

## **man pages section 9: DDI and DKI Kernel Functions**

Copyright © 1993, 2012, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée d'The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation.

# Contents

---

<b>Preface</b> .....	19
<b>Introduction</b> .....	23
Intro(9F) .....	24
<b>Kernel Functions for Drivers</b> .....	53
adjmsg(9F) .....	54
allobc(9F) .....	55
allobc_tmpl(9F) .....	58
anocancel(9F) .....	59
aphysio(9F) .....	60
ASSERT(9F) .....	62
atomic_add(9F) .....	63
atomic_and(9F) .....	65
atomic_bits(9F) .....	67
atomic_cas(9F) .....	68
atomic_dec(9F) .....	69
atomic_inc(9F) .....	71
atomic_ops(9F) .....	73
atomic_or(9F) .....	74
atomic_swap(9F) .....	76
audio_dev_add_control(9F) .....	77
audio_dev_add_engine(9F) .....	82
audio_dev_alloc(9F) .....	83
audio_dev_register(9F) .....	85
audio_dev_set_description(9F) .....	86
audio_dev_suspend(9F) .....	87

---

audio_dev_warn(9F) .....	88
audio_engine_alloc(9F) .....	89
audio_engine_set_private(9F) .....	90
audio_init_ops(9F) .....	91
backq(9F) .....	93
bcanput(9F) .....	94
bcmp(9F) .....	95
bcopy(9F) .....	96
bioaligned(9F) .....	98
bioclone(9F) .....	99
biodone(9F) .....	102
bioerror(9F) .....	104
biofini(9F) .....	105
bioinit(9F) .....	106
biomodified(9F) .....	107
bioreset(9F) .....	108
biosize(9F) .....	109
biowait(9F) .....	110
bp_copyin(9F) .....	111
bp_copyout(9F) .....	112
bp_mapin(9F) .....	113
bp_mapout(9F) .....	114
btop(9F) .....	115
btopr(9F) .....	116
bufcall(9F) .....	117
bzero(9F) .....	120
canput(9F) .....	121
canputnext(9F) .....	122
clrbuf(9F) .....	123
cmn_err(9F) .....	124
condvar(9F) .....	130
copyb(9F) .....	134
copyin(9F) .....	136
copymsg(9F) .....	138
copyout(9F) .....	140
csx_AccessConfigurationRegister(9F) .....	142

---

csx_ConvertSize(9F) .....	144
csx_ConvertSpeed(9F) .....	145
csx_CS_DDI_Info(9F) .....	146
csx_DeregisterClient(9F) .....	148
csx_DupHandle(9F) .....	149
csx_Error2Text(9F) .....	151
csx_Event2Text(9F) .....	152
csx_FreeHandle(9F) .....	153
csx_Get8(9F) .....	154
csx_GetFirstClient(9F) .....	155
csx_GetFirstTuple(9F) .....	157
csx_GetHandleOffset(9F) .....	159
csx_GetMappedAddr(9F) .....	160
csx_GetStatus(9F) .....	161
csx_GetTupleData(9F) .....	164
csx_MakeDeviceNode(9F) .....	166
csx_MapLogSocket(9F) .....	168
csx_MapMemPage(9F) .....	169
csx_ModifyConfiguration(9F) .....	170
csx_ModifyWindow(9F) .....	173
csx_Parse_CISTPL_BATTERY(9F) .....	175
csx_Parse_CISTPL_BYTEORDER(9F) .....	177
csx_Parse_CISTPL_CFTABLE_ENTRY(9F) .....	179
csx_Parse_CISTPL_CONFIG(9F) .....	184
csx_Parse_CISTPL_DATE(9F) .....	186
csx_Parse_CISTPL_DEVICE(9F) .....	187
csx_Parse_CISTPL_DEVICEGEO(9F) .....	190
csx_Parse_CISTPL_DEVICEGEO_A(9F) .....	192
csx_Parse_CISTPL_FORMAT(9F) .....	194
csx_Parse_CISTPL_FUNCE(9F) .....	196
csx_Parse_CISTPL_FUNCID(9F) .....	202
csx_Parse_CISTPL_GEOMETRY(9F) .....	204
csx_Parse_CISTPL_JEDEC_C(9F) .....	206
csx_Parse_CISTPL_LINKTARGET(9F) .....	208
csx_Parse_CISTPL_LONGLINK_A(9F) .....	210
csx_Parse_CISTPL_LONGLINK_MFC(9F) .....	212

---

csx_Parse_CISTPL_MANFID(9F) .....	214
csx_Parse_CISTPL_ORG(9F) .....	215
csx_Parse_CISTPL_SPCL(9F) .....	216
csx_Parse_CISTPL_SWIL(9F) .....	218
csx_Parse_CISTPL_VERS_1(9F) .....	219
csx_Parse_CISTPL_VERS_2(9F) .....	220
csx_ParseTuple(9F) .....	222
csx_Put8(9F) .....	224
csx_RegisterClient(9F) .....	225
csx_ReleaseConfiguration(9F) .....	228
csx_RepGet8(9F) .....	230
csx_RepPut8(9F) .....	232
csx_RequestConfiguration(9F) .....	234
csx_RequestIO(9F) .....	239
csx_RequestIRQ(9F) .....	244
csx_RequestSocketMask(9F) .....	246
csx_RequestWindow(9F) .....	248
csx_ResetFunction(9F) .....	253
csx_SetEventMask(9F) .....	254
csx_SetHandleOffset(9F) .....	256
csx_ValidateCIS(9F) .....	257
datams(9F) .....	258
DB_BASE(9F) .....	259
ddi_add_event_handler(9F) .....	260
ddi_add_intr(9F) .....	262
ddi_add_softintr(9F) .....	265
ddi_binding_name(9F) .....	272
ddi_btop(9F) .....	273
ddi_can_receive_sig(9F) .....	274
ddi_cb_register(9F) .....	275
ddi_check_acc_handle(9F) .....	290
ddi_copyin(9F) .....	292
ddi_copyout(9F) .....	295
ddi_create_minor_node(9F) .....	298
ddi_cred(9F) .....	300
ddi_device_copy(9F) .....	302

---

ddi_device_zero(9F) .....	304
ddi_devid_compare(9F) .....	305
ddi_dev_is_sid(9F) .....	309
ddi_dev_nintrs(9F) .....	310
ddi_dev_nregs(9F) .....	311
ddi_dev_regsz(9F) .....	312
ddi_dev_report_fault(9F) .....	313
ddi_dma_addr_bind_handle(9F) .....	316
ddi_dma_addr_setup(9F) .....	320
ddi_dma_alloc_handle(9F) .....	322
ddi_dma_buf_bind_handle(9F) .....	324
ddi_dma_buf_setup(9F) .....	328
ddi_dma_burstsizes(9F) .....	330
ddi_dma_coff(9F) .....	331
ddi_dma_curwin(9F) .....	332
ddi_dma_devalign(9F) .....	333
ddi_dmae(9F) .....	334
ddi_dma_free(9F) .....	338
ddi_dma_free_handle(9F) .....	339
ddi_dma_get_attr(9F) .....	340
ddi_dma_getwin(9F) .....	341
ddi_dma_htoc(9F) .....	343
ddi_dma_mem_alloc(9F) .....	344
ddi_dma_mem_free(9F) .....	347
ddi_dma_movwin(9F) .....	348
ddi_dma_nextcookie(9F) .....	350
ddi_dma_nextseg(9F) .....	352
ddi_dma_nextwin(9F) .....	354
ddi_dma_numwin(9F) .....	356
ddi_dma_segtocookie(9F) .....	357
ddi_dma_set_sbus64(9F) .....	359
ddi_dma_setup(9F) .....	360
ddi_dma_sync(9F) .....	362
ddi_dma_unbind_handle(9F) .....	364
ddi_driver_major(9F) .....	365
ddi_driver_name(9F) .....	366

ddi_enter_critical(9F) .....	367
ddi_ffs(9F) .....	368
ddi_fm_acc_err_clear(9F) .....	369
ddi_fm_acc_err_get(9F) .....	370
ddi_fm_ereport_post(9F) .....	372
ddi_fm_handler_register(9F) .....	374
ddi_fm_init(9F) .....	376
ddi_fm_service_impact(9F) .....	378
ddi_get8(9F) .....	379
ddi_get_cred(9F) .....	381
ddi_get_devstate(9F) .....	382
ddi_get_driver_private(9F) .....	384
ddi_get_eventcookie(9F) .....	385
ddi_getiminor(9F) .....	386
ddi_get_instance(9F) .....	387
ddi_get_kt_did(9F) .....	388
ddi_get_lbolt(9F) .....	389
ddi_get_parent(9F) .....	390
ddi_get_pid(9F) .....	391
ddi_get_time(9F) .....	392
ddi_in_panic(9F) .....	393
ddi_intr_add_handler(9F) .....	394
ddi_intr_add_softint(9F) .....	396
ddi_intr_alloc(9F) .....	403
ddi_intr_dup_handler(9F) .....	406
ddi_intr_enable(9F) .....	411
ddi_intr_get_cap(9F) .....	414
ddi_intr_get_hilevel_pri(9F) .....	416
ddi_intr_get_nintrs(9F) .....	418
ddi_intr_get_pending(9F) .....	420
ddi_intr_get_pri(9F) .....	422
ddi_intr_get_supported_types(9F) .....	424
ddi_intr_hilevel(9F) .....	426
ddi_intr_set_mask(9F) .....	427
ddi_intr_set_nreq(9F) .....	429
ddi_io_get8(9F) .....	431



---

ddi_iomin(9F) .....	432
ddi_iopb_alloc(9F) .....	433
ddi_io_put8(9F) .....	435
ddi_io_rep_get8(9F) .....	436
ddi_io_rep_put8(9F) .....	438
ddi_log_sysevent(9F) .....	440
ddi_map_regs(9F) .....	444
ddi_mem_alloc(9F) .....	446
ddi_mem_get8(9F) .....	448
ddi_mem_put8(9F) .....	450
ddi_mem_rep_get8(9F) .....	452
ddi_mem_rep_put8(9F) .....	454
ddi_mmap_get_model(9F) .....	456
ddi_model_convert_from(9F) .....	458
ddi_modopen(9F) .....	460
ddi_node_name(9F) .....	465
ddi_no_info(9F) .....	466
ddi_peek(9F) .....	467
ddi_periodic_add(9F) .....	469
ddi_periodic_delete(9F) .....	472
ddi_poke(9F) .....	474
ddi_prop_create(9F) .....	476
ddi_prop_exists(9F) .....	481
ddi_prop_get_int(9F) .....	483
ddi_prop_lookup(9F) .....	485
ddi_prop_op(9F) .....	490
ddi_prop_update(9F) .....	494
ddi_put8(9F) .....	498
ddi_regs_map_free(9F) .....	500
ddi_regs_map_setup(9F) .....	501
ddi_remove_event_handler(9F) .....	503
ddi_remove_minor_node(9F) .....	504
ddi_removing_power(9F) .....	505
ddi_rep_get8(9F) .....	507
ddi_report_dev(9F) .....	509
ddi_rep_put8(9F) .....	510

ddi_root_node(9F) .....	512
ddi_segmap(9F) .....	513
ddi_slaveonly(9F) .....	516
ddi_soft_state(9F) .....	517
ddi_strtol(9F) .....	522
ddi_strtoll(9F) .....	524
ddi_strtoul(9F) .....	526
ddi_umem_alloc(9F) .....	528
ddi_umem_iosetup(9F) .....	530
ddi_umem_lock(9F) .....	532
delay(9F) .....	534
devmap_default_access(9F) .....	536
devmap_devmem_setup(9F) .....	539
devmap_do_ctxmgt(9F) .....	543
devmap_set_ctx_timeout(9F) .....	546
devmap_setup(9F) .....	547
devmap_unload(9F) .....	549
disksort(9F) .....	551
dlbindack(9F) .....	552
drv_getparm(9F) .....	554
drv_hztousec(9F) .....	556
drv_priv(9F) .....	557
drv_usectohz(9F) .....	558
drv_usecwait(9F) .....	559
dupb(9F) .....	560
dupmsg(9F) .....	563
enableok(9F) .....	564
esballoc(9F) .....	565
esbcall(9F) .....	567
flushband(9F) .....	568
flushq(9F) .....	569
freeb(9F) .....	571
freemsg(9F) .....	572
freerbuf(9F) .....	573
freezestr(9F) .....	574
geterror(9F) .....	575

---

gethrtime(9F) .....	576
getmajor(9F) .....	577
getminor(9F) .....	578
get_pktiopb(9F) .....	579
getq(9F) .....	581
getrbuf(9F) .....	582
gld(9F) .....	583
hat_getkpfnum(9F) .....	586
hook_alloc(9F) .....	587
hook_free(9F) .....	588
id32_alloc(9F) .....	589
inb(9F) .....	590
insq(9F) .....	592
IOC_CONVERT_FROM(9F) .....	594
kiconv(9F) .....	595
kiconv_close(9F) .....	599
kiconv_open(9F) .....	601
kiconvstr(9F) .....	605
kmem_alloc(9F) .....	609
kmem_cache_create(9F) .....	611
kstat_create(9F) .....	619
kstat_delete(9F) .....	621
kstat_install(9F) .....	622
kstat_named_init(9F) .....	623
kstat_queue(9F) .....	624
ldi_add_event_handler(9F) .....	626
ldi_aread(9F) .....	628
ldi_devmap(9F) .....	629
ldi_dump(9F) .....	630
ldi_ev_finalize(9F) .....	631
ldi_ev_get_cookie(9F) .....	633
ldi_ev_get_type(9F) .....	634
ldi_ev_notify(9F) .....	635
ldi_ev_register_callbacks(9F) .....	637
ldi_ev_remove_callbacks(9F) .....	645
ldi_get_dev(9F) .....	646

---

ldi_get_eventcookie(9F) .....	647
ldi_get_size(9F) .....	648
ldi_ident_from_dev(9F) .....	649
ldi_ioctl(9F) .....	650
ldi_open_by_dev(9F) .....	652
ldi_poll(9F) .....	655
ldi_prop_exists(9F) .....	657
ldi_prop_get_int(9F) .....	659
ldi_prop_lookup_int_array(9F) .....	661
ldi_putmsg(9F) .....	666
ldi_read(9F) .....	667
ldi_remove_event_handler(9F) .....	668
ldi_strategy(9F) .....	669
linkb(9F) .....	670
list_create(9F) .....	671
mac(9F) .....	673
mac_hcksum_get(9F) .....	676
mac_lso_get(9F) .....	678
mac_prop_info_set_perm(9F) .....	679
makecom(9F) .....	681
makedevice(9F) .....	683
max(9F) .....	684
MBLKHEAD(9F) .....	685
mcopyin(9F) .....	687
mcopymsg(9F) .....	688
mcopyout(9F) .....	689
membar_ops(9F) .....	690
memchr(9F) .....	692
merror(9F) .....	694
mexchange(9F) .....	695
min(9F) .....	696
mioc2ack(9F) .....	697
miocack(9F) .....	698
miocnak(9F) .....	699
miocpullup(9F) .....	700
mkiocb(9F) .....	701

---

mod_install(9F) .....	704
msgdsize(9F) .....	706
msgpullup(9F) .....	707
msgsize(9F) .....	708
mt-streams(9F) .....	709
mutex(9F) .....	711
net_event_notify_register(9F) .....	714
net_getifname(9F) .....	716
net_getlifaddr(9F) .....	717
net_getmtu(9F) .....	719
net_getnetid(9F) .....	720
net_getpmtuenabled(9F) .....	721
net_hook_register(9F) .....	722
net_hook_unregister(9F) .....	724
netinfo(9F) .....	725
net_inject(9F) .....	726
net_inject_alloc(9F) .....	728
net_inject_free(9F) .....	729
net_instance_alloc(9F) .....	730
net_instance_free(9F) .....	731
net_instance_notify_register(9F) .....	732
net_instance_register(9F) .....	734
net_instance_unregister(9F) .....	735
net_ispartialchecksum(9F) .....	736
net_isvalidchecksum(9F) .....	737
net_kstat_create(9F) .....	738
net_kstat_delete(9F) .....	740
net_lifgetnext(9F) .....	741
net_netidtozonid(9F) .....	743
net_phygetnext(9F) .....	744
net_phylookup(9F) .....	746
net_protocol_lookup(9F) .....	747
net_protocol_notify_register(9F) .....	748
net_protocol_release(9F) .....	750
net_protocol_walk(9F) .....	751
net_routeto(9F) .....	752

---

net_zoneidtonetid(9F) .....	753
nochpoll(9F) .....	754
nodev(9F) .....	755
noenable(9F) .....	756
nulldev(9F) .....	757
nvlist_add_boolean(9F) .....	758
nvlist_alloc(9F) .....	761
nvlist_lookup_boolean(9F) .....	767
nvlist_lookup_nvpair(9F) .....	771
nvlist_next_nvpair(9F) .....	772
nvlist_remove(9F) .....	774
nvpair_value_byte(9F) .....	775
OTHERQ(9F) .....	777
outb(9F) .....	778
packet_events(9F) .....	780
pci_config_get8(9F) .....	783
pci_config_setup(9F) .....	785
pci_ereport_setup(9F) .....	786
pci_param_free(9F) .....	789
pci_param_get(9F) .....	790
pci_param_get_ioctl(9F) .....	792
pci_plist_get(9F) .....	793
pci_plist_getvf(9F) .....	794
pci_plist_lookup(9F) .....	795
pci_report_pmcap(9F) .....	797
pci_save_config_regs(9F) .....	799
pciv_send(9F) .....	801
pciv_vf_config(9F) .....	803
physio(9F) .....	805
pm_busy_component(9F) .....	807
pm_power_has_changed(9F) .....	809
pm_raise_power(9F) .....	812
pm_trans_check(9F) .....	816
pollwakeup(9F) .....	818
priv_getbyname(9F) .....	819
priv_policy(9F) .....	821

---

proc_signal(9F) .....	823
ptob(9F) .....	825
pullupmsg(9F) .....	826
put(9F) .....	828
putbq(9F) .....	829
putctl1(9F) .....	830
putctl(9F) .....	831
putnext(9F) .....	833
putnextctl1(9F) .....	834
putnextctl(9F) .....	835
putq(9F) .....	837
qassociate(9F) .....	838
qbufcall(9F) .....	840
qenable(9F) .....	841
qprocson(9F) .....	842
qreply(9F) .....	843
qsize(9F) .....	845
qtimeout(9F) .....	846
queue(9F) .....	847
qunbufcall(9F) .....	862
quntimeout(9F) .....	863
qwait(9F) .....	864
qwriter(9F) .....	866
RD(9F) .....	867
rmalloc(9F) .....	868
rmallocmap(9F) .....	871
rmalloc_wait(9F) .....	872
rmfree(9F) .....	873
rmvb(9F) .....	874
rmvq(9F) .....	876
rwlock(9F) .....	878
SAMESTR(9F) .....	881
scsi_abort(9F) .....	882
scsi_alloc_consistent_buf(9F) .....	883
scsi_cname(9F) .....	885
scsi_destroy_pkt(9F) .....	887

---

scsi_dmaget(9F) .....	888
scsi_errmsg(9F) .....	890
scsi_ext_sense_fields(9F) .....	893
scsi_find_sense_descr(9F) .....	895
scsi_free_consistent_buf(9F) .....	896
scsi_get_device_type_scsi_options(9F) .....	897
scsi_get_device_type_string(9F) .....	899
scsi_hba_attach_setup(9F) .....	901
scsi_hba_init(9F) .....	904
scsi_hba_lookup_capstr(9F) .....	905
scsi_hba_pkt_alloc(9F) .....	907
scsi_hba_pkt_comp(9F) .....	909
scsi_hba_probe(9F) .....	910
scsi_hba_tran_alloc(9F) .....	911
scsi_ifgetcap(9F) .....	912
scsi_init_pkt(9F) .....	916
scsi_log(9F) .....	920
scsi_pktalloc(9F) .....	921
scsi_poll(9F) .....	923
scsi_probe(9F) .....	924
scsi_reset(9F) .....	926
scsi_reset_notify(9F) .....	927
scsi_sense_key(9F) .....	928
scsi_setup_cdb(9F) .....	929
scsi_slave(9F) .....	930
scsi_sync_pkt(9F) .....	932
scsi_transport(9F) .....	933
scsi_unprobe(9F) .....	934
scsi_validate_sense(9F) .....	935
scsi_vu_errmsg(9F) .....	936
semaphore(9F) .....	939
sof_bypass(9F) .....	941
sof_flowctrl_data_in(9F) .....	942
sof_inject_data_in(9F) .....	943
sof_mblk_msghdr(9F) .....	944
sof_newconn_ready(9F) .....	947



---

sof_register(9F) .....	948
sprintf(9F) .....	949
stoi(9F) .....	951
string(9F) .....	952
strlog(9F) .....	956
strqget(9F) .....	958
strqset(9F) .....	959
STRUCT_DECL(9F) .....	960
swab(9F) .....	965
taskq(9F) .....	966
testb(9F) .....	969
timeout(9F) .....	971
tree(9F) .....	973
u8_strcmp(9F) .....	980
u8_textprep_str(9F) .....	984
u8_validate(9F) .....	988
uconv_u16tou32(9F) .....	992
uiomove(9F) .....	998
unbufcall(9F) .....	999
unlinkb(9F) .....	1000
untimeout(9F) .....	1001
ureadc(9F) .....	1003
usb_alloc_request(9F) .....	1004
usb_client_attach(9F) .....	1007
usb_clr_feature(9F) .....	1010
usb_create_pm_components(9F) .....	1012
usb_get_addr(9F) .....	1014
usb_get_alt_if(9F) .....	1015
usb_get_cfg(9F) .....	1019
usb_get_current_frame_number(9F) .....	1022
usb_get_dev_data(9F) .....	1024
usb_get_max_pkts_per_isoc_request(9F) .....	1028
usb_get_status(9F) .....	1030
usb_get_string_descr(9F) .....	1032
usb_handle_remote_wakeup(9F) .....	1034
usb_lookup_ep_data(9F) .....	1035

usb_parse_data(9F) .....	1037
usb_pipe_bulk_xfer(9F) .....	1039
usb_pipe_close(9F) .....	1042
usb_pipe_ctrl_xfer(9F) .....	1045
usb_pipe_drain_reqs(9F) .....	1051
usb_pipe_get_max_bulk_transfer_size(9F) .....	1054
usb_pipe_get_state(9F) .....	1056
usb_pipe_intr_xfer(9F) .....	1058
usb_pipe_isoc_xfer(9F) .....	1064
usb_pipe_open(9F) .....	1070
usb_pipe_reset(9F) .....	1074
usb_pipe_set_private(9F) .....	1077
usb_register_hotplug_cbs(9F) .....	1079
usb_reset_device(9F) .....	1081
uwritec(9F) .....	1085
va_arg(9F) .....	1086
vsprintf(9F) .....	1088
WR(9F) .....	1091

# Preface

---

Both novice users and those familiar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

## Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9E describes the DDI (Device Driver Interface)/DKI (Driver/Kernel Interface), DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report,

there is no BUGS section. See the intro pages for more information and detail about each section, and [man\(1\)](#) for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	<p>This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.</p> <p>The following special characters are used in this section:</p> <ul style="list-style-type: none"><li>[ ] Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.</li><li>. . . Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, “filename . . .”.</li><li>  Separator. Only one of the arguments separated by this character can be specified at a time.</li><li>{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.</li></ul>
PROTOCOL	This section occurs only in subsection 3R to indicate the protocol description file.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.
IOCTL	This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the <a href="#">ioctl(2)</a> system call is called <code>ioctl</code> and generates its own heading. <code>ioctl</code> calls for a specific device are listed alphabetically (on the man page for that specific device).

---

	<p><code>ioctl</code> calls are used for a particular class of devices all of which have an <code>io</code> ending, such as <code>mtio(7I)</code>.</p>
OPTIONS	<p>This section lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.</p>
OPERANDS	<p>This section lists the command operands and describes how they affect the actions of the command.</p>
OUTPUT	<p>This section describes the output – standard output, standard error, or output files – generated by the command.</p>
RETURN VALUES	<p>If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or -1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.</p>
ERRORS	<p>On failure, most functions place an error code in the global variable <code>errno</code> indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.</p>
USAGE	<p>This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality:</p> <ul style="list-style-type: none"><li>Commands</li><li>Modifiers</li><li>Variables</li><li>Expressions</li><li>Input Grammar</li></ul>
EXAMPLES	<p>This section provides examples of usage or of how to use a command or function. Wherever possible a complete</p>

	<p>example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as <code>example%</code>, or if the user must be superuser, <code>example#</code>. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.</p>
ENVIRONMENT VARIABLES	<p>This section lists any environment variables that the command or function affects, followed by a brief description of the effect.</p>
EXIT STATUS	<p>This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.</p>
FILES	<p>This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.</p>
ATTRIBUTES	<p>This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See <a href="#">attributes(5)</a> for more information.</p>
SEE ALSO	<p>This section lists references to other man pages, in-house documentation, and outside publications.</p>
DIAGNOSTICS	<p>This section lists diagnostic messages with a brief explanation of the condition causing the error.</p>
WARNINGS	<p>This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.</p>
NOTES	<p>This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.</p>
BUGS	<p>This section describes known bugs and, wherever possible, suggests workarounds.</p>

**R E F E R E N C E**

**Introduction**

**Name** Intro – introduction to DDI/DKI functions

**Description** Section 9F describes the kernel functions available for use by device drivers. See [Intro\(9E\)](#) for an overview of device driver interfaces.

In this section, the information for each driver function is organized under the following headings:

- **NAME** summarizes the function's purpose.
- **SYNOPSIS** shows the syntax of the function's entry point in the source code. `#include` directives are shown for required headers.
- **INTERFACE LEVEL** describes any architecture dependencies.
- **ARGUMENTS** describes any arguments required to invoke the function.
- **DESCRIPTION** describes general information about the function.
- **RETURN VALUES** describes the return values and messages that can result from invoking the function.
- **CONTEXT** indicates from which driver context (user, kernel, interrupt, or high-level interrupt) the function can be called.
- A driver function has *user context* if it was directly invoked because of a user thread. The [read\(9E\)](#) entry point of the driver, invoked by a [read\(2\)](#) system call, has user context.
- A driver function has *kernel context* if was invoked by some other part of the kernel. In a block device driver, the [strategy\(9E\)](#) entry point may be called by the page daemon to write pages to the device. The page daemon has no relation to the current user thread, so in this case [strategy\(9E\)](#) has kernel context.
- *Interrupt context* is kernel context, but also has an interrupt level associated with it. Driver interrupt routines have interrupt context.

Note that a mutex acquired in user or kernel context that can also be acquired in interrupt context means that the user or kernel context thread holding that mutex is subject to all the restrictions imposed by interrupt context, for the duration of the ownership of that mutex. Please see the [mutex\(9F\)](#) man page for a more complete discussion of proper mutex handling for drivers.

- *High-level interrupt context* is a more restricted form of interrupt context. If a driver interrupt priority returned from [ddi\\_intr\\_get\\_pri\(9F\)](#) is greater than the priority returned from [ddi\\_intr\\_get\\_hilevel\\_pri\(9F\)](#) this indicates the interrupt handler will run in high-level interrupt context. These interrupt routines are only allowed to call [ddi\\_intr\\_trigger\\_softint\(9F\)](#), [mutex\\_enter\(9F\)](#), and [mutex\\_exit\(9F\)](#). Furthermore, [mutex\\_enter\(9F\)](#) and [mutex\\_exit\(9F\)](#) may only be called on mutexes initialized with the interrupt priority returned by [ddi\\_intr\\_get\\_pri\(9F\)](#).
- **SEE ALSO** indicates functions that are related by usage and sources, and which can be referred to for further information.
- **EXAMPLES** shows how the function can be used in driver code.



Every driver MUST include `<sys/ddi.h>` and `<sys/sunddi.h>`, in that order, and as the last files the driver includes.

### Streams Kernel Function Summary

The following table summarizes the STREAMS functions described in this section.

Routine	Type
<code>adjmsg</code>	DDI/DKI
<code>allocb</code>	DDI/DKI
<code>allocb_tmpl</code>	Solaris DDI
<code>backq</code>	DDI/DKI
<code>bcanput</code>	DDI/DKI
<code>bcanputnext</code>	DDI/DKI
<code>bufcall</code>	DDI/DKI
<code>canput</code>	DDI/DKI
<code>canputnext</code>	DDI/DKI
<code>clrbuf</code>	DDI/DKI
<code>copyb</code>	DDI/DKI
<code>copymsg</code>	DDI/DKI
<code>DB_BASE</code>	Solaris DDI
<code>DB_LIM</code>	Solaris DDI
<code>DB_REF</code>	Solaris DDI
<code>DB_TYPE</code>	Solaris DDI
<code>datamsg</code>	DDI/DKI
<code>dupb</code>	DDI/DKI
<code>dupmsg</code>	DDI/DKI
<code>enableleok</code>	DDI/DKI
<code>esballoc</code>	DDI/DKI
<code>esbcall</code>	DDI/DKI
<code>flushband</code>	DDI/DKI
<code>flushq</code>	DDI/DKI
<code>freeb</code>	DDI/DKI

---

<b>Routine</b>	<b>Type</b>
freemsg	DDI/DKI
freezestr	DDI/DKI
getq	DDI/DKI
IOC_CONVER_FROM	Solaris DDI
insq	DDI/DKI
linkb	DDI/DKI
MBLKHEAD	Solaris DDI
MBLKIN	Solaris DDI
MBLKL	Solaris DDI
MBLKSIZE	Solaris DDI
MBLKTAIL	Solaris DDI
mcopyin	Solaris DDI
mcopymsg	Solaris DDI
mcopyout	Solaris DDI
merror	Solaris DDI
mexchange	Solaris DDI
mioc2ack	Solaris DDI
miocack	Solaris DDI
mexchange	Solaris DDI
miocpullup	Solaris DDI
mkiocb	Solaris DDI
msgdsize	DDI/DKI
msgpullup	DDI/DKI
msgsize	Solaris DDI
mt-streams	Solaris DDI
noenable	DDI/DKI
OTHERQ	DDI/DKI
pullupmsg	DDI/DKI

---

Routine	Type
put	DDI/DKI
putbq	DDI/DKI
putctl	DDI/DKI
putctl1	DDI/DKI
putnext	DDI/DKI
putnextctl	DDI/DKI
putq	DDI/DKI
qassociate	Solaris DDI
qbufcall	Solaris DDI
qenable	DDI/DKI
qprocson	DDI/DKI
qprocsoff	DDI/DKI
qreply	DDI/DKI
qsize	DDI/DKI
qtimeout	Solaris DDI
qunbufcall	Solaris DDI
quntimeout	Solaris DDI
qwait	Solaris DDI
qwait_sig	Solaris DDI
qwriter	Solaris DDI
RD	DDI/DKI
rmvb	DDI/DKI
rmvq	DDI/DKI
SAMESTR	DDI/DKI
strlog	DDI/DKI
strqget	DDI/DKI
strqset	DDI/DKI
testb	DDI/DKI

---

<b>Routine</b>	<b>Type</b>
unbufcall	DDI/DKI
unfreezestr	DDI/DKI
unlinkb	DDI/DKI
WR	DDI/DKI

The following table summarizes the functions not specific to STREAMS.

---

<b>Routine</b>	<b>Type</b>
ASSERT	DDI/DKI
anocancel	Solaris DDI
aphysio	Solaris DDI
atomic_add	DDI/DKI
atomic_and	DDI/DKI
atomic_bits	DDI/DKI
atomic_cas	DDI/DKI
atomic_dec	DDI/DKI
atomic_inc	DDI/DKI
atomic_ops	DDI/DKI
atomic_or	DDI/DKI
atomic_swap	DDI/DKI
bcmp	DDI/DKI
bcopy	DDI/DKI
bioclone	Solaris DDI
biodone	DDI/DKI
biofini	Solaris DDI
bioinit	Solaris DDI
biomodified	Solaris DDI
biosize	Solaris DDI
bioerror	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
bioreset	Solaris DDI
biowait	DDI/DKI
bp_copyin	DDI/DKI
bp_copyout	DDI/DKI
bp_mapin	DDI/DKI
bp_mapout	DDI/DKI
btop	DDI/DKI
btopr	DDI/DKI
bzero	DDI/DKI
cmn_err	DDI/DKI
condvar	Solaris DDI
copyin	DDI/DKI
copyout	DDI/DKI
csx_AccessConfigurationRegister	Solaris DDI
csx_ConvertSize	Solaris DDI
csx_ConvertSpeed	Solaris DDI
csx_CS_DDI_Info	Solaris DDI
csx_DeregisterClient	Solaris DDI
csx_DupHandle	Solaris DDI
csx_Error2Text	Solaris DDI
csx_Event2Text	Solaris DDI
csx_FreeHandle	Solaris DDI
csx_Get8	Solaris DDI
csx_GetFirstClient	Solaris DDI
csx_GetFirstTuple	Solaris DDI
csx_GetHandleOffset	Solaris DDI
csx_GetMappedAddr	Solaris DDI
csx_GetStatus	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>csx_GetTupleData</code>	Solaris DDI
<code>csx_MakeDeviceNode</code>	Solaris DDI
<code>csx_MapLogSocket</code>	Solaris DDI
<code>csx_MapMemPage</code>	Solaris DDI
<code>csx_ModifyConfiguration</code>	Solaris DDI
<code>csx_ModifyWindow</code>	Solaris DDI
<code>csx_Parse_CISTPL_BATTERY</code>	Solaris DDI
<code>csx_Parse_CISTPL_BYTEORDER</code>	Solaris DDI
<code>csx_Parse_CISTPL_CFTABLE_ENTRY</code>	Solaris DDI
<code>csx_Parse_CISTPL_CONFIG</code>	Solaris DDI
<code>csx_Parse_CISTPL_DATE</code>	Solaris DDI
<code>csx_Parse_CISTPL_DEVICE</code>	Solaris DDI
<code>csx_Parse_CISTPL_DEVICEGEO</code>	Solaris DDI
<code>csx_Parse_CISTPL_DEVICEGEO_A</code>	Solaris DDI
<code>csx_Parse_CISTPL_FORMAT</code>	Solaris DDI
<code>csx_Parse_CISTPL_FUNCE</code>	Solaris DDI
<code>csx_Parse_CISTPL_FUNCID</code>	Solaris DDI
<code>csx_Parse_CISTPL_GEOMETRY</code>	Solaris DDI
<code>csx_Parse_CISTPL_JEDEC_C</code>	Solaris DDI
<code>csx_Parse_CISTPL_LINKTARGET</code>	Solaris DDI
<code>csx_Parse_CISTPL_LONGLINK_A</code>	Solaris DDI
<code>csx_Parse_CISTPL_LONGLINK_MFC</code>	Solaris DDI
<code>csx_Parse_CISTPL_MANFID</code>	Solaris DDI
<code>csx_Parse_CISTPL_ORG</code>	Solaris DDI
<code>csx_Parse_CISTPL_SPCL</code>	Solaris DDI
<code>csx_Parse_CISTPL_SWIL</code>	Solaris DDI
<code>csx_Parse_CISTPL_VERS_1</code>	Solaris DDI
<code>csx_Parse_CISTPL_VERS_2</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>csx_ParseTuple</code>	Solaris DDI
<code>csx_Put8</code>	Solaris DDI
<code>csx_RegisterClient</code>	Solaris DDI
<code>csx_ReleaseConfiguration</code>	Solaris DDI
<code>csx_RepGet8</code>	Solaris DDI
<code>csx_RepPut8</code>	Solaris DDI
<code>csx_RequestConfiguration</code>	Solaris DDI
<code>csx_RequestIO</code>	Solaris DDI
<code>csx_RequestIRQ</code>	Solaris DDI
<code>csx_RequestSocketMask</code>	Solaris DDI
<code>csx_RequestWindow</code>	Solaris DDI
<code>csx_ResetFunction</code>	Solaris DDI
<code>csx_SetEventMask</code>	Solaris DDI
<code>csx_SetHandleOffset</code>	Solaris DDI
<code>csx_ValidateCIS</code>	Solaris DDI
<code>cv_broadcast</code>	Solaris DDI
<code>cv_destroy</code>	Solaris DDI
<code>cv_init</code>	Solaris DDI
<code>cv_signal</code>	Solaris DDI
<code>cv_timedwait</code>	Solaris DDI
<code>cv_wait</code>	Solaris DDI
<code>cv_wait_sig</code>	Solaris DDI
<code>ddi_add_event_handler</code>	Solaris DDI
<code>ddi_add_intr</code>	Solaris DDI
<code>ddi_add_softintr</code>	Solaris DDI
<code>ddi_binding_name</code>	Solaris DDI
<code>ddi_btop</code>	Solaris DDI
<code>ddi_btopr</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>ddi_can_receive_sig</code>	Solaris DDI
<code>ddi_check_acc_handle</code>	Solaris DDI
<code>ddi_copyin</code>	Solaris DDI
<code>ddi_copyout</code>	Solaris DDI
<code>ddi_create_minor_node</code>	Solaris DDI
<code>ddi_cred</code>	Solaris DDI
<code>ddi_dev_is_sid</code>	Solaris DDI
<code>ddi_dev_nintrs</code>	Solaris DDI
<code>ddi_dev_nregs</code>	Solaris DDI
<code>ddi_dev_regsz</code>	Solaris DDI
<code>ddi_device_copy</code>	Solaris DDI
<code>ddi_device_zero</code>	Solaris DDI
<code>ddi_devmap_segmap</code>	Solaris DDI
<code>ddi_dma_addr_bind_handle</code>	Solaris DDI
<code>ddi_dma_addr_setup</code>	Solaris DDI
<code>ddi_dma_alloc_handle</code>	Solaris DDI
<code>ddi_dma_buf_bind_handle</code>	Solaris DDI
<code>ddi_dma_buf_setup</code>	Solaris DDI
<code>ddi_dma_burstsizes</code>	Solaris DDI
<code>ddi_dma_coff</code>	Solaris SPARC DDI
<code>ddi_dma_curwin</code>	Solaris SPARC DDI
<code>ddi_dma_devalign</code>	Solaris DDI
<code>ddi_dma_free</code>	Solaris DDI
<code>ddi_dma_free_handle</code>	Solaris DDI
<code>ddi_dma_getwin</code>	Solaris DDI
<code>ddi_dma_get_attr</code>	Solaris DDI
<code>ddi_dma_htoc</code>	Solaris SPARC DDI
<code>ddi_dma_mem_alloc</code>	Solaris DDI



