# Solaris 8 Software Developer Supplement

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

Part Number 806-5184–10
October 2000

---

Adobe PostScript

**Please Recycle**

# Contents

# Preface

The *Solaris 8 Software Developer Supplement* describes new or changed functionality in Solaris™ Update releases. The information here supplements or supersedes information in the previous releases of Solaris 8 documentation sets. Solaris documentation is available on the Solaris 8 Documentation CD included with this release.

---

**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$^{SM}$ 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 up-to-date 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 | First Released in... |
|---|---|
| Drivers | |
| The Generic LAN driver (GLD) can be used 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 only available for Solaris *Intel Platform Edition* network drivers. Now GLD is available for Solaris *SPARC™ Platform Edition* network drivers as well.<br>For more information, see Chapter 2. | 10/00 |
| Chapter 3 provides a detailed description of how to design drivers to support High Availability through driver hardening and ensuring serviceability. This material extends information provided the Solaris 8 *Writing Device Drivers*.<br>For more information, see Chapter 3. | 10/00 |
| Software Developer | |

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

| **Feature** | **First Released in...** |
|---|---|
| Additional partial locales for European Solaris software includes the addition of UTF-8 locales for Russian and Polish and two new locales for Catalan.<br><br>For more information, see "Additional Partial Locales for European Solaris Software" on page 45. | 10/00 |
| A number of new features have been added for linkers and libraries.<br><br>For more information, see "Linkers and Libraries Guide" on page 53. | 10/00 |
| Java | |
| 32–bit: With the addition of the `mod_jserv` module and related files, the Apache web server now supports Java™ Servlets.<br><br>For more information, see "Java Servlet Support in Apache Web Server" on page 52. | 10/00 |
| The JDK™ 1.2.2_05a contains the following new features:<br>■ Scalability improvements to over 20 CPUs<br>■ Improved JIT compiler optimizations<br>■ Text rendering performance improvements<br>■ `poller` class demo package<br>■ Swing improvements<br><br>For more information, see "Enhancements in Java 2 Standard Edition for Solaris v. 1.2.2_05a" on page 51. | 10/00 |
| The Solaris 8 10/00 software release includes the JDK 1.1.8_10 which is improved with bug fixes since the last release. | 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. | 10/00 |

CHAPTER **2**

# Drivers for Network Devices

**Note -** For the most up-to-date 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*.

The following functionality 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 release, 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 the man pages: `gld(7D)`, `kstat(7D)`, `dlpi(7P)`, `attach(9E)`, `gld(9E)`, `open(9E)`, `gld(9F)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `kstat(9S)`.

## 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 adhere 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 that length in the MAC frame header `type` field. 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 Sub-Net Access Protocol (SNAP) 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 fall into the format specified above. Packets that do fall into this format are matched to 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` above.

## 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 the multiple routes that are available. 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 abridged) network.

## Style 1 and 2 Providers

GLD implements both Style 1 and Style 2 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. If both Style 1 and Style 2 `open`s, 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 called 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 cannot be issued 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 each Stream initializes upon completion of the processing of the DL_BIND_REQ. Multiple Streams can be bound to the same SAP; 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. A set of multicast addresses can be iteratively created and modified on a per-Stream basis using 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. Physical level promiscuous mode can be specified for all packets on the medium or for 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 returns the MAC address currently associated with the PPA attached to the Stream, in 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 can 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 physical address of a PPA can be changed by the superuser 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 is successful.

## 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 snoop1M command. The DLIOCRAW command puts the Stream into a raw mode, which, upon receipt, causes the full MAC-level packet to be sent upstream in an M_DATA message instead of being transformed into the DL_UNITDATA_IND form 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(s). 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>. They 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( ). It then also returns 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). The 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` | Number of 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` | 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 are maintained by device-specific drivers of that type, as above.

| | |
|---|---|
| `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, fcs, and too-long 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 cannot make any assumptions about the length of this structure, which might be different 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 */
u_int               (*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; may 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 can be zero if the device-specific driver can handle 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 allocate space to hold the broadcast address, fill it in with the appropriate value, and set `gldm_broadcast_addr` to point at 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 allocate space to hold the address, fill it in with information read from the device, and set `gldm_vendor_addr` to point at 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 received 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 cannot make any assumptions about the length of this structure, which might be different 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` can also be specified. |
| `glds_duplex` | Current duplex state of the interface. Supported values are `GLD_DUPLEX_HALF` and `GLD_DUPLEX_FULL`. `GLD_DUPLEX_UNKNOWN` can also be specified. |

# 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` if 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 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, GLD returns `GLD_NORESOURCES`. Return `GLD_NOTSUPPORTED` if 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_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_PROMISC_MULTI`, then reception of all multicast packets will be enabled. If *promiscflag* is set to `GLD_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` if 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_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 chain through 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( ). It then 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 can 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 been 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 cannot 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.

