

### Driver Data Structures

Linux	Solaris	Comments
file_operations tty_operations block_device_operations etc.	modlinkage(9s) modldrv(9s) dev_ops(9s) cb_ops(9s)	Structures used to link driver into system and to specify various I/O operations in driver. All Solaris drivers have the Solaris listed structures. Linux uses a structure depending on the type of device.
struct device struct net_device	dev_info_t	Structure used to maintain device tree. In Solaris, fields may be accessed via DDI routines. To observe in Solaris, use mdb.
struct xxx_state	struct xxx_state	Device specific state. Solaris provides routines to help manage state. (ddi_soft_state_init(9f), etc.).
struct bio struct request	struct buf	Used for block device, regular file, and paging I/O. Contains transfer size, device address, memory address, and direction of I/O.
struct sk_buff	mblk_t dblk_t	Structures used to manage STREAMS messages. In Solaris, network drivers (and the network code) use mblk_t and dblk_t.

### Driver Entry Points

Linux	Solaris	Comments
module_init(fcn)	_init()	_init() - load and link driver module_init(fcn) - load, link, probe, and initialize driver/device. Done via xxx_attach on Solaris.
	xxx_attach()	Initialize an instance of the device. Must exist once in Solaris driver.
	xxx_probe()	Optional routine to determine if device exists. May be nulldev() on Solaris.
module_exit(fcn)	_fini()	_fini() - unload driver. May return EBUSY. module_exit(fcn) - reset device and (possibly) unload driver.
	xxx_detach()	Un-initialize instance(s) of the device. Must exist once in Solaris driver.
	xxx_getinfo()	Returns instance number or devinfo_t for given device number. Must exist for leaf node drivers.
	xxx_power()	Power manage device. Optional routine, nodev() if device is not to be power managed. Done via pm_register()/pm_unregister() via module_init(fcn)/module_exit(fcn) in Linux.
xxx_open()	xxx_open()	Called when device is opened. This is a file_operations in linux, cb_ops in Solaris. A Linux driver may have multiple file_operations, only one cb_ops. Solaris uses minor device number to differentiate. Code calling Linux driver uses the correct file_operations structure.
xxx_close()	xxx_close()	Called when device is closed. Also file_operations in Linux, cb_ops in Solaris.
xxx_read()	xxx_read()	Called when device is read. Also file_operations in Linux, cb_ops in Solaris.
xxx_write()	xxx_write()	Called when device is written. Also file_operations in Linux, cb_ops in Solaris.
xxx_ioctl()	xxx_ioctl()	Called for ioctl. Not all drivers need support this. For Solaris, must specify nodev in cb_ops if not supported.
xxx_mmap()	xxx_devmap() xxx_segmap() xxx_mmap()	Functions to support mmap(2). segmap(9e) and devmap(9e) give driver more control over context management, ie., driver can be notified of user context changes via driver callbacks.
xxx_poll()	xxx_poll()	Poll device for events. In Solaris, used with poll(2)/select(2).
	xxx_strategy()	Schedule I/O from/to device for disks. Used for read/write block device, raw (char) interface to block device, and for paging. On Linux, the closest routine is the init_command routine specified in the scsi_driver structure.