---

<b>Routine</b>	<b>Type</b>
ddi_dma_mem_free	Solaris DDI
ddi_dma_movwin	Solaris SPARC DDI
ddi_dma_nextcookie	Solaris DDI
ddi_dma_nextseg	Solaris DDI
ddi_dma_nextwin	Solaris DDI
ddi_dma_numwin	Solaris DDI
ddi_dma_segtocookie	Solaris DDI
ddi_dma_set_sbus64	Solaris DDI
ddi_dma_setup	Solaris DDI
ddi_dma_sync	Solaris DDI
ddi_dma_unbind_handle	Solaris DDI
ddi_dmae	Solaris x86 DDI
ddi_dmae_1stparty	Solaris x86 DDI
ddi_dmae_alloc	Solaris x86 DDI
ddi_dmae_disable	Solaris x86 DDI
ddi_dmae_enable	Solaris x86 DDI
ddi_dmae_getattr	Solaris x86 DDI
ddi_dmae_getcnt	Solaris x86 DDI
ddi_dmae_getlim	Solaris x86 DDI
ddi_dmae_prog	Solaris x86 DDI
ddi_dmae_release	Solaris x86 DDI
ddi_dmae_stop	Solaris x86 DDI
ddi_driver_major	Solaris DDI
ddi_driver_name	Solaris DDI
ddi_enter_critical	Solaris DDI
ddi_exit_critical	Solaris DDI
ddi_ffs	Solaris DDI
ddi_fls	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>ddi_fm_acc_err_clear</code>	Solaris DDI
<code>ddi_fm_acc_err_get</code>	Solaris DDI
<code>ddi_fm_ereport_post</code>	Solaris DDI
<code>ddi_fm_handler_register</code>	Solaris DDI
<code>ddi_fm_init</code>	Solaris DDI
<code>ddi_fm_service_impact</code>	Solaris DDI
<code>ddi_get16</code>	Solaris DDI
<code>ddi_get32</code>	Solaris DDI
<code>ddi_get64</code>	Solaris DDI
<code>ddi_get8</code>	Solaris DDI
<code>ddi_get_cred</code>	Solaris DDI
<code>ddi_get_devstate</code>	Solaris DDI
<code>ddi_get_driver_private</code>	Solaris DDI
<code>ddi_get_eventcookie</code>	Solaris DDI
<code>ddi_get_iblock_cookie</code>	Solaris DDI
<code>ddi_get_iminor</code>	Solaris DDI
<code>ddi_get_instance</code>	Solaris DDI
<code>ddi_get_kt_did</code>	Solaris DDI
<code>ddi_get_lbolt</code>	Solaris DDI
<code>ddi_get_lbolt64</code>	Solaris DDI
<code>ddi_get_name</code>	Solaris DDI
<code>ddi_get_parent</code>	Solaris DDI
<code>ddi_get_pid</code>	Solaris DDI
<code>ddi_get_soft_iblock_cookie</code>	Solaris DDI
<code>ddi_get_soft_state</code>	Solaris DDI
<code>ddi_getb</code>	Solaris DDI
<code>ddi_getl</code>	Solaris DDI
<code>ddi_getll</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>ddi_getlongprop</code>	Solaris DDI
<code>ddi_getlongprop_buf</code>	Solaris DDI
<code>ddi_getprop</code>	Solaris DDI
<code>ddi_getproplen</code>	Solaris DDI
<code>ddi_getw</code>	Solaris DDI
<code>ddi_intr_add_handler</code>	Solaris DDI
<code>ddi_intr_add_softint</code>	Solaris DDI
<code>ddi_intr_alloc</code>	Solaris DDI
<code>ddi_intr_block_disable</code>	Solaris DDI
<code>ddi_intr_block_enable</code>	Solaris DDI
<code>ddi_intr_clr_mask</code>	Solaris DDI
<code>ddi_intr_dup_handler</code>	Solaris DDI
<code>ddi_intr_disable</code>	Solaris DDI
<code>ddi_intr_enable</code>	Solaris DDI
<code>ddi_intr_free</code>	Solaris DDI
<code>ddi_intr_get_cap</code>	Solaris DDI
<code>ddi_intr_get_hilevel_pri</code>	Solaris DDI
<code>ddi_intr_get_navail</code>	Solaris DDI
<code>ddi_intr_get_nintrs</code>	Solaris DDI
<code>ddi_intr_get_pending</code>	Solaris DDI
<code>ddi_intr_get_pri</code>	Solaris DDI
<code>ddi_intr_get_softint_pri</code>	Solaris DDI
<code>ddi_intr_get_supported_types</code>	Solaris DDI
<code>ddi_intr_remove_handler</code>	Solaris DDI
<code>ddi_intr_remove_softint</code>	Solaris DDI
<code>ddi_intr_set_cap</code>	Solaris DDI
<code>ddi_intr_set_mask</code>	Solaris DDI
<code>ddi_intr_set_pri</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
<code>ddi_intr_set_softint_pri</code>	Solaris DDI
<code>ddi_intr_trigger_softint</code>	Solaris DDI
<code>ddi_io_get16</code>	Solaris DDI
<code>ddi_io_get32</code>	Solaris DDI
<code>ddi_io_get8</code>	Solaris DDI
<code>ddi_io_getb</code>	Solaris DDI
<code>ddi_io_getl</code>	Solaris DDI
<code>ddi_io_getw</code>	Solaris DDI
<code>ddi_io_put16</code>	Solaris DDI
<code>ddi_io_put32</code>	Solaris DDI
<code>ddi_io_put8</code>	Solaris DDI
<code>ddi_io_putb</code>	Solaris DDI
<code>ddi_io_putl</code>	Solaris DDI
<code>ddi_io_putw</code>	Solaris DDI
<code>ddi_io_rep_get16</code>	Solaris DDI
<code>ddi_io_rep_get32</code>	Solaris DDI
<code>ddi_io_rep_get8</code>	Solaris DDI
<code>ddi_io_rep_getb</code>	Solaris DDI
<code>ddi_io_rep_getl</code>	Solaris DDI
<code>ddi_io_rep_getw</code>	Solaris DDI
<code>ddi_io_rep_put16</code>	Solaris DDI
<code>ddi_io_rep_put32</code>	Solaris DDI
<code>ddi_io_rep_put8</code>	Solaris DDI
<code>ddi_io_rep_putb</code>	Solaris DDI
<code>ddi_io_rep_putl</code>	Solaris DDI
<code>ddi_io_rep_putw</code>	Solaris DDI
<code>ddi_iomin</code>	Solaris DDI
<code>ddi_iopb_alloc</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
ddi_iopb_free	Solaris DDI
ddi_log_sysevent	Solaris DDI
ddi_map_regs	Solaris DDI
ddi_mapdev	Solaris DDI
ddi_mapdev_intercept	Solaris DDI
ddi_mapdev_nointercept	Solaris DDI
ddi_mapdev_set_device_acc_attr	Solaris DDI
ddi_mem_alloc	Solaris DDI
ddi_mem_free	Solaris DDI
ddi_mem_get16	Solaris DDI
ddi_mem_get32	Solaris DDI
ddi_mem_get64	Solaris DDI
ddi_mem_get8	Solaris DDI
ddi_mem_getb	Solaris DDI
ddi_mem_getl	Solaris DDI
ddi_mem_getll	Solaris DDI
ddi_mem_getw	Solaris DDI
ddi_mem_put16	Solaris DDI
ddi_mem_put32	Solaris DDI
ddi_mem_put64	Solaris DDI
ddi_mem_put8	Solaris DDI
ddi_mem_putb	Solaris DDI
ddi_mem_putl	Solaris DDI
ddi_mem_putll	Solaris DDI
ddi_mem_putw	Solaris DDI
ddi_mem_rep_get16	Solaris DDI
ddi_mem_rep_get32	Solaris DDI
ddi_mem_rep_get64	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
ddi_mem_rep_get8	Solaris DDI
ddi_mem_rep_getb	Solaris DDI
ddi_mem_rep_getl	Solaris DDI
ddi_mem_rep_getll	Solaris DDI
ddi_mem_rep_getw	Solaris DDI
ddi_mem_rep_put16	Solaris DDI
ddi_mem_rep_put32	Solaris DDI
ddi_mem_rep_put64	Solaris DDI
ddi_mem_rep_put8	Solaris DDI
ddi_mem_rep_putb	Solaris DDI
ddi_mem_rep_putl	Solaris DDI
ddi_mem_rep_putll	Solaris DDI
ddi_mem_rep_putw	Solaris DDI
ddi_mmap_get_model	Solaris DDI
ddi_model_convert_from	Solaris DDI
ddi_modopen	Solaris DDI
ddi_no_info	Solaris DDI
ddi_node_name	Solaris DDI
ddi_peek16	Solaris DDI
ddi_peek32	Solaris DDI
ddi_peek64	Solaris DDI
ddi_peek8	Solaris DDI
ddi_peekc	Solaris DDI
ddi_peekd	Solaris DDI
ddi_peekl	Solaris DDI
ddi_peeks	Solaris DDI
ddi_periodic_add	Solaris DDI
ddi_periodic_delete	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
ddi_poke16	Solaris DDI
ddi_poke32	Solaris DDI
ddi_poke64	Solaris DDI
ddi_poke8	Solaris DDI
ddi_pokec	Solaris DDI
ddi_poked	Solaris DDI
ddi_pokel	Solaris DDI
ddi_pokes	Solaris DDI
ddi_prop_create	Solaris DDI
ddi_prop_exists	Solaris DDI
ddi_prop_free	Solaris DDI
ddi_prop_get_int	Solaris DDI
ddi_prop_lookup	Solaris DDI
ddi_prop_lookup_byte_array	Solaris DDI
ddi_prop_lookup_int_array	Solaris DDI
ddi_prop_lookup_string	Solaris DDI
ddi_prop_lookup_string_array	Solaris DDI
ddi_prop_modify	Solaris DDI
ddi_prop_op	Solaris DDI
ddi_prop_remove	Solaris DDI
ddi_prop_remove_all	Solaris DDI
ddi_prop_undefine	Solaris DDI
ddi_prop_update	Solaris DDI
ddi_prop_update_byte_array	Solaris DDI
ddi_prop_update_int	Solaris DDI
ddi_prop_update_int_array	Solaris DDI
ddi_prop_update_string	Solaris DDI
ddi_prop_update_string_array	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
ddi_ptob	Solaris DDI
ddi_put16	Solaris DDI
ddi_put32	Solaris DDI
ddi_put64	Solaris DDI
ddi_put8	Solaris DDI
ddi_putb	Solaris DDI
ddi_putl	Solaris DDI
ddi_putll	Solaris DDI
ddi_putw	Solaris DDI
ddi_regs_map_free	Solaris DDI
ddi_regs_map_setup	Solaris DDI
ddi_remove_event_handler	Solaris DDI
ddi_remove_intr	Solaris DDI
ddi_remove_minor_node	Solaris DDI
ddi_remove_softintr	Solaris DDI
ddi_removing_power	Solaris DDI
ddi_rep_get16	Solaris DDI
ddi_rep_get32	Solaris DDI
ddi_rep_get64	Solaris DDI
ddi_rep_get8	Solaris DDI
ddi_rep_getb	Solaris DDI
ddi_rep_getl	Solaris DDI
ddi_rep_getll	Solaris DDI
ddi_rep_getw	Solaris DDI
ddi_rep_put16	Solaris DDI
ddi_rep_put32	Solaris DDI
ddi_rep_put64	Solaris DDI
ddi_rep_put8	Solaris DDI



Routine	Type
ddi_rep_putb	Solaris DDI
ddi_rep_putl	Solaris DDI
ddi_rep_putll	Solaris DDI
ddi_rep_putw	Solaris DDI
ddi_report_dev	Solaris DDI
ddi_root_node	Solaris DDI
ddi_segmap	Solaris DDI
ddi_segmap_setup	Solaris DDI
ddi_set_driver_private	Solaris DDI
ddi_slaveonly	Solaris DDI
ddi_soft_state	Solaris DDI
ddi_soft_state_fini	Solaris DDI
ddi_soft_state_free	Solaris DDI
ddi_soft_state_init	Solaris DDI
ddi_soft_state_zalloc	Solaris DDI
ddi_strlol	Solaris DDI
ddi_strloul	Solaris DDI
ddi_trigger_softintr	Solaris DDI
ddi_umem_alloc	Solaris DDI
ddi_umem_free	Solaris DDI
ddi_umem_iosetup	Solaris DDI
ddi_umem_lock	Solaris DDI
ddi_unmap_regs	Solaris DDI
delay	DDI/DKI
devmap_default_access	Solaris DDI
devmap_devmem_setup	Solaris DDI
devmap_do_ctxmgt	Solaris DDI
devmap_load	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
devmap_set_ctx_timeout	Solaris DDI
devmap_setup	Solaris DDI
devmap_umem_setup	Solaris DDI
devmap_unload	Solaris DDI
disksort	Solaris DDI
dlbindack	Solaris DDI
drv_getparm	DDI/DKI
drv_hztousec	DDI/DKI
drv_priv	DDI/DKI
drv_usectohz	DDI/DKI
drv_usecwait	DDI/DKI
free_pktiopb	Solaris DDI
freerbuf	DDI/DKI
get_pktiopb	Solaris DDI
geterror	DDI/DKI
gethrtime	DDI/DKI
getmajor	DDI/DKI
getminor	DDI/DKI
getrbuf	DDI/DKI
gld	Solaris DDI
hat_getkpfnum	DKI only
id32_alloc	Solaris DDI
inb	Solaris x86 DDI
inl	Solaris x86 DDI
inw	Solaris x86 DDI
kiconv	Solaris DDI
kiconv_close	Solaris DDI
kiconv_open	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
kiconvstr	Solaris DDI
kmem_alloc	DDI/DKI
kmem_cache_create	Solaris DDI
kmem_free	DDI/DKI
kmem_zalloc	DDI/DKI
kstat_create	Solaris DDI
kstat_delete	Solaris DDI
kstat_install	Solaris DDI
kstat_named_init	Solaris DDI
kstat_queue	Solaris DDI
kstat_runq_back_to_waitq	Solaris DDI
kstat_runq_enter	Solaris DDI
kstat_runq_exit	Solaris DDI
kstat_waitq_enter	Solaris DDI
kstat_waitq_exit	Solaris DDI
kstat_waitq_to_runq	Solaris DDI
ldi_add_event_handler	Solaris DDI
ldi_aread	Solaris DDI
ldi_devmap	Solaris DDI
ldi_dump	Solaris DDI
ldi_ev_finalize	Solaris DDI
ldi_ev_get_cookie	Solaris DDI
ldi_ev_get_type	Solaris DDI
ldi_ev_notify	Solaris DDI
ldi_ev_register_callbacks	Solaris DDI
ldi_ev_remove_callbacks	Solaris DDI
ldi_get_dev	Solaris DDI
ldi_get_eventcookie	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
ldi_get_size	Solaris DDI
ldi_ident_from_dev	Solaris DDI
ldi_ioctl	Solaris DDI
ldi_open_by_dev	Solaris DDI
ldi_poll	Solaris DDI
ldi_prop_exists	Solaris DDI
ldi_prop_get_int	Solaris DDI
ldi_prop_get_lookup_int_array	Solaris DDI
ldi_putmsg	Solaris DDI
ldi_read	Solaris DDI
ldi_remove_event_handler	Solaris DDI
ldi_strategy	Solaris DDI
makecom_g0	Solaris DDI
makecom_g0_s	Solaris DDI
makecom_g1	Solaris DDI
makecom_g5	Solaris DDI
makedevice	DDI/DKI
max	DDI/DKI
max	DDI/DKI
membar_ops	Solaris DDI
memchr	Solaris DDI
minphys	Solaris DDI
mod_info	Solaris DDI
mod_install	Solaris DDI
mod_remove	Solaris DDI
mutex_destroy	Solaris DDI
mutex_enter	Solaris DDI
mutex_exit	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
mutex_init	Solaris DDI
mutex_owned	Solaris DDI
mutex_tryenter	Solaris DDI
nochpoll	Solaris DDI
nodev	DDI/DKI
nulldev	DDI/DKI
numtos	Solaris DDI
nvlist_add_boolean	Solaris DDI
nvlist_alloc	Solaris DDI
nvlist_lookup_boolean	Solaris DDI
nvlist_lookup_nvpair	Solaris DDI
nvlist_next_nvpair	Solaris DDI
nvlist_remove	Solaris DDI
nvlist_value_byte	Solaris DDI
outb	Solaris x86 DDI
outl	Solaris x86 DDI
outw	Solaris x86 DDI
pci_config_get16	Solaris DDI
pci_config_get32	Solaris DDI
pci_config_get64	Solaris DDI
pci_config_get8	Solaris DDI
pci_config_getb	Solaris DDI
pci_config_getl	Solaris DDI
pci_config_getw	Solaris DDI
pci_config_put16	Solaris DDI
pci_config_put32	Solaris DDI
pci_config_put64	Solaris DDI
pci_config_put8	Solaris DDI

---

Routine	Type
pci_config_putb	Solaris DDI
pci_config_putl	Solaris DDI
pci_config_putw	Solaris DDI
pci_config_setup	Solaris DDI
pci_config_teardown	Solaris DDI
pci_ereport_setup	Solaris DDI
pci_report_pmcap	Solaris DDI
pci_save_config_regs	Solaris DDI
physio	Solaris DDI
pm_busy_component	Solaris DDI
pm_power_has_changed	Solaris DDI
pm_raise_power	Solaris DDI
pm_trans_check	Solaris DDI
pollwakeup	DDI/DKI
pci_config_teardown	Solaris DDI
pci_config_teardown	Solaris DDI
priv_getbyname	Solaris DDI
priv_policy	Solaris DDI
proc_signal	Solaris DDI
proc_unref	Solaris DDI
ptob	DDI/DKI
repinsb	Solaris x86 DDI
repinsd	Solaris x86 DDI
repinsw	Solaris x86 DDI
repoutsb	Solaris x86 DDI
repoutsd	Solaris x86 DDI
repoutsw	Solaris x86 DDI
rmalloc	DDI/DKI

---

<b>Routine</b>	<b>Type</b>
<code>rmalloc_wait</code>	DDI/DKI
<code>rmallocmap</code>	DDI/DKI
<code>rmallocmap_wait</code>	DDI/DKI
<code>rmfree</code>	DDI/DKI
<code>rmfreemap</code>	DDI/DKI
<code>rw_destroy</code>	Solaris DDI
<code>rw_downgrade</code>	Solaris DDI
<code>rw_enter</code>	Solaris DDI
<code>rw_exit</code>	Solaris DDI
<code>rw_init</code>	Solaris DDI
<code>rw_read_locked</code>	Solaris DDI
<code>rw_tryenter</code>	Solaris DDI
<code>rw_tryupgrade</code>	Solaris DDI
<code>scsi_abort</code>	Solaris DDI
<code>scsi_alloc_consistent_buf</code>	Solaris DDI
<code>scsi_cname</code>	Solaris DDI
<code>scsi_destroy_pkt</code>	Solaris DDI
<code>scsi_dmafree</code>	Solaris DDI
<code>scsi_dmaget</code>	Solaris DDI
<code>scsi_dname</code>	Solaris DDI
<code>scsi_errmsg</code>	Solaris DDI
<code>scsi_ext_sense_fields</code>	Solaris DDI
<code>scsi_find_sense_descr</code>	Solaris DDI
<code>scsi_free_consistent_buf</code>	Solaris DDI
<code>scsi_get_device_type_scsi_options</code>	Solaris DDI
<code>scsi_get_device_type_string</code>	Solaris DDI
<code>scsi_hba_attach</code>	Solaris DDI
<code>scsi_hba_attach_setup</code>	Solaris DDI

---

<b>Routine</b>	<b>Type</b>
scsi_hba_detach	Solaris DDI
scsi_hba_fini	Solaris DDI
scsi_hba_init	Solaris DDI
scsi_hba_lookup_capstr	Solaris DDI
scsi_hba_pkt_alloc	Solaris DDI
scsi_hba_pkt_free	Solaris DDI
scsi_hba_probe	Solaris DDI
scsi_hba_tran_alloc	Solaris DDI
scsi_hba_tran_free	Solaris DDI
scsi_ifgetcap	Solaris DDI
scsi_ifsetcap	Solaris DDI
scsi_init_pkt	Solaris DDI
scsi_log	Solaris DDI
scsi_mname	Solaris DDI
scsi_pktalloc	Solaris DDI
scsi_pktfree	Solaris DDI
scsi_poll	Solaris DDI
scsi_probe	Solaris DDI
scsi_realloc	Solaris DDI
scsi_reset	Solaris DDI
scsi_reset_notify	Solaris DDI
scsi_resfree	Solaris DDI
scsi_rname	Solaris DDI
scsi_sense_key	Solaris DDI
scsi_setup_cdb	Solaris DDI
scsi_slave	Solaris DDI
scsi_sname	Solaris DDI
scsi_sync_pkt	Solaris DDI



---

Routine	Type
scsi_transport	Solaris DDI
scsi_unprobe	Solaris DDI
scsi_unslave	Solaris DDI
scsi_validate_sense	Solaris DDI
scsi_vu_errmsg	Solaris DDI
sema_destroy	Solaris DDI
sema_init	Solaris DDI
sema_p	Solaris DDI
sema_p_sig	Solaris DDI
sema_tryv	Solaris DDI
sema_v	Solaris DDI
sprintf	Solaris DDI
stoi	Solaris DDI
strchr	Solaris DDI
strcmp	Solaris DDI
strcpy	Solaris DDI
strlen	Solaris DDI
strncmp	Solaris DDI
strncpy	Solaris DDI
STRUCT_DECL	Solaris DDI
swab	DDI/DKI
taskq	Solaris DDI
timeout	DDI/DKI
u8_strcmp	Solaris DDI
u8_textprep_str	Solaris DDI
u8_validate	Solaris DDI
uconv_u16tou32	Solaris DDI
uiomove	DDI/DKI

---

<b>Routine</b>	<b>Type</b>
untimeout	DDI/DKI
ureadc	DDI/DKI
usb_alloc_request	Solaris DDI
usb_client_attach	Solaris DDI
usb_clr_feature	Solaris DDI
usb_create_pm_components	Solaris DDI
usb_get_addr	Solaris DDI
usb_get_alt_if	Solaris DDI
usb_get_cfg	Solaris DDI
usb_get_current_frame_number	Solaris DDI
usb_get_dev_data	Solaris DDI
usb_get_max_pkts_per_ioc_request	Solaris DDI
usb_get_status	Solaris DDI
usb_get_string_desc	Solaris DDI
usb_handle_remote_wakeup	Solaris DDI
usb_lookup_ep_data	Solaris DDI
usb_parse_data	Solaris DDI
usb_pipe_bulk_xfer	Solaris DDI
usb_pipe_close	Solaris DDI
usb_pipe_ctrl_xfer	Solaris DDI
usb_pipe_drain_reqs	Solaris DDI
usb_pipe_get_max_bulk_transfer_size	Solaris DDI
usb_pipe_get_state	Solaris DDI
usb_pipe_intr_xfer	Solaris DDI
usb_pipe_isoc_xfer	Solaris DDI
usb_pipe_open	Solaris DDI
usb_pipe_reset	Solaris DDI
usb_pipe_set_private	Solaris DDI

<b>Routine</b>	<b>Type</b>
usb_register_hotplug_cbs	Solaris DDI
usb_reset_device	Solaris DDI
uwritec	DDI/DKI
va_arg	Solaris DDI
va_end	Solaris DDI
va_start	Solaris DDI
vcmn_err	DDI/DKI
vsprintf	Solaris DDI

**See Also** [Intro\(9E\)](#), [mutex\(9F\)](#)



## REFERENCE

# Kernel Functions for Drivers

**Name** adjmsg – trim bytes from a message

**Synopsis** #include <sys/stream.h>

```
int adjmsg(mblk_t *mp, ssize_t len);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the message to be trimmed.

*len* The number of bytes to be removed.

**Description** The `adjmsg()` function removes bytes from a message.  $|len|$  (the absolute value of *len*) specifies the number of bytes to be removed. The `adjmsg()` function only trims bytes across message blocks of the same type.

The `adjmsg()` function finds the maximal leading sequence of message blocks of the same type as that of *mp* and starts removing bytes either from the head of that sequence or from the tail of that sequence. If *len* is greater than 0, `adjmsg()` removes bytes from the start of the first message block in that sequence. If *len* is less than 0, it removes bytes from the end of the last message block in that sequence.

The `adjmsg()` function fails if  $|len|$  is greater than the number of bytes in the maximal leading sequence it finds.

The `adjmsg()` function may remove any except the first zero-length message block created during adjusting. It may also remove any zero-length message blocks that occur within the scope of  $|len|$ .

**Return Values** The `adjmsg()` function returns:

1 Successful completion.

0 An error occurred.

**Context** The `adjmsg()` function can be called from user, interrupt, or kernel context.

**See Also** [STREAMS Programming Guide](#)

**Name** allocb – allocate a message block

**Synopsis** #include <sys/stream.h>

```
mblk_t *allocb(size_t size, uint_t pri);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Description** The `allocb()` function tries to allocate a STREAMS message block. Buffer allocation fails only when the system is out of memory. If no buffer is available, the `bufcall(9F)` function can help a module recover from an allocation failure.

A STREAMS message block is composed of three structures. The first structure is a message block (`mblk_t`). See [msgb\(9S\)](#). The `mblk_t` structure points to a data block structure (`dblk_t`). See [datab\(9S\)](#). Together these two structures describe the message type (if applicable) and the size and location of the third structure, the data buffer. The data buffer contains the data for this message block. The allocated data buffer is at least double-word aligned, so it can hold any C data structure.

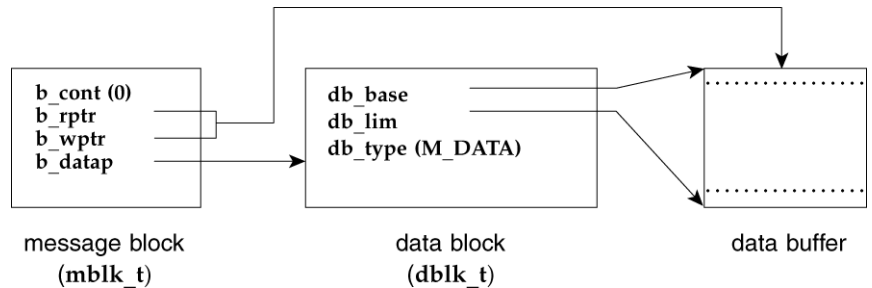
The fields in the `mblk_t` structure are initialized as follows:

<code>b_cont</code>	set to NULL
<code>b_rptr</code>	points to the beginning of the data buffer
<code>b_wptr</code>	points to the beginning of the data buffer
<code>b_datap</code>	points to the <code>dblk_t</code> structure

The fields in the `dblk_t` structure are initialized as follows:

<code>db_base</code>	points to the first byte of the data buffer
<code>db_lim</code>	points to the last byte + 1 of the buffer
<code>db_type</code>	set to <code>M_DATA</code>

The following figure identifies the data structure members that are affected when a message block is allocated.



**Parameters** *size* The number of bytes in the message block.  
*pri* Priority of the request (no longer used).

**Return Values** Upon success, `allocb()` returns a pointer to the allocated message block of type `M_DATA`. On failure, `allocb()` returns a `NULL` pointer.

**Context** The `allocb()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 `allocb()` Code Sample

Given a pointer to a queue (*q*) and an error number (*err*), the `send_error()` routine sends an `M_ERROR` type message to the stream head.

If a message cannot be allocated, `NULL` is returned, indicating an allocation failure (line 8). Otherwise, the message type is set to `M_ERROR` (line 10). Line 11 increments the write pointer (`bp->b_wptr`) by the size (one byte) of the data in the message.

A message must be sent up the read side of the stream to arrive at the stream head. To determine whether *q* points to a read queue or to a write queue, the `q->q_flag` member is tested to see if `QREADR` is set (line 13). If it is not set, *q* points to a write queue, and in line 14 the [RD\(9F\)](#) function is used to find the corresponding read queue. In line 15, the [putnext\(9F\)](#) function is used to send the message upstream, returning 1 if successful.

```

1  send_error(q,err)
2  queue_t *q;
3  unsigned char err;
4  {
5  mblk_t *bp;
6
7  if ((bp = allocb(1, BPRI_HI)) == NULL) /* allocate msg. block */
8      return(0);
9
10  bp->b_datap->db_type = M_ERROR; /* set msg type to M_ERROR */
11  *bp->b_wptr++ = err; /* increment write pointer */
12

```



**EXAMPLE 1** allocb() Code Sample (Continued)

```
13  if (!(q->q_flag & QREADR))      /* if not read queue */
14      q = RD(q);                  /* get read queue */
15  putnext(q, bp);                 /* send message upstream */
16  return(1);
17 }
```

**See Also** [RD\(9F\)](#), [bufcall\(9F\)](#), [esballoc\(9F\)](#), [esbbcall\(9F\)](#), [putnext\(9F\)](#), [testb\(9F\)](#), [datab\(9S\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The *pri* argument is no longer used, but is retained for compatibility with existing drivers.

**Name** allocb\_tmpl – allocate a message block using a template

**Synopsis** #include <sys/stream.h>

```
mblk_t *allocb_tmpl(size_t size, const mblk_t *tmpl);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *size* The number of bytes in the message block.

*tmpl* The template message block.

**Description** The `allocb_tmpl()` function tries to allocate a STREAMS message block using [allocb\(9F\)](#). If the allocation is successful, the `db_type` field in the data block structure (`dblk_t`, see [datab\(9S\)](#)), as well as some implementation-private data, are copied from the `dblk_t` associated with *tmpl*.

The `allocb_tmpl()` function should be used when a new STREAMS message block is allocated. This block is then used to contain data derived from another STREAMS message block. The original message is used as the *tmpl* argument.

**Return Values** Upon success, `allocb_tmpl()` returns a pointer to the allocated message block of the same type as *tmpl*. On failure, `allocb_tmpl()` returns a NULL pointer.

**Context** The `allocb_tmpl()` function can be called from user, interrupt, or kernel context.

**See Also** [allocb\(9F\)](#), [datab\(9S\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** anocancel – prevent cancellation of asynchronous I/O request

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
int anocancel( );
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** `anocancel( )` should be used by drivers that do not support canceling asynchronous I/O requests. `anocancel( )` is passed as the driver cancel routine parameter to [aphysio\(9F\)](#).

**Return Values** `anocancel( )` returns `ENXIO`.

**See Also** [aread\(9E\)](#), [awrite\(9E\)](#), [aphysio\(9F\)](#)

*Writing Device Drivers*

**Name** `aphysio`, `aminphys` – perform asynchronous physical I/O

**Synopsis**

```
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>
#include <sys/aio_req.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int  aphysio(int (*strat)( struct buf*), int (*cancel)(struct buf*),
          dev_t dev, int rw, void (*mincnt)(struct buf*),
          struct aio_req *aio_req);
```

**Parameters**

<i>strat</i>	Pointer to device strategy routine.
<i>cancel</i>	Pointer to driver cancel routine. Used to cancel a submitted request. The driver must pass the address of the function <a href="#">anocancel(9F)</a> because cancellation is not supported.
<i>dev</i>	The device number.
<i>rw</i>	Read/write flag. This is either <code>B_READ</code> when reading from the device or <code>B_WRITE</code> when writing to the device.
<i>mincnt</i>	Routine which bounds the maximum transfer unit size.
<i>aio_req</i>	Pointer to the <a href="#">aio_req(9S)</a> structure which describes the user I/O request.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** `aphysio()` performs asynchronous I/O operations between the device and the address space described by `aio_req`→`aio_uio`.

Prior to the start of the transfer, `aphysio()` verifies the requested operation is valid. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, `strat`, is then called one or more times to perform the physical I/O operations. `aphysio()` does not wait for each transfer to complete, but returns as soon as the necessary requests have been made.

`aphysio()` calls `mincnt` to bound the maximum transfer unit size to a sensible default for the device and the system. Drivers which do not provide their own local `mincnt` routine should call `aphysio()` with [minphys\(9F\)](#). [minphys\(9F\)](#) is the system `mincnt` routine. [minphys\(9F\)](#) ensures the transfer size does not exceed any system limits.

If a driver supplies a local `mincnt` routine, this routine should perform the following actions:

- If `bp`→`b_bcount` exceeds a device limit, `physio()` returns `ENOTSUP`.

- Call [aminphys\(9F\)](#) to ensure that the driver does not circumvent additional system limits. If [aminphys\(9F\)](#) does not return 0, return ENOTSUP.

**Return Values** `aphysio()` returns:

- |          |               |
|----------|---------------|
| 0        | Upon success. |
| non-zero | Upon failure. |

**Context** `aphysio()` can be called from user context only.

**See Also** [aread\(9E\)](#), [awrite\(9E\)](#), [strategy\(9E\)](#), [anocancel\(9F\)](#), [biodone\(9F\)](#), [biowait\(9F\)](#), [minphys\(9F\)](#), [physio\(9F\)](#), [aio\\_req\(9S\)](#), [buf\(9S\)](#), [uio\(9S\)](#)

*Writing Device Drivers*

**Warnings** It is the driver's responsibility to call [biodone\(9F\)](#) when the transfer is complete.

**Bugs** Cancellation is not supported in this release. The address of the function [anocancel\(9F\)](#) must be used as the *cancel* argument.

**Name** ASSERT, assert – expression verification

**Synopsis** `#include <sys/debug.h>`

`void ASSERT(EX);`

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *EX* boolean expression.

**Description** The ASSERT() macro checks to see if the expression *EX* is true. If it is not, then ASSERT() causes an error message to be logged to the console and the system to panic. ASSERT() works only if the preprocessor symbol DEBUG is defined.

**Context** The ASSERT() macro can be used from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)

**Name** atomic\_add, atomic\_add\_8, atomic\_add\_char, atomic\_add\_16, atomic\_add\_short, atomic\_add\_32, atomic\_add\_int, atomic\_add\_long, atomic\_add\_64, atomic\_add\_ptr, atomic\_add\_8\_nv, atomic\_add\_char\_nv, atomic\_add\_16\_nv, atomic\_add\_short\_nv, atomic\_add\_32\_nv, atomic\_add\_int\_nv, atomic\_add\_long\_nv, atomic\_add\_64\_nv, atomic\_add\_ptr\_nv – atomic add operations

**Synopsis** #include <sys/atomic.h>