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 can be specified as `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 will return 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, these additional values might be returned by some of the GLD entry point functions:

| | |
|---|---|
| `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 mac_info pointer 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 then initialize several of its structure elements. See `lgld_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 also 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 cannot 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 because of recursive mutex entry.

```
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, it 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( ) can 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 **3**

# High Availability Drivers

---

**Note -** For the most up-to-date 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*.

---

This functionality is new in the Solaris 8 10/00 release.

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 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 resulting 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.

Additional information about how to create a Solaris device driver can be found in *Writing Device Drivers*.

## Driver Hardening

Hardening is the process of ensuring that a driver works correctly in spite of faults 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, 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 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. It 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 up 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 fail-over 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). Using 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 timeout 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 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 be misleading. 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 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 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 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 going 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 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 may 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 indicating 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. These are:

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

### Thread Interaction

Kernel panics in a device driver are often caused by 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 may 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 upon 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 `ioctls`.

## Adaptive Strategies

A driver can continue to provide service with faulty hardware, attempting 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, the driver must be enabled 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 upon to handle a device that is `OFFLINE`. Generally, the device state will reflect earlier device fault reports, possibly modified by any reconfiguration activities that have taken place.

### 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, returning (if possible) an appropriate error code (for example, `EIO`). For a STREAMS driver, `M_ERROR` or `M_HANGUP`, as appropriate, should be put upstream to indicate that the driver is not usable.

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

The `ioctl()` entry point presents a problem in this respect: `ioctl` operations that imply I/O to the device (for example, formatting a disk) should fail if the device is unusable, while others (such as 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, this will eventually result in the interrupt being 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 will cause the device state to be changed to `DOWN` and a service-degraded fault will cause 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 using `ddi_dev_report_fault`(9F) as well as 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.

As a general rule, 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 where the device can be the secondary or fail-over 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 whose 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` used by download), and event counters. Reading an updated predictable value from the device gives a degree of confidence that things are proceeding satisfactorily.
- Time-stamp outgoing requests (transmit blocks or commands) when issued by the driver.

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

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

CHAPTER **4**

# Software Developer

This chapter describes new partial locales.

---

**Note -** For the most up-to-date 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*.

---

## Additional Partial Locales for European Solaris Software

This functionality is new in the Solaris 8 10/00 release.

The current identified 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 additional locales are partial locales, because there is no language support (translation of messages and GUI).

## Localization in the Base and Multilingual Solaris Product

### Central Europe

**TABLE 4–1** 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 Europe

**TABLE 4–2** 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 | Russia (Russia) |
| `ru_RU.UTF-8` | 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) |

## South Europe

**TABLE 4–3** 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 4–4** 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 |
| 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 **5**

# Java for Developers

This chapter describes new Java features.

---

**Note -** For the most up-to-date 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*.

---

## Enhancements in Java 2 Standard Edition for Solaris v. 1.2.2_05a

The Java 2 Standard Edition v. 1.2.2_05a is the latest release of Java 2 platform for the Solaris operating environment. It 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; Ultra 10; the Solaris Operating Environment, Intel Platform Edition; 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

## Java Servlet Support in Apache Web Server

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, maintained by a group external to Sun. This group seeks to maintain compatibility with previous releases of Apache and `mod_jserv`.

## Java Development Kit (JDK) 1.1.8_10

The Solaris 8 10/00 software release includes the JDK 1.1.8_10 which is improved with bug fixes since the last release.

CHAPTER **6**

# Summary of Changes to Solaris 8 Books

Some Solaris 8 books have been revised and are included in the Solaris 8 10/00 Update Collection. This chapter describes changes to these books since the 6/00 Update release.

---

**Note -** For the most up-to-date 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*.

---

## System Interface Guide

For Solaris 8 6/00, the *System Interface Guide* is updated to incorporate bug fixes. This release corrects several typographical errors in text and source code examples. See *System Interface Guide*.

## Linkers and Libraries Guide

The *Linker and Libraries Guide* has been updated with the following new information for Solaris 8 10/00.

- 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 `rd_init()` function in the section Agent Manipulation. This update expands the `rl_flags, rl_bend`, and `rl_dynamic` fields of the `rd_loadobj_st` 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 using the `link-editors -64` option. See the section, "Generating a Standard Filter."
- Some explanatory notes on why `$ORIGIN` dynamic string token expansion is restricted within secure applications are provided. See the section, "Security."
- The search paths used to locate the dependencies of dynamic objects can be inspected using `dlinfo(3DL)`.
- `dlsym(3DL)` and `dlinfo(3DL)` lookup semantics have been expanded with a new handle `RTLD_SELF`.
- The runtime symbol lookup mechanism used to relocate dynamic objects can be significantly reduced by establishing direct binding information within each dynamic object. See the section, "External Bindings and Direct Binding."

## Solaris Modular Debugger Guide Updates

This information is new in the Solaris 8 10/00 software release.

The following updates are included in the *Solaris Modular Debugger Guide*:

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