## Kernel Functions

### Synchronization

Linux	Solaris	Comments
atomic_add() atomic_sub() atomic_set() atomic_read() atomic_sub_and_test() atomic_inc() atomic_dec() atomic_dec_and_test() atomic_inc_and_test() atomic_add_negative()	atomic_add_16() atomic_add_32() atomic_add_long() atomic_add_64() atomic_add_16_nv() atomic_add_32_nv() atomic_add_long_nv() atomic_add_64_nv() atomic_inc() atomic_dec() etc.	Strictly speaking, the Solaris functions are not in the DDI/DKI. However, they exist and drivers do use them. The "_nv" forms return the new value. Subtraction is done by adding negative value. There are manual pages for some of these, (see atomic_add_32(3c)).
test_bit() set_bit() clear_bit() change_bit() test_and_set_bit() test_and_clear_bit() test_and_change_bit() atomic_clear_mask() atomic_set_mask()	atomic_or_uint() atomic_or_32() atomic_and_uint() atomic_and_32() cas32() cas64() etc.	As above, these are not defined in the DDI/DKI. Basically, atomic operations exist in both Linux and Solaris.
mb() rmb() wmb() smp_mb() smp_rmb() smp_wmb()		Solaris has no defined memory barrier operations. Existing synchronization mechanisms in Solaris use barriers where they are required. If you need to use a barrier separate from other synchronization, you may implement your own.
spin_lock_init() spin_lock() spin_unlock() spin_unlock_wait() spin_is_locked() spin_trylock()	mutex_init() mutex_enter() mutex_exit()  mutex_tryenter() mutex_destroy() mutex_owned()	mutex_init() is used to initialize spin locks and adaptive mutexes. Driver mutexes are adaptive or spin depending on the iblock_cookie_t passed to the mutex_init(). Note that this function and all mutex functions work the same on both uniprocessor and multiprocessor. (Kernel preemption can occur on single processor, so locks are needed even in that case).

**Synchronization (cont.)**

Linux	Solaris	Comments
rwlock_init() read_lock() write_lock() read_unlock write_unlock	rw_init() rw_enter()  rw_exit()  rw_tryenter() rw_tryupgrade() rw_downgrade() rw_destroy()	Reader/writer lock. In Solaris, writers get preference (i.e., readers can be starved).
Sema_init() up() down() down_trylock() down_interruptible()	sema_init() sema_v() sema_p() sema_tryup() sema_p_sig() sema_destroy()	Kernel semaphore operations.
wait_event() wait_event_interruptible() wait_event_timeout() wait_event_interruptible_timeout() wake_up() wake_up_all() wake_up_interruptible() etc.	cv_init() cv_wait() cv_wait_sig() cv_timedwait() cv_timedwait_sig() cv_signal() cv_broadcast()	Called condition variables in Solaris. Provides a "sleep/wakeup" mechanism. Use of these requires mutex.
Local_irq_disable() local_irq_enable() cli() sti()	spl<i>x</i>()	Mask interrupts. Solaris drivers should not need to use the spl routines. Instead, mutex_enter() will mask interrupts if needed (i.e., for spin mutexes only). Otherwise, device specific interrupt enable/disable can be done.

Dynamic Memory Allocation		
Linux	Solaris	Comments
kmem_cache_create()	kmem_cache_create()	Dynamic memory allocation in both Linux and Solaris use the SLAB allocator.
Kmem_cache_alloc()	kmem_cache_alloc()	Allocate space from a cache.
Kmalloc(size, flag)	kmem_alloc(size, flags	Dynamically allocate kernel memory. Flag is KM_SLEEP/KM_NOSLEEP in Solaris (i.e., if space not available).
Kmem_cache_free()	kmem_cache_free()	Return space to a cache.
Kfree()	kmem_free()	Free memory.

Timers		
init_timer() add_timer() del_timer()	timeout() untimeout() drv_usecwait() delay()	Solaris timeout() uses time delta in ticks, Linux uses absolute time in jiffies+ticks. Clock tick is same on both Solaris and Linux (typically, 10ms/tick). Both timers are not interval timers. They must be re-programmed if you want them to run again. Drv_usecwait() busy loops. Delay() sleeps.

Device Register Handling		
check_region() request_region() release_region() check_mem_region() request_mem_region() release_mem_region() ioremap() ioremap_nocache() iounmap()  inb/inw/inl() readb/readw/readl() outb/outw/outl() writeb/writew/writel() insb/insw/insl() outsb/outsw/outsl()	ddi_regs_map_setup() ddi_regs_map_free() ddi_peek8/16/32/64() ddi_poke8/16/32/64() ddi_get8/16/32/64()  ddi_put8/16/32/64()  ddi_rep_get8/16/32/64() ddi_rep_put8/16/32/64()	On Solaris, to access I/O ports, PCI configuration space, device I/O space, and device memory space, one uses ddi_regs_map_setup(9f) to obtain a <i>handle</i> for the space. The handle is passed to the various ddi_get/ddi_put functions to access the device. In effect, ddi_regs_map_setup(9f) and ddi_regs_map_free(9f) do the equivalent of the xxx_region and ioremap functions. ddi_peek and ddi_poke are used for probes, as they protect the system from bus timeouts if the device being probed does not exist. The ddi_get/ddi_put functions do no protection and are implemented to be as fast as possible.