```
void atomic_add_8(volatile uint8_t *target, int8_t delta);
void atomic_add_char(volatile uchar_t *target, signed char delta);
void atomic_add_16(volatile uint16_t *target, int16_t delta);
void atomic_add_short(volatile ushort_t *target, short delta);
void atomic_add_32(volatile uint32_t *target, int32_t delta);
void atomic_add_int(volatile uint_t *target, int delta);
void atomic_add_long(volatile ulong_t *target, long delta);
void atomic_add_64(volatile uint64_t *target, int64_t delta);
void atomic_add_ptr(volatile void *target, ssize_t delta);
uint8_t atomic_add_8_nv(volatile uint8_t *target, int8_t delta);
uchar_t atomic_add_char_nv(volatile uchar_t *target, signed char delta);
uint16_t atomic_add_16_nv(volatile uint16_t *target, int16_t delta);
ushort_t atomic_add_short_nv(volatile ushort_t *target, shortdelta);
uint32_t atomic_add_32_nv(volatile uint32_t *target, int32_t delta);
uint_t atomic_add_int_nv(volatile uint_t *target, int delta);
ulong_t atomic_add_long_nv(volatile ulong_t *target, long delta);
uint64_t atomic_add_64_nv(volatile uint64_t *target, int64_t delta);
void *atomic_add_ptr_nv(volatile void *target, ssize_t delta);
```

**Description** These functions enable the addition of *delta* to the value stored in *target* to occur in an atomic manner.

**Return Values** The \*\_nv() variants of these functions return the new value of *target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** The \*\_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically* (for example, when decrementing a reference count and checking whether it went to zero).



**Name** atomic\_and, atomic\_and\_8, atomic\_and\_uchar, atomic\_and\_16, atomic\_and\_ushort, atomic\_and\_32, atomic\_and\_uint, atomic\_and\_ulong, atomic\_and\_64, atomic\_and\_8\_nv, atomic\_and\_uchar\_nv, atomic\_and\_16\_nv, atomic\_and\_ushort\_nv, atomic\_and\_32\_nv, atomic\_and\_uint\_nv, atomic\_and\_ulong\_nv, atomic\_and\_64\_nv – atomic AND operations

**Synopsis** #include <sys/atomic.h>

```
void atomic_and_8(volatile uint8_t *target, uint8_t bits);
void atomic_and_uchar(volatile uchar_t *target, uchar_t bits);
void atomic_and_16(volatile uint16_t *target, uint16_t bits);
void atomic_and_ushort(volatile ushort_t *target, ushort_t bits);
void atomic_and_32(volatile uint32_t *target, uint32_t bits);
void atomic_and_uint(volatile uint_t *target, uint_t bits);
void atomic_and_ulong(volatile ulong_t *target, ulong_t bits);
void atomic_and_64(volatile uint64_t *target, uint64_t bits);
uint8_t atomic_and_8_nv(volatile uint8_t *target, uint8_t bits);
uchar_t atomic_and_uchar_nv(volatile uchar_t *target, uchar_t bits);
uint16_t atomic_and_16_nv(volatile uint16_t *target, uint16_t bits);
ushort_t atomic_and_ushort_nv(volatile ushort_t *target, ushort_t bits);
uint32_t atomic_and_32_nv(volatile uint32_t *target, uint32_t bits);
uint_t atomic_and_uint_nv(volatile uint_t *target, uint_t bits);
ulong_t atomic_and_ulong_nv(volatile ulong_t *target, ulong_t bits);
uint64_t atomic_and_64_nv(volatile uint64_t *target, uint64_t bits);
```

**Description** These functions enable the bitwise AND of *bits* to the value stored in *target* to occur in an atomic manner.

**Return Values** The \*\_nv() variants of these functions return the new value of *target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** The \*\_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

**Name** atomic\_bits, atomic\_set\_long\_excl, atomic\_clear\_long\_excl – atomic set and clear bit operations

**Synopsis** #include <sys/atomic.h>

```
int atomic_set_long_excl(volatile ulong_t *target, uint_t bit);
int atomic_clear_long_excl(volatile ulong_t *target, uint_t bit);
```

**Description** The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions perform an exclusive atomic bit set or clear operation on *target*. The value of *bit* specifies the number of the bit to be modified within target. Bits are numbered from zero to one less than the maximum number of bits in a long. If the value of *bit* falls outside of this range, the result of the operation is undefined.

**Return Values** The `atomic_set_long_excl()` and `atomic_clear_long_excl()` functions return 0 if *bit* was successfully set or cleared. They return -1 if *bit* was already set or cleared.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Name** atomic\_cas, atomic\_cas\_8, atomic\_cas\_uchar, atomic\_cas\_16, atomic\_cas\_ushort, atomic\_cas\_32, atomic\_cas\_uint, atomic\_cas\_ulong, atomic\_cas\_64, atomic\_cas\_ptr – atomic compare and swap operations

**Synopsis** #include <sys/atomic.h>

```
uint8_t atomic_cas_8(volatile uint8_t *target, uint8_t cmp, uint8_t
    newval);
uchar_t atomic_cas_uchar(volatile uchar_t *target, uchar_t cmp, uchar_t
    newval);
uint16_t atomic_cas_16(volatile uint16_t *target, uint16_t cmp, uint16_t
    newval);
ushort_t atomic_cas_ushort(volatile ushort_t *target, ushort_t cmp,
    ushort_t newval);
uint32_t atomic_cas_32(volatile uint32_t *target, uint32_t cmp, uint32_t
    newval);
uint_t atomic_cas_uint(volatile uint_t *target, uint_t cmp, uint_t newval);
ulong_t atomic_cas_ulong(volatile ulong_t *target, ulong_t cmp, ulong_t
    newval);
uint64_t atomic_cas_64(volatile uint64_t *target, uint64_t cmp, uint64_t
    newval);
void *atomic_cas_ptr(volatile void *target, void *cmp, void *newval);
```

**Description** These functions enable a compare and swap operation to occur atomically. The value stored in *target* is compared with *cmp*. If these values are equal, the value stored in *target* is replaced with *newval*. The old value stored in *target* is returned by the function whether or not the replacement occurred.

**Return Values** These functions return the old of *\*target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Name** atomic\_dec, atomic\_dec\_8, atomic\_dec\_uchar, atomic\_dec\_16, atomic\_dec\_ushort, atomic\_dec\_32, atomic\_dec\_uint, atomic\_dec\_ulong, atomic\_dec\_64, atomic\_dec\_ptr, atomic\_dec\_8\_nv, atomic\_dec\_uchar\_nv, atomic\_dec\_16\_nv, atomic\_dec\_ushort\_nv, atomic\_dec\_32\_nv, atomic\_dec\_uint\_nv, atomic\_dec\_ulong\_nv, atomic\_dec\_64\_nv, atomic\_dec\_ptr\_nv – atomic decrement operations

**Synopsis** #include <sys/atomic.h>

```
void atomic_dec_8(volatile uint8_t *target);
void atomic_dec_uchar(volatile uchar_t *target);
void atomic_dec_16(volatile uint16_t *target);
void atomic_dec_ushort(volatile ushort_t *target);
void atomic_dec_32(volatile uint32_t *target);
void atomic_dec_uint(volatile uint_t *target);
void atomic_dec_ulong(volatile ulong_t *target);
void atomic_dec_64(volatile uint64_t *target);
void atomic_dec_ptr(volatile void *target);
uint8_t atomic_dec_8_nv(volatile uint8_t *target);
uchar_t atomic_dec_uchar_nv(volatile uchar_t *target);
uint16_t atomic_dec_16_nv(volatile uint16_t *target);
ushort_t atomic_dec_ushort_nv(volatile ushort_t *target);
uint32_t atomic_dec_32_nv(volatile uint32_t *target);
uint_t atomic_dec_uint_nv(volatile uint_t *target);
ulong_t atomic_dec_ulong_nv(volatile ulong_t *target);
uint64_t atomic_dec_64_nv(volatile uint64_t *target);
void *atomic_dec_ptr_nv(volatile void *target);
```

**Description** These functions enable the decrementing (by one) of the value stored in *target* to occur in an atomic manner.

**Return Values** The \*\_nv() variants of these functions return the new value of *target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** The \*\_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value atomically (for example, when decrementing a reference count and checking whether it went to zero).

**Name** atomic\_inc, atomic\_inc\_8, atomic\_inc\_uchar, atomic\_inc\_16, atomic\_inc\_ushort, atomic\_inc\_32, atomic\_inc\_uint, atomic\_inc\_ulong, atomic\_inc\_64, atomic\_inc\_ptr, atomic\_inc\_8\_nv, atomic\_inc\_uchar\_nv, atomic\_inc\_16\_nv, atomic\_inc\_ushort\_nv, atomic\_inc\_32\_nv, atomic\_inc\_uint\_nv, atomic\_inc\_ulong\_nv, atomic\_inc\_64\_nv, atomic\_inc\_ptr\_nv – atomic increment operations

**Synopsis** #include <sys/atomic.h>

```
void atomic_inc_8(volatile uint8_t *target);
void atomic_inc_uchar(volatile uchar_t *target);
void atomic_inc_16(volatile uint16_t *target);
void atomic_inc_ushort(volatile ushort_t *target);
void atomic_inc_32(volatile uint32_t *target);
void atomic_inc_uint(volatile uint_t *target);
void atomic_inc_ulong(volatile ulong_t *target);
void atomic_inc_64(volatile uint64_t *target);
void atomic_inc_ptr(volatile void *target);
uint8_t atomic_inc_8_nv(volatile uint8_t *target);
uchar_t atomic_inc_uchar_nv(volatile uchar_t *target);
uint16_t atomic_inc_16_nv(volatile uint16_t *target);
ushort_t atomic_inc_ushort_nv(volatile ushort_t *target);
uint32_t atomic_inc_32_nv(volatile uint32_t *target);
uint_t atomic_inc_uint_nv(volatile uint_t *target);
ulong_t atomic_inc_ulong_nv(volatile ulong_t *target);
uint64_t atomic_inc_64_nv(volatile uint64_t *target);
void *atomic_inc_ptr_nv(volatile void *target);
```

**Description** These functions enable the incrementing (by one) of the value stored in *target* to occur in an atomic manner.

**Return Values** The \*\_nv() variants of these functions return the new value of *target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** The \*\_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.



**Name** atomic\_ops – atomic operations

**Synopsis** #include <sys/atomic.h>

**Description** This collection of functions provides atomic memory operations. There are 8 different classes of atomic operations:

- [atomic\\_add\(9F\)](#) These functions provide an atomic addition of a signed value to a variable.
- [atomic\\_and\(9F\)](#) These functions provide an atomic logical 'and' of a value to a variable.
- [atomic\\_bits\(9F\)](#) These functions provide atomic bit setting and clearing within a variable.
- [atomic\\_cas\(9F\)](#) These functions provide an atomic comparison of a value with a variable. If the comparison is equal, then swap in a new value for the variable, returning the old value of the variable in either case.
- [atomic\\_dec\(9F\)](#) These functions provide an atomic decrement on a variable.
- [atomic\\_inc\(9F\)](#) These functions provide an atomic increment on a variable.
- [atomic\\_or\(9F\)](#) These functions provide an atomic logical 'or' of a value to a variable.
- [atomic\\_swap\(9F\)](#) These functions provide an atomic swap of a value with a variable, returning the old value of the variable.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#)

**Notes** Atomic instructions ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used. See [membar\\_ops\(9F\)](#).

Atomic instructions can be expensive, since they require synchronization to occur at a hardware level. This means they should be used with care to ensure that forcing hardware level synchronization occurs a minimum number of times. For example, if you have several variables that need to be incremented as a group, and each needs to be done atomically, then do so with a mutex lock protecting all of them being incremented rather than using the [atomic\\_inc\(9F\)](#) operation on each of them.

**Name** atomic\_or, atomic\_or\_8, atomic\_or\_uchar, atomic\_or\_16, atomic\_or\_ushort, atomic\_or\_32, atomic\_or\_uint, atomic\_or\_ulong, atomic\_or\_64, atomic\_or\_8\_nv, atomic\_or\_uchar\_nv, atomic\_or\_16\_nv, atomic\_or\_ushort\_nv, atomic\_or\_32\_nv, atomic\_or\_uint\_nv, atomic\_or\_ulong\_nv, atomic\_or\_64\_nv – atomic OR operations

**Synopsis** #include <sys/atomic.h>

```
void atomic_or_8(volatile uint8_t *target, uint8_t bits);
void atomic_or_uchar(volatile uchar_t *target, uchar_t bits);
void atomic_or_16(volatile uint16_t *target, uint16_t bits);
void atomic_or_ushort(volatile ushort_t *target, ushort_t bits);
void atomic_or_32(volatile uint32_t *target, uint32_t bits);
void atomic_or_uint(volatile uint_t *target, uint_t bits);
void atomic_or_ulong(volatile ulong_t *target, ulong_t bits);
void atomic_or_64(volatile uint64_t *target, uint64_t bits);
uint8_t atomic_or_8_nv(volatile uint8_t *target, uint8_t bits);
uchar_t atomic_or_uchar_nv(volatile uchar_t *target, uchar_t bits);
uint16_t atomic_or_16_nv(volatile uint16_t *target, uint16_t bits);
ushort_t atomic_or_ushort_nv(volatile ushort_t *target, ushort_t bits);
uint32_t atomic_or_32_nv(volatile uint32_t *target, uint32_t bits);
uint_t atomic_or_uint_nv(volatile uint_t *target, uint_t bits);
ulong_t atomic_or_ulong_nv(volatile ulong_t *target, ulong_t bits);
uint64_t atomic_or_64_nv(volatile uint64_t *target, uint64_t bits);
```

**Description** These functions enable the bitwise OR of *bits* to the value stored in *target* to occur in an atomic manner.

**Return Values** The \*\_nv() variants of these functions return the new value of *target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_swap\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** The \*\_nv() variants are substantially more expensive on some platforms than the versions that do not return values. Do not use them unless you need to know the new value *atomically*.

**Name** atomic\_swap, atomic\_swap\_8, atomic\_swap\_uchar, atomic\_swap\_16, atomic\_swap\_ushort, atomic\_swap\_32, atomic\_swap\_uint, atomic\_swap\_ulong, atomic\_swap\_64, atomic\_swap\_ptr – atomic swap operations

**Synopsis** #include <sys/atomic.h>

```
uint8_t atomic_swap_8(volatile uint8_t *target, uint8_t newval);
uchar_t atomic_swap_uchar(volatile uchar_t *target, uchar_t newval);
uint16_t atomic_swap_16(volatile uint16_t *target, uint16_t newval);
ushort_t atomic_swap_ushort(volatile ushort_t *target, ushort_t newval);
uint32_t atomic_swap_32(volatile uint32_t *target, uint32_t newval);
uint_t atomic_swap_uint(volatile uint_t *target, uint_t newval);
ulong_t atomic_swap_ulong(volatile ulong_t *target, ulong_t newval);
uint64_t atomic_swap_64(volatile uint64_t *target, uint64_t newval);
void *atomic_swap_ptr(volatile void *target, void *newval);
```

**Description** These functions enable a swap operation to occur atomically. The value stored in *target* is replaced with *newval*. The old value is returned by the function.

**Return Values** These functions return the old of *\*target*.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_cas\(9F\)](#), [membar\\_ops\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Name** audio\_dev\_add\_control, audio\_dev\_del\_control, audio\_dev\_update\_controls, audio\_dev\_add\_soft\_volume, audio\_control\_read, audio\_control\_write – audio device control functions

**Synopsis** #include <sys/audio/audio\_driver.h>

```
typedef int (*audio_ctrl_wr_t)(void *, uint64_t);
typedef int (*audio_ctrl_rd_t)(void *, uint64_t *);

audio_ctrl_t *audio_dev_add_control(audio_dev_t *adev,
    audio_ctrl_desc_t *desc, audio_ctrl_rd_t readfn,
    audio_ctrl_wr_t writefn, void *data);

void audio_dev_del_control(audio_ctrl_t *ctrl);

void audio_dev_update_controls(audio_dev_t *adev);

void audio_dev_add_soft_volume(audio_dev_t *adev);

int audio_control_read(audio_ctrl_t *ctrl, uint64_t *valuep);

int audio_control_write(audio_ctrl_t *ctrl, uint64_t value);
```

**Parameters**

*adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#)

*desc* pointer to a structure describing the control. The structure has the following definition:

```
struct audio_ctrl_desc {
    const char *acd_name; /* Controls Mnemonic */
    uint32_t acd_type; /* Entry type */
    uint64_t acd_flags; /* Characteristics */
    uint64_t acd_maxvalue; /* Max value control */
    uint64_t acd_minvalue; /* Min value control */
    const char *acd_enum[64]; /* Enum values */
};
```

*readfn* pointer to a function used to read the value of the control

*writefn* pointer to a function used to write the value of the control

*data* driver private data pointer passed to read and write functions

*ctrl* opaque handle to audio control previously returned by `audio_dev_add_control()`

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** Controls for audio devices are adjustable values that affect the operation of the device. The most common use of a control is to affect volume settings.

The `audio_dev_add_control()` function registers a control for the audio device and associates handlers with it.

The *desc* parameter describes the control in more detail. It has the following structure:

```
struct audio_ctrl_desc {
    const char *acd_name;      /* Controls Mnemonic */
    uint32_t   acd_type;      /* Entry type */
    uint64_t   acd_flags;     /* Characteristics */
    uint64_t   acd_maxvalue;  /* Max value control */
    uint64_t   acd_minvalue;  /* Min value control */
    const char *acd_enum[64]; /* Enum values */
};
```

The *acd\_name* field is a simple ASCII string identifying the control. The convention is to follow the same naming rules as used for C identifiers.

The following predefined control names are available and should be used when possible. Other names may be used, but might not be presented consistently to all users in all locales, so they should be used with caution.

AUDIO_CTRL_ID_VOLUME	Master play back volume
AUDIO_CTRL_ID_LINEOUT	Volume on the line out jack
AUDIO_CTRL_ID_FRONT	Front speaker volume
AUDIO_CTRL_ID_REAR	Rear speaker volume
AUDIO_CTRL_ID_HEADPHONE	Headphone volume
AUDIO_CTRL_ID_CENTER	Center speaker volume
AUDIO_CTRL_ID_LFE	Low-frequency effect volume
AUDIO_CTRL_ID_SPEAKER	Built-in speaker volume
AUDIO_CTRL_ID_MIC	Microphone gain
AUDIO_CTRL_ID_CD	CD input gain
AUDIO_CTRL_ID_LINEIN	Line input gain
AUDIO_CTRL_ID_REC_GAIN	Master record gain
AUDIO_CTRL_ID_MONGAIN	Monitor gain
AUDIO_CTRL_ID_RECSRC	Record source

The *acd\_type* field indicates the type of control. The following types are available:

AUDIO_CTRL_TYPE_BOOLEAN	A simple boolean on/off value
AUDIO_CTRL_TYPE_STEREO	A stereophonic control. The value for the right channel is in the low order byte, and for the right channel is in the 2nd least significant byte.

AUDIO_CTRL_TYPE_MONO	A monophonic value
AUDIO_CTRL_TYPE_ENUM	An enumeration

The *acd\_flags* field is a bitmask that provides additional detail about the control. The following bits are defined; all other bits are reserved and must not be set by the driver:

AUDIO_CTRL_FLAG_READABLE	Control readable
AUDIO_CTRL_FLAG_WRITEABLE	Control writable
AUDIO_CTRL_FLAG_RW	Read/writable
AUDIO_CTRL_FLAG_MAINVOL	Main analog volume control
AUDIO_CTRL_FLAG_PCMVOL	PCM output volume
AUDIO_CTRL_FLAG_RECVOL	Record volume
AUDIO_CTRL_FLAG_MONVOL	Monitor volume
AUDIO_CTRL_FLAG_PLAY	Control relates to playback
AUDIO_CTRL_FLAG_REC	Control relates to recording
AUDIO_CTRL_FLAG_MONITOR	Control relates to monitoring
AUDIO_CTRL_FLAG_MULTI	Enumeration is a bitmask that allows multiple selections

The *acd\_minvalue* and *acd\_maxvalue* fields constrain the values that the control can take. For boolean values, they should be 0 and 1. For monophonic and stereophonic controls, they are the values for one channel, and must not exceed 255. For enumerations, *acd\_minvalue* indicates enumeration values that may be changed and *acd\_maxvalue* indicates the full mask of possible options. It is nonsensical for *acd\_minvalue* to have value different from *acd\_maxvalue* if AUDIO\_CTRL\_FLAG\_MULTI is not set. If a bit is present in *acd\_maxval* but not in *acd\_minvalue*, then the corresponding bit must always be set in the value.

The *acd\_enum* field is used with AUDIO\_CTRL\_TYPE\_ENUM and provides ASCII strings identifying each bit in the enumeration. The following definitions are available for use and should be used whenever possible.

```
AUDIO_PORT_MIC
AUDIO_PORT_CD
AUDIO_PORT_VIDEO
AUDIO_PORT_AUX1OUT
AUDIO_PORT_AUX2OUT
AUDIO_PORT_LINEOUT
AUDIO_PORT_LINEIN
AUDIO_PORT_AUX1IN
AUDIO_PORT_AUX2IN
```

AUDIO\_PORT\_HEADPHONES  
AUDIO\_PORT\_SPDIFIN  
AUDIO\_PORT\_SPDIFOUT  
AUDIO\_PORT\_DIGOUT  
AUDIO\_PORT\_DIGIN  
AUDIO\_PORT\_HDMI  
AUDIO\_VALUE\_ON  
AUDIO\_VALUE\_OFF  
AUDIO\_VALUE\_VERYLOW  
AUDIO\_VALUE\_LOW  
AUDIO\_VALUE\_MEDIUM  
AUDIO\_VALUE\_HIGH  
AUDIO\_VALUE\_VERYHIGH

The `readfn()` function takes the supplied driver private data as its first argument and a pointer to receive the controls current value as its second argument. It returns 0 on success, or an `errno` on failure. It may be called in interrupt context and must not block.

The `writfn()` function takes the supplied driver private data as its first argument and a new value to set for the control as its second argument. It return 0 on success or an `errno` on failure. It may be called in interrupt context and must not block.

The data argument is a pointer to driver private state passed as is to the `readfn()` and `writfn()` functions.

If a control with the supplied name is already present, the previously associated control is automatically removed and its resources released.

The `audio_dev_del_control()` function removes the supplied control from the audio device and frees any related resources.

The `audio_dev_update_controls()` function is called by the audio driver to indicate that the value for one or more of the controls on the audio device has changed asynchronously.

The `audio_dev_add_soft_volume()` function is used to create a synthetic volume control for the audio device. It can be used for audio devices that lack any physical volume control capability. The control is registered using `AUDIO_CTRL_ID_VOLUME`.

The `audio_control_read()` function is used by a driver to call the read function for the control. This can be useful when used with layered systems which override control functions.

The `audio_control_write()` function is used by a driver to call the write function for the control. This can be useful when used with layered systems that override control functions.



**Return Values** The `audio_dev_add_control()` function returns an opaque handle to the control.

The control read and write functions return 0 on success or an error number on failure.

**Context** The `audio_dev_add_control()`, `audio_dev_del_control()`, and `audio_dev_add_soft_volume()` functions may be called from user or kernel context only.

The `audio_dev_update_controls()`, `audio_control_read()`, and `audio_control_write()` functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [audio\\_dev\\_alloc\(9F\)](#), [audio\\_dev\\_register\(9F\)](#), [audio\\_dev\\_suspend\(9F\)](#)

**Notes** The audio framework provides the following synchronization guarantees for audio controls:

- Only one thread will be executing an audio control accessor function for a given audio device at any one time.
- No threads will be executing any audio control accessor functions for a given audio device if that device is suspended. See [audio\\_dev\\_suspend\(9F\)](#).

Each control must have a unique name. An attempt to register a control with a name that is already in use will overwrite the previous control and return the same handle that was obtained originally. This can be used to reconfigure the settings of a control. However, care must be taken to ensure that the control is only deleted once during cleanup, or else a system crash will probably result.

Any controls added to an audio engine with `audio_dev_add_control()` must be destroyed with `audio_dev_del_control()` before the device itself is freed with [audio\\_dev\\_free\(9F\)](#).

If `audio_dev_add_soft_volume()` is performed for a device, the caller must ensure that no `AUDIO_CTRL_ID_VOLUME` controls are created for it using `audio_dev_add_control()`. Unlike other controls, the soft volume control is destroyed automatically during the execution of `audio_dev_free()`.

**Name** audio\_dev\_add\_engine, audio\_dev\_remove\_engine – add or remove an audio engine to or from a device

**Synopsis** #include <sys/audio/audio\_driver.h>

```
void audio_dev_add_engine(audio_dev_t *adev, audio_engine_t *engine);
void audio_dev_remove_engine(audio_dev_t *adev, audio_engine_t *engine);
```

**Parameters** *adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#)  
*engine* pointer to an audio engine allocated with [audio\\_engine\\_alloc\(9F\)](#)

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_dev_add_engine()` function associates an allocated and initialized engine with an audio device.

Multiple engines may be added to an audio device in this fashion. Usually device drivers perform this at least twice: once for a playback engine and once for a record engine. Multiple playback engines can be especially useful in allowing the framework to avoid software mixing overhead or to support engines running with different parameters. For example, different engines may support different audio formats or different sample rates.

Generally, `audio_dev_add_engine()` should be called before registering the audio device with the audio framework using [audio\\_dev\\_register\(9F\)](#).

The `audio_dev_remove_engine()` function removes an engine from the list of engines associated with a device. This is generally called during [detach\(9E\)](#) processing.

**Context** These functions may be called from user or kernel context only.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [detach\(9E\)](#), [audio\\_dev\\_alloc\(9F\)](#), [audio\\_dev\\_register\(9F\)](#), [audio\\_dev\\_unregister\(9F\)](#), [audio\\_engine\\_alloc\(9F\)](#)

**Name** audio\_dev\_alloc, audio\_dev\_free – allocate or free an audio device

**Synopsis** #include <sys/audio/audio\_driver.h>

```
audio_dev_t *audio_dev_alloc(dev_info_t *dip, int devno);
void audio_dev_free(audio_dev_t *adev);
```

**Parameters**

- dip* the device instance structure for the device
- devno* an integer representing the “logical device” number of the device. If 0 is supplied, the instance number returned from [ddi\\_get\\_instance\(9F\)](#) is used instead.
- adev* pointer to an audio device allocated with `audio_dev_alloc()`

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_dev_alloc()` function allocates a structure representing a logical audio device. Normally, there is one such audio device per `dev_info_t`. Some devices, however, may support the notion of multiple logical devices, where different audio devices with completely independent configurations can be supported. These devices will usually have completely independent physical ports as well. In such cases, the driver may supply a non-zero value for the *devno*.

If *devno* is used, it is the responsibility of the driver to ensure that all values of *devno* are unique for that particular driver, and a *devno* must be supplied for all hardware instances in the system to avoid collisions with device instance numbers.

An audio device (`audio_dev_t`) may have multiple DMA or PIO engines “streams”, but generally all of these engines are subject to the same set of controllable parameters (such as gain, port selection, etc.)

The `audio_dev_free()` function frees a previously allocated audio device handle and releases any resources associated with it.

**Return Values** The `audio_dev_alloc()` function returns an opaque handle for the audio device. It always succeeds.

**Context** The `audio_dev_alloc()` and `audio_dev_free()` functions may be called from user or kernel context only.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [audio\\_dev\\_add\\_control\(9F\)](#), [audio\\_dev\\_add\\_engine\(9F\)](#), [audio\\_dev\\_register\(9F\)](#), [audio\\_dev\\_set\\_description\(9F\)](#), [audio\\_dev\\_suspend\(9F\)](#)

*Writing Device Drivers*

**Name** audio\_dev\_register, audio\_dev\_unregister – register or unregister an audio device from the system

**Synopsis** #include <sys/audio/audio\_driver.h>

```
int audio_dev_register(audio_dev_t *adev);
int audio_dev_unregister(audio_dev_t *adev);
```

**Parameters** *adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#)

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_dev_register()` function is called by the driver to complete configuration of the audio device and register the device (along with all of its engines and controls) with the audio framework.

Use of this function also results in making the device accessible to applications through the creation of minor nodes and other such entities.

The `audio_dev_register()` function is often the last function called by a driver during [attach\(9E\)](#) processing.

The `audio_dev_unregister()` function is the inverse operation, and deregisters the device (and engines and controls) from the framework. However, it also checks to make sure that the device is not in use before doing so. As such, it is usually the first function called during [detach\(9E\)](#) processing.

**Return Values** The `audio_dev_register()` function returns `DDI_SUCCESS` on success or `DDI_FAILURE` on failure.

The `audio_dev_unregister()` function returns `DDI_SUCCESS` if it unregistered successfully, or `DDI_FAILURE` if it did not. Generally, `DDI_FAILURE` indicates that the audio device is still in use.

**Context** These functions may only be called in user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [audio\\_dev\\_alloc\(9F\)](#)

**Name** `audio_dev_set_description`, `audio_dev_set_version`, `audio_dev_add_info` – set human-readable information fields for an audio device

**Synopsis** `#include <sys/audio/audio_driver.h>`

```
void audio_dev_set_description(audio_dev_t *adev, const char *desc);
void audio_dev_set_version(audio_dev_t *adev, const char *vers);
void audio_dev_add_info(audio_dev_t *adev, const char *info);
```

**Parameters**

- adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#)
- desc* a short description of the device
- vers* a human-readable version for the device
- info* additional human-readable details about the device

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_dev_set_description()` and `audio_dev_version()` functions set strings describing the device for applications to display this information to users.

The `audio_dev_add_info()` function provides more detailed information, such as codec details. Successive calls to this function are additive. Each addition will represent one line of output in any detailed device report.

The strings are copied by the framework, so drivers may use local or temporary storage for them. They must be properly terminated with null bytes. Only the ASCII character set is supported.

The supplied strings may be truncated to 16 characters by some applications.

**Context** These functions may be called from user or kernel context only.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [audio\\_dev\\_alloc\(9F\)](#)

- Name** audio\_dev\_suspend, audio\_dev\_resume – suspend or resume an audio device
- Synopsis** #include <sys/audio/audio\_driver.h>
- ```
void audio_dev_suspend(audio_dev_t *adev);
void audio_dev_resume(audio_dev_t *adev);
```
- Parameters** *adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#)
- Interface Level** Solaris DDI specific (Solaris DDI)
- Description** The `audio_dev_suspend()` function suspends all functions associated with an audio device. The intention is for the audio driver to call this as the first part of its handling for `DDI_SUSPEND` in [detach\(9E\)](#).
- The `audio_dev_resume()` function resumes functionality associated with the audio device. This function will generally be the last thing executed during the handling for `DDI_RESUME` in [attach\(9E\)](#).
- The framework ensures no entry points associated with the audio device are executing while the device is suspended. Furthermore, the framework guarantees that any engines that may have been running are stopped with [audio\\_engine\\_stop\(9E\)](#) before returning from `audio_dev_suspend()`. Those engines will be restarted using [audio\\_engine\\_start\(9E\)](#) by `audio_dev_resume()`.
- Any control settings are retrieved during suspend and restored as during resume.
- The driver must not hold any locks while calling either of these functions.
- Context** These functions may be called from user or kernel context only.
- Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [attach\(9E\)](#), [audio\\_engine\\_start\(9E\)](#), [audio\\_engine\\_stop\(9E\)](#), [detach\(9E\)](#), [audio\\_dev\\_add\\_control\(9F\)](#), [audio\\_dev\\_alloc\(9F\)](#)

*Writing Device Drivers*

**Name** audio\_dev\_warn – log a warning message for an audio device

**Synopsis** #include <sys/audio/audio\_driver.h>

```
void audio_dev_warn(audio_dev_t *adev, const char *fmt, ...);
```

**Parameters** *adev* pointer to an audio device allocated with [audio\\_dev\\_alloc\(9F\)](#). This parameter can be NULL if no suitable device pointer is available.

*fmt* format of message to be displayed

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_dev_warn()` function is a convenience wrapper that sends a message to the system console or logs using [cmn\\_err\(9F\)](#) with a level of `CE_WARN`. It also prefixes the message with information identifying the device if *adev* is not NULL.

The format should not contain any trailing newlines, as the newline will be appended by `cmn_err()` automatically. Unlike `cmn_err()`, however, no special handling of the first character of the format string is performed.

**Context** This function may be called in any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [audio\\_dev\\_alloc\(9F\)](#), [cmn\\_err\(9F\)](#)



**Name** audio\_engine\_alloc, audio\_engine\_free – allocate or free an audio engine

**Synopsis** #include <sys/audio/audio\_driver.h>

```
audio_engine_t *audio_engine_alloc(audio_engine_ops_t *ops,
    unsigned flags);
```

```
void audio_engine_free(audio_engine_t *engine);
```

**Parameters** *ops* pointer to an operations vector describing the engine

*flags* integer mask of flags indicating further details about the engine and its capabilities. ENGINE\_INPUT\_CAP indicates the engine supports recording ENGINE\_OUTPUT\_CAP indicates the engine supports playback. All other possible bits are reserved and should be unset.

*engine* pointer to an audio engine allocated with audio\_engine\_alloc()

**Description** The audio\_engine\_alloc() function allocates a structure representing an audio DMA or PIO engine (sometimes called a “stream”). The allocated structure is opaque to device drivers, but is used when the driver needs to supply a reference to the engine to other functions in the Solaris Audio DDI.

The audio\_engine\_free() function frees a previously allocated structure and releases any resources associated with it. It is important that device drivers not call this while the engine remains registered with the audio device. This should not be called until after the engine is deregistered with audio\_dev\_remove\_engine(9F) or before it is registered with audio\_dev\_add\_engine(9F).

**Return Values** The audio\_engine\_alloc function returns an opaque handle for the audio engine. It always succeeds.

**Context** These functions may be called from user or kernel context only.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [audio\\_dev\\_add\\_engine\(9F\)](#), [audio\\_dev\\_remove\\_engine\(9F\)](#), [audio\\_engine\\_set\\_private\(9F\)](#), [audio\\_engine\\_ops\(9S\)](#)

**Name** audio\_engine\_set\_private, audio\_engine\_get\_private – set or get private state for an audio engine

**Synopsis** #include <sys/audio/audio\_driver.h>

```
void *audio_engine_get_private(audio_engine_t *eng);
void audio_engine_set_private(audio_engine_t *eng, void *state)
```

**Parameters** *eng* pointer to an audio engine allocated with [audio\\_engine\\_alloc\(9F\)](#)  
*state* a private soft state structure for the device driver

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The audio\_engine\_set\_private() function stores a pointer to the driver private state for the engine in the audio engine structure.

The pointer to the private state established by audio\_engine\_set\_private() is passed as the first argument to the engine's entry points registered in the [audio\\_engine\\_ops\(9S\)](#) structure.

The audio\_engine\_get\_private() function retrieves the previously stored private state pointer.

**Return Values** The audio\_engine\_get\_private() function returns the device driver private state pointer.

**Context** These functions may be called from any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [audio\(7D\)](#), [audio\\_engine\\_alloc\(9F\)](#), [audio\\_engine\\_ops\(9S\)](#)

**Name** audio\_init\_ops, audio\_fini\_ops – initialize DDI entry points for an audio device driver

**Synopsis** #include <sys/audio/audio\_driver.h>

```
void audio_init_ops(struct dev_ops *devops, const char *name);
```

```
void audio_fini_ops(struct dev_ops *devops);
```

**Parameters** *devops* pointer to the [dev\\_ops\(9S\)](#) vector for the device instance

*name* name of the audio driver, which must be the same as the kernel module name. For example, “audiohd”.

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `audio_init_ops()` function initializes the device operations vector for an audio driver, setting a number of entry points to common implementation from the audio framework. It must be executed during the [\\_init\(9E\)](#) function for the driver.

The actual values supplied by `audio_init_ops()` are the fields for *devo\_cb\_ops* and *devo\_getinfo*.

Device drivers need to supply any other entry points, especially the [attach\(9E\)](#), [detach\(9E\)](#), and [quiesce\(9E\)](#) entry points.

The [ddi\\_driver\\_name\(9F\)](#) function cannot be used to determine the device driver's name, as there is not necessarily a `dev_info_t` to use during the [\\_init\(9E\)](#) entry point. A constant string must be supplied by the driver.

The `audio_fini_ops()` function undoes the work of `audio_init_ops()` and releases resources allocated by `audio_init_ops()`.

**Context** These functions may be called from the driver's [\\_init\(9E\)](#) and [\\_fini\(9E\)](#) entry points only.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [\\_fini\(9E\)](#), [\\_init\(9E\)](#), [quiesce\(9E\)](#), [ddi\\_driver\\_name\(9F\)](#), [dev\\_ops\(9S\)](#)

*Writing Device Drivers*

**Notes** Audio device driver names are limited to at most 12 characters.

The audio DDI is provided by the drv/audio module. Developers should supply “-dy -Ndrv/audio” on the final link command line to ensure that symbol dependencies can be satisfied when the driver is loaded.

**Name** backq – get pointer to the queue behind the current queue

**Synopsis** #include <sys/stream.h>

```
queue_t *backq(queue_t *cq);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *cq* The pointer to the current queue. `queue_t` is an alias for the [queue\(9S\)](#) structure.

**Description** The `backq()` function returns a pointer to the queue preceding *cq* (the current queue). If *cq* is a read queue, `backq()` returns a pointer to the queue downstream from *cq*, unless it is the stream end. If *cq* is a write queue, `backq()` returns a pointer to the next queue upstream from *cq*, unless it is the stream head.

**Return Values** If successful, `backq()` returns a pointer to the queue preceding the current queue. Otherwise, it returns `NULL`.

**Context** The `backq()` function can be called from user, interrupt, or kernel context.

**See Also** [queue\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** bcanput – test for flow control in specified priority band

**Synopsis** #include <sys/stream.h>

```
int bcanput(queue_t *q, unsigned char pri);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the message queue.

*pri* Message priority.

**Description** The `bcanput()` function searches through the stream (starting at *q*) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message of priority *pri* in the queue.

If *pri* is 0, `bcanput()` is equivalent to a call with `canput(9F)`.

`canputnext(q)` and `bcanputnext(q, pri)` should always be used in preference to `canput(q→q_next)` and `bcanput(q→q_next, pri)` respectively.

**Return Values** 1 If a message of priority *pri* can be placed on the queue.

0 If the priority band is full.

**Context** The `bcanput()` function can be called from user, interrupt, or kernel context.

**See Also** `bcanputnext(9F)`, `canput(9F)`, `canputnext(9F)`, `putbq(9F)`, `putnext(9F)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** Drivers are responsible for both testing a queue with `bcanput()` and refraining from placing a message on the queue if `bcanput()` fails.

- 
- Name** bcmp – compare two byte arrays
- Synopsis** `#include <sys/types.h>`  
`#include <sys/ddi.h>`
- ```
int bcmp(const void *s1, const void *s2, size_t len);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *s1* Pointer to the first character string.  
*s2* Pointer to the second character string.  
*len* Number of bytes to be compared.
- Description** The `bcmp()` function compares two byte arrays of length *len*.
- Return Values** The `bcmp()` function returns 0 if the arrays are identical, or 1 if they are not.
- Context** The `bcmp()` function can be called from user, interrupt, or kernel context.
- See Also** [strcmp\(9F\)](#)  
[Writing Device Drivers](#)
- Notes** Unlike [strcmp\(9F\)](#), `bcmp()` does not terminate when it encounters a null byte.

**Name** bcopy – copy data between address locations in the kernel

**Synopsis** #include <sys/types.h>  
#include <sys/sunddi.h>

```
void bcopy(const void *from, void *to, size_t bcount);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *from* Source address from which the copy is made.

*to* Destination address to which copy is made.

*bcount* The number of bytes moved.

**Description** The `bcopy()` function copies *bcount* bytes from one kernel address to another. If the input and output addresses overlap, the command executes, but the results may not be as expected.

Note that `bcopy()` should never be used to move data in or out of a user buffer, because it has no provision for handling page faults. The user address space can be swapped out at any time, and `bcopy()` always assumes that there will be no paging faults. If `bcopy()` attempts to access the user buffer when it is swapped out, the system will panic. It is safe to use `bcopy()` to move data within kernel space, since kernel space is never swapped out.

**Context** The `bcopy()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Copying data between address locations in the kernel:

An I/O request is made for data stored in a RAM disk. If the I/O operation is a read request, the data is copied from the RAM disk to a buffer (line 8). If it is a write request, the data is copied from a buffer to the RAM disk (line 15). `bcopy()` is used since both the RAM disk and the buffer are part of the kernel address space.

```
1 #define RAMDNBLK 1000           /* blocks in the RAM disk */
2 #define RAMDBSIZ 512           /* bytes per block */
3 char ramdbls[RAMDNBLK][RAMDBSIZ]; /* blocks forming RAM
                                     /* disk
...
4
5 if (bp->b_flags & B_READ)       /* if read request, copy data */
6                                 /* from RAM disk data block */
7                                 /* to system buffer */
8     bcopy(&ramdbls[bp->b_blkno][0], bp->b_un.b_addr,
9          bp->b_bcount);
10
11 else                            /* else write request, */
12                                 /* copy data from a */
13                                 /* system buffer to RAM disk */
14                                 /* data block */
```



**EXAMPLE 1** Copying data between address locations in the kernel: *(Continued)*

```
15         bcopy(bp->b_un.b_addr, &ramdblk[bp->b_blkno][0],
16         bp->b_bcount);
```

**See Also** [copyin\(9F\)](#), [copyout\(9F\)](#)

*Writing Device Drivers*

**Warnings** The *from* and *to* addresses must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.

**Name** bioaligned – check alignment of buffer data

**Synopsis** `#include<sys/ddi.h>`  
`#include<sys/sunddi.h>`

```
int bioaligned(struct buf *bp, size_t align, int flags);
```

**Parameters** *bp* Pointer to the [buf\(9S\)](#) structure describing the original I/O request.

*align* Power of two alignment being checked.

*flags* Determines the type of alignment being checked. Flag values supported are BIOALIGNED\_BEGIN (which checks the beginning address alignment), and BIOALIGNED\_END (which checks the ending address alignment).

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The `bioaligned()` function checks the alignment of memory associated with a buffer. It returns 1 if the buffer is aligned and 0 if the buffer is unaligned.

**Return Values** The `bioaligned()` function returns 1 if the buffer is aligned and 0 if the buffer is unaligned.

**Context** The `bioaligned()` function can be called from any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [biodone\(9F\)](#), [bp\\_mapin\(9F\)](#), [freerbuf\(9F\)](#), [getrbuf\(9F\)](#), [buf\(9S\)](#)

**Name** bioclone – clone another buffer

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
struct buf *bioclone(struct buf *bp, off_t off, size_t len, dev_t dev,
    daddr_t blkno, int (*iodone) (struct buf *), struct buf *bp_mem,
    int sleepflag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- bp* Pointer to the [buf\(9S\)](#) structure describing the original I/O request.
- off* Offset within original I/O request where new I/O request should start.
- len* Length of the I/O request.
- dev* Device number.
- blkno* Block number on device.
- iodone* Specific [biodone\(9F\)](#) routine.
- bp\_mem* Pointer to a buffer structure to be filled in or NULL.
- sleepflag* Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return NULL immediately if memory is not available.

**Description** The `bioclone()` function returns an initialized buffer to perform I/O to a portion of another buffer. The new buffer will be set up to perform I/O to the range within the original I/O request specified by the parameters *off* and *len*. An offset 0 starts the new I/O request at the same address as the original request. *off* + *len* must not exceed *b\_bcount*, the length of the original request. The device number *dev* specifies the device to which the buffer is to perform I/O. *blkno* is the block number on device. It will be assigned to the *b\_blkno* field of the cloned buffer structure. *iodone* lets the driver identify a specific [biodone\(9F\)](#) routine to be called by the driver when the I/O is complete. *bp\_mem* determines from where the space for the buffer should be allocated. If *bp\_mem* is NULL, `bioclone()` will allocate a new buffer using [getrbuf\(9F\)](#). If *sleepflag* is set to `KM_SLEEP`, the driver may sleep until space is freed up. If *sleepflag* is set to `KM_NOSLEEP`, the driver will not sleep. In either case, a pointer to the allocated space is returned or NULL to indicate that no space was available. After the transfer is completed, the buffer has to be freed using [freerbuf\(9F\)](#). If *bp\_mem* is not NULL, it will be used as the space for the buffer structure. The driver has to ensure that *bp\_mem* is initialized properly either using [getrbuf\(9F\)](#) or [bioinit\(9F\)](#).

If the original buffer is mapped into the kernel virtual address space using [bp\\_mapin\(9F\)](#) before calling `bioclone()`, a clone buffer will share the kernel mapping of the original buffer. An additional `bp_mapin()` to get a kernel mapping for the clone buffer is not necessary.

The driver has to ensure that the original buffer is not freed while any of the clone buffers is still performing I/O. The `biodone()` function has to be called on all clone buffers before it is called on the original buffer.

**Return Values** The `bioclone()` function returns a pointer to the initialized buffer header, or `NULL` if no space is available.

**Context** The `bioclone()` function can be called from user, interrupt, or interrupt context. Drivers must not allow `bioclone()` to sleep if called from an interrupt routine.

**Examples** **EXAMPLE 1** Using `bioclone()` for Disk Striping

A device driver can use `bioclone()` for disk striping. For each disk in the stripe, a clone buffer is created which performs I/O to a portion of the original buffer.

```
static int
stripe_strategy(struct buf *bp)
{
    ...
    bp_orig = bp;
    bp_1 = bioclone(bp_orig, 0, size_1, dev_1, blkno_1,
                   stripe_done, NULL, KM_SLEEP);
    fragment++;
    ...
    bp_n = bioclone(bp_orig, offset_n, size_n, dev_n,
                   blkno_n, stripe_done, NULL, KM_SLEEP);
    fragment++;
    /* submit bp_1 ... bp_n to device */
    xxstrategy(bp_x);
    return (0);
}

static uint_t
xxintr(caddr_t arg)
{
    ...
    /*
     * get bp of completed subrequest. biodone(9F) will
     * call stripe_done()
     */
    biodone(bp);
    return (0);
}

static int
stripe_done(struct buf *bp)
{
    ...
```

**EXAMPLE 1** Using `bioclon()` for Disk Striping (Continued)

```
freerbuf(bp);
fragment--;
if (fragment == 0) {
    /* get bp_orig */
    biodone(bp_orig);
}
return (0);
}
```

**See Also** [biodone\(9F\)](#), [bp\\_mapin\(9F\)](#), [freerbuf\(9F\)](#), [getrbuf\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** biodone – release buffer after buffer I/O transfer and notify blocked threads

**Synopsis** `#include <sys/types.h>`  
`#include <sys/buf.h>`

```
void biodone(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to a [buf\(9S\)](#) structure.

**Description** The `biodone()` function notifies blocked processes waiting for the I/O to complete, sets the `B_DONE` flag in the `b_flags` field of the [buf\(9S\)](#) structure, and releases the buffer if the I/O is asynchronous. `biodone()` is called by either the driver interrupt or [strategy\(9E\)](#) routines when a buffer I/O request is complete.

The `biodone()` function provides the capability to call a completion routine if *bp* describes a kernel buffer. The address of the routine is specified in the `b_iodone` field of the [buf\(9S\)](#) structure. If such a routine is specified, `biodone()` calls it and returns without performing any other actions. Otherwise, it performs the steps above.

**Context** The `biodone()` function can be called from user, interrupt, or kernel context.

**Examples** Generally, the first validation test performed by any block device [strategy\(9E\)](#) routine is a check for an end-of-file (EOF) condition. The [strategy\(9E\)](#) routine is responsible for determining an EOF condition when the device is accessed directly. If a [read\(2\)](#) request is made for one block beyond the limits of the device (line 10), it will report an EOF condition. Otherwise, if the request is outside the limits of the device, the routine will report an error condition. In either case, report the I/O operation as complete (line 27).

```

1  #define RAMDNBLK    1000    /* Number of blocks in RAM disk */
2  #define RAMDBSIZ    512    /* Number of bytes per block */
3  char ramdbls[RAMDNBLK][RAMDBSIZ]; /* Array containing RAM disk */
4
5  static int
6  ramdstrategy(struct buf *bp)
7  {
8      daddr_t blkno = bp->b_blkno; /* get block number */
9
10     if ((blkno < 0) || (blkno >= RAMDNBLK)) {
11         /*
12          * If requested block is outside RAM disk
13          * limits, test for EOF which could result
14          * from a direct (physio) request.
15          */
16         if ((blkno == RAMDNBLK) && (bp->b_flags & B_READ)) {
17             /*
18              * If read is for block beyond RAM disk

```

```

19         * limits, mark EOF condition.
20         */
21         bp->b_resid = bp->b_bcount;    /* compute return value */
22
23     } else {    /* I/O attempt is beyond */
24         bp->b_error = ENXIO;    /* limits of RAM disk */
25         bp->b_flags |= B_ERROR;    /* return error */
26     }
27     biodone(bp);    /* mark I/O complete (B_DONE) */
28     /*
29     * Wake any processes awaiting this I/O
30     * or release buffer for asynchronous
31     * (B_ASYNC) request.
32     */
33     return (0);
34 }

```

**See Also** [read\(2\)](#), [strategy\(9E\)](#), [biowait\(9F\)](#), [ddi\\_add\\_intr\(9F\)](#), [delay\(9F\)](#), [timeout\(9F\)](#), [untimeout\(9F\)](#), [buf\(9S\)](#)

#### *Writing Device Drivers*

**Warnings** After calling `biodone()`, `bp` is no longer available to be referred to by the driver. If the driver makes any reference to `bp` after calling `biodone()`, a panic may result.

**Notes** Drivers that use the `b_iodone` field of the [buf\(9S\)](#) structure to specify a substitute completion routine should save the value of `b_iodone` before changing it, and then restore the old value before calling `biodone()` to release the buffer.

**Name** bioerror – indicate error in buffer header

**Synopsis**

```
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>
```

```
void bioerror(struct buf *bp, int error);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *bp* Pointer to the [buf\(9S\)](#) structure describing the transfer.  
*error* Error number to be set, or zero to clear an error indication.

**Description** If *error* is non-zero, `bioerror()` indicates an error has occurred in the [buf\(9S\)](#) structure. A subsequent call to [geterror\(9F\)](#) will return *error*.

If *error* is 0, the error indication is cleared and a subsequent call to [geterror\(9F\)](#) will return 0.

**Context** `bioerror()` can be called from any context.

**See Also** [strategy\(9E\)](#), [geterror\(9F\)](#), [getrbuf\(9F\)](#), [buf\(9S\)](#)



- 
- Name** biofini – uninitialized a buffer structure
- Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`
- ```
void biofini(struct buf *bp);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *bp* Pointer to the buffer header structure.
- Description** The `biofini()` function uninitialized a [buf\(9S\)](#) structure. If a buffer structure has been allocated and initialized using [kmem\\_alloc\(9F\)](#) and [bioinit\(9F\)](#) it needs to be uninitialized using `biofini()` before calling [kmem\\_free\(9F\)](#). It is not necessary to call `biofini()` before freeing a buffer structure using [freerbuf\(9F\)](#) because `freerbuf()` will call `biofini()` directly.
- Context** The `biofini()` function can be called from any context.
- Examples** `EXAMPLE1 Usingbiofini()`
- ```
struct buf *bp = kmem_alloc(biosize(), KM_SLEEP);
bioinit(bp);
/* use buffer */
biofini(bp);
kmem_free(bp, biosize());
```
- See Also** [bioinit\(9F\)](#), [bioreset\(9F\)](#), [biosize\(9F\)](#), [freerbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [kmem\\_free\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** bioinit – initialize a buffer structure

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
void bioinit(struct buf *bp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *bp* Pointer to the buffer header structure.

**Description** The `bioinit()` function initializes a [buf\(9S\)](#) structure. A buffer structure contains state information which has to be initialized if the memory for the buffer was allocated using [kmem\\_alloc\(9F\)](#). This is not necessary for a buffer allocated using [getrbuf\(9F\)](#) because `getrbuf()` will call `bioinit()` directly.

**Context** The `bioinit()` function can be called from any context.

**Examples** EXAMPLE1 Using `bioinit()`

```
struct buf *bp = kmem_alloc(biosize(), KM_SLEEP);
bioinit(bp);
/* use buffer */
```

**See Also** [biofini\(9F\)](#), [bioreset\(9F\)](#), [biosize\(9F\)](#), [getrbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

- 
- Name** biomodified – check if a buffer is modified
- Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int biomodified(struct buf *bp);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *bp* Pointer to the buffer header structure.
- Description** The `biomodified()` function returns status to indicate if the memory associated with a [buf\(9S\)](#) structure is modified. The `biomodified()` function may only be supported for some types of `buf` structures (like a paged-I/O request). If called on an unsupported `buf` structure, `-1` is returned. The `biomodified()` function will check the memory pages associated with a buffer and determine whether the Virtual Memory system's modification bit is set. If at least one of these pages is modified, the buffer is indicated as modified. A filesystem will mark the pages unmodified when it writes the pages to the backing store. The `biomodified()` function can be used to detect any modifications to the memory pages while I/O is in progress.
- A device driver can use `biomodified()` for disk mirroring. An application is allowed to `mmap` a file which can reside on a disk which is mirrored by multiple submirrors. If the file system writes the file to the backing store, it is written to all submirrors in parallel. It must be ensured that the copies on all submirrors are identical. The `biomodified()` function can be used in the device driver to detect any modifications to the buffer by the user program during the time the buffer is written to multiple submirrors.
- Return Values** The `biomodified()` function returns the following values:
- 1 Buffer is modified.
  - 0 Buffer is not modified.
  - 1 Unknown whether buffer is modified.
- Context** `biomodified()` can be called from any context.
- See Also** [bioaligned\(9F\)](#), [bp\\_mapin\(9F\)](#), [buf\(9S\)](#)
- Writing Device Drivers*

**Name** bioreset – reuse a private buffer header after I/O is complete

**Synopsis**

```
#include <sys/buf.h>
#include <sys/ddi.h>
```

```
void bioreset(struct buf *bp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *bp* Pointer to the [buf\(9S\)](#) structure.

**Description** `bioreset()` is used by drivers that allocate private buffers with [getrbuf\(9F\)](#) or [kmem\\_alloc\(9F\)](#) and want to reuse them in multiple transfers before freeing them with [freerbuf\(9F\)](#) or [kmem\\_free\(9F\)](#). `bioreset()` resets the buffer header to the state it had when initially allocated by `getrbuf()` or initialized by [bioinit\(9F\)](#).

**Context** `bioreset()` can be called from any context.

**See Also** [strategy\(9E\)](#), [bioinit\(9F\)](#), [biofini\(9F\)](#), [freerbuf\(9F\)](#), [getrbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [kmem\\_free\(9F\)](#), [buf\(9S\)](#)

**Notes** *bp* must not describe a transfer in progress.

**Name** biosize – returns size of a buffer structure

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
size_t biosize(void)
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The biosize() function returns the size in bytes of the [buf\(9S\)](#) structure. The biosize() function is used by drivers in combination with [kmem\\_alloc\(9F\)](#) and [bioinit\(9F\)](#) to allocate buffer structures embedded in other data structures.

**Context** The biosize() function can be called from any context.

**See Also** [biofini\(9F\)](#), [bioinit\(9F\)](#), [getrbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** biowait – suspend processes pending completion of block I/O

**Synopsis** `#include <sys/types.h>`  
`#include <sys/buf.h>`

```
int biowait(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the buf structure describing the transfer.

**Description** Drivers allocating their own buf structures with [getrbuf\(9F\)](#) can use the `biowait()` function to suspend the current thread and wait for completion of the transfer.

Drivers must call [biodone\(9F\)](#) when the transfer is complete to notify the thread blocked by `biowait()`. `biodone()` is usually called in the interrupt routine.

**Return Values** `0` Upon success  
non-zero Upon I/O failure. `biowait()` calls [geterror\(9F\)](#) to retrieve the error number which it returns.

**Context** `biowait()` can be called from user context only.

**See Also** [biodone\(9F\)](#), [geterror\(9F\)](#), [getrbuf\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** bp\_copyin – copy from a buf(9S) into a driver buffer

**Synopsis** #include <sys/types.h>  
#include <sys/buf.h>

```
int bp_copyin(struct buf *bp, void *driverbuf, offset_t offset,  
              size_t size);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the buffer header structure to copy from.  
*driverbuf* Driver buffer to copy to.  
*offset* Offset into *bp* where to start copying.  
*size* Size of copy.

**Description** The bp\_copyin() function copies *size* bytes into the memory associated with *bp* to the destination driver buffer *driverbuf*. The *offset* only applies to *bp*.

**Return Values** Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if *bp* references invalid pages.

**Context** The bp\_copyin() function can be called from user or kernel context only.

**See Also** [bp\\_copyout\(9F\)](#), [bp\\_mapin\(9F\)](#), [bp\\_mapout\(9F\)](#), [ddi\\_copyout\(9F\)](#), [buf\(9S\)](#)

**Name** bp\_copyout – copy from a driver buffer into a buf(9S)

**Synopsis** #include <sys/types.h>  
#include <sys/buf.h>

```
int bp_copyout(void *driverbuf, struct buf *bp, offset_t offset,  
              size_t size);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the buffer header structure to copy to.  
*driverbuf* Driver buffer to copy from.  
*offset* Offset into *bp* where to start copying.  
*size* Size of copy.

**Description** The bp\_copyout () function copies *size* bytes starting from the driver buffer *driverbuf* to *offset* bytes into the memory associated with *bp*. The *offset* only applies to *bp*.

**Return Values** Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if *bp* references invalid pages.

**Context** The bp\_copyout () function can be called from user or kernel context only.

**See Also** bp\_copyin(9F), bp\_mapin(9F), bp\_mapout(9F), ddi\_copyout(9F), buf(9S)



**Name** bp\_mapin – allocate virtual address space

**Synopsis** #include <sys/types.h>  
#include <sys/buf.h>

```
void bp_mapin(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the buffer header structure.

**Description** The `bp_mapin()` function makes buffer data accessible to a driver by mapping the data into a contiguous kernel virtual address region. After `bp_mapin()`, the mapped buffer is accessible via the `bp->b_un.b_addr` field of the [buf\(9S\)](#) structure.

If `bp_mapin()` is unable to establish a mapping, it returns without allocating space and no mapping is performed.

Any resources associated with establishing the mapping are deallocated by the [bp\\_mapout\(9F\)](#) function.

**Context** The `bp_mapin()` function can be called from user and kernel contexts.

**See Also** [bioaligned\(9F\)](#), [bp\\_mapout\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Warnings** Because `bp_mapin()` does not keep a reference count, `bp_mapout()` will wipe out any kernel mapping, including mappings that a layer above the device driver might still rely on. See [bp\\_mapout\(9F\)](#) for details.

When a driver needs to look directly at buffer data, it should call `bp_mapin()` without conditioning the call on other aspects of the [buf\(9S\)](#) structure like `b_flags`.

To check alignment of buffer memory, it may be more efficient for a driver to use [bioaligned\(9F\)](#) instead of `bp_mapin()`.

**Name** bp\_mapout – deallocate virtual address space

**Synopsis**

```
#include <sys/types.h>
#include <sys/buf.h>
```

```
void bp_mapout(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the buffer header structure.

**Description** bp\_mapout() deallocates system virtual address space allocated by a previous call to [bp\\_mapin\(9F\)](#). bp\_mapout() should only be called on buffers which have been allocated and are owned by the device driver. It must not be called on buffers passed to the driver through the [strategy\(9E\)](#) entry point (for example a filesystem). Because [bp\\_mapin\(9F\)](#) does not keep a reference count, bp\_mapout() will wipe out any kernel mapping that a layer above the device driver might rely on.

**Context** bp\_mapout() can be called from user context only.

**See Also** [strategy\(9E\)](#), [bp\\_mapin\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** btop – convert size in bytes to size in pages (round down)

**Synopsis** #include <sys/ddi.h>

```
unsigned long btop(unsigned long numbytes);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *numbytes* Number of bytes.

**Description** The `btop()` function returns the number of memory pages that are contained in the specified number of bytes, with downward rounding in the case that the byte count is not a page multiple. For example, if the page size is 2048, then `btop(4096)` returns 2, and `btop(4097)` returns 2 as well. `btop(0)` returns 0.

**Return Values** The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

**Context** The `btop()` function can be called from user, interrupt, or kernel context.

**See Also** [btopr\(9F\)](#), [ddi\\_btop\(9F\)](#), [ptob\(9F\)](#)

*Writing Device Drivers*

**Name** btopr – convert size in bytes to size in pages (round up)

**Synopsis** `#include <sys/ddi.h>`

```
unsigned long btopr(unsigned long numbytes);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *numbytes* Number of bytes.

**Description** The `btopr()` function returns the number of memory pages contained in the specified number of bytes memory, rounded up to the next whole page. For example, if the page size is 2048, then `btopr(4096)` returns 2, and `btopr(4097)` returns 3.

**Return Values** The return value is always the number of pages. There are no invalid input values, and therefore no error return values.

**Context** The `btopr()` function can be called from user, interrupt, or kernel context.

**See Also** [btop\(9F\)](#), [ddi\\_btopr\(9F\)](#), [ptob\(9F\)](#)

*Writing Device Drivers*

- 
- Name** bufcall – call a function when a buffer becomes available
- Synopsis**

```
#include <sys/types.h>
#include <sys/stream.h>

bufcall_id_t bufcall(size_t size, uint_t pri, void (*func)(void *arg),
                    void * arg);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters**
- size* Number of bytes required for the buffer.
  - pri* Priority of the [allocb\(9F\)](#) allocation request (not used).
  - func* Function or driver routine to be called when a buffer becomes available.
  - arg* Argument to the function to be called when a buffer becomes available.
- Description** The `bufcall()` function serves as a [timeout\(9F\)](#) call of indeterminate length. When a buffer allocation request fails, `bufcall()` can be used to schedule the routine *func*, to be called with the argument *arg* when a buffer becomes available. *func* may call `allocb()` or it may do something else.
- Return Values** If successful, `bufcall()` returns a `bufcall` ID that can be used in a call to `unbufcall()` to cancel the request. If the `bufcall()` scheduling fails, *func* is never called and 0 is returned.
- Context** The `bufcall()` function can be called from user, interrupt, or kernel context.
- Examples** **EXAMPLE 1** Calling a function when a buffer becomes available:
- The purpose of this [srv\(9E\)](#) service routine is to add a header to all `M_DATA` messages. Service routines must process all messages on their queues before returning, or arrange to be rescheduled
- While there are messages to be processed (line 13), check to see if it is a high priority message or a normal priority message that can be sent on (line 14). Normal priority message that cannot be sent are put back on the message queue (line 34). If the message was a high priority one, or if it was normal priority and [canputnext\(9F\)](#) succeeded, then send all but `M_DATA` messages to the next module with [putnext\(9F\)](#) (line 16).
- For `M_DATA` messages, try to allocate a buffer large enough to hold the header (line 18). If no such buffer is available, the service routine must be rescheduled for a time when a buffer is available. The original message is put back on the queue (line 20) and `bufcall` (line 21) is used to attempt the rescheduling. It will succeed if the rescheduling succeeds, indicating that `qenable` will be called subsequently with the argument *q* once a buffer of the specified size (`sizeof (struct hdr)`) becomes available. If it does, [qenable\(9F\)](#) will put *q* on the list of queues to have their service routines called. If `bufcall()` fails, [timeout\(9F\)](#) (line 22) is used to try again in about a half second.

EXAMPLE 1 Calling a function when a buffer becomes available: (Continued)

If the buffer allocation was successful, initialize the header (lines 25–28), make the message type `M_PROTO` (line 29), link the `M_DATA` message to it (line 30), and pass it on (line 31).

Note that this example ignores the bookkeeping needed to handle `bufcall()` and `timeout(9F)` cancellation for ones that are still outstanding at close time.

```

1  struct hdr {
2      unsigned int h_size;
3      int          h_version;
4  };
5
6  void xxxsrv(q)
7      queue_t *q;
8  {
9      mblk_t *bp;
10     mblk_t *mp;
11     struct hdr *hp;
12
13     while ((mp = getq(q)) != NULL) { /* get next message */
14         if (mp->b_datap->db_type >= QPCTL || /* if high priority */
15             canputnext(q)) { /* normal & can be passed */
16             if (mp->b_datap->db_type != M_DATA)
17                 putnext(q, mp); /* send all but M_DATA */
18             else {
19                 bp = allocb(sizeof(struct hdr), BPRI_LO);
20                 if (bp == NULL) { /* if unsuccessful */
21                     putbq(q, mp); /* put it back */
22                     if (!bufcall(sizeof(struct hdr), BPRI_LO,
23                                 qenable, q)) /* try to reschedule */
24                         timeout(qenable, q, drv_usecstohz(500000));
25                     return (0);
26                 }
27                 hp = (struct hdr *)bp->b_wptr;
28                 hp->h_size = msgdsize(mp); /* initialize header */
29                 hp->h_version = 1;
30                 bp->b_wptr += sizeof(struct hdr);
31                 bp->b_datap->db_type = M_PROTO; /* make M_PROTO */
32                 bp->b_cont = mp; /* link it */
33                 putnext(q, bp); /* pass it on */
34             }
35         } else { /* normal priority, canputnext failed */
36             putbq(q, mp); /* put back on the message queue */
37             return (0);
38         }
39     }
40 }

```

**EXAMPLE 1** Calling a function when a buffer becomes available: *(Continued)*

```
        return (0);  
38    }
```

**See Also** [srv\(9E\)](#), [allocb\(9F\)](#), [canputnext\(9F\)](#), [esballoc\(9F\)](#), [esbbscall\(9F\)](#), [putnext\(9F\)](#), [genable\(9F\)](#), [testb\(9F\)](#), [timeout\(9F\)](#), [unbufcall\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** Even when *func* is called by `bufcall()`, [allocb\(9F\)](#) can fail if another module or driver had allocated the memory before *func* was able to call [allocb\(9F\)](#).

**Name** bzero – clear memory for a given number of bytes

**Synopsis** `#include <sys/types.h>`  
`#include <sys/ddi.h>`

```
void bzero(void *addr, size_t bytes);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *addr* Starting virtual address of memory to be cleared.  
*bytes* The number of bytes to clear starting at *addr*.

**Description** The `bzero()` function clears a contiguous portion of memory by filling it with zeros.

**Context** The `bzero()` function can be called from user, interrupt, or kernel context.

**See Also** [bcopy\(9F\)](#), [clrbuf\(9F\)](#), [kmem\\_zalloc\(9F\)](#)

*Writing Device Drivers*

**Warnings** The address range specified must be within the kernel space. No range checking is done. If an address outside of the kernel space is selected, the driver may corrupt the system in an unpredictable way.



- 
- Name** canput – test for room in a message queue
- Synopsis** `#include <sys/stream.h>`
- ```
int canput(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the message queue.
- Description** canput ( ) searches through the stream (starting at *q*) until it finds a queue containing a service routine where the message can be enqueued, or until it reaches the end of the stream. If found, the queue containing the service routine is tested to see if there is room for a message in the queue.
- canputnext(*q*) and bcanputnext(*q*, *pri*) should always be used in preference to canput(*q*→*q\_next*) and bcanput(*q*→*q\_next*, *pri*) respectively.
- Return Values**
- 1 If the message queue is not full.
  - 0 If the queue is full.
- Context** canput ( ) can be called from user or interrupt context.
- See Also** [bcanput\(9F\)](#), [bcanputnext\(9F\)](#), [canputnext\(9F\)](#), [putbq\(9F\)](#), [putnext\(9F\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*
- Warnings** Drivers are responsible for both testing a queue with canput ( ) and refraining from placing a message on the queue if canput ( ) fails.

**Name** canputnext, bcanputnext – test for room in next module's message queue

**Synopsis** #include <sys/stream.h>

```
int canputnext(queue_t *q);  
int bcanputnext(queue_t *q, unsigned char pri);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to a message queue belonging to the invoking module.  
*pri* Minimum priority level.

**Description** The invocation `canputnext(q)`; is an atomic equivalent of the `canput(q→q_next)`; routine. That is, the STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its `q_next` field and then invoking `canput(9F)` proceeds without interference from other threads.

`bcanputnext(q, pri)`; is the equivalent of the `bcanput(q→q_next, pri)`; routine.

`canputnext(q)`; and `bcanputnext(q, pri)`; should always be used in preference to `canput(q→q_next)`; and `bcanput(q→q_next, pri)`; respectively.

See `canput(9F)` and `bcanput(9F)` for further details.

**Return Values** 1 If the message queue is not full.  
0 If the queue is full.

**Context** The `canputnext()` and `bcanputnext()` functions can be called from user, interrupt, or kernel context.

**Warnings** Drivers are responsible for both testing a queue with `canputnext()` or `bcanputnext()` and refraining from placing a message on the queue if the queue is full.

**See Also** `bcanput(9F)`, `canput(9F)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** clrbuf – erase the contents of a buffer

**Synopsis**

```
#include <sys/types.h>
#include <sys/buf.h>
```

```
void clrbuf(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the [buf\(9S\)](#) structure.

**Description** The `clrbuf()` function zeros a buffer and sets the `b_resid` member of the [buf\(9S\)](#) structure to 0. Zeros are placed in the buffer starting at `bp->b_un.b_addr` for a length of `bp->b_bcount` bytes. `b_un.b_addr` and `b_bcount` are members of the [buf\(9S\)](#) data structure.

**Context** The `clrbuf()` function can be called from user, interrupt, or kernel context.

**See Also** [getrbuf\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** cmn\_err, vcmn\_err, zcmn\_err – display an error message or panic the system

**Synopsis**

```
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

void cmn_err(int level, char *format...

#include <sys/varargs.h>

void vcmn_err(int level, char *format, va_list ap);

#include <sys/types.h>

void zcmn_err(zoneid_t zoneid, int level, char *format...);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

### Parameters

cmn\_err() *level* A constant indicating the severity of the error condition.

*format* Message to be displayed.

vcmn\_err() The vcmn\_err() function takes *level* and *format* as described for cmn\_err(), but its third argument is different:

*ap* Variable argument list passed to the function.

zcmn\_err() The zcmn\_err() function works exactly like cmn\_err(), but includes an additional argument:

*zoneid* Zone to which log messages should be directed. See [zones\(5\)](#).

### Description

cmn\_err() The cmn\_err() function displays a specified message on the console. cmn\_err() can also panic the system. When the system panics, it attempts to save recent changes to data, display a “panic message” on the console, attempt to write a core file, and halt system processing. See the `CE_PANIC` *level* below.

*level* is a constant indicating the severity of the error condition. The four severity levels are:

`CE_CONT` Used to continue another message or to display an informative message not associated with an error. Note that multiple `CE_CONT` messages without a newline may or may not appear on the system console or in the system log as a single line message. A single line message may be produced by constructing the message with [sprintf\(9F\)](#) or [vsprintf\(9F\)](#) before calling cmn\_err().

`CE_NOTE` Used to display a message preceded with NOTICE. This message is used to report system events that do not necessarily require user action, but may interest the

system administrator. For example, a message saying that a sector on a disk needs to be accessed repeatedly before it can be accessed correctly might be noteworthy.

- CE\_WARN** Used to display a message preceded with **WARNING**. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
- CE\_PANIC** Used to display a message preceded with “panic”, and to panic the system. Drivers should specify this level only under the most severe conditions or when debugging a driver. A valid use of this level is when the system cannot continue to function. If the error is recoverable, or not essential to continued system operation, do not panic the system.

*format* is the message to be displayed. It is a character string which may contain plain characters and conversion specifications. By default, the message is sent both to the system console and to the system log.

Each conversion specification in *format* is introduced by the % character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional l (ll) specifying that a following d, D, o, O, x, X, or u conversion character applies to a long (long long) integer argument. An l (ll) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

- d,D,o,O,x,X,u** The integer argument is converted to signed decimal (d, D), unsigned octal (o, O), unsigned hexadecimal (x, X), or unsigned decimal (u), respectively, and displayed. The letters abcdef are used for x and X conversion.
- c** The character value of the argument is displayed.
- b** The %b conversion specification allows bit values to be displayed meaningfully. Each %b takes an integer value and a format string from the argument list. The first character of the format string should be the output base encoded as a control character. This base is used to display the integer argument. The remaining groups of characters in the format string consist of a bit number (between 1 and 32, also encoded as a control character) and the next characters (up to the next control character or '\0') give the name of the bit field. The string corresponding to the bit fields set in the integer argument is displayed after the numerical value. See EXAMPLE section.

- p** The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void \* when using the %p format specifier.
- s** The argument is taken to be a string (character pointer), and characters from the string are displayed until a null character is encountered. If the character pointer is NULL, the string <null string> is used in its place.
- %** Copy a %; no argument is converted.

The first character in *format* affects where the message will be written:

- !** The message goes only to the system log.
- ^** The message goes only to the console.
- ?** If *level* is also CE\_CONT, the message is always sent to the system log, but is only written to the console when the system has been booted in verbose mode. See [kernel\(1M\)](#). If neither condition is met, the '?' character has no effect and is simply ignored.

Refer to [syslogd\(1M\)](#) to determine where the system log is written.

The `cmn_err()` function sends log messages to the log of the global zone. `cmn_err()` appends a \n to each *format*, except when *level* is CE\_CONT.

`vcmn_err()` The `vcmn_err()` function is identical to `cmn_err()` except that its last argument, *ap*, is a pointer to a variable list of arguments. *ap* contains the list of arguments used by the conversion specifications in *format*. *ap* must be initialized by calling [va\\_start\(9F\)](#). [va\\_end\(9F\)](#) is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by [va\\_start\(9F\)](#) and [va\\_end\(9F\)](#), are possible.

`zcmn_err()` With the exception of its first argument (*zoneid*), `zcmn_err()` is identical to `cmn_err()`. *zoneid* is the numeric ID of the zone to which the message should be directed. Note that *zoneid* only has an effect if the message is sent to the system log. Using *zoneid* will cause messages to be sent to the log associated with the specified local zone rather than the log in the global zone. This is accomplished by the message being received and processed by the [syslogd\(1M\)](#) process running in the specified zone instead of the one running in the global zone. You can retrieve a process zone ID from its credential structure using [crgetzzoneid\(9F\)](#).

**Return Values** None. However, if an unknown *level* is passed to `cmn_err()`, the following panic error message is displayed:

```
panic: unknown level in cmn_err (level=level, msg=format)
```

**Context** The `cmn_err()` function can be called from user, kernel, interrupt, or high-level interrupt context.

**Examples** EXAMPLE 1 Using `cmn_err()`

This first example shows how `cmn_err()` can record tracing and debugging information only in the system log (lines 17); display problems with a device only on the system console (line 23); or display problems with the device on both the system console and in the system log (line 28).

```

1  struct  reg {
2      uchar_t data;
3      uchar_t csr;
4  };
5
6  struct  xxstate {
7      . . .
8      dev_info_t *dip;
9      struct reg *regp;
10     . . .
11 };
12
13 dev_t dev;
14 struct xxstate *xsp;
15     . . .
16 #ifdef DEBUG /* in debugging mode, log function call */
17     cmn_err(CE_CONT, "!!%s%d: xxopen function called.",
18           ddi_binding_name(xsp->dip), getminor(dev));
19 #endif /* end DEBUG */
20     . . .
21 /* display device power failure on system console */
22     if ((xsp->regp->csr & POWER) == OFF)
23         cmn_err(CE_NOTE, "^OFF.",
24               ddi_binding_name(xsp->dip), getminor(dev));
25     . . .
26 /* display warning if device has bad VTOC */
27     if (xsp->regp->csr & BADVTOC)
28         cmn_err(CE_WARN, "%s%d: xxopen: Bad VTOC.",
29               ddi_binding_name(xsp->dip), getminor(dev));

```

**EXAMPLE 2** Using the `%b` conversion specification

This example shows how to use the `%b` conversion specification. Because of the leading `'?` character in the format string, this message will always be logged, but it will only be displayed when the kernel is booted in verbose mode.

```
cmn_err(CE_CONT, "?reg=0x%b\n", regval, "\020\3Intr\2Err\1Enable");
```

**EXAMPLE 3** Using `regval`

When `regval` is set to (decimal) 13, the following message would be displayed:

```
reg=0xd<Intr, ,Enable>
```

**EXAMPLE 4** Error Routine

This example shows an error reporting routine which accepts a variable number of arguments and displays a single line error message both in the system log and on the system console. Note the use of `vsprintf()` to construct the error message before calling `cmn_err()`.

```
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#define MAX_MSG 256;

void
xxerror(dev_info_t *dip, int level, const char *fmt, . . . )
{
    va_list    ap;
    int        instance;
    char        buf[MAX_MSG], *name;

    instance = ddi_get_instance(dip);
    name = ddi_binding_name(dip);

    /* format buf using fmt and arguments contained in ap */

    va_start(ap, fmt);
    vsprintf(buf, fmt, ap);
    va_end(ap);

    /* pass formatted string to cmn_err(9F) */

    cmn_err(level, "%s%d: %s", name, instance, buf);
}
```

**EXAMPLE 5** Log to Current Zone

This example shows how messages can be sent to the log of the zone in which a thread is currently running, when applicable. Note that most hardware-related messages should instead be sent to the global zone using `cmn_err()`.

```
zcmn_err(crgetzoneid(ddi_get_cred()), CE_NOTE, "out of processes0");
```

**See Also** [dmesg\(1M\)](#), [kernel\(1M\)](#), [printf\(3C\)](#), [zones\(5\)](#), [ddi\\_binding\\_name\(9F\)](#), [ddi\\_cred\(9F\)](#), [sprintf\(9F\)](#), [va\\_arg\(9F\)](#), [va\\_end\(9F\)](#), [va\\_start\(9F\)](#), [vsprintf\(9F\)](#)

*Writing Device Drivers*

**Warnings** The `cmn_err()` function with the `CE_CONT` argument can be used by driver developers as a driver code debugging tool. However, using `cmn_err()` in this capacity can change system timing characteristics.



**Notes** Messages of arbitrary length can be generated using `cmn_err()`, but if the call to `cmn_err()` is made from high-level interrupt context and insufficient memory is available to create a buffer of the specified size, the message will be truncated to `LOG_MSGSIZE` bytes (see `<sys/log.h>`). For this reason, callers of `cmn_err()` that require complete and accurate message generation should post down from high-level interrupt context before calling `cmn_err()`.

**Name** condvar, cv\_init, cv\_destroy, cv\_wait, cv\_signal, cv\_broadcast, cv\_wait\_sig, cv\_timedwait, cv\_timedwait\_sig, cv\_reltimedwait, cv\_reltimedwait\_sig – condition variable routines

**Synopsis** #include <sys/ksynch.h>

```
void cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
void cv_destroy(kcondvar_t *cvp);
void cv_wait(kcondvar_t *cvp, kmutex_t *mp);
void cv_signal(kcondvar_t *cvp);
void cv_broadcast(kcondvar_t *cvp);
int cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp);
clock_t cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
clock_t cv_reltimedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t delta,
                        time_res_t resolution);
clock_t cv_reltimedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t delta,
                            time_res_t resolution);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

|                   |                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Parameters</b> | <i>cvp</i>        | A pointer to an abstract data type <code>kcondvar_t</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|                   | <i>mp</i>         | A pointer to a mutual exclusion lock ( <code>kmutex_t</code> ), initialized by <a href="#">mutex_init(9F)</a> and held by the caller.                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                   | <i>name</i>       | Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they're a waste of kernel memory.)                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|                   | <i>type</i>       | The constant <code>CV_DRIVER</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                   | <i>arg</i>        | A type-specific argument, drivers should pass <code>arg</code> as NULL.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|                   | <i>timeout</i>    | A time, in absolute ticks since boot, when <code>cv_timedwait()</code> or <code>cv_timedwait_sig()</code> should return.                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|                   | <i>delta</i>      | A time, in relative ticks, when <code>cv_reltimedwait()</code> or <code>cv_reltimedwait_sig()</code> should return.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|                   | <i>resolution</i> | A flag that specifies how accurate the relative time interval should be. Possible values are <code>TR_NANOSEC</code> , <code>TR_MICROSEC</code> , <code>TR_MILLISEC</code> , <code>TR_SEC</code> , or <code>TR_CLOCK_TICK</code> . <code>TR_CLOCK_TICK</code> indicates that the interval should be aligned to system clock ticks. This information is used to anticipate or defer the timeout expiration in order to batch process similarly expiring events, allowing the system to stay idle for longer periods of time and enhance its power efficiency. |

**Description** Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling `cv_init()`, and must be deallocated by calling `cv_destroy()`.

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, `cv_wait()` is called with a related condition variable and the mutex. At some later point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with `cv_signal()` or `cv_broadcast()`, and then release the mutex.

The `cv_wait()` function suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

The `cv_signal()` function signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling `cv_broadcast()`. `cv_signal()` and `cv_broadcast()` can be called by a thread even if it does not hold the mutex passed into `cv_wait()`, though holding the mutex is necessary to ensure predictable scheduling.

The `cv_wait_sig()` function is similar to `cv_wait()` but returns 0 if a signal (for example, by `kill(2)`) is sent to the thread. In any case, the mutex is reacquired before returning.

The `cv_timedwait()` function is similar to `cv_wait()`, except that it returns -1 without the condition being signaled after the timeout time has been reached.

The `cv_timedwait_sig()` function is similar to `cv_timedwait()` and `cv_wait_sig()`, except that it returns -1 without the condition being signaled after the timeout time has been reached, or 0 if a signal (for example, by `kill(2)`) is sent to the thread.

For both `cv_timedwait()` and `cv_timedwait_sig()`, time is in absolute clock ticks since the last system reboot. The current time may be found by calling `ddi_get_lbolt(9F)`.

The `cv_reltimedwait()` function is similar to `cv_timedwait()`, except that it takes a relative time value as argument and it also takes an additional argument to specify the accuracy of such interval. The `cv_reltimedwait_sig()` function is analogous to `cv_timedwait_sig()` but takes the same arguments as `cv_reltimedwait()`.

**Return Values** 0 For `cv_wait_sig()`, `cv_timedwait_sig()`, and `cv_reltimedwait_sig()`, this value indicates that the condition was not necessarily signaled and the function returned because a signal (as in `kill(2)`) was pending.

- 1 For `cv_timedwait()`, `cv_timedwait_sig()`, `cv_reltimedwait()`, and `cv_reltimedwait_sig()`, this value indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.
- >0 For `cv_wait_sig()`, `cv_timedwait()`, `cv_timedwait_sig()`, `cv_reltimedwait()`, and `cv_reltimedwait_sig()`, a value greater than 0 indicates that the condition was met and the function returned due to a call to either `cv_signal()` or `cv_broadcast()`, or due to a premature wakeup (see NOTES).

**Context** These functions can be called from user, kernel or interrupt context. In most cases, however, `cv_wait()`, `cv_timedwait()`, `cv_wait_sig()`, `cv_timedwait_sig()`, `cv_reltimedwait()`, and `cv_reltimedwait_sig()` should not be called from interrupt context, and cannot be called from a high-level interrupt context.

If `cv_wait()`, `cv_timedwait()`, `cv_wait_sig()`, `cv_timedwait_sig()`, `cv_reltimedwait()`, or `cv_reltimedwait_sig()` are used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the wakeup becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

**Examples** **EXAMPLE 1** Waiting for a Flag Value in a Driver's Unit

Here the condition being waited for is a flag value in a driver's unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
    cv_wait(&un->un_cv, &un->un_lock);
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

**EXAMPLE 2** Unblocking Threads Blocked by the Code in Example 1

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

**Notes** It is possible for `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()`, `cv_timedwait_sig()`, `cv_reltimedwait()`, and `cv_reltimedwait_sig()` to return prematurely, that is, not due to a call to `cv_signal()` or `cv_broadcast()`. This occurs most commonly in the case of `cv_wait_sig()` and `cv_timedwait_sig()` and `cv_reltimedwait_sig()` when the thread is stopped and restarted by job control signals or by a debugger, but can happen in other cases as well, even for `cv_wait()`. Code that calls these functions must always recheck the reason for blocking and call again if the reason for blocking is still true.

If your driver needs to wait on behalf of processes that have real-time constraints, use `cv_timedwait()` or `cv_reltimedwait()` rather than `delay(9F)`. The `delay()` function calls `timeout(9F)`, which can be subject to priority inversions.

Not all threads can receive signals from user level processes. In cases where such reception is impossible (such as during execution of `close(9E)` due to `exit(2)`), `cv_wait_sig()` behaves as `cv_wait()`, `cv_timedwait_sig()` behaves as `cv_timedwait()`, and `cv_reltimedwait_sig()` behaves as `cv_reltimedwait()`. To avoid unkillable processes, users of these functions may need to protect against waiting indefinitely for events that might not occur. The `ddi_can_receive_sig(9F)` function is provided to detect when signal reception is possible.

**See Also** `kill(2)`, `ddi_can_receive_sig(9F)`, `ddi_get_lbolt(9F)`, `mutex(9F)`, `mutex_init(9F)`

*Writing Device Drivers*

**Name** copyb – copy a message block

**Synopsis** #include <sys/stream.h>

```
mblk_t *copyb(mblk_t *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to the message block from which data is copied.

**Description** The copyb() function allocates a new message block, and copies into it the data from the block that *bp* denotes. The new block will be at least as large as the block being copied. copyb() uses the *b\_rptr* and *b\_wptr* members of *bp* to determine how many bytes to copy.

**Return Values** If successful, copyb() returns a pointer to the newly allocated message block containing the copied data. Otherwise, it returns a NULL pointer.

**Context** The copyb() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using copyb

For each message in the list, test to see if the downstream queue is full with the [canputnext\(9F\)](#) function (line 21). If it is not full, use copyb to copy a header message block, and [dupmsg\(9F\)](#) to duplicate the data to be retransmitted. If either operation fails, reschedule a timeout at the next valid interval.

Update the new header block with the correct destination address (line 34), link the message to it (line 35), and send it downstream (line 36). At the end of the list, reschedule this routine.

```

1 struct retrans {
2     mblk_t      *r_mp;
3     int         r_address;
4     queue_t     *r_outq;
5     struct retrans *r_next;
6 };
7
8 struct protoheader {
9     ...
10    int         h_address;
11    ...
12 };
13
14 void
15 retransmit(struct retrans *ret)
16 {
17     mblk_t *bp, *mp;
18     struct protoheader *php;
```

## EXAMPLE 1 Using copyb (Continued)

```

19
20     while (ret) {
21         if (!canputnext(ret->r_outq)) { /* no room */
22             ret = ret->r_next;
23             continue;
24         }
25         bp = copyb(header);           /* copy header msg. block */
26         if (bp == NULL)
27             break;
28         mp = dupmsg(ret->r_mp);        /* duplicate data */
29         if (mp == NULL) {            /* if unsuccessful */
30             freeb(bp);               /* free the block */
31             break;
32         }
33         php = (struct protoheader *)bp->b_rptr;
34         php->h_address = ret->r_address; /* new header */
35         bp->bp_cont = mp;            /* link the message */
36         putnext(ret->r_outq, bp);    /* send downstream */
37         ret = ret->r_next;
38     }
39     /* reschedule */
40     (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
41 }

```

**See Also** [allocb\(9F\)](#), [canputnext\(9F\)](#), [dupmsg\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** copyin – copy data from a user program to a driver buffer

**Synopsis** `#include <sys/types.h>`  
`#include <sys/ddi.h>`

```
int copyin(const void *userbuf, void *driverbuf, size_t cn);
```

**Interface Level** This interface is obsolete. `ddi_copyin(9F)` should be used instead.

**Parameters** *userbuf* User program source address from which data is transferred.

*driverbuf* Driver destination address to which data is transferred.

*cn* Number of bytes transferred.

**Description** `copyin()` copies data from a user program source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move according to address alignment.

**Return Values** Under normal conditions, a 0 is returned indicating a successful copy. Otherwise, a -1 is returned if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.

If a -1 is returned to the caller, driver entry point routines should return EFAULT.

**Context** `copyin()` can be called from user context only.

**Examples** EXAMPLE 1 An `ioctl()` Routine

A driver `ioctl(9E)` routine (line 10) can be used to get or set device attributes or registers. In the `XX_GETREGS` condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```
1 struct device {          /* layout of physical device registers */
2     int     control;    /* physical device control word */
3     int     status;    /* physical device status word */
4     short   recv_char; /* receive character from device */
```



**EXAMPLE 1** An ioctl() Routine (Continued)

```

5     short    xmit_char; /* transmit character to device */
6 };
7
8 extern struct device xx_addr[]; /* phys. device regs. location */
9     . . .
10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,
11         cred_t *cred_p, int *rval_p)
12     . . .
13 {
14     register struct device *rp = &xx_addr[getminor(dev) >> 4];
15     switch (cmd) {
16
17     case XX_GETREGS: /* copy device regs. to user program */
18         if (copyin(arg, rp, sizeof(struct device)))
19             return(EFAULT);
20         break;
21         . . .
22     }
23     . . .
24 }

```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ioctl\(9E\)](#), [bcopy\(9F\)](#), [copyout\(9F\)](#), [ddi\\_copyin\(9F\)](#), [ddi\\_copyout\(9F\)](#), [uiomove\(9F\)](#).

### *Writing Device Drivers*

**Notes** Driver writers who intend to support layered ioctls in their [ioctl\(9E\)](#) routines should use [ddi\\_copyin\(9F\)](#) instead.

Driver defined locks should not be held across calls to this function.

`copyin()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*.

**Name** copymsg – copy a message

**Synopsis** #include <sys/stream.h>

```
mblk_t *copymsg(mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the message to be copied.

**Description** The copymsg() function forms a new message by allocating new message blocks, and copying the contents of the message referred to by *mp* (using the copyb(9F) function). It returns a pointer to the new message.

**Return Values** If the copy is successful, copymsg() returns a pointer to the new message. Otherwise, it returns a NULL pointer.

**Context** The copymsg() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 : Using copymsg

The routine lctouc() converts all the lowercase ASCII characters in the message to uppercase. If the reference count is greater than one (line 8), then the message is shared, and must be copied before changing the contents of the data buffer. If the call to the copymsg() function fails (line 9), return NULL (line 10), otherwise, free the original message (line 11). If the reference count was equal to 1, the message can be modified. For each character (line 16) in each message block (line 15), if it is a lowercase letter, convert it to an uppercase letter (line 18). A pointer to the converted message is returned (line 21).

```

1 mblk_t *lctouc(mp)
2   mblk_t *mp;
3 {
4   mblk_t *cmp;
5   mblk_t *tmp;
6   unsigned char *cp;
7
8   if (mp->b_datap->db_ref > 1) {
9       if ((cmp = copymsg(mp)) == NULL)
10          return (NULL);
11       freemsg(mp);
12   } else {
13       cmp = mp;
14   }
15   for (tmp = cmp; tmp; tmp = tmp->b_cont) {
16       for (cp = tmp->b_rptr; cp < tmp->b_wptr; cp++) {
17           if ((*cp <= 'z') && (*cp >= 'a'))
18               *cp -= 0x20;
19       }

```

**EXAMPLE 1** : Using copymsg (Continued)

```
20     }  
21     return(cmp);  
22 }
```

**See Also** [allocb\(9F\)](#), [copyb\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** copyout – copy data from a driver to a user program

**Synopsis** `#include <sys/types.h>`  
`#include <sys/ddi.h>`

```
int copyout(const void *driverbuf, void *userbuf, size_t cn);
```

**Interface Level** This interface is obsolete. [ddi\\_copyout\(9F\)](#) should be used instead.

**Parameters** *driverbuf* Source address in the driver from which the data is transferred.  
*userbuf* Destination address in the user program to which the data is transferred.  
*cn* Number of bytes moved.

**Description** `copyout()` copies data from driver buffers to user data space.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

**Return Values** Under normal conditions, a 0 is returned to indicate a successful copy. Otherwise, a -1 is returned if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.

If a -1 is returned to the caller, driver entry point routines should return EFAULT.

**Context** `copyout()` can be called from user context only.

**Examples** EXAMPLE 1 An `ioctl()` Routine

A driver [ioctl\(9E\)](#) routine (line 10) can be used to get or set device attributes or registers. In the `XX_GETREGS` condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```
1 struct device {          /* layout of physical device registers */
2     int     control;     /* physical device control word */
3     int     status;     /* physical device status word */
4     short  recv_char;   /* receive character from device */
5     short  xmit_char;   /* transmit character to device */
6 };
7
```

**EXAMPLE 1** An ioctl() Routine (Continued)

```

 8 extern struct device xx_addr[]; /* phys. device regs. location */
 9   . . .
10 xx_ioctl(dev_t dev, int cmd, int arg, int mode,
11         cred_t *cred_p, int *rval_p)
12         ...
13 {
14     register struct device *rp = &xx_addr[getminor(dev) >> 4];
15     switch (cmd) {
16
17     case XX_GETREGS: /* copy device regs. to user program */
18         if (copyout(rp, arg, sizeof(struct device)))
19             return(EFAULT);
20         break;
21         ...
22     }
23         ...
24 }

```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ioctl\(9E\)](#), [bcopy\(9F\)](#), [copyin\(9F\)](#), [ddi\\_copyin\(9F\)](#), [ddi\\_copyout\(9F\)](#), [uiomove\(9F\)](#)

### *Writing Device Drivers*

**Notes** Driver writers who intend to support layered ioctls in their [ioctl\(9E\)](#) routines should use [ddi\\_copyout\(9F\)](#) instead.

Driver defined locks should not be held across calls to this function.

`copyout()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*.

**Name** csx\_AccessConfigurationRegister – read or write a PC Card Configuration Register

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_AccessConfigurationRegister(client_handle_t ch,
    access_config_reg_t *acr);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*acr* Pointer to an `access_config_reg_t` structure.

**Description** This function allows a client to read or write a PCCard Configuration Register.

**Structure Members** The structure members of `access_config_reg_t` are:

```
uint32_t Socket; /* socket number*/
uint32_t Action; /* register access operation*/
uint32_t Offset; /* config register offset*/
uint32_t Value; /* value read or written*/
```

The fields are defined as follows:

- Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
- Action** May be set to `CONFIG_REG_READ` or `CONFIG_REG_WRITE`. All other values in the `Action` field are reserved for future use. If the `Action` field is set to `CONFIG_REG_WRITE`, the `Value` field is written to the specified configuration register. Card Services does not read the configuration register after a write operation. For that reason, the `Value` field is only updated by a `CONFIG_REG_READ` request.
- Offset** Specifies the byte offset for the desired configuration register from the PC Card configuration register base specified in [csx\\_RequestConfiguration\(9F\)](#).
- Value** Contains the value read from the PC Card Configuration Register for a read operation. For a write operation, the `Value` field contains the value to write to the configuration register. As noted above, on return from a write request, the `Value` field is the value written to the PC Card and not any changed value that may have resulted from the write request (that is, no read after write is performed).

A client must be very careful when writing to the COR (Configuration Option Register) at offset 0. This has the potential to change the type of interrupt request generated by the PC Card or place the card in the reset state. Either request may have undefined results. The client should read the register to determine the appropriate setting for the interrupt mode (Bit 6) before writing to the register.

If a client wants to reset a PCCard, the `csx_ResetFunction(9F)` function should be used. Unlike `csx_AccessConfigurationRegister()`, the `csx_ResetFunction(9F)` function generates a series of event notifications to all clients using the PC Card, so they can re-establish the appropriate card state after the reset operation is complete.

|                      |                                      |                                                                                                                                                                                |
|----------------------|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Return Values</b> | <code>CS_SUCCESS</code>              | Successful operation.                                                                                                                                                          |
|                      | <code>CS_BAD_ARGS</code>             | Specified arguments are invalid. Client specifies an <code>Offset</code> that is out of range or neither <code>CONFIG_REG_READ</code> or <code>CONFIG_REG_WRITE</code> is set. |
|                      | <code>CS_UNSUPPORTED_MODE</code>     | Client has not called <code>csx_RequestConfiguration(9F)</code> before calling this function.                                                                                  |
|                      | <code>CS_BAD_HANDLE</code>           | Client handle is invalid.                                                                                                                                                      |
|                      | <code>CS_NO_CARD</code>              | No PC card in socket.                                                                                                                                                          |
|                      | <code>CS_UNSUPPORTED_FUNCTION</code> | No PCMCIA hardware installed.                                                                                                                                                  |

**Context** This function may be called from user or kernel context.

**See Also** `csx_ParseTuple(9F)`, `csx_RegisterClient(9F)`, `csx_RequestConfiguration(9F)`, `csx_ResetFunction(9F)`

*PCCard 95 Standard, PCMCIA/JEIDA*

**Name** csx\_ConvertSize – convert device sizes

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ConvertSize(convert_size_t *cs);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *cs* Pointer to a `convert_size_t` structure.

**Description** `csx_ConvertSize()` is a Solaris-specific extension that provides a method for clients to convert from one type of device size representation to another, that is, from *devsize* format to *bytes* and vice versa.

**Structure Members** The structure members of `convert_size_t` are:

```
uint32_t  Attributes;  
uint32_t  bytes;  
uint32_t  devsize;
```

The fields are defined as follows:

**Attributes** This is a bit-mapped field that identifies the type of size conversion to be performed. The field is defined as follows:

`CONVERT_BYTES_TO_DEVSIZE` Converts *bytes* to *devsize* format.

`CONVERT_DEVSIZE_TO_BYTES` Converts *devsize* format to *bytes*.

**bytes** If `CONVERT_BYTES_TO_DEVSIZE` is set, the value in the `bytes` field is converted to a *devsize* format and returned in the `devsize` field.

**devsize** If `CONVERT_DEVSIZE_TO_BYTES` is set, the value in the `devsize` field is converted to a *bytes* value and returned in the `bytes` field.

**Return Values** `CS_SUCCESS` Successful operation.

`CS_BAD_SIZE` Invalid *bytes* or *devsize*.

`CS_UNSUPPORTED_FUNCTION` No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_ModifyWindow\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PCCard 95 Standard, PCMCIA/JEIDA*



- 
- Name** csx\_ConvertSpeed – convert device speeds
- Synopsis** #include <sys/pccard.h>
- ```
int32_t csx_ConvertSpeed(convert_speed_t *cs);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters** *cs* Pointer to a `convert_speed_t` structure.
- Description** This function is a Solaris-specific extension that provides a method for clients to convert from one type of device speed representation to another, that is, from *devspeed* format to *nS* and vice versa.
- Structure Members** The structure members of `convert_speed_t` are:
- ```
uint32_t    Attributes;
uint32_t    nS;
uint32_t    devspeed;
```
- The fields are defined as follows:
- Attributes** This is a bit-mapped field that identifies the type of speed conversion to be performed. The field is defined as follows:
- ```
CONVERT_NS_TO_DEVSPEED    Converts nS to devspeed format
CONVERT_DEVSPEED_TO_NS    Converts devspeed format to nS
```
- nS** If `CONVERT_NS_TO_DEVSPEED` is set, the value in the `nS` field is converted to a *devspeed* format and returned in the `devspeed` field.
- devspeed** If `CONVERT_DEVSPEED_TO_NS` is set, the value in the `devspeed` field is converted to an *nS* value and returned in the `nS` field.
- Return Values**
- ```
CS_SUCCESS                Successful operation.
CS_BAD_SPEED              Invalid nS or devspeed.
CS_BAD_ATTRIBUTE          Bad Attributes value.
CS_UNSUPPORTED_FUNCTION   No PCMCIA hardware installed.
```
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_ModifyWindow\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)  
*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_CS\_DDI\_Info – obtain DDI information

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_CS_DDI_Info(cs_ddi_info_t *cdi);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *cdi* Pointer to a `cs_ddi_info_t` structure.

**Description** This function is a Solaris-specific extension that is used by clients that need to provide the `xx_getinfo` driver entry point (see [getinfo\(9E\)](#)). It provides a method for clients to obtain DDI information based on their socket number and client driver name.

**Structure Members** The structure members of `cs_ddi_info_t` are:

```
uint32_t      Socket;          /* socket number */
char*         driver_name;     /* unique driver name */
dev_info_t   *dip;            /* dip */
int32_t      instance;        /* instance */
```

The fields are defined as follows:

**Socket** This field must be set to the physical socket number that the client is interested in getting information about.

**driver\_name** This field must be set to a string containing the name of the client driver to get information about.

If `csx_CS_DDI_Info()` is used in a client's `xx_getinfo` function, then the client will typically extract the `Socket` value from the `*arg` argument and it *must* set the `driver_name` field to the same string used with [csx\\_RegisterClient\(9F\)](#).

If the `driver_name` is found on the `Socket`, the `csx_CS_DDI_Info()` function returns both the `dev_info` pointer and the `instance` fields for the requested driver instance.

**Return Values**

|                                      |                                           |
|--------------------------------------|-------------------------------------------|
| <code>CS_SUCCESS</code>              | Successful operation.                     |
| <code>CS_BAD_SOCKET</code>           | Client not found on <code>Socket</code> . |
| <code>CS_UNSUPPORTED_FUNCTION</code> | No PCMCIA hardware installed.             |

**Context** This function may be called from user or kernel context.

**Examples** **EXAMPLE 1** : Using `csx_CS_DDI_Info`

The following example shows how a client might call the `csx_CS_DDI_Info()` in the client's `xx_getinfo` function to return the `dip` or the instance number:

```
static int
pcepp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
```

EXAMPLE 1 : Using csx\_CS\_DDI\_Info (Continued)

```

  void **result)
{
    int                error = DDI_SUCCESS;
    pcepp_state_t     *pps;
    cs_ddi_info_t     cs_ddi_info;

    switch (cmd) {

        case DDI_INFO_DEVT2DEVINFO:
            cs_ddi_info.Socket = getminor((dev_t)arg) & 0x3f;
            cs_ddi_info.driver_name = pcepp_name;
            if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS)
                return (DDI_FAILURE);
            if (!(pps = ddi_get_soft_state(pcepp_soft_state_p,
                cs_ddi_info.instance))) {
                *result = NULL;
            } else {
                *result = pps->dip;
            }
            break;

        case DDI_INFO_DEVT2INSTANCE:
            cs_ddi_info.Socket = getminor((dev_t)arg) & 0x3f;
            cs_ddi_info.driver_name = pcepp_name;
            if (csx_CS_DDI_Info(&cs_ddi_info) != CS_SUCCESS)
                return (DDI_FAILURE);
            *result = (void *)cs_ddi_info.instance;
            break;

        default:
            error = DDI_FAILURE;
            break;

    }

    return (error);
}

```

**See Also** [getinfo\(9E\)](#), [csx\\_RegisterClient\(9F\)](#), [ddi\\_get\\_instance\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_DeregisterClient – remove client from Card Services list

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_DeregisterClient(client_handle_t ch);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

**Description** This function removes a client from the list of registered clients maintained by Card Services. The Client Handle returned by [csx\\_RegisterClient\(9F\)](#) is passed in the *client\_handle\_t* argument.

The client must have returned all requested resources before this function is called. If any resources have not been released, CS\_IN\_USE is returned.

**Return Values**

|                         |                                        |
|-------------------------|----------------------------------------|
| CS_SUCCESS              | Successful operation.                  |
| CS_BAD_HANDLE           | Client handle is invalid.              |
| CS_IN_USE               | Resources not released by this client. |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed.          |

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Warnings** Clients should be prepared to receive callbacks until Card Services returns from this request successfully.

**Name** csx\_DupHandle – duplicate access handle

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_DupHandle(acc_handle_t handle1, acc_handle_t *handle2,
                    uint32_t flags);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle1* The access handle returned from `csx_RequestIO(9F)` or `csx_RequestWindow(9F)` that is to be duplicated.

*handle2* A pointer to the newly-created duplicated data access handle.

*flags* The access attributes that will be applied to the new handle.

**Description** This function duplicates the handle, *handle1*, into a new handle, *handle2*, that has the access attributes specified in the *flags* argument. Both the original handle and the new handle are active and can be used with the common access functions.

Both handles must be explicitly freed when they are no longer necessary.

The *flags* argument is bit-mapped. The following bits are defined:

|                         |                                       |
|-------------------------|---------------------------------------|
| WIN_ACC_NEVER_SWAP      | Host endian byte ordering             |
| WIN_ACC_BIG_ENDIAN      | Big endian byte ordering              |
| WIN_ACC_LITTLE_ENDIAN   | Little endian byte ordering           |
| WIN_ACC_STRICT_ORDER    | Program ordering references           |
| WIN_ACC_UNORDERED_OK    | May re-order references               |
| WIN_ACC_MERGING_OK      | Merge stores to consecutive locations |
| WIN_ACC_LOADCACHING_OK  | May cache load operations             |
| WIN_ACC_STORECACHING_OK | May cache store operations            |

WIN\_ACC\_BIG\_ENDIAN and WIN\_ACC\_LITTLE\_ENDIAN describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When WIN\_ACC\_BIG\_ENDIAN or WIN\_ACC\_LITTLE\_ENDIAN is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities. When WIN\_ACC\_NEVER\_SWAP is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following *flags* bits. Only one of the following bits may be specified:

|                      |                                                                                                        |
|----------------------|--------------------------------------------------------------------------------------------------------|
| WIN_ACC_STRICT_ORDER | The data references must be issued by a CPU in program order. Strict ordering is the default behavior. |
|----------------------|--------------------------------------------------------------------------------------------------------|

|                         |                                                                                                                                                                                                                                                                                                                  |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| WIN_ACC_UNORDERED_OK    | The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).                                                                                                                                            |
| WIN_ACC_MERGING_OK      | The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. Setting this bit also implies re-ordering. |
| WIN_ACC_LOADCACHING_OK  | The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. Setting this bit also implies merging and re-ordering.                                                                                                                   |
| WIN_ACC_STORECACHING_OK | The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. Setting this bit also implies load caching, merging, and re-ordering.                                                                           |

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together.

|                      |                         |                                                                                   |
|----------------------|-------------------------|-----------------------------------------------------------------------------------|
| <b>Return Values</b> | CS_SUCCESS              | Successful operation.                                                             |
|                      | CS_FAILURE              | Error in <i>flags</i> argument or handle could not be duplicated for some reason. |
|                      | CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed.                                                     |

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_Get8\(9F\)](#), [csx\\_GetMappedAddr\(9F\)](#), [csx\\_Put8\(9F\)](#), [csx\\_RepGet8\(9F\)](#), [csx\\_RepPut8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Error2Text – convert error return codes to text strings

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Error2Text(error2text_t *er);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *er* Pointer to an error2text\_t structure.

**Description** This function is a Solaris-specific extension that provides a method for clients to convert Card Services error return codes to text strings.

**Structure Members** The structure members of error2text\_t are:

```
uint32_t    item;                /*the error code*/
char        test[CS_ERROR_MAX_BUFSIZE]; /*the error code*/
```

A pointer to the text for the Card Services error return code in the *item* field is returned in the *text* field if the error return code is found. The client is not responsible for allocating a buffer to hold the text. If the Card Services error return code specified in the *item* field is not found, the *text* field will be set to a string of the form:

```
"{unknown Card Services return code}"
```

**Return Values**

|                         |                               |
|-------------------------|-------------------------------|
| CS_SUCCESS              | Successful operation.         |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed. |

**Context** This function may be called from user or kernel context.

**Examples** EXAMPLE 1 : Using the csxError2Text function

```
if ((ret = csx_RegisterClient(&client_handle, &
    client_reg)) != CS_SUCCESS)
{
    error2text_t error2text;
    error2text.item = ret;
    csx_Error2Text(&error2text);
    cmn_err(CE_CONT, "RegisterClient failed %s (0x%x)",
        error2text.text, ret);
}
```

**See Also** [csx\\_Event2Text\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Event2Text – convert events to text strings

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Event2Text(event2text_t *ev);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ev* Pointer to an event2text\_t structure.

**Description** This function is a Solaris-specific extension that provides a method for clients to convert Card Services events to text strings.

**Structure Members** The structure members of event2text\_t are:

```
event_t    event;           /*the event code*/
char       text[CS_EVENT_MAX_BUFSIZE] /*the event code*/
```

The fields are defined as follows:

*event* The text for the event code in the event field is returned in the *text* field.

*text* The text string describing the name of the event.

**Return Values** CS\_SUCCESS Successful operation.

CS\_UNSUPPORTED\_FUNCTION No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**Examples** EXAMPLE 1 :Using csx\_Event2Text()

```
xx_event(event_t event, int priority, event_callback_args_t *eca)
{
    event2text_t    event2text;

    event2text.event = event;
    csx_Event2Text(&event2text);
    cmn_err(CE_CONT, "event %s (0x%x)", event2text.text, (int)event);
}
```

**See Also** [csx\\_event\\_handler\(9E\)](#), [csx\\_Error2Text\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*



- 
- Name** csx\_FreeHandle – free access handle
- Synopsis** #include <sys/pccard.h>
- ```
int32_t csx_FreeHandle(acc_handle_t *handle);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters** *handle* The access handle returned from [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#), or [csx\\_DupHandle\(9F\)](#).
- Description** This function frees the handle, *handle*. If the handle was created by the [csx\\_DupHandle\(9F\)](#) function, this function will free the storage associated with this handle, but will not modify any resources that the original handle refers to. If the handle was created by a common access setup function, this function will release the resources associated with this handle.
- Return Values**
- |                         |                               |
|-------------------------|-------------------------------|
| CS_SUCCESS              | Successful operation.         |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed. |
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)
- PC Card95 Standard, PCMCIA/JEIDA*

**Name** csx\_Get8, csx\_Get16, csx\_Get32, csx\_Get64 – read data from device address

**Synopsis** #include <sys/pccard.h>

```
uint8_t csx_Get8(acc_handle_t handle, uint32_t offset);
uint16_t csx_Get16(acc_handle_t handle, uint32_t offset);
uint32_t csx_Get32(acc_handle_t handle, uint32_t offset);
uint64_t csx_Get64(acc_handle_t handle, uint64_t offset);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* The access handle returned from [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#), or [csx\\_DupHandle\(9F\)](#).

*offset* The offset in bytes from the base of the mapped resource.

**Description** These functions generate a read of various sizes from the mapped memory or device register.

The `csx_Get8()`, `csx_Get16()`, `csx_Get32()`, and `csx_Get64()` functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the *handle*, at an offset in bytes represented by the *offset*, *offset*.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

**Return Values** These functions return the value read from the mapped address.

**Context** These functions may be called from user, kernel, or interrupt context.

**See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_GetMappedAddr\(9F\)](#), [csx\\_Put8\(9F\)](#), [csx\\_RepGet8\(9F\)](#), [csx\\_RepPut8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_GetFirstClient, csx\_GetNextClient – return first or next client

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetFirstClient(get_firstnext_client_t *fnc);
int32_t csx_GetNextClient(get_firstnext_client_t *fnc);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *fnc* Pointer to a get\_firstnext\_client\_t structure.

**Description** The functions csx\_GetFirstClient() and csx\_GetNextClient() return information about the first or subsequent PC cards, respectively, that are installed in the system.

**Structure Members** The structure members of get\_firstnext\_client\_t are:

```
uint32_t      Socket;          /* socket number */
uint32_t      Attributes;     /* attributes */
client_handle_t client_handle; /* client handle */
uint32_t      num_clients;    /* number of clients */
```

The fields are defined as follows:

Socket	If the CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY attribute is set, return information only on the PC card installed in this socket.				
Attributes	This field indicates the type of client. The field is bit-mapped; the following bits are defined: <table> <tbody> <tr> <td>CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS</td> <td>Return information on all clients.</td> </tr> <tr> <td>CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY</td> <td>Return client information for the specified socket only.</td> </tr> </tbody> </table>	CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS	Return information on all clients.	CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY	Return client information for the specified socket only.
CS_GET_FIRSTNEXT_CLIENT_ALL_CLIENTS	Return information on all clients.				
CS_GET_FIRSTNEXT_CLIENT_SOCKET_ONLY	Return client information for the specified socket only.				
client_handle	The client handle of the PC card driver is returned in this field.				
num_clients	The number of clients is returned in this field.				

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_BAD_SOCKET	Socket number is invalid.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_MORE_ITEMS	PC Card driver does not handle the CS_EVENT_CLIENT_INFO event.

CS\_UNSUPPORTED\_FUNCTION      No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_event\\_handler\(9E\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_GetFirstTuple, csx\_GetNextTuple – return Card Information Structure tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetFirstTuple(client_handle_t ch, tuple_t *tu);
int32_t csx_GetNextTuple(client_handle_t ch, tuple_t *tu);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure.

**Description** The functions `csx_GetFirstTuple()` and `csx_GetNextTuple()` return the first and next tuple, respectively, of the specified type in the Card Information Structure (CIS) for the specified socket.

**Structure Members** The structure members of `tuple_t` are:

```
uint32_t   Socket;           /* socket number */
uint32_t   Attributes;      /* Attributes */
cisdata_t  DesiredTuple;    /* tuple to search for or flags */
cisdata_t  TupleCode;       /* tuple type code */
cisdata_t  TupleLink;       /* tuple data body size */
```

The fields are defined as follows:

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Attributes** This field is bit-mapped. The following bits are defined:

**TUPLE\_RETURN\_LINK**

Return link tuples if set. The following are link tuples and will only be returned by this function if the `TUPLE_RETURN_LINK` bit in the `Attributes` field is set:

```
CISTPL_NULL      CISTPL_LONGLINK_MFC
CISTPL_LONGLINK_A  CISTPL_LINKTARGET
CISTPL_LONGLINK_C  CISTPL_NO_LINK
CISTPL_LONGLINK_CB CISTPL_END
```

**TUPLE\_RETURN\_IGNORED\_TUPLES**

Return ignored tuples if set. Ignored tuples will be returned by this function if the `TUPLE_RETURN_IGNORED_TUPLES` bit in the `Attributes` field is set, see [tuple\(9S\)](#) for more information. The CIS is parsed from the location setup by the previous `csx_GetFirstTuple()` or `csx_GetNextTuple()` request.

DesiredTuple	This field is the tuple value desired. If it is RETURN_FIRST_TUPLE, the very first tuple of the CIS is returned (if it exists). If this field is set to RETURN_NEXT_TUPLE, the very next tuple of the CIS is returned (if it exists). If the DesiredTuple field is any other value on entry, the CIS is searched in an attempt to locate a tuple which matches.
TupleCode, TupleLink	These fields are the values returned from the tuple found. If there are no tuples on the card, CS_NO_MORE_ITEMS is returned.

Since the `csx_GetFirstTuple()`, `csx_GetNextTuple()`, and `csx_GetTupleData(9F)` functions all share the same `tuple_t` structure, some fields in the `tuple_t` structure are unused or reserved when calling this function and these fields must not be initialized by the client.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC card.
	CS_NO_MORE_ITEMS	Desired tuple not found.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_GetTupleData\(9F\)](#), [csx\\_ParseTuple\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95Standard, PCMCIA/JEIDA*

**Name** csx\_GetHandleOffset – return current access handle offset

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetHandleOffset(acc_handle_t handle, uint32_t *offset);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* Access handle returned by [csx\\_RequestIRQ\(9F\)](#) or [csx\\_RequestIO\(9F\)](#).

*offset* Pointer to a `uint32_t` in which the current access handle offset is returned.

**Description** This function returns the current offset for the access handle, *handle*, in *offset*.

**Return Values** `CS_SUCCESS` Successful operation.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_RequestIO\(9F\)](#), [csx\\_RequestIRQ\(9F\)](#), [csx\\_SetHandleOffset\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_GetMappedAddr – return mapped virtual address

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetMappedAddr(acc_handle_t handle, void **addr);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* The access handle returned from [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#), or [csx\\_DupHandle\(9F\)](#).

*addr* The virtual or I/O port number represented by the handle.

**Description** This function returns the mapped virtual address or the mapped I/O port number represented by the handle, *handle*.

**Return Values**

CS_SUCCESS	The resulting address or I/O port number can be directly accessed by the caller.
CS_FAILURE	The resulting address or I/O port number can not be directly accessed by the caller; the caller must make all accesses to the mapped area via the common access functions.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user, kernel, or interrupt context.

**See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_Get8\(9F\)](#), [csx\\_Put8\(9F\)](#), [csx\\_RepGet8\(9F\)](#), [csx\\_RepPut8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*



**Name** csx\_GetStatus – return the current status of a PC Card and its socket

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetStatus(client_handle_t ch, get_status_t *gs);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*gs* Pointer to a `get_status_t` structure.

**Description** This function returns the current status of a PC Card and its socket.

**Structure Members** The structure members of `get_status_t` are:

```
uint32_t Socket;          /* socket number*/
uint32_t CardState;      /* "live" card status for this client*/
uint32_t SocketState;    /* latched socket values */
uint32_t raw_CardState;  /* raw live card status */
```

The fields are defined as follows:

Socket	Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.																						
CardState	The <code>CardState</code> field is the bit-mapped output data returned from Card Services. The bits identify what Card Services thinks the current state of the installed PC Card is. The bits are:																						
	<table> <tbody> <tr> <td>CS_STATUS_WRITE_PROTECTED</td> <td>Card is write protected</td> </tr> <tr> <td>CS_STATUS_CARD_LOCKED</td> <td>Card is locked</td> </tr> <tr> <td>CS_STATUS_EJECTION_REQUEST</td> <td>Ejection request in progress</td> </tr> <tr> <td>CS_STATUS_INSERTION_REQUEST</td> <td>Insertion request in progress</td> </tr> <tr> <td>CS_STATUS_BATTERY_DEAD</td> <td>Card battery is dead</td> </tr> <tr> <td>CS_STATUS_BATTERY_DEAD</td> <td>Card battery is dead (BVD1)</td> </tr> <tr> <td>CS_STATUS_BATTERY_LOW</td> <td>Card battery is low (BVD2)</td> </tr> <tr> <td>CS_STATUS_CARD_READY</td> <td>Card is READY</td> </tr> <tr> <td>CS_STATUS_CARD_INSERTED</td> <td>Card is inserted</td> </tr> <tr> <td>CS_STATUS_REQ_ATTEN</td> <td>Extended status attention request</td> </tr> <tr> <td>CS_STATUS_RES_EVT1</td> <td>Extended status reserved event status</td> </tr> </tbody> </table>	CS_STATUS_WRITE_PROTECTED	Card is write protected	CS_STATUS_CARD_LOCKED	Card is locked	CS_STATUS_EJECTION_REQUEST	Ejection request in progress	CS_STATUS_INSERTION_REQUEST	Insertion request in progress	CS_STATUS_BATTERY_DEAD	Card battery is dead	CS_STATUS_BATTERY_DEAD	Card battery is dead (BVD1)	CS_STATUS_BATTERY_LOW	Card battery is low (BVD2)	CS_STATUS_CARD_READY	Card is READY	CS_STATUS_CARD_INSERTED	Card is inserted	CS_STATUS_REQ_ATTEN	Extended status attention request	CS_STATUS_RES_EVT1	Extended status reserved event status
CS_STATUS_WRITE_PROTECTED	Card is write protected																						
CS_STATUS_CARD_LOCKED	Card is locked																						
CS_STATUS_EJECTION_REQUEST	Ejection request in progress																						
CS_STATUS_INSERTION_REQUEST	Insertion request in progress																						
CS_STATUS_BATTERY_DEAD	Card battery is dead																						
CS_STATUS_BATTERY_DEAD	Card battery is dead (BVD1)																						
CS_STATUS_BATTERY_LOW	Card battery is low (BVD2)																						
CS_STATUS_CARD_READY	Card is READY																						
CS_STATUS_CARD_INSERTED	Card is inserted																						
CS_STATUS_REQ_ATTEN	Extended status attention request																						
CS_STATUS_RES_EVT1	Extended status reserved event status																						

CS_STATUS_RES_EVT2	Extended status reserved event status
CS_STATUS_RES_EVT3	Extended status reserved event status
CS_STATUS_VCC_50	5.0 Volts Vcc Indicated
CS_STATUS_VCC_33	3.3 Volts Vcc Indicated
CS_STATUS_VCC_XX	X.X Volts Vcc Indicated

The state of the CS\_STATUS\_CARD\_INSERTED bit indicates whether the PC Card associated with this driver instance, not just any card, is inserted in the socket. If an I/O card is installed in the specified socket, card state is returned from the PRR (Pin Replacement Register) and the ESR (Extended Status Register) (if present). If certain state bits are not present in the PRR or ESR, a simulated state bit value is returned as defined below:

CS_STATUS_WRITE_PROTECTED	Not write protected
CS_STATUS_BATTERY_DEAD	Power good
PCS_STATUS_BATTERY_LOW	Power good
CS_STATUS_CARD_READY	Ready
CS_STATUS_REQ_ATTEN	Not set
CS_STATUS_RES_EVT1	Not set
CS_STATUS_RES_EVT2	Not set
CS_STATUS_RES_EVT3	Not set

SocketState The SocketState field is a bit-map of the current card and socket state. The bits are:

CS SOCK STATUS WRITE PROTECT CHANGE	Write Protect
ECS SOCK STATUS CARD LOCK CHANGE	Card Lock Change
CS SOCK STATUS EJECTION PENDING	Ejection Request
CS SOCK STATUS INSERTION PENDING	Insertion Request
CS SOCK STATUS BATTERY DEAD CHANGE	Battery Dead

CS SOCK STATUS BATTERY LOW CHANGE

Battery Low

CS SOCK STATUS CARD READY CHANGE

Ready Change

CS SOCK STATUS CARD INSERTION CHANGE

Card is inserted

The state reported in the SocketState field may be different from the state reported in the CardState field. Clients should normally depend only on the state reported in the CardState field.

The state reported in the SocketState field may be different from the state reported in the CardState field. Clients should normally depend only on the state reported in the CardState field.

`raw_CardState` The `raw_CardState` field is a Solaris-specific extension that allows the client to determine if any card is inserted in the socket. The bit definitions in the `raw_CardState` field are identical to those in the `CardState` field with the exception that the `CS_STATUS_CARD_INSERTED` bit in the `raw_CardState` field is set whenever any card is inserted into the socket.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_BAD_SOCKET	Error getting socket state.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

CS\_NO\_CARD will not be returned if there is no PC Card present in the socket.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_GetTupleData – return the data portion of a tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_GetTupleData(client_handle_t ch, tuple_t *tu);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure.

**Description** This function returns the data portion of a tuple, as returned by the [csx\\_GetFirstTuple\(9F\)](#) and [csx\\_GetNextTuple\(9F\)](#) functions.

**Structure Members** The structure members of `tuple_t` are:

The fields are defined as follows:

```
uint32_t Socket; /* socket number */
uint32_t Attributes; /* tuple attributes*/
cisdata_t DesiredTuple; /* tuple to search for*/
cisdata_t TupleOffset; /* tuple data offset*/
cisdata_t TupleDataMax; /* max tuple data size*/
cisdata_t TupleDataLen; /* actual tuple data length*/
cisdata_t TupleData[CIS_MAX_TUPLE_DATA_LEN]; /* tuple body data buffer*/
cisdata_t TupleCode; /* tuple type code*/
cisdata_t TupleLink; /* tuple link */
```

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Attributes** Initialized by [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#); the client must not modify the value in this field.

**DesiredTuple** Initialized by [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#); the client must not modify the value in this field.

**TupleOffset** This field allows partial tuple information to be retrieved, starting anywhere within the tuple.

**TupleDataMax** This field is the size of the tuple data buffer that Card Services uses to return raw tuple data from [csx\\_GetTupleData](#). It can be larger than the number of bytes in the tuple data body. Card Services ignores any value placed here by the client.

**TupleDataLen** This field is the actual size of the tuple data body. It represents the number of tuple data body bytes returned.

**TupleData** This field is an array of bytes containing the raw tuple data body contents.

**tupleCode** Initialized by `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`; the client must not modify the value in this field.

**tupleLink** Initialized by `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`; the client must not modify the value in this field.

**Return Values**

<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_HANDLE</code>	Client handle is invalid.
<code>CS_BAD_ARGS</code>	Data from prior <code>csx_GetFirstTuple(9F)</code> or <code>csx_GetNextTuple(9F)</code> is corrupt.
<code>CS_NO_CARD</code>	No PC Card in socket.
<code>CS_NO_CIS</code>	No Card Information Structure (CIS) on PC Card.
<code>CS_NO_MORE_ITEMS</code>	Card Services was not able to read the tuple from the PC Card.
<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** `csx_GetFirstTuple(9F)`, `csx_ParseTuple(9F)`, `csx_RegisterClient(9F)`, `csx_ValidateCIS(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_MakeDeviceNode, csx\_RemoveDeviceNode – create and remove minor nodes on behalf of the client

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_MakeDeviceNode(client_handle_t ch, make_device_node_t *dn);
int32_t csx_RemoveDeviceNode(client_handle_t ch, remove_device_node_t *dn);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*dn* Pointer to a `make_device_node_t` or `remove_device_node_t` structure.

**Description** `csx_MakeDeviceNode()` and `csx_RemoveDeviceNode()` are Solaris-specific extensions to allow the client to request that device nodes in the filesystem are created or removed, respectively, on its behalf.

**Structure Members** The structure members of `make_device_node_t` are:

```
uint32_t      Action;          /* device operation */
uint32_t      NumDevNodes;     /* number of nodes to create */
devnode_desc_t *devnode_desc; /* description of device nodes */
```

The structure members of `remove_device_node_t` are:

```
uint32_t      Action;          /* device operation */
uint32_t      NumDevNodes;     /* number of nodes to remove */
devnode_desc_t *devnode_desc; /* description of device nodes */
```

The structure members of `devnode_desc_t` are:

```
char          *name;          /* device node path and name */
int32_t       spec_type;     /* device special type (block or char) */
int32_t       minor_num;     /* device node minor number */
char          *node_type;    /* device node type */
```

The Action field is used to specify the operation that `csx_MakeDeviceNode()` and `csx_RemoveDeviceNode()` should perform.

The following Action values are defined for `csx_MakeDeviceNode()`:

CREATE\_DEVICE\_NODE     Create NumDevNodes minor nodes

The following Action values are defined for `csx_RemoveDeviceNode()`:

REMOVE\_DEVICE\_NODE     Remove NumDevNodes minor nodes

REMOVE\_ALL\_DEVICE\_NODES     Remove all minor nodes for this client

For `csx_MakeDeviceNode()`, if the Action field is:

**CREATE\_DEVICE\_NODE** The NumDevNodes field must be set to the number of minor devices to create, and the client must allocate the quantity of devnode\_desc\_t structures specified by NumDevNodes and fill out the fields in the devnode\_desc\_t structure with the appropriate minor node information. The meanings of the fields in the devnode\_desc\_t structure are identical to the parameters of the same name to the [ddi\\_create\\_minor\\_node\(9F\)](#) DDI function.

For `csx_RemoveDeviceNode()`, if the Action field is:

**REMOVE\_DEVICE\_NODE** The NumDevNodes field must be set to the number of minor devices to remove, and the client must allocate the quantity of devnode\_desc\_t structures specified by NumDevNodes and fill out the fields in the devnode\_desc\_t structure with the appropriate minor node information. The meanings of the fields in the devnode\_desc\_t structure are identical to the parameters of the same name to the [ddi\\_remove\\_minor\\_node\(9F\)](#) DDI function.

**REMOVE\_ALL\_DEVICE\_NODES** The NumDevNodes field must be set to 0 and the devnode\_desc\_t structure pointer must be set to NULL. All device nodes for this client will be removed from the filesystem.

<b>Return Values</b>	<b>CS_SUCCESS</b>	Successful operation.
	<b>CS_BAD_HANDLE</b>	Client handle is invalid.
	<b>CS_BAD_ATTRIBUTE</b>	The value of one or more arguments is invalid.
	<b>CS_BAD_ARGS</b>	Action is invalid.
	<b>CS_OUT_OF_RESOURCE</b>	Unable to create or remove device node.
	<b>CS_UNSUPPORTED_FUNCTION</b>	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#), [ddi\\_create\\_minor\\_node\(9F\)](#), [ddi\\_remove\\_minor\\_node\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_MapLogSocket – return the physical socket number associated with the client handle

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_MapLogSocket(client_handle_t ch, map_log_socket_t *ls);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*ls* Pointer to a map\_log\_socket\_t structure.

**Description** This function returns the physical socket number associated with the client handle.

**Structure Members** The structure members of map\_log\_socket\_t are:

```
uint32_t    LogSocket;        /* logical socket number */
uint32_t    PhyAdapter;      /* physical adapter number */
uint32_t    PhySocket;       /* physical socket number */
```

The fields are defined as follows:

**LogSocket** Not used by this implementation of Card Services and can be set to any arbitrary value.

**PhyAdapter** Returns the physical adapter number, which is always 0 in the Solaris implementation of Card Services.

**PhySocket** Returns the physical socket number associated with the client handle. The physical socket number is typically used as part of an error or message string or if the client creates minor nodes based on the physical socket number.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*



- 
- Name** csx\_MapMemPage – map the memory area on a PC Card
- Synopsis** `#include <sys/pccard.h>`
- ```
int32_t csx_MapMemPage(window_handle_t wh, map_mem_page_t *mp);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters** *wh* Window handle returned from [csx\\_RequestWindow\(9F\)](#).  
*mp* Pointer to a `map_mem_page_t` structure.
- Description** This function maps the memory area on a PC Card into a page of a window allocated with the [csx\\_RequestWindow\(9F\)](#) function.
- Structure Members** The structure members of `map_mem_page_t` are:
- ```
uint32_t    CardOffset;    /* card offset */
uint32_t    Page;         /* page number */
```
- The fields are defined as follows:
- CardOffset* The absolute offset in bytes from the beginning of the PC Card to map into system memory.
- Page* Used internally by Card Services; clients must set this field to 0 before calling this function.
- Return Values**
- |                                      |                               |
|--------------------------------------|-------------------------------|
| <code>CS_SUCCESS</code>              | Successful operation.         |
| <code>CS_BAD_HANDLE</code>           | Client handle is invalid.     |
| <code>CS_BAD_OFFSET</code>           | Offset is invalid.            |
| <code>CS_BAD_PAGE</code>             | Page is not zero.             |
| <code>CS_NO_CARD</code>              | No PC Card in socket.         |
| <code>CS_UNSUPPORTED_FUNCTION</code> | No PCMCIA hardware installed. |
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_ModifyWindow\(9F\)](#), [csx\\_ReleaseWindow\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)  
*PC Card 95 Standard*, *PCMCIA/JEIDA*

**Name** csx\_ModifyConfiguration – modify socket and PC Card Configuration Register

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ModifyConfiguration(client_handle_t ch, modify_config_t *mc);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*mc* Pointer to a `modify_config_t` structure.

**Description** This function allows a socket and PC Card configuration to be modified. This function can only modify a configuration requested via [csx\\_RequestConfiguration\(9F\)](#).

**Structure Members** The structure members of `modify_config_t` are:

```
uint32_t Socket;          /* socket number */
uint32_t Attributes;     /* attributes to modify */
uint32_t Vpp1;           /* Vpp1 value */
uint32_t Vpp2;           /* Vpp2 value */
```

The fields are defined as follows:

Socket	Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
Attributes	This field is bit-mapped. The following bits are defined:
CONF_ENABLE_IRQ_STEERING	Enable IRQ steering. Set to connect the PC Card IREQ line to a previously selected system interrupt.
CONF_IRQ_CHANGE_VALID	IRQ change valid. Set to request the IRQ steering enable to be changed.
CONF_VPP1_CHANGE_VALID	Vpp1 change valid. These bits are set to request a change to the corresponding voltage level for the PC Card.
CONF_VPP2_CHANGE_VALID	Vpp2 change valid. These bits are set to request a change to the corresponding voltage level for the PC Card.
CONF_VSOVERRIDE	Override VS pins. For Low Voltage keyed cards, must be set if a client desires to apply a voltage inappropriate for this card to any

pin. After card insertion and prior to the first [csx\\_RequestConfiguration\(9F\)](#) call for this client, the voltage levels applied to the card will be those specified by the Card Interface Specification. (See WARNINGS.)

Vpp1, Vpp2 Represent voltages expressed in tenths of a volt. Values from 0 to 25.5 volts may be set. To be valid, the exact voltage must be available from the system. To be compliant with the *PC Card 95 Standard*, *PCMCIA/JEIDA*, systems must always support 5.0 volts for both Vcc and Vpp. (See WARNINGS.)

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid or <a href="#">csx_RequestConfiguration(9F)</a> not done.
	CS_BAD_SOCKET	Error getting/setting socket hardware parameters.
	CS_BAD_VPP	Requested Vpp is not available on socket.
	CS_NO_CARD	No PC Card in socket.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#), [csx\\_ReleaseConfiguration\(9F\)](#), [csx\\_ReleaseIO\(9F\)](#), [csx\\_ReleaseIRQ\(9F\)](#), [csx\\_RequestConfiguration\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestIRQ\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

- Warnings**
1. CONF\_VSOVERRIDE is provided for clients that have a need to override the information provided in the CIS. The client must exercise caution when setting this as it overrides any voltage level protection provided by Card Services.
  2. Using [csx\\_ModifyConfiguration\(\)](#) to set Vpp to 0 volts may result in the loss of a PC Card's state. Any client setting Vpp to 0 volts is responsible for insuring that the PC Card's state is restored when power is re-applied to the card.

**Notes** Mapped IO addresses can only be changed by first releasing the current configuration and IO resources with [csx\\_ReleaseConfiguration\(9F\)](#) and [csx\\_ReleaseIO\(9F\)](#), requesting new IO resources and a new configuration with [csx\\_RequestIO\(9F\)](#), followed by [csx\\_RequestConfiguration\(9F\)](#).

IRQ priority can only be changed by first releasing the current configuration and IRQ resources with `csx_ReleaseConfiguration(9F)` and `csx_ReleaseIRQ(9F)`, requesting new IRQ resources and a new configuration with `csx_RequestIRQ(9F)`, followed by `csx_RequestConfiguration(9F)`.

Vcc can not be changed using `csx_ModifyConfiguration()`. Vcc may be changed by first invoking `csx_ReleaseConfiguration(9F)`, followed by `csx_RequestConfiguration(9F)` with a new Vcc value.

**Name** csx\_ModifyWindow – modify window attributes

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ModifyWindow(window_handle_t wh, modify_win_t *mw);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *wh* Window handle returned from [csx\\_RequestWindow\(9F\)](#).

*mw* Pointer to a `modify_win_t` structure.

**Description** This function modifies the attributes of a window allocated by the [csx\\_RequestWindow\(9F\)](#) function.

Only some of the window attributes or the access speed field may be modified by this request. The [csx\\_MapMemPage\(9F\)](#) function is also used to set the offset into PC Card memory to be mapped into system memory for paged windows. The [csx\\_RequestWindow\(9F\)](#) and [csx\\_ReleaseWindow\(9F\)](#) functions must be used to change the window base or size.

**Structure Members** The structure members of `modify_win_t` are:

```
uint32_t    Attributes;           /* window flags */
uint32_t    AccessSpeed;         /* window access speed */
```

The fields are defined as follows:

**Attributes** This field is bit-mapped and defined as follows:

WIN_MEMORY_TYPE_CM	Window points to Common Memory area. Set this to map the window to Common Memory.
WIN_MEMORY_TYPE_AM	Window points to Attribute Memory area. Set this to map the window to Attribute Memory.
WIN_ENABLE	Enable Window. The client must set this to enable the window.
WIN_ACCESS_SPEED_VALID	AccessSpeed valid. The client must set this when the AccessSpeed field has a value that the client wants set for the window.

**AccessSpeed** The bit definitions for this field use the format of the extended speed byte of the Device ID tuple. If the mantissa is 0 (noted as reserved in the *PC Card 95 Standard*), the lower bits are a binary code representing a speed from the list below. Numbers in the first column are codes; items in the second column are speeds.

0	Reserved: do not use
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5 - 7	Reserved: do not use

It is recommended that clients use the [csx\\_ConvertSpeed\(9F\)](#) function to generate the appropriate `AccessSpeed` values rather than manually perturbing the `AccessSpeed` field.

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_HANDLE</code>	Window handle is invalid.
	<code>CS_NO_CARD</code>	No PC Card in socket.
	<code>CS_BAD_OFFSET</code>	Error getting/setting window hardware parameters.
	<code>CS_BAD_WINDOW</code>	Error getting/setting window hardware parameters.
	<code>CS_BAD_SPEED</code>	<code>AccessSpeed</code> is invalid.
	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_ConvertSpeed\(9F\)](#), [csx\\_MapMemPage\(9F\)](#), [csx\\_ReleaseWindow\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_BATTERY – parse the Battery Replacement Date tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_BATTERY(client_handle_t ch, tuple_t *tu,
                                cistpl_battery_t *cb);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cb* Pointer to a `cistpl_battery_t` structure which contains the parsed CISTPL\_BATTERY tuple information upon return from this function.

**Description** This function parses the Battery Replacement Date tuple, CISTPL\_BATTERY, into a form usable by PC Card drivers.

The CISTPL\_BATTERY tuple is an optional tuple which shall be present only in PC Cards with battery-backed storage. It indicates the date on which the battery was replaced, and the date on which the battery is expected to need replacement. Only one CISTPL\_BATTERY tuple is allowed per PC Card.

**Structure Members** The structure members of `cistpl_battery_t` are:

```
uint32_t rday; /* date battery last replaced */
uint32_t xday; /* date battery due for replacement */
```

The fields are defined as follows:

*rday* This field indicates the date on which the battery was last replaced.

*xday* This field indicates the date on which the battery should be replaced.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*



**Name** csx\_Parse\_CISTPL\_BYTEORDER – parse the Byte Order tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_BYTEORDER(client_handle_t ch, tuple_t *tu,
    cistpl_byteorder_t *cbo);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from `csx_RegisterClient(9F)`.
- tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- cbo* Pointer to a `cistpl_byteorder_t` structure which contains the parsed CISTPL\_BYTEORDER tuple information upon return from this function.

**Description** This function parses the Byte Order tuple, CISTPL\_BYTEORDER, into a form usable by PC Card drivers.

The CISTPL\_BYTEORDER tuple shall only appear in a partition tuple set for a memory-like partition. It specifies two parameters: the order for multi-byte data, and the order in which bytes map into words for 16-bit cards.

**Structure Members** The structure members of `cistpl_byteorder_t` are:

```
uint32_t    order;    /* byte order code */
uint32_t    map;     /* byte mapping code */
```

The fields are defined as follows:

order	This field specifies the byte order for multi-byte numeric data.
TPLBYTEORD_LOW	Little endian order
TPLBYTEORD_VS	Vendor specific
map	This field specifies the byte mapping for 16-bit or wider cards.
TPLBYTEMAP_LOW	Byte zero is least significant byte
TPLBYTEMAP_HIGH	Byte zero is most significant byte
TPLBYTEMAP_VS	Vendor specific mapping

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) PC Card.

CS\_UNSUPPORTED\_FUNCTION      No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#),  
[csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_CFTABLE\_ENTRY – parse 16-bit Card Configuration Table Entry tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_CFTABLE_ENTRY(client_handle_t ch, tuple_t *tu,
    cistpl_cftable_entry_t *cft);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- cft* Pointer to a `cistpl_cftable_entry_t` structure which contains the parsed CISTPL\_CFTABLE\_ENTRY tuple information upon return from this function.

**Description** This function parses the 16 bit Card Configuration Table Entry tuple, CISTPL\_CFTABLE\_ENTRY, into a form usable by PC Card drivers.

The CISTPL\_CFTABLE\_ENTRY tuple is used to describe each possible configuration of a PC Card and to distinguish among the permitted configurations. The CISTPL\_CONFIG tuple must precede all CISTPL\_CFTABLE\_ENTRY tuples.

**Structure Members** The structure members of `cistpl_cftable_entry_t` are:

```
uint32_t          flags;      /* valid descriptions */
uint32_t          ifc;       /* interface description */
                    /* information */
uint32_t          pin;       /* values for PRR */
uint32_t          index;    /* configuration index number */
cistpl_cftable_entry_pd_t  pd; /* power requirements */
                    /* description */
cistpl_cftable_entry_speed_t  speed; /* device speed description */
cistpl_cftable_entry_io_t    io; /* device I/O map */
cistpl_cftable_entry_irq_t   irq; /* device IRQ utilization */
cistpl_cftable_entry_mem_t   mem; /* device memory space */
cistpl_cftable_entry_misc_t  misc; /* miscellaneous
                    /* device features */
```

The `flags` field is defined and bit-mapped as follows:

CISTPL_CFTABLE_TPCE_DEFAULT	This is a default configuration
CISTPL_CFTABLE_TPCE_IF	If configuration byte exists
CISTPL_CFTABLE_TPCE_FS_PWR	Power information exists
CISTPL_CFTABLE_TPCE_FS_TD	Timing information exists
CISTPL_CFTABLE_TPCE_FS_IO	I/O information exists

CISTPL_CFTABLE_TPCE_FS_IRQ	IRQ information exists
CISTPL_CFTABLE_TPCE_FS_MEM	MEM space information exists
CISTPL_CFTABLE_TPCE_FS_MISC	MISC information exists
CISTPL_CFTABLE_TPCE_FS_STCE_EV	STCE_EV exists
CISTPL_CFTABLE_TPCE_FS_STCE_PD	STCE_PD exists

If the CISTPL\_CFTABLE\_TPCE\_IF flag is set, the `ifc` field is bit-mapped and defined as follows:

CISTPL_CFTABLE_TPCE_IF_MEMORY	Memory interface
CISTPL_CFTABLE_TPCE_IF_IO_MEM	IO and memory
CISTPL_CFTABLE_TPCE_IF_CUSTOM_0	Custom interface 0
CISTPL_CFTABLE_TPCE_IF_CUSTOM_1	Custom interface 1
CISTPL_CFTABLE_TPCE_IF_CUSTOM_2	Custom interface 2
CISTPL_CFTABLE_TPCE_IF_CUSTOM_3	Custom interface 3
CISTPL_CFTABLE_TPCE_IF_MASK	Interface type mask
CISTPL_CFTABLE_TPCE_IF_BVD	BVD active in PRR
CISTPL_CFTABLE_TPCE_IF_WP	WP active in PRR
CISTPL_CFTABLE_TPCE_IF_RDY	RDY active in PRR
CISTPL_CFTABLE_TPCE_IF_MWAIT	WAIT - mem cycles

`pin` is a value for the Pin Replacement Register.

`index` is a configuration index number.

The structure members of `cistpl_cftable_entry_pd_t` are:

```
uint32_t          flags;          /* which descriptions are valid */
cistpl_cftable_entry_pwr_t pd_vcc; /* VCC power description */
cistpl_cftable_entry_pwr_t pd_vpp1; /* Vpp1 power description */
cistpl_cftable_entry_pwr_t pd_vpp2; /* Vpp2 power description */
```

This `flags` field is bit-mapped and defined as follows:

CISTPL_CFTABLE_TPCE_FS_PWR_VCC	Vcc description valid
CISTPL_CFTABLE_TPCE_FS_PWR_VPP1	Vpp1 description valid
CISTPL_CFTABLE_TPCE_FS_PWR_VPP2	Vpp2 description valid

The structure members of `cistpl_cftable_entry_pwr_t` are:

```

uint32_t    nomV;        /* nominal supply voltage */
uint32_t    nomV_flags;
uint32_t    minV;       /* minimum supply voltage */
uint32_t    minV_flags;
uint32_t    maxV;       /* maximum supply voltage */
uint32_t    maxV_flags;
uint32_t    staticI;    /* continuous supply current */
uint32_t    staticI_flags;
uint32_t    avgI;       /* max current required averaged over 1 sec. */
uint32_t    avgI_flags;
uint32_t    peakI;     /* max current required averaged over 10mS */
uint32_t    peakI_flags;
uint32_t    pdownI;    /* power down supply current required */
uint32_t    pdownI_flags;

```

nomV, minV, maxV, staticI, avgI, peakI\_flag, and pdownI are defined and bit-mapped as follows:

CISTPL_CFTABLE_PD_NOMV	Nominal supply voltage
CISTPL_CFTABLE_PD_MINV	Minimum supply voltage
CISTPL_CFTABLE_PD_MAXV	Maximum supply voltage
CISTPL_CFTABLE_PD_STATICI	Continuous supply current
CISTPL_CFTABLE_PD_AVGI	Maximum current required averaged over 1 second
CISTPL_CFTABLE_PD_PEAKI	Maximum current required averaged over 10mS
CISTPL_CFTABLE_PD_PDOWNI	Power down supply current required

nomV\_flags, minV\_flags, maxV\_flags, staticI\_flags, avgI\_flags, peakI\_flags, and pdownI\_flags are defined and bit-mapped as follows:

CISTPL_CFTABLE_PD_EXISTS	This parameter exists
CISTPL_CFTABLE_PD_MUL10	Multiply return value by 10
CISTPL_CFTABLE_PD_NC_SLEEP	No connection on sleep/power down
CISTPL_CFTABLE_PD_ZERO	Zero value required
CISTPL_CFTABLE_PD_NC	No connection ever

The structure members of `cistpl_cftable_entry_speed_t` are:

```

uint32_t    flags;      /* which timing information is present */
uint32_t    wait;      /* max WAIT time in device speed format */
uint32_t    nS_wait;   /* max WAIT time in nS */
uint32_t    rdybsy;    /* max RDY/BSY time in device speed format */
uint32_t    nS_rdybsy; /* max RDY/BSY time in nS */

```

```
uint32_t  rsvd;          /* max RSVD time in device speed format */
uint32_t  nS_rsvd;     /* max RSVD time in nS */
```

The flags field is bit-mapped and defined as follows:

```
CISTPL_CFTABLE_TPCE_FS_TD_WAIT    WAIT timing exists
CISTPL_CFTABLE_TPCE_FS_TD_RDY     RDY/BSY timing exists
CISTPL_CFTABLE_TPCE_FS_TD_RSVD    RSVD timing exists
```

The structure members of `cistpl_cftable_entry_io_t` are:

```
uint32_t  flags;        /* direct copy of TPCE_IO byte in tuple */
uint32_t  addr_lines;  /* number of decoded I/O address lines */
uint32_t  ranges;      /* number of I/O ranges */
cistpl_cftable_entry_io_range_t
    range[CISTPL_CFTABLE_ENTRY_MAX_IO_RANGES];
```

The flags field is defined and bit-mapped as follows:

```
CISTPL_CFTABLE_TPCE_FS_IO_BUS     Bus width mask
CISTPL_CFTABLE_TPCE_FS_IO_BUS8    8-bit flag
CISTPL_CFTABLE_TPCE_FS_IO_BUS16   16-bit flag
CISTPL_CFTABLE_TPCE_FS_IO_RANGE   IO address ranges exist
```

The structure members of `cistpl_cftable_entry_io_range_t` are:

```
uint32_t  addr;        /* I/O start address */
uint32_t  length;     /* I/O register length */
```

The structure members of `cistpl_cftable_entry_irq_t` are:

```
uint32_t  flags;      /* direct copy of TPCE_IR byte in tuple */
uint32_t  irqs;      /* bit mask for each allowed IRQ */
```

The structure members of `cistpl_cftable_entry_mem_t` are:

```
uint32_t  flags;      /* memory descriptor type and host addr info */
uint32_t  windows;   /* number of memory space descriptors */
cistpl_cftable_entry_mem_window_t
    window[CISTPL_CFTABLE_ENTRY_MAX_MEM_WINDOWS];
```

The flags field is defined and bit-mapped as follows:

```
CISTPL_CFTABLE_TPCE_FS_MEM3       Space descriptors
CISTPL_CFTABLE_TPCE_FS_MEM2       host_addr=card_addr
CISTPL_CFTABLE_TPCE_FS_MEM1       Card address=0 any host address
```

CISTPL\_CFTABLE\_TPCE\_FS\_MEM\_HOST     If host address is present in MEM3

The structure members of `cistpl_cftable_entry_mem_window_t` are:

```
uint32_t    length;    /* length of this window */
uint32_t    card_addr; /* card address */
uint32_t    host_addr; /* host address */
```

The structure members of `cistpl_cftable_entry_misc_t` are:

```
uint32_t    flags;    /* miscellaneous features flags */
```

The `flags` field is defined and bit-mapped as follows:

CISTPL_CFTABLE_TPCE_MI_MTC_MASK	Max twin cards mask
CISTPL_CFTABLE_TPCE_MI_AUDIO	Audio on BVD2
CISTPL_CFTABLE_TPCE_MI_READONLY	R/O storage
CISTPL_CFTABLE_TPCE_MI_PWRDOWN	Powerdown capable
CISTPL_CFTABLE_TPCE_MI_DRQ_MASK	DMAREQ mask
CISTPL_CFTABLE_TPCE_MI_DRQ_SPK	DMAREQ on SPKR
CISTPL_CFTABLE_TPCE_MI_DRQ_IOIS	DMAREQ on IOIS16
CISTPL_CFTABLE_TPCE_MI_DRQ_INP	DMAREQ on INPACK
CISTPL_CFTABLE_TPCE_MI_DMA_8	DMA width 8 bits
CISTPL_CFTABLE_TPCE_MI_DMA_16	DMA width 16 bits

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_CONFIG\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_CONFIG – parse Configuration tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_CONFIG(client_handle_t ch, tuple_t *tu,
                               cistpl_config_t *cc);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cc* Pointer to a `cistpl_config_t` structure which contains the parsed CISTPL\_CONFIG tuple information upon return from this function.

**Description** This function parses the Configuration tuple, CISTPL\_CONFIG, into a form usable by PC Card drivers. The CISTPL\_CONFIG tuple is used to describe the general characteristics of 16-bit PC Cards containing I/O devices or using custom interfaces. It may also describe PC Cards, including Memory Only cards, which exceed nominal power supply specifications, or which need descriptions of their power requirements or other information.

**Structure Members** The structure members of `cistpl_config_t` are:

```
uint32_t    present;    /* register present flags */
uint32_t    nr;        /* number of config registers found */
uint32_t    hr;        /* highest config register index found */
uint32_t    regs[CISTPL_CONFIG_MAX_CONFIG_REGS]; /* reg offsets */
uint32_t    base;     /* base offset of config registers */
uint32_t    last;     /* last config index */
```

The fields are defined as follows:

present	This field indicates which configuration registers are present on the PC Card.
CONFIG_OPTION_REG_PRESENT	Configuration Option Register present
CONFIG_STATUS_REG_PRESENT	Configuration Status Register present
CONFIG_PINREPL_REG_PRESENT	Pin Replacement Register present
CONFIG_COPY_REG_PRESENT	Copy Register present
CONFIG_EXSTAT_REG_PRESENT	Extended Status Register present
CONFIG_IOBASE0_REG_PRESENT	IO Base 0 Register present
CONFIG_IOBASE1_REG_PRESENT	IO Base 1 Register present
CONFIG_IOBASE2_REG_PRESENT	IO Base2 Register present



	CONFIG_IOBASE3_REG_PRESENT	IO Base3 Register present
	CONFIG_IOLIMIT_REG_PRESENT	IO Limit Register present
nr	This field specifies the number of configuration registers that are present on the PC Card.	
hr	This field specifies the highest configuration register number that is present on the PC Card.	
regs	This array contains the offset from the start of Attribute Memory space for each configuration register that is present on the PC Card. If a configuration register is not present on the PC Card, the value in the corresponding entry in the regs array is undefined.	
base	This field contains the offset from the start of Attribute Memory space to the base of the PC Card configuration register space.	
last	This field contains the value of the last valid configuration index for this PC Card.	

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_CFTABLE\\_ENTRY\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Notes** PC Card drivers should not attempt to use configurations beyond the “last” member in the `cistpl_config_t` structure.

**Name** csx\_Parse\_CISTPL\_DATE – parse the Card Initialization Date tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_DATE(client_handle_t ch, tuple_t *tu,
                             cistpl_date_t *cd);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cd* Pointer to a `cistpl_date_t` structure which contains the parsed CISTPL\_DATE tuple information upon return from this function.

**Description** This function parses the Card Initialization Date tuple, CISTPL\_DATE, into a form usable by PC Card drivers.

The CISTPL\_DATE tuple is an optional tuple. It indicates the date and time at which the card was formatted. Only one CISTPL\_DATE tuple is allowed per PC Card.

**Structure Members** The structure members of `cistpl_date_t` are:

```
uint32_t    time;
uint32_t    day
```

The fields are defined as follows:

*time* This field indicates the time at which the PC Card was initialized.

*day* This field indicates the date the PC Card was initialized.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** `csx_Parse_CISTPL_DEVICE`, `csx_Parse_CISTPL_DEVICE_A`,  
`csx_Parse_CISTPL_DEVICE_OC`, `csx_Parse_CISTPL_DEVICE_OA` – parse Device  
Information tuples

**Synopsis** `#include <sys/pccard.h>`

```
int32_t csx_Parse_CISTPL_DEVICE(client_handle_t ch, tuple_t *tu,
                               cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_A(client_handle_t ch, tuple_t *tu,
                                  cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OC(client_handle_t ch, tuple_t *tu,
                                   cistpl_device_t *cd);

int32_t csx_Parse_CISTPL_DEVICE_OA(client_handle_t ch, tuple_t *tu,
                                    cistpl_device_t *cd);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from `csx_RegisterClient(9F)`.

*tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.

*cd* Pointer to a `cistpl_device_t` structure which contains the parsed `CISTPL_DEVICE`, `CISTPL_DEVICE_A`, `CISTPL_DEVICE_OC`, or `CISTPL_DEVICE_OA` tuple information upon return from these functions, respectively.

**Description** `csx_Parse_CISTPL_DEVICE()` and `csx_Parse_CISTPL_DEVICE_A()` parse the 5 volt Device Information tuples, `CISTPL_DEVICE` and `CISTPL_DEVICE_A`, respectively, into a form usable by PC Card drivers.

`csx_Parse_CISTPL_DEVICE_OC()` and `csx_Parse_CISTPL_DEVICE_OA()` parse the Other Condition Device Information tuples, `CISTPL_DEVICE_OC` and `CISTPL_DEVICE_OA`, respectively, into a form usable by PC Card drivers.

The `CISTPL_DEVICE` and `CISTPL_DEVICE_A` tuples are used to describe the card's device information, such as device speed, device size, device type, and address space layout information for Common Memory or Attribute Memory space, respectively.

The `CISTPL_DEVICE_OC` and `CISTPL_DEVICE_OA` tuples are used to describe the information about the card's device under a set of operating conditions for Common Memory or Attribute Memory space, respectively.

**Structure Members** The structure members of `cistpl_device_t` are:

```
uint32_t          num_devices;    /* number of devices found */
cistpl_device_node_t devnode[CISTPL_DEVICE_MAX_DEVICES];
```

The structure members of `cistpl_device_node_t` are:

```
uint32_t    flags;           /* flags specific to this device */
uint32_t    speed;          /* device speed in device
                             /* speed code format */
uint32_t    nS_speed;      /* device speed in nS */
uint32_t    type;          /* device type */
uint32_t    size;          /* device size */
uint32_t    size_in_bytes; /* device size in bytes */
```

The fields are defined as follows:

**flags** This field indicates whether or not the device is writable, and describes a Vcc voltage at which the PC Card can be operated.

`CISTPL_DEVICE_WPS` Write Protect Switch bit is set

Bits which are applicable only for `CISTPL_DEVICE_OC` and `CISTPL_DEVICE_OA` are:

`CISTPL_DEVICE_OC_MWAIT` Use MWAIT

`CISTPL_DEVICE_OC_Vcc_MASK` Mask for Vcc value

`CISTPL_DEVICE_OC_Vcc5` 5.0 volt operation

`CISTPL_DEVICE_OC_Vcc33` 3.3 volt operation

`CISTPL_DEVICE_OC_VccXX` X.X volt operation

`CISTPL_DEVICE_OC_VccYY` Y.Y volt operation

**speed** The device speed value described in the device speed code unit. If this field is set to `CISTPL_DEVICE_SPEED_SIZE_IGNORE`, then the speed information will be ignored.

**nS\_speed** The device speed value described in nanosecond units.

**size** The device size value described in the device size code unit. If this field is set to `CISTPL_DEVICE_SPEED_SIZE_IGNORE`, then the size information will be ignored.

**size\_in\_bytes** The device size value described in byte units.

**type** This is the device type code field which is defined as follows:

`CISTPL_DEVICE_DTYPE_NULL` No device

`CISTPL_DEVICE_DTYPE_ROM` Masked ROM

`CISTPL_DEVICE_DTYPE_OTPROM` One Time Programmable ROM

`CISTPL_DEVICE_DTYPE_EPROM` UV EPROM

CISTPL_DEVICE_DTYPE_EEPROM	EEPROM
CISTPL_DEVICE_DTYPE_FLASH	FLASH
CISTPL_DEVICE_DTYPE_SRAM	Static RAM
CISTPL_DEVICE_DTYPE_DRAM	Dynamic RAM
CISTPL_DEVICE_DTYPE_FUNCSPEC	Function-specific memory address range
CISTPL_DEVICE_DTYPE_EXTEND	Extended type follows

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_JEDEC\\_C\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_DEVICEGEO – parse the Device Geo tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_DEVICEGEO(client_handle_t ch, tuple_t *tp,
    cistpl_devicegeo_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tp* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*pt* Pointer to a `cistpl_devicegeo_t` structure which contains the parsed Device Geo tuple information upon return from this function.

**Description** This function parses the Device Geo tuple, `CISTPL_DEVICEGEO`, into a form usable by PC Card drivers.

The `CISTPL_DEVICEGEO` tuple describes the device geometry of common memory partitions.

**Structure Members** The structure members of `cistpl_devicegeo_t` are:

```
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil;
```

The fields are defined as follows:

<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus</code>	This field indicates the card interface width in bytes for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs</code>	This field indicates the minimum erase block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs</code>	This field indicates the minimum read block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs</code>	This field indicates the minimum write block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part</code>	This field indicates the segment partition subdivisions for the given partition.

`info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil` This field indicates the hardware interleave

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_HANDLE</code>	Client handle is invalid.
	<code>CS_UNKNOWN_TUPLE</code>	Parser does not know how to parse tuple.
	<code>CS_NO_CARD</code>	No PC Card in socket.
	<code>CS_NO_CIS</code>	No Card Information Structure (CIS) on PC Card.
	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** `csx_GetFirstTuple(9F)`, `csx_GetNextTuple(9F)`, `csx_GetTupleData(9F)`, `csx_Parse_CISTPL_DEVICEGEO_A(9F)`, `csx_RegisterClient(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_DEVICEGEO\_A – parse the Device Geo A tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_DEVICEGEO_A(client_handle_t ch, tuple_t *tp,
    cistpl_devicegeo_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tp* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*pt* Pointer to a `cistpl_devicegeo_t` structure which contains the parsed Device Geo A tuple information upon return from this function.

**Description** This function parses the Device Geo A tuple, `CISTPL_DEVICEGEO_A`, into a form usable by PC Card drivers.

The `CISTPL_DEVICEGEO_A` tuple describes the device geometry of attribute memory partitions.

**Structure Members** The structure members of `cistpl_devicegeo_t` are:

```
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part;
uint32_t    info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil;
```

The fields are defined as follows:

<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].bus</code>	This field indicates the card interface width in bytes for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].ebs</code>	This field indicates the minimum erase block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].rbs</code>	This field indicates the minimum read block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].wbs</code>	This field indicates the minimum write block size for the given partition.
<code>info[CISTPL_DEVICEGEO_MAX_PARTITIONS].part</code>	This field indicates the segment partition subdivisions for the given partition.



`info[CISTPL_DEVICEGEO_MAX_PARTITIONS].hwil` This field indicates the hardware interleave for the given partition.

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_HANDLE</code>	Client handle is invalid.
	<code>CS_UNKNOWN_TUPLE</code>	Parser does not know how to parse tuple.
	<code>CS_NO_CARD</code>	No PC Card in socket.
	<code>CS_NO_CIS</code>	No Card Information Structure (CIS) on PC Card.
	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** `csx_GetFirstTuple(9F)`, `csx_GetNextTuple(9F)`, `csx_GetTupleData(9F)`, `csx_Parse_CISTPL_DEVICEGEO(9F)`, `csx_RegisterClient(9F)`, `tuple(9S)`

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_FORMAT – parse the Data Recording Format tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_FORMAT(client_handle_t ch, tuple_t *tu,
                               cistpl_format_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from `csx_RegisterClient(9F)`.
- tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.
- pt* Pointer to a `cistpl_format_t` structure which contains the parsed CISTPL\_FORMAT tuple information upon return from this function.

**Description** This function parses the Data Recording Format tuple, CISTPL\_FORMAT, into a form usable by PC Card drivers.

The CISTPL\_FORMAT tuple indicates the data recording format for a device partition.

**Structure Members** The structure members of `cistpl_format_t` are:

```
uint32_t    type;
uint32_t    edc_length;
uint32_t    edc_type;
uint32_t    offset;
uint32_t    nbytes;
uint32_t    dev.disk.bksize;
uint32_t    dev.disk.nblocks;
uint32_t    dev.disk.edcloc;
uint32_t    dev.mem.flags;
uint32_t    dev.mem.reserved;
caddr_t     dev.mem.address;
uint32_t    dev.mem.edcloc;
```

The fields are defined as follows:

<code>type</code>	This field indicates the type of device:
	<code>TPLFMTTYPE_DISK</code> disk-like device
	<code>TPLFMTTYPE_MEM</code> memory-like device
	<code>TPLFMTTYPE_VS</code> vendor-specific device
<code>edc_length</code>	This field indicates the error detection code length.
<code>edc_type</code>	This field indicates the error detection code type.
<code>offset</code>	This field indicates the offset of the first byte of data in this partition.

nbytes	This field indicates the number of bytes of data in this partition
dev.disk.bksize	This field indicates the block size, for disk devices.
dev.disk.nblocks	This field indicates the number of blocks, for disk devices.
dev.disk.edcloc	This field indicates the location of the error detection code, for disk devices.
dev.mem.flags	This field provides flags, for memory devices. Valid flags are: TPLFMTFLAGS_ADDR     address is valid TPLFMTFLAGS_AUTO     automatically map memory region
dev.mem.reserved	This field is reserved.
dev.mem.address	This field indicates the physical address, for memory devices.
dev.mem.edcloc	This field indicates the location of the error detection code, for memory devices.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_FUNCCE – parse Function Extension tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_FUNCCE(client_handle_t ch, tuple_t *tu,
                               cistpl_funcce_t *cf,
                               uint32_t fid);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- cf* Pointer to a `cistpl_funcce_t` structure which contains the parsed CISTPL\_FUNCCE tuple information upon return from this function.
- fid* The function ID code to which this CISTPL\_FUNCCE tuple refers. See [csx\\_Parse\\_CISTPL\\_FUNCID\(9F\)](#).

**Description** This function parses the Function Extension tuple, CISTPL\_FUNCCE, into a form usable by PC Card drivers.

The CISTPL\_FUNCCE tuple is used to describe information about a specific PCCard function. The information provided is determined by the Function Identification tuple, CISTPL\_FUNCID, that is being extended. Each function has a defined set of extension tuples.

**Structure Members** The structure members of `cistpl_funcce_t` are:

```
uint32_t    function;           /* type of extended data */
uint32_t    subfunction;
union {
    struct serial {
        uint32_t ua;           /* UART in use */
        uint32_t uc;           /* UART capabilities */
    } serial;
    struct modem {
        uint32_t fc;           /* supported flow control methods */
        uint32_t cb;           /* size of DCE command buffer */
        uint32_t eb;           /* size of DCE to DCE buffer */
        uint32_t tb;           /* size of DTE to DCE buffer */
    } modem;
    struct data_modem {
        uint32_t ud;           /* highest data rate */
        uint32_t ms;           /* modulation standards */
        uint32_t em;           /* err correct proto and
                               /* non-CCITT modulation */
        uint32_t dc;           /* data compression protocols */
        uint32_t cm;           /* command protocols */
    } data_modem;
}
```

```

        uint32_t ex;      /* escape mechanisms */
        uint32_t dy;      /* standardized data encryption */
        uint32_t ef;      /* miscellaneous end user features */
        uint32_t ncd;     /* number of country codes */
        uchar_t  cd[16];  /* CCITT country code */
    } data_modem;
    struct fax {
        uint32_t uf;      /* highest data rate in DTE/UART */
        uint32_t fm;      /* CCITT modulation standards */
        uint32_t fy;      /* standardized data encryption */
        uint32_t fs;      /* feature selection */
        uint32_t ncf;     /* number of country codes */
        uchar_t  cf[16];  /* CCITT country codes */
    } fax;
    struct voice {
        uint32_t uv;      /* highest data rate */
        uint32_t nsr;     /* voice sampling rates (*100) */
        uint32_t sr[16];  /* voice sampling rates (*100) */
        uint32_t nss;     /* voice sample sizes (*10) */
        uint32_t ss[16];  /* voice sample sizes (*10) */
        uint32_t nsc;     /* voice compression methods */
        uint32_t sc[16];  /* voice compression methods */
    } voice;
    struct lan {
        uint32_t tech;     /* network technology */
        uint32_t speed;    /* media bit or baud rate */
        uint32_t media;    /* network media supported */
        uint32_t con;      /* open/closed connector standard */
        uint32_t id_sz;    /* length of lan station id */
        uchar_t  id[16];   /* station ID */
    } lan;
} data;

```

The fields are defined as follows:

function	This field identifies the type of extended information provided about a function by the CISTPL_FUNCE tuple. This field is defined as follows:
TPLFE_SUB_SERIAL	Serial port interface
TPLFE_SUB_MODEM_COMMON	Common modem interface
TPLFE_SUB_MODEM_DATA	Data modem services
TPLFE_SUB_MODEM_FAX	Fax modem services
TPLFE_SUB_VOICE	Voice services
TPLFE_CAP_MODEM_DATA	Capabilities of the data modem interface

	TPLFE_CAP_MODEM_FAX	Capabilities of the fax modem interface
	TPLFE_CAP_MODEM_VOICE	Capabilities of the voice modem interface
	TPLFE_CAP_SERIAL_DATA	Serial port interface for data modem services
	TPLFE_CAP_SERIAL_FAX	Serial port interface for fax modem services
	TPLFE_CAP_SERIAL_VOICE	Serial port interface for voice modem services
subfunction	This is for identifying a sub-category of services provided by a function in the CISTPL_FUNCE tuple. The numeric value of the code is in the range of 1 to 15.	
ua	This is the serial port UART identification and is defined as follows:	
	TPLFE_UA_8250	Intel 8250
	TPLFE_UA_16450	NS 16450
	TPLFE_UA_16550	NS 16550
uc	This identifies the serial port UART capabilities and is defined as follows:	
	TPLFE_UC_PARITY_SPACE	Space parity supported
	TPLFE_UC_PARITY_MARK	Mark parity supported
	TPLFE_UC_PARITY_ODD	Odd parity supported
	TPLFE_UC_PARITY_EVEN	Even parity supported
	TPLFE_UC_CS5	5 bit characters supported
	TPLFE_UC_CS6	6 bit characters supported
	TPLFE_UC_CS7	7 bit characters supported
	TPLFE_UC_CS8	8 bit characters supported
	TPLFE_UC_STOP_1	1 stop bit supported
	TPLFE_UC_STOP_15	1.5 stop bits supported
	TPLFE_UC_STOP_2	2 stop bits supported
fc	This identifies the modem flow control methods and is defined as follows:	
	TPLFE_FC_TX_XONOFF	Transmit XON/XOFF

TPLFE_FC_RX_XONOFF	Receiver XON/XOFF
TPLFE_FC_TX_HW	Transmit hardware flow control (CTS)
TPLFE_FC_RX_HW	Receiver hardware flow control (RTS)
TPLFE_FC_TRANS	Transparent flow control

ms This identifies the modem modulation standards and is defined as follows:

TPLFE_MS_BELL103	300bps
TPLFE_MS_V21	300bps (V.21)
TPLFE_MS_V23	600/1200bps (V.23)
TPLFE_MS_V22AB	1200bps (V.22A V.22B)
TPLFE_MS_BELL212	2400bps (US Bell 212)
TPLFE_MS_V22BIS	2400bps (V.22bis)
TPLFE_MS_V26	2400bps leased line (V.26)
TPLFE_MS_V26BIS	2400bps (V.26bis)
TPLFE_MS_V27BIS	4800/2400bps leased line (V.27bis)
TPLFE_MS_V29	9600/7200/4800 leased line (V.29)
TPLFE_MS_V32	Up to 9600bps (V.32)
TPLFE_MS_V32BIS	Up to 14400bps (V.32bis)
TPLFE_MS_VFAST	Up to 28800 V.FAST

em This identifies modem error correction/detection protocols and is defined as follows:

TPLFE_EM_MNP	MNP levels 2-4
TPLFE_EM_V42	CCITT LAPM (V.42)

dc This identifies modem data compression protocols and is defined as follows:

TPLFE_DC_V42BI	CCITT compression V.42
TPLFE_DC_MNP5	MNP compression (uses MNP 2, 3 or 4)

cm This identifies modem command protocols and is defined as follows:

TPLFE_CM_AT1	ANSI/EIA/TIA 602 "Action" commands
TPLFE_CM_AT2	ANSI/EIA/TIA 602 "ACE/DCE IF Params"
TPLFE_CM_AT3	ANSI/EIA/TIA 602 "Ace Parameters"

---

	TPLFE_CM_MNP_AT	MNP specification AT commands
	TPLFE_CM_V25BIS	V.25bis calling commands
	TPLFE_CM_V25A	V.25bis test procedures
	TPLFE_CM_DMCL	DMCL command mode
ex		This identifies the modem escape mechanism and is defined as follows:
	TPLFE_EX_BREAK	BREAK support standardized
	TPLFE_EX_PLUS	+++ returns to command mode
	TPLFE_EX_UD	User defined escape character
dy		This identifies modem standardized data encryption and is a reserved field for future use and must be set to 0.
ef		This identifies modem miscellaneous features and is defined as follows:
	TPLFE_EF_CALLERID	Caller ID is supported
fm		This identifies fax modulation standards and is defined as follows:
	TPLFE_FM_V21C2	300bps (V.21-C2)
	TPLFE_FM_V27TER	4800/2400bps (V.27ter)
	TPLFE_FM_V29	9600/7200/4800 leased line (V.29)
	TPLFE_FM_V17	14.4K/12K/9600/7200bps (V.17)
	TPLFE_FM_V33	4.4K/12K/9600/7200 leased line (V.33)
fs		This identifies the fax feature selection and is defined as follows:
	TPLFE_FS_T3	Group 2 (T.3) service class
	TPLFE_FS_T4	Group 3 (T.4) service class
	TPLFE_FS_T6	Group 4 (T.6) service class
	TPLFE_FS_ECM	Error Correction Mode
	TPLFE_FS_VOICEREQ	Voice requests allowed
	TPLFE_FS_POLLING	Polling support
	TPLFE_FS_FTP	File transfer support
	TPLFE_FS_PASSWORD	Password support
tech		This identifies the LAN technology type and is defined as follows:
	TPLFE_LAN_TECH_ARCNET	Arcnet



	TPLFE_LAN_TECH_ETHERNET	Ethernet
	TPLFE_LAN_TECH_TOKENRING	Token Ring
	TPLFE_LAN_TECH_LOCALTALK	Local Talk
	TPLFE_LAN_TECH_FDDI	FDDI/CDDI
	TPLFE_LAN_TECH_ATM	ATM
	TPLFE_LAN_TECH_WIRELESS	Wireless
<b>media</b>	This identifies the LAN media type and is defined as follows:	
	TPLFE_LAN_MEDIA_INHERENT	Generic interface
	TPLFE_LAN_MEDIA_UTP	Unshielded twisted pair
	TPLFE_LAN_MEDIA_STP	Shielded twisted pair
	TPLFE_LAN_MEDIA_THIN_COAX	Thin coax
	TPLFE_LAN_MEDIA_THICK_COAX	Thick coax
	TPLFE_LAN_MEDIA_FIBER	Fiber
	TPLFE_LAN_MEDIA_SSR_902	Spread spectrum radio 902-928 MHz
	TPLFE_LAN_MEDIA_SSR_2_4	Spread spectrum radio 2.4 GHz
	TPLFE_LAN_MEDIA_SSR_5_4	Spread spectrum radio 5.4 GHz
	TPLFE_LAN_MEDIA_DIFFUSE_IR	Diffuse infra red
	TPLFE_LAN_MEDIA_PTP_IR	Point to point infra red

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_FUNCID\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_FUNCID – parse Function Identification tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_FUNCID(client_handle_t ch, tuple_t *tu,
                               cistpl_funcid_t *cf);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cf* Pointer to a `cistpl_funcid_t` structure which contains the parsed CISTPL\_FUNCID tuple information upon return from this function.

**Description** This function parses the Function Identification tuple, CISTPL\_FUNCID, into a form usable by PC Card drivers.

The CISTPL\_FUNCID tuple is used to describe information about the functionality provided by a PC Card. Information is also provided to enable system utilities to decide if the PC Card should be configured during system initialization. If additional function specific information is available, one or more function extension tuples of type CISTPL\_FUNCID follow this tuple (see [csx\\_Parse\\_CISTPL\\_FUNCID\(9F\)](#)).

**Structure Members** The structure members of `cistpl_funcid_t` are:

```
uint32_t    function;    /* PC Card function code */
uint32_t    sysinit;    /* system initialization mask */
```

The fields are defined as follows:

function	This is the function type for CISTPL_FUNCID:
TPLFUNC_MULTI	Vendor-specific multifunction card
TPLFUNC_MEMORY	Memory card
TPLFUNC_SERIAL	Serial I/O port
TPLFUNC_PARALLEL	Parallel printer port
TPLFUNC_FIXED	Fixed disk, silicon or removable
TPLFUNC_VIDEO	Video interface
TPLFUNC_LAN	Local Area Network adapter
TPLFUNC_AIMS	Auto Incrementing Mass Storage
TPLFUNC_SCSI	SCSI bridge

---

	TPLFUNC_SECURITY	Security cards
	TPLFUNC_VENDOR_SPECIFIC	Vendor specific
	TPLFUNC_UNKNOWN	Unknown function(s)
sysinit	This field is bit-mapped and defined as follows:	
	TPLINIT_POST	POST should attempt configure
	TPLINIT_ROM	Map ROM during sys init
<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.
<b>Context</b>	This function may be called from user or kernel context.	
<b>See Also</b>	<a href="#">csx_GetFirstTuple(9F)</a> , <a href="#">csx_GetTupleData(9F)</a> , <a href="#">csx_Parse_CISTPL_FUNCID(9F)</a> , <a href="#">csx_RegisterClient(9F)</a> , <a href="#">csx_ValidateCIS(9F)</a> , <a href="#">tuple(9S)</a>	
	<i>PC Card 95 Standard, PCMCIA/JEIDA</i>	

**Name** csx\_Parse\_CISTPL\_GEOMETRY – parse the Geometry tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_GEOMETRY(client_handle_t ch, tuple_t *tu,
    cistpl_geometry_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- pt* Pointer to a `cistpl_geometry_t` structure which contains the parsed CISTPL\_GEOMETRY tuple information upon return from this function.

**Description** This function parses the Geometry tuple, CISTPL\_GEOMETRY, into a form usable by PC Card drivers.

The CISTPL\_GEOMETRY tuple indicates the geometry of a disk-like device.

**Structure Members** The structure members of `cistpl_geometry_t` are:

```
uint32_t    spt;
uint32_t    tpc;
uint32_t    ncyl;
```

The fields are defined as follows:

*spt* This field indicates the number of sectors per track.  
*tpc* This field indicates the number of tracks per cylinder.  
*ncyl* This field indicates the number of cylinders.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_JEDEC\_C, csx\_Parse\_CISTPL\_JEDEC\_A – parse JEDEC Identifier tuples

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_JEDEC_C(client_handle_t ch, tuple_t *tu,
                                cistpl_jedec_t *cj);
```

```
int32_t csx_Parse_CISTPL_JEDEC_A(client_handle_t ch, tuple_t *tu,
                                cistpl_jedec_t *cj);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- cj* Pointer to a `cistpl_jedec_t` structure which contains the parsed CISTPL\_JEDEC\_C or CISTPL\_JEDEC\_A tuple information upon return from these functions, respectively.

**Description** `csx_Parse_CISTPL_JEDEC_C()` and `csx_Parse_CISTPL_JEDEC_A()` parse the JEDEC Identifier tuples, CISTPL\_JEDEC\_C and CISTPL\_JEDEC\_A, respectively, into a form usable by PC Card drivers.

The CISTPL\_JEDEC\_C and CISTPL\_JEDEC\_A tuples are optional tuples provided for cards containing programmable devices. They describe information for Common Memory or Attribute Memory space, respectively.

**Structure Members** The structure members of `cistpl_jedec_t` are:

```
uint32_t      nid;    /* # of JEDEC identifiers present */
jedec_ident_t jid[CISTPL_JEDEC_MAX_IDENTIFIERS];
```

The structure members of `jedec_ident_t` are:

```
uint32_t      id;     /* manufacturer id */
uint32_t      info;  /* manufacturer specific info */
```

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_DEVICE\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_LINKTARGET – parse the Link Target tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_LINKTARGET(client_handle_t ch, tuple_t *tu,
    cistpl_linktarget_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*pt* Pointer to a `cistpl_linktarget_t` structure which contains the parsed CISTPL\_LINKTARGET tuple information upon return from this function.

**Description** This function parses the Link Target tuple, CISTPL\_LINKTARGET, into a form usable by PCCard drivers.

The CISTPL\_LINKTARGET tuple is used to verify that tuple chains other than the primary chain are valid. All secondary tuple chains are required to contain this tuple as the first tuple of the chain.

**Structure Members** The structure members of `cistpl_linktarget_t` are:

```
uint32_t length;
char      tpltg_tag[CIS_MAX_TUPLE_DATA_LEN];
```

The fields are defined as follows:

`length` This field indicates the number of bytes in `tpltg_tag`.

`tpltg_tag` This field provides the Link Target tuple information.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)



*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_LONGLINK\_A, csx\_Parse\_CISTPL\_LONGLINK\_C – parse the Long Link A and C tuples

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_LONGLINK_A(client_handle_t ch, tuple_t *tu,
    cistpl_longlink_ac_t *pt);
```

```
int32_t csx_Parse_CISTPL_LONGLINK_C(client_handle_t ch, tuple_t *tu,
    cistpl_longlink_ac_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- pt* Pointer to a `cistpl_longlink_ac_t` structure which contains the parsed CISTPL\_LONGLINK\_A or CISTPL\_LONGLINK\_C tuple information upon return from this function.

**Description** This function parses the Long Link A and C tuples, CISTPL\_LONGLINK\_A and CISTPL\_LONGLINK\_C, into a form usable by PC Card drivers.

The CISTPL\_LONGLINK\_A and CISTPL\_LONGLINK\_C tuples provide links to Attribute and Common Memory.

**Structure Members** The structure members of `cistpl_longlink_ac_t` are:

```
uint32_t flags;
uint32_t tpll_addr;
```

The fields are defined as follows:

<code>flags</code>	This field indicates the type of memory:
<code>CISTPL_LONGLINK_AC_AM</code>	long link to Attribute Memory
<code>CISTPL_LONGLINK_AC_CM</code>	long link to Common Memory
<code>tpll_addr</code>	This field provides the offset from the beginning of the specified address space.

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_HANDLE</code>	Client handle is invalid.
	<code>CS_UNKNOWN_TUPLE</code>	Parser does not know how to parse tuple.
	<code>CS_NO_CARD</code>	No PC Card in socket.
	<code>CS_NO_CIS</code>	No Card Information Structure (CIS) on PC Card.

CS\_UNSUPPORTED\_FUNCTION      No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#),  
[csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_Parse\_CISTPL\_LONGLINK\_MFC – parse the Multi-Function tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_LONGLINK_MFC(client_handle_t ch, tuple_t *tu,
    cistpl_longlink_mfc_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*pt* Pointer to a `cistpl_longlink_mfc_t` structure which contains the parsed CISTPL\_LONGLINK\_MFC tuple information upon return from this function.

**Description** This function parses the Multi-Function tuple, CISTPL\_LONGLINK\_MFC, into a form usable by PC Card drivers.

The CISTPL\_LONGLINK\_MFC tuple describes the start of the function-specific CIS for each function on a multi-function card.

**Structure Members** The structure members of `cistpl_longlink_mfc_t` are:

```
uint32_t    nfuncs;
uint32_t    nregs;
uint32_t    function[CIS_MAX_FUNCTIONS].tas
uint32_t    function[CIS_MAX_FUNCTIONS].addr
```

The fields are defined as follows:

<code>nfuncs</code>	This field indicates the number of functions on the PC card.				
<code>nregs</code>	This field indicates the number of configuration register sets.				
<code>function[CIS_MAX_FUNCTIONS].tas</code>	This field provides the target address space for each function on the PC card. This field can be one of: <table> <tbody> <tr> <td><code>CISTPL_LONGLINK_MFC_TAS_AM</code></td> <td>CIS in attribute memory</td> </tr> <tr> <td><code>CISTPL_LONGLINK_MFC_TAS_CM</code></td> <td>CIS in common memory</td> </tr> </tbody> </table>	<code>CISTPL_LONGLINK_MFC_TAS_AM</code>	CIS in attribute memory	<code>CISTPL_LONGLINK_MFC_TAS_CM</code>	CIS in common memory
<code>CISTPL_LONGLINK_MFC_TAS_AM</code>	CIS in attribute memory				
<code>CISTPL_LONGLINK_MFC_TAS_CM</code>	CIS in common memory				
<code>function[CIS_MAX_FUNCTIONS].addr</code>	This field				

provides the target address offset for each function on the PC card.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_Parse\_CISTPL\_MANFID – parse Manufacturer Identification tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_MANFID(client_handle_t ch, tuple_t *tu,
                               cistpl_manfid_t *cm);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cm* Pointer to a `cistpl_manfid_t` structure which contains the parsed CISTPL\_MANFID tuple information upon return from this function.

**Description** This function parses the Manufacturer Identification tuple, CISTPL\_MANFID, into a form usable by PC Card drivers.

The CISTPL\_MANFID tuple is used to describe the information about the manufacturer of a PC Card. There are two types of information, the PC Card's manufacturer and a manufacturer card number.

**Structure Members** The structure members of `cistpl_manfid_t` are:

```
uint32_t manf; /* PCMCIA assigned manufacturer code */
uint32_t card; /* manufacturer information
                (part number and/or revision) */
```

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

- Name** csx\_Parse\_CISTPL\_ORG – parse the Data Organization tuple
- Synopsis** #include <sys/pccard.h>
- ```
int32_t csx_Parse_CISTPL_ORG(client_handle_t ch, tuple_t *tu,
                             cistpl_org_t *pt);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters**
- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
  - tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
  - pt* Pointer to a `cistpl_org_t` structure which contains the parsed CISTPL\_ORG tuple information upon return from this function.
- Description** This function parses the Data Organization tuple, CISTPL\_ORG, into a form usable by PC Card drivers.
- The CISTPL\_ORG tuple provides a text description of the organization.
- Structure Members** The structure members of `cistpl_org_t` are:
- ```
uint32_t  type;
char      desc[CIS_MAX_TUPLE_DATA_LEN];
```
- The fields are defined as follows:
- |   |  |
|---|--|
| <code>type</code>                         | This field indicates type of data organization.                |
| <code>desc[CIS_MAX_TUPLE_DATA_LEN]</code> | This field provides the text description of this organization. |
- Return Values**
- |                                      |   |
|--------------------------------------|---|
| <code>CS_SUCCESS</code>              | Successful operation.                           |
| <code>CS_BAD_HANDLE</code>           | Client handle is invalid.                       |
| <code>CS_UNKNOWN_TUPLE</code>        | Parser does not know how to parse tuple.        |
| <code>CS_NO_CARD</code>              | No PC Card in socket.                           |
| <code>CS_NO_CIS</code>               | No Card Information Structure (CIS) on PC Card. |
| <code>CS_UNSUPPORTED_FUNCTION</code> | No PCMCIA hardware installed.                   |
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)
- PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Parse\_CISTPL\_SPCL – parse the Special Purpose tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_SPCL(client_handle_t ch, tuple_t *tu,
                             cistpl_spcl_t *csp);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from `csx_RegisterClient(9F)`.

*tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.

*csp* Pointer to a `cistpl_spcl_t` structure which contains the parsed CISTPL\_SPCL tuple information upon return from this function.

**Description** This function parses the Special Purpose tuple, CISTPL\_SPCL, into a form usable by PC Card drivers.

The CISTPL\_SPCL tuple is identified by an identification field that is assigned by PCMCIA or JEIDA. A sequence field allows a series of CISTPL\_SPCL tuples to be used when the data exceeds the size that can be stored in a single tuple; the maximum data area of a series of CISTPL\_SPCL tuples is unlimited. Another field gives the number of bytes in the data field in this tuple.

**Structure Members** The structure members of `cistpl_date_t` are:

```
uint32_t id;          /* tuple contents identification */
uint32_t seq;        /* data sequence number */
uint32_t bytes;      /* number of bytes following */
uchar_t data[CIS_MAX_TUPLE_DATA_LEN];
```

The fields are defined as follows:

*id* This field contains a PCMCIA or JEIDA assigned value that identifies this series of one or more CISTPL\_SPCL tuples. These field values are assigned by contacting either PCMCIA or JEIDA.

*seq* This field contains a data sequence number. CISTPL\_SPCL\_SEQ\_END is the last tuple in sequence.

*bytes* This field contains the number of data bytes in the `data[CIS_MAX_TUPLE_DATA_LEN]`.

*data* The data component of this tuple.

**Return Values** CS\_SUCCESS Successful operation.

CS\_BAD\_HANDLE Client handle is invalid.



CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_Parse\_CISTPL\_SWIL – parse the Software Interleaving tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_SWIL(client_handle_t ch, tuple_t *tu,
                             cistpl_swil_t *pt);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- pt* Pointer to a `cistpl_swil_t` structure which contains the parsed CISTPL\_SWIL tuple information upon return from this function.

**Description** This function parses the Software Interleaving tuple, CISTPL\_SWIL, into a form usable by PC Card drivers.

The CISTPL\_SWIL tuple provides the software interleaving of data within a partition on the card.

**Structure Members** The structure members of `cistpl_swil_t` are:

```
uint32_t    intrlv;
```

The fields are defined as follows:

`intrlv` This field provides the software interleaving for a partition.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
CS_NO_CARD	No PC Card in socket.
CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

- Name** csx\_Parse\_CISTPL\_VERS\_1 – parse Level-1 Version/Product Information tuple
- Synopsis** #include <sys/pccard.h>
- ```
int32_t csx_Parse_CISTPL_VERS_1(client_handle_t ch, tuple_t *tu,
                               cistpl_vers_1_t *cv1);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters**
- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- tu* Pointer to a [tuple\\_t](#) structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).
- cv1* Pointer to a [cistpl\\_vers\\_1\\_t](#) structure which contains the parsed CISTPL\_VERS\_1 tuple information upon return from this function.
- Description** This function parses the Level-1 Version/Product Information tuple, CISTPL\_VERS\_1, into a form usable by PC Card drivers.
- The CISTPL\_VERS\_1 tuple is used to describe the card Level-1 version compliance and card manufacturer information.
- Structure Members** The structure members of [cistpl\\_vers\\_1\\_t](#) are:
- ```
uint32_t major; /* major version number */
uint32_t minor; /* minor version number */
uint32_t ns; /* number of information strings */
char pi[CISTPL_VERS_1_MAX_PROD_STRINGS]
      [CIS_MAX_TUPLE_DATA_LEN];
      /* pointers to product information strings */
```
- Return Values**
- |                         |   |
|-------------------------|---|
| CS_SUCCESS              | Successful operation.                           |
| CS_BAD_HANDLE           | Client handle is invalid.                       |
| CS_UNKNOWN_TUPLE        | Parser does not know how to parse tuple.        |
| CS_NO_CARD              | No PC Card in socket.                           |
| CS_NO_CIS               | No Card Information Structure (CIS) on PC Card. |
| CS_UNSUPPORTED_FUNCTION | No PCMCIA hardware installed.                   |
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)
- PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_Parse\_CISTPL\_VERS\_2 – parse Level-2 Version and Information tuple

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_Parse_CISTPL_VERS_2(client_handle_t ch, tuple_t *tu,
                               cistpl_vers_2_t *cv2);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from `csx_RegisterClient(9F)`.

*tu* Pointer to a `tuple_t` structure (see `tuple(9S)`) returned by a call to `csx_GetFirstTuple(9F)` or `csx_GetNextTuple(9F)`.

*cv2* Pointer to a `cistpl_vers_2_t` structure which contains the parsed CISTPL\_VERS\_2 tuple information upon return from this function.

**Description** This function parses the Level-2 Version and Information tuple, CISTPL\_VERS\_2, into a form usable by PC Card drivers.

The CISTPL\_VERS\_2 tuple is used to describe the card Level-2 information which has the logical organization of the card's data.

**Structure Members** The structure members of `cistpl_vers_2_t` are:

```
uint32_t vers;      /* version number */
uint32_t comply;   /* level of compliance */
uint32_t dindex;   /* byte address of first data byte in card */
uint32_t vspec8;   /* vendor specific (byte 8) */
uint32_t vspec9;   /* vendor specific (byte 9) */
uint32_t nhdr;     /* number of copies of CIS present on device */
char oem[CIS_MAX_TUPLE_DATA_LEN];
                /* Vendor of software that formatted card */
char info[CIS_MAX_TUPLE_DATA_LEN];
                /* Informational message about card */
```

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** `csx_GetFirstTuple(9F)`, `csx_GetTupleData(9F)`, `csx_RegisterClient(9F)`,  
`csx_ValidateCIS(9F)`, `tuple(9S)`

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_ParseTuple – generic tuple parser

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ParseTuple(client_handle_t ch, tuple_t *tu, cisparsed_t *cp,
                      cisdata_t cd);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*tu* Pointer to a `tuple_t` structure (see [tuple\(9S\)](#)) returned by a call to [csx\\_GetFirstTuple\(9F\)](#) or [csx\\_GetNextTuple\(9F\)](#).

*cp* Pointer to a `cisparsed_t` structure that unifies all tuple parsing structures.

*cd* Extended tuple data for some tuples.

**Description** This function is the generic tuple parser entry point.

**Structure Members** The structure members of `cisparsed_t` are:

```
typedef union cisparsed_t {
    cistpl_config_t      cistpl_config;
    cistpl_device_t     cistpl_device;
    cistpl_vers_1_t     cistpl_vers_1;
    cistpl_vers_2_t     cistpl_vers_2;
    cistpl_jedec_t      cistpl_jedec;
    cistpl_format_t     cistpl_format;
    cistpl_geometry_t   cistpl_geometry;
    cistpl_byteorder_t  cistpl_byteorder;
    cistpl_date_t       cistpl_date;
    cistpl_battery_t    cistpl_battery;
    cistpl_org_t        cistpl_org;
    cistpl_manfid_t     cistpl_manfid;
    cistpl_funcid_t     cistpl_funcid;
    cistpl_func_t       cistpl_func;
    cistpl_cftable_entry_t cistpl_cftable_entry;
    cistpl_linktarget_t cistpl_linktarget;
    cistpl_longlink_ac_t cistpl_longlink_ac;
    cistpl_longlink_mfc_t cistpl_longlink_mfc;
    cistpl_spcl_t       cistpl_spcl;
    cistpl_swil_t       cistpl_swil;
    cistpl_bar_t        cistpl_bar;
    cistpl_devicegeo_t  cistpl_devicegeo;
    cistpl_longlink_cb_t cistpl_longlink_cb;
    cistpl_get_tuple_name_t cistpl_get_tuple_name;
} cisparsed_t;
```

---

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid.
	CS_UNKNOWN_TUPLE	Parser does not know how to parse tuple.
	CS_NO_CARD	No PC Card in socket.
	CS_BAD_CIS	Generic parser error.
	CS_NO_CIS	No Card Information Structure (CIS) on PC Card.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_Parse\\_CISTPL\\_BATTERY\(9F\)](#), [csx\\_Parse\\_CISTPL\\_BYTEORDER\(9F\)](#), [csx\\_Parse\\_CISTPL\\_CFTABLE\\_ENTRY\(9F\)](#), [csx\\_Parse\\_CISTPL\\_CONFIG\(9F\)](#), [csx\\_Parse\\_CISTPL\\_DATE\(9F\)](#), [csx\\_Parse\\_CISTPL\\_DEVICE\(9F\)](#), [csx\\_Parse\\_CISTPL\\_FUNCE\(9F\)](#), [csx\\_Parse\\_CISTPL\\_FUNCID\(9F\)](#), [csx\\_Parse\\_CISTPL\\_JEDEC\\_C\(9F\)](#), [csx\\_Parse\\_CISTPL\\_MANFID\(9F\)](#), [csx\\_Parse\\_CISTPL\\_SPCL\(9F\)](#), [csx\\_Parse\\_CISTPL\\_VERS\\_1\(9F\)](#), [csx\\_Parse\\_CISTPL\\_VERS\\_2\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ValidateCIS\(9F\)](#), [tuple\(9S\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_Put8, csx\_Put16, csx\_Put32, csx\_Put64 – write to device register

**Synopsis** #include <sys/pccard.h>

```
void csx_Put8(acc_handle_t handle, uint32_t offset, uint8_t value);
void csx_Put16(acc_handle_t handle, uint32_t offset, uint16_t value);
void csx_Put32(acc_handle_t handle, uint32_t offset, uint32_t value);
void csx_Put64(acc_handle_t handle, uint32_t offset, uint64_t value);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* The access handle returned from [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#), or [csx\\_DupHandle\(9F\)](#).

*offset* The offset in bytes from the base of the mapped resource.

*value* The data to be written to the device.

**Description** These functions generate a write of various sizes to the mapped memory or device register.

The [csx\\_Put8\(\)](#), [csx\\_Put16\(\)](#), [csx\\_Put32\(\)](#), and [csx\\_Put64\(\)](#) functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, *handle*, at an offset in bytes represented by the *offset*, *offset*.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

**Context** These functions may be called from user, kernel, or interrupt context.

**See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_Get8\(9F\)](#), [csx\\_GetMappedAddr\(9F\)](#), [csx\\_RepGet8\(9F\)](#), [csx\\_RepPut8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*



**Name** csx\_RegisterClient – register a client

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RegisterClient(client_handle_t *ch, client_reg_t *cr);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Pointer to a `client_handle_t` structure.

*mc* Pointer to a `client_reg_t` structure.

**Description** This function registers a client with Card Services and returns a unique client handle for the client. The client handle must be passed to [csx\\_DeregisterClient\(9F\)](#) when the client terminates.

**Structure Members** The structure members of `client_reg_t` are:

```
uint32_t          Attributes;
uint32_t          EventMask;
event_callback_args_t event_callback_args;
uint32_t          Version;           /* CS version to expect */
csfunction_t      *event_handler;
ddi_iblock_cookie_t *iblk_cookie;   /* event iblk cookie */
ddi_idevice_cookie_t *idev_cookie;  /* event idev cookie */
dev_info_t        *dip;             /* client's dip */
char              driver_name[MODMAXNAMELEN];
```

The fields are defined as follows:

Attributes	This field is bit-mapped and defined as follows:  <b>INFO_MEM_CLIENT</b> Memory client device driver.  <b>INFO_MTD_CLIENT</b> Memory Technology Driver client.  <b>INFO_IO_CLIENT</b> IO client device driver.  <b>INFO_CARD_SHARE</b> Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events.  <b>INFO_CARD_EXCL</b> Generate artificial CS_EVENT_CARD_INSERTION and CS_EVENT_REGISTRATION_COMPLETE events.  <b>INFO_MEM_CLIENT</b> <b>INFO_MTD_CLIENT</b>
------------	--

**INFO\_IO\_CLIENT**

These bits are mutually exclusive (that is, only one bit may be set), but one of the bits must be set.

**INFO\_CARD\_SHARE****INFO\_CARD\_EXCL**

If either of these bits is set, the client will receive a `CS_EVENT_REGISTRATION_COMPLETE` event when Card Services has completed its internal client registration processing and after a successful call to [csx\\_RequestSocketMask\(9F\)](#).

Also, if either of these bits is set, and if a card of the type that the client can control is currently inserted in the socket (and after a successful call to [csx\\_RequestSocketMask\(9F\)](#)), the client will receive an artificial `CS_EVENT_CARD_INSERTION` event.

Event Mask	This field is bit-mapped and specifies the client's global event mask. Card Services performs event notification based on this field. See <a href="#">csx_event_handler(9E)</a> for valid event definitions and for additional information about handling events.
event_callback_args	The <code>event_callback_args_t</code> structure members are: <pre>void    *client_data;</pre> <p>The <code>client_data</code> field may be used to provide data available to the event handler (see <a href="#">csx_event_handler(9E)</a>). Typically, this is the client driver's soft state pointer.</p>
Version	This field contains the specific Card Services version number that the client expects to use. Typically, the client will use the <code>CS_VERSION</code> macro to specify to Card Services which version of Card Services the client expects.
event_handler	The client event callback handler entry point is passed in the <code>event_handler</code> field.
iblk_cookie idev_cookie	These fields must be used by the client to set up mutexes that are used in the client's event callback handler when handling high priority events.
dip	The client must set this field with a pointer to the client's dip.
driver_name	The client must copy a driver-unique name into this member. This name must be identical across all instances of the driver.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_ATTRIBUTE	No client type or more than one client type specified.
	CS_OUT_OF_RESOURCE	Card Services is unable to register client.
	CS_BAD_VERSION	Card Services version is incompatible with client.
	CS_BAD_HANDLE	Client has already registered for this socket.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_DeregisterClient\(9F\)](#), [csx\\_RequestSocketMask\(9F\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_ReleaseConfiguration – release PC Card and socket configuration

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ReleaseConfiguration(client_handle_t ch, release_config_t *rc);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*rc* Pointer to a `release_config_t` structure.

**Description** This function returns a PC Card and socket to a simple memory only interface and sets the card to configuration zero by writing a 0 to the PC card's COR (Configuration Option Register).

Card Services may remove power from the socket if no clients have indicated their usage of the socket by an active [csx\\_RequestConfiguration\(9F\)](#) or [csx\\_RequestWindow\(9F\)](#).

Card Services is prohibited from resetting the PC Card and is not required to cycle power through zero (0) volts.

After calling `csx_ReleaseConfiguration()` any resources requested via the request functions [csx\\_RequestIO\(9F\)](#), [csx\\_RequestIRQ\(9F\)](#), or [csx\\_RequestWindow\(9F\)](#) that are no longer needed should be returned to Card Services via the corresponding [csx\\_ReleaseIO\(9F\)](#), [csx\\_ReleaseIRQ\(9F\)](#), or [csx\\_ReleaseWindow\(9F\)](#) functions. `csx_ReleaseConfiguration()` must be called to release the current card and socket configuration before releasing any resources requested by the driver via the request functions named above.

**Structure Members** The structure members of `release_config_t` are:

```
uint32_t    Socket;    /* socket number */
```

The `Socket` field is not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_HANDLE</code>	Client handle is invalid or <a href="#">csx_RequestConfiguration(9F)</a> not done.
	<code>CS_BAD_SOCKET</code>	Error getting or setting socket hardware parameters.
	<code>CS_NO_CARD</code>	No PC card in socket.
	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** `csx_RegisterClient(9F)`, `csx_RequestConfiguration(9F)`, `csx_RequestIO(9F)`,  
`csx_RequestIRQ(9F)`, `csx_RequestWindow(9F)`

*PC Card 95 Standard*, PCMCIA/JEIDA

**Name** csx\_RepGet8, csx\_RepGet16, csx\_RepGet32, csx\_RepGet64 – read repetitively from the device register

**Synopsis** #include <sys/pccard.h>

```
void csx_RepGet8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset,
                uint32_t recount, uint32_t flags);
```

```
void csx_RepGet16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);
```

```
void csx_RepGet32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);
```

```
void csx_RepGet64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* The access handle returned from [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#), or [csx\\_DupHandle\(9F\)](#).

*hostaddr* Source host address.

*offset* The offset in bytes from the base of the mapped resource.

*recount* Number of data accesses to perform.

*flags* Device address flags.

**Description** These functions generate multiple reads of various sizes from the mapped memory or device register.

The `csx_RepGet8()`, `csx_RepGet16()`, `csx_RepGet32()`, and `csx_RepGet64()` functions generate *recount* reads of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address represented by the handle, *handle*, at an offset in bytes represented by the offset, *offset*. The data read is stored consecutively into the buffer pointed to by the host address pointer, *hostaddr*.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the *flags* argument is set to `CS_DEV_AUTOINCR`, these functions increment the device offset, *offset*, after each datum read operation. However, when the *flags* argument is set to `CS_DEV_NO_AUTOINCR`, the same device offset will be used for every datum access. For example, this flag may be useful when reading from a data register.

**Context** These functions may be called from user, kernel, or interrupt context.

**See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_Get8\(9F\)](#), [csx\\_GetMappedAddr\(9F\)](#), [csx\\_Put8\(9F\)](#), [csx\\_RepPut8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_RepPut8, csx\_RepPut16, csx\_RepPut32, csx\_RepPut64 – write repetitively to the device register

**Synopsis** #include <sys/pccard.h>

```
void csx_RepPut8(acc_handle_t handle, uint8_t *hostaddr, uint32_t offset,
                uint32_t recount, uint32_t flags);

void csx_RepPut16(acc_handle_t handle, uint16_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);

void csx_RepPut32(acc_handle_t handle, uint32_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);

void csx_RepPut64(acc_handle_t handle, uint64_t *hostaddr, uint32_t offset,
                 uint32_t recount, uint32_t flags);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

<i>handle</i>	The access handle returned from <a href="#">csx_RequestIO(9F)</a> , <a href="#">csx_RequestWindow(9F)</a> , or <a href="#">csx_DupHandle(9F)</a> .
<i>hostaddr</i>	Source host address.
<i>offset</i>	The offset in bytes from the base of the mapped resource.
<i>recount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags.

**Description** These functions generate multiple writes of various sizes to the mapped memory or device register.

The `csx_RepPut8()`, `csx_RepPut16()`, `csx_RepPut32()`, and `csx_RepPut64()` functions generate *recount* writes of 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address represented by the handle, *handle*, at an offset in bytes represented by the offset, *offset*. The data written is read consecutively from the buffer pointed to by the host address pointer, *hostaddr*.

Data that consists of more than one byte will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte swapping if the host and the device have incompatible endian characteristics.

When the *flags* argument is set to `CS_DEV_AUTOINCR`, these functions increment the device offset, *offset*, after each datum write operation. However, when the *flags* argument is set to `CS_DEV_NO_AUTOINCR`, the same device offset will be used for every datum access. For example, this flag may be useful when writing to a data register.



**Context** These functions may be called from user, kernel, or interrupt context.

**See Also** [csx\\_DupHandle\(9F\)](#), [csx\\_Get8\(9F\)](#), [csx\\_GetMappedAddr\(9F\)](#), [csx\\_Put8\(9F\)](#), [csx\\_RepGet8\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestWindow\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_RequestConfiguration – configure the PC Card and socket

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RequestConfiguration(client_handle_t ch, config_req_t *cr);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*cr* Pointer to a config\_req\_t structure.

**Description** This function configures the PC Card and socket. It must be used by clients that require I/O or IRQ resources for their PC Card.

[csx\\_RequestIO\(9F\)](#) and [csx\\_RequestIRQ\(9F\)](#) must be used before calling this function to specify the I/O and IRQ requirements for the PC Card and socket if necessary. [csx\\_RequestConfiguration\(\)](#) establishes the configuration in the socket adapter and PC Card, and it programs the Base and Limit registers of multi-function PC Cards if these registers exist. The values programmed into these registers depend on the IO requirements of this configuration.

**Structure Members** The structure members of config\_req\_t are:

```
uint32_t    Socket;           /* socket number */
uint32_t    Attributes;      /* configuration attributes */
uint32_t    Vcc;             /* Vcc value */
uint32_t    Vpp1;           /* Vpp1 value */
uint32_t    Vpp2;           /* Vpp2 value */
uint32_t    IntType;        /* socket interface type - mem or IO */
uint32_t    ConfigBase;     /* offset from start of AM space */
uint32_t    Status;        /* value to write to STATUS register */
uint32_t    Pin;           /* value to write to PRR */
uint32_t    Copy;          /* value to write to COPY register */
uint32_t    ConfigIndex;    /* value to write to COR */
uint32_t    Present;       /* which config registers present */
uint32_t    ExtendedStatus; /* value to write to EXSTAT register */
```

The fields are defined as follows:

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Attributes** This field is bit-mapped. It indicates whether the client wishes the IRQ resources to be enabled and whether Card Services should ignore the VS bits on the socket interface. The following bits are defined:

**CONF\_ENABLE\_IRQ\_STEERING**

Enable IRQ Steering. Set to connect the PC Card IREQ line to a system interrupt previously selected by a call to [csx\\_RequestIRQ\(9F\)](#). If **CONF\_ENABLE\_IRQ\_STEERING** is set, once [csx\\_RequestConfiguration\(\)](#) has successfully returned, the client may start receiving IRQ callbacks at the IRQ callback handler established in the call to [csx\\_RequestIRQ\(9F\)](#).

**CONF\_VSOVERRIDE**

Override VS pins. After card insertion and prior to the first successful [csx\\_RequestConfiguration\(\)](#), the voltage levels applied to the card shall be those indicated by the card's physical key and/or the VS[2:1] voltage sense pins. For Low Voltage capable host systems (hosts which are capable of VS pin decoding), if a client desires to apply a voltage not indicated by the VS pin decoding, then **CONF\_VSOVERRIDE** must be set in the **Attributes** field; otherwise, **CS\_BAD\_VCC** shall be returned.

Vcc, Vpp1, Vpp2

These fields all represent voltages expressed in tenths of a volt. Values from zero (0) to 25.5 volts may be set. To be valid, the exact voltage must be available from the system. PC Cards indicate multiple Vcc voltage capability in their CIS via the **CISTPL\_CFTABLE\_ENTRY** tuple. After card insertion, Card Services processes the CIS, and when multiple Vcc voltage capability is indicated, Card Services will allow the client to apply Vcc voltage levels which are contrary to the VS pin decoding without requiring the client to set **CONF\_VSOVERRIDE**.

IntType

This field is bit-mapped. It indicates how the socket should be configured. The following bits are defined:

**SOCKET\_INTERFACE\_MEMORY**

Memory only interface.

	SOCKET_INTERFACE_MEMORY_AND_IO	Memory and I/O interface.
ConfigBase	This field is the offset in bytes from the beginning of attribute memory of the configuration registers.	
Present	This field identifies which of the configuration registers are present. If present, the corresponding bit is set. This field is bit-mapped as follows:	
	CONFIG_OPTION_REG_PRESENT	Configuration Option Register (COR) present
	CONFIG_STATUS_REG_PRESENT	Configuration Status Register (CCSR) present
	CONFIG_PINREPL_REG_PRESENT	Pin Replacement Register (PRR) present
	CONFIG_COPY_REG_PRESENT	Socket and Copy Register (SCR) present
	CONFIG_ESR_REG_PRESENT	Extended Status Register (ESR) present
Status, Pin, Copy, ExtendedStatus	These fields represent the initial values that should be written to those registers if they are present, as indicated by the Present field.	
	The Pin field is also used to inform Card Services which pins in the PC Card's PRR (Pin Replacement Register) are valid. Only those bits	

which are set are considered valid. This affects how status is returned by the `csx_GetStatus(9F)` function. If a particular signal is valid in the PRR, both the *mask* (STATUS) bit and the *change* (EVENT) bit must be set in the Pin field. The following PRR bit definitions are provided for client use:

PRR_WP_STATUS	WRITE PROTECT mask
PRR_READY_STATUS	READY mask
PRR_BVD2_STATUS	BVD2 mask
PRR_BVD1_STATUS	BVD1 mask
PRR_WP_EVENT	WRITE PROTECT changed
PRR_READY_EVENT	READY changed
PRR_BVD2_EVENT	BVD2 changed
PRR_BVD1_EVENT	BVD1 changed

ConfigIndex

This field is the value written to the COR (Configuration Option Register) for the configuration index required by the PC Card. Only the least significant six bits of the ConfigIndex field are significant; the upper two (2) bits are ignored. The interrupt type in the COR is always set to *level* mode by Card Services.

<b>Return Values</b>	CS_SUCCESS	Successful operation.
	CS_BAD_HANDLE	Client handle is invalid or <code>csx_RequestConfiguration()</code> not done.
	CS_BAD_SOCKET	Error in getting or setting socket hardware parameters.
	CS_BAD_VCC	Requested Vcc is not available on socket.
	CS_BAD_VPP	Requested Vpp is not available on socket.
	CS_NO_CARD	No PC Card in socket.
	CS_BAD_TYPE	I/O and memory interface not supported on socket.
	CS_CONFIGURATION_LOCKED	<code>csx_RequestConfiguration()</code> already done.
	CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_AccessConfigurationRegister\(9F\)](#), [csx\\_GetStatus\(9F\)](#), [csx\\_RegisterClient\(9F\)](#),  
[csx\\_ReleaseConfiguration\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestIRQ\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_RequestIO, csx\_ReleaseIO – request or release I/O resources for the client

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RequestIO(client_handle_t ch, io_req_t *ir);
```

```
int32_t csx_ReleaseIO(client_handle_t ch, io_req_t *ir);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*ir* Pointer to an `io_req_t` structure.

**Description** The functions `csx_RequestIO()` and `csx_ReleaseIO()` request or release, respectively, I/O resources for the client.

If a client requires I/O resources, `csx_RequestIO()` must be called to request I/O resources from Card Services; then [csx\\_RequestConfiguration\(9F\)](#) must be used to establish the configuration. `csx_RequestIO()` can be called multiple times until a successful set of I/O resources is found. [csx\\_RequestConfiguration\(9F\)](#) only uses the last configuration specified.

`csx_RequestIO()` fails if it has already been called without a corresponding `csx_ReleaseIO()`.

`csx_ReleaseIO()` releases previously requested I/O resources. The Card Services window resource list is adjusted by this function. Depending on the adapter hardware, the I/O window might also be disabled.

**Structure Members** The structure members of `io_req_t` are:

```
uint32_t      Socket;          /* socket number*/

uint32_t      Baseport1.base;  /* IO range base port address */
acc_handle_t  Baseport1.handle; /* IO range base address
/* or port num */

uint32_t      NumPorts1;       /* first IO range number contiguous
/* ports */

uint32_t      Attributes1;     /* first IO range attributes */

uint32_t      Baseport2.base;  /* IO range base port address */
acc_handle_t  Baseport2.handle; /* IO range base address or port num */
uint32_t      NumPorts2;       /* second IO range number contiguous
/* ports */

uint32_t      Attributes2;     /* second IO range attributes */

uint32_t      IOAddrLines;     /* number of IO address lines decoded */
```

The fields are defined as follows:

Socket	Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.												
BasePort1.base BasePort1.handle BasePort2.base BasePort2.handle	<p>Two I/O address ranges can be requested by <code>csx_RequestIO()</code>. Each I/O address range is specified by the <code>BasePort</code>, <code>NumPorts</code>, and <code>Attributes</code> fields. If only a single I/O range is being requested, the <code>NumPorts2</code> field must be reset to 0.</p> <p>When calling <code>csx_RequestIO()</code>, the <code>BasePort.base</code> field specifies the first port address requested. Upon successful return from <code>csx_RequestIO()</code>, the <code>BasePort.handle</code> field contains an access handle, corresponding to the first byte of the allocated I/O window, which the client must use when accessing the PC Card's I/O space via the common access functions. A client <i>must not</i> make any assumptions as to the format of the returned <code>BasePort.handle</code> field value.</p> <p>If the <code>BasePort.base</code> field is set to 0, Card Services returns an I/O resource based on the available I/O resources and the number of contiguous ports requested. When <code>BasePort.base</code> is 0, Card Services aligns the returned resource in the host system's I/O address space on a boundary that is a multiple of the number of contiguous ports requested, rounded up to the nearest power of two. For example, if a client requests two I/O ports, the resource returned will be a multiple of two. If a client requests five contiguous I/O ports, the resource returned will be a multiple of eight.</p> <p>If multiple ranges are being requested, at least one of the <code>BasePort.base</code> fields must be non-zero.</p>												
NumPorts	This field is the number of contiguous ports being requested.												
Attributes	<p>This field is bit-mapped. The following bits are defined:</p> <table> <tr> <td>IO_DATA_WIDTH_8</td> <td>I/O resource uses 8-bit data path.</td> </tr> <tr> <td>IO_DATA_WIDTH_16</td> <td>I/O resource uses 16-bit data path.</td> </tr> <tr> <td>WIN_ACC_NEVER_SWAP</td> <td>Host endian byte ordering.</td> </tr> <tr> <td>WIN_ACC_BIG_ENDIAN</td> <td>Big endian byte ordering</td> </tr> <tr> <td>WIN_ACC_LITTLE_ENDIAN</td> <td>Little endian byte ordering.</td> </tr> <tr> <td>WIN_ACC_STRICT_ORDER</td> <td>Program ordering references.</td> </tr> </table>	IO_DATA_WIDTH_8	I/O resource uses 8-bit data path.	IO_DATA_WIDTH_16	I/O resource uses 16-bit data path.	WIN_ACC_NEVER_SWAP	Host endian byte ordering.	WIN_ACC_BIG_ENDIAN	Big endian byte ordering	WIN_ACC_LITTLE_ENDIAN	Little endian byte ordering.	WIN_ACC_STRICT_ORDER	Program ordering references.
IO_DATA_WIDTH_8	I/O resource uses 8-bit data path.												
IO_DATA_WIDTH_16	I/O resource uses 16-bit data path.												
WIN_ACC_NEVER_SWAP	Host endian byte ordering.												
WIN_ACC_BIG_ENDIAN	Big endian byte ordering												
WIN_ACC_LITTLE_ENDIAN	Little endian byte ordering.												
WIN_ACC_STRICT_ORDER	Program ordering references.												



<code>WIN_ACC_UNORDERED_OK</code>	May re-order references.
<code>WIN_ACC_MERGING_OK</code>	Merge stores to consecutive locations.
<code>WIN_ACC_LOADCACHING_OK</code>	May cache load operations.
<code>WIN_ACC_STORECACHING_OK</code>	May cache store operations.

For some combinations of host system busses and adapter hardware, the width of an I/O resource can not be set via `RequestIO()`; on those systems, the host bus cycle access type determines the I/O resource data path width on a per-cycle basis.

`WIN_ACC_BIG_ENDIAN` and `WIN_ACC_LITTLE_ENDIAN` describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When `WIN_ACC_BIG_ENDIAN` or `WIN_ACC_LITTLE_ENDIAN` is set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

When `WIN_ACC_NEVER_SWAP` is specified, byte swapping will not be invoked in the data access functions. The ability to specify the order in which the CPU will reference data is provided by the following `Attributes` bits. Only one of the following bits may be specified:

`WIN_ACC_STRICT_ORDER`

The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

`WIN_ACC_UNORDERED_OK`

The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

`WIN_ACC_MERGING_OK`

The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. `IO_MERGING_OK_ACC` also implies re-ordering.

**WIN\_ACC\_LOADCACHING\_OK**

The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. **WIN\_ACC\_LOADCACHING\_OK** also implies merging and re-ordering.

**WIN\_ACC\_STORECACHING\_OK**

The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. **WIN\_ACC\_STORECACHING\_OK** also implies load caching, merging, and re-ordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together. All other bits in the **Attributes** field must be set to 0.

**IOAddrLines** This field is the number of I/O address lines decoded by the PC Card in the specified socket.

On some systems, multiple calls to **csx\_RequestIO()** with different **BasePort**, **NumPorts**, and/or **IOAddrLines** values will have to be made to find an acceptable combination of parameters that can be used by Card Services to allocate I/O resources for the client. (See **NOTES**).

<b>Return Values</b>		
<b>CS_SUCCESS</b>		Successful operation.
<b>CS_BAD_ATTRIBUTE</b>		Invalid <b>Attributes</b> specified.
<b>CS_BAD_BASE</b>		<b>BasePort</b> value is invalid.
<b>CS_BAD_HANDLE</b>		Client handle is invalid.
<b>CS_CONFIGURATION_LOCKED</b>		<a href="#">csx_RequestConfiguration(9F)</a> has already been done.
<b>CS_IN_USE</b>		<b>csx_RequestIO()</b> has already been done without a corresponding <b>csx_ReleaseIO()</b> .
<b>CS_NO_CARD</b>		No PC Card in socket.
<b>CS_BAD_WINDOW</b>		Unable to allocate I/O resources.
<b>CS_OUT_OF_RESOURCE</b>		Unable to allocate I/O resources.
<b>CS_UNSUPPORTED_FUNCTION</b>		No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_RegisterClient\(9F\)](#), [csx\\_RequestConfiguration\(9F\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA

**Notes** It is important for clients to try to use the minimum amount of I/O resources necessary. One way to do this is for the client to parse the CIS of the PC Card and call `csx_RequestIO()` first with any `IOAddrLines` values that are 0 or that specify a minimum number of address lines necessary to decode the I/O space on the PC Card. Also, if no convenient minimum number of address lines can be used to decode the I/O space on the PC Card, it is important to try to avoid system conflicts with well-known architectural hardware features.

**Name** csx\_RequestIRQ, csx\_ReleaseIRQ – request or release IRQ resource

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RequestIRQ(client_handle_t ch, irq_req_t *ir);
int32_t csx_ReleaseIRQ(client_handle_t ch, irq_req_t *ir);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*ir* Pointer to an `irq_req_t` structure.

**Description** The function `csx_RequestIRQ()` requests an IRQ resource and registers the client's IRQ handler with Card Services.

If a client requires an IRQ, `csx_RequestIRQ()` must be called to request an IRQ resource as well as to register the client's IRQ handler with Card Services. The client will not receive callbacks at the IRQ callback handler until [csx\\_RequestConfiguration\(9F\)](#) or [csx\\_ModifyConfiguration\(9F\)](#) has successfully returned when either of these functions are called with the `CONF_ENABLE_IRQ_STEERING` bit set.

The function `csx_ReleaseIRQ()` releases a previously requested IRQ resource.

The Card Services IRQ resource list is adjusted by `csx_ReleaseIRQ()`. Depending on the adapter hardware, the host bus IRQ connection might also be disabled. Client IRQ handlers always run above lock level and so should take care to perform only Solaris operations that are appropriate for an above-lock-level IRQ handler.

`csx_RequestIRQ()` fails if it has already been called without a corresponding `csx_ReleaseIRQ()`.

**Structure Members** The structure members of `irq_req_t` are:

```
uint32_t          Socket;          /* socket number */
uint32_t          Attributes;      /* IRQ attribute flags */
csfunction_t     *irq_handler;    /* IRQ handler */
void             *irq_handler_arg; /* IRQ handler argument */
ddi_iblock_cookie_t *iblk_cookie; /* IRQ interrupt
                                   /* block cookie */
ddi_idevice_cookie_t *idev_cookie; /* IRQ interrupt device
                                   /* cookie */
```

The fields are defined as follows:

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

Attributes	This field is bit-mapped. It specifies details about the type of IRQ desired by the client. The following bits are defined:  IRQ_TYPE_EXCLUSIVE    IRQ is exclusive to this socket. This bit must be set. It indicates that the system IRQ is dedicated to this PC Card.
irq_handler	The client IRQ callback handler entry point is passed in the irq_handler field.
irq_handler_arg	The client can use the irq_handler_arg field to pass client-specific data to the client IRQ callback handler.
iblk_cookie idev_cookie	These fields must be used by the client to set up mutexes that are used in the client's IRQ callback handler.

For a specific `csx_ReleaseIRQ()` call, the values in the `irq_req_t` structure must be the same as those returned from the previous `csx_RequestIRQ()` call; otherwise, `CS_BAD_ARGS` is returned and no changes are made to Card Services resources or the socket and adapter hardware.

<b>Return Values</b>	<code>CS_SUCCESS</code>	Successful operation.
	<code>CS_BAD_ARGS</code>	IRQ description does not match allocation.
	<code>CS_BAD_ATTRIBUTE</code>	<code>IRQ_TYPE_EXCLUSIVE</code> not set, or an unsupported or reserved bit is set.
	<code>CS_BAD_HANDLