closing down network connections, saving session state, and deleting temporary files.

  New *delete-on-close* mode for opening Zip and Jar files has been added so that long-running server applications can delete no-longer-needed `JarFile` objects and data to keep disk space free.

Java 2 SDK, Standard Edition, version 1.2.2_07a and Previous Releases

The following describes new features in J2SE releases.

TABLE 18–1 Previous Java 2 Standard Edition (J2SE) Releases

Java Release	Update Release
J2SE 1.2.2_07a contains fixes for bugs identified in previous releases in the J2SE 1.2.2 series. An important bug fix in J2SE 1.2.2_07a is a fix for a performance regression that was introduced in J2SE 1.2.2_05. For more information about bug fixes in J2SE 1.2.2_07a, see this URL: `http://java.sun.com/j2se/1.2/ReleaseNotes.html`.	4/01
The J2SE 1.2.2_06 is improved with bug fixes since the last release.	1/01
The J2SE v. 1.2.2_05a is a bug-fix release of v. 1.2.2_05 (without the "a") of the same product and includes the following new features and enhancements. ■ Scalability improvements to over 20 CPUs Improved handling of concurrency primitives and threads has increased the performance of multithreaded programs and significantly reduced garbage-collection pause times for programs that use many threads. ■ Improved JIT compiler optimizations The JIT compiler performs the following new optimizations: inlining of virtual and non-virtual methods, CSE within extended basic blocks, loop analysis to eliminate array bounds checking, and fast type checks. ■ Text-rendering performance improvements Several graphics optimizations have significantly improved text-rendering performance for Java 2 Standard Edition on Solaris software platforms without direct graphics access (DGA) support. These platforms include Ultra 5 and Ultra 10, the Solaris (*Intel Platform Edition*) Operating Environment, and all remote display systems. ■ `poller` class demo package Provides Java applications with the ability to efficiently access the functions of the C `poll(2)` routine and is provided as a demo package with a sample usage server. ■ Swing improvements Significant improvements in quality and performance have been made to the Swing classes. For additional information on these improvements, see the following URLs: ▪ http://Java.sun.com/products/jdk/1.2/changes.html ▪ http://java.sun.com/products/jdk/1.2/fixedbugs/index.html	10/00

JDK Releases

The following are JDK releases that have been included in Update releases.

TABLE 18–2 JDK Releases

JDK Releases	Update Release
JDK 1.1.8_12 is improved with bug fixes since the last release.	1/01
The JDK 1.1.8_10 is improved with bug fixes.	10/00

Java Servlet Support in Apache Web Server

In the Solaris 8 10/00 release, with the addition of mod_jserv module and related files, the Apache web server software now supports Java servlets. The following configuration files are now stored in /etc/apache:

- zone.properties
- jserv.properties
- jserv.conf

The mod_jserv module, like the rest of Apache software, is open source code that is maintained by a group external to Sun. This group seeks to maintain compatibility with previous releases of Apache and mod_jserv.