## Interrupt Handling

request_irq()	ddi_add_intr()	Register interrupt handler for device instance. Called via xxx_attach(9e) entry point. The IRQ is hidden in Solaris, i.e., you don't specify the IRQ in ddi_add_intr(). IRQ is chosen by nexus driver or specified in the driver.conf(4) file.
free_irq()	ddi_remove_intr()	Remove the interrupt handler.
probe_irq_on() probe_irq_off()		Detect IRQ for a device. Should not be needed in Solaris, but if you poke around in the source, you might find something that works.
xxx_interrupt()	xxx_intr()	The interrupt handler itself. Solaris allows one argument to be passed. The argument is specified by the driver when the interrupt is registered via ddi_add_intr(). It is typically a pointer to state for the given instance of the device. State may contain anything needed to process the interrupt (as well as other stuff). Solaris interrupt handlers must return DDI_INTR_CLAIMED if the interrupt was handled.
DECLARE_TASKLET() tasklet_schedule()	ddi_add_softintr() ddi_trigger_softintr() ddi_remove_softintr()	Strictly speaking, the Solaris routines are implementing software interrupt handlers. Typically, they are used with devices that generate high priority (>10) interrupts. The driver's attach function registers a high priority handler via ddi_add_intr(), and a soft level (low priority) handler via ddi_add_softintr(). Then, when the device interrupts, the high level handler is called which in turn triggers the soft handler via ddi_trigger_softintr(). Note that there is a task queue mechanism in the kernel, but it is not directly visible in the DDI/DKI. (The timeout(9f) mechanism uses it). Also, timeout(9f) can be used to implement the Linux notion of a "bottom half" handler.
sti() cli() disable_irq() disable_irq_nosync() enable_irq()	ddi_enter_critical() ddi_exit_critical()	The ddi_enter_critical() and ddi_exit_critical disable/re-enable interrupts on the processor on which they are executed. The (undocumented and un-needed) spl routines do the same for a given interrupt level. The spl routines are not needed since they are not sufficient (i.e., you must still use mutexes), and the mutex mechanism masks interrupts where needed.

## DMA Handling

	<code>ddi_dma_alloc_handle()</code> <code>ddi_dma_free_handle()</code>	Allocate a "handle" to be used in subsequent DMA-related routines. This routine is passed a <code>ddi_dma_attr_t</code> that defines the DMA attributes of the device.
<code>dma_alloc_coherent()</code> <code>dma_free_coherent()</code> <code>dma_alloc_noncoherent()</code> <code>pci_alloc_consistent()</code> <code>pci_free_consistent()</code> <code>pci_map_sg()</code> <code>pci_unmap_sg()</code> <code>sg_dma_address()</code> <code>sg_dma_len()</code> <code>pci_dma_sync_sg()</code> etc.	<code>ddi_dma_mem_alloc()</code> <code>ddi_dma_addr_bind_handle()</code> <code>ddi_dma_unbind_handle()</code> <code>ddi_dma_buf_bind_handle()</code> <code>ddi_dma_mem_free()</code> <code>ddi_umem_iosetup()</code> <code>ddi_dma_sync()</code> etc.	Solaris DMA routines use the handle allocated by <code>ddi_dma_alloc_handle</code> . Note that there are no specific routines for PCI or any other bus. The bus specifics are handled within the <code>ddi_dma_*</code> calls by calling the nexus driver on which the device doing DMA is attached. The <code>dma_alloc_(non)coherent</code> and <code>dma_free_(non)coherent</code> accomplish much the same task as <code>ddi_dma_mem_alloc</code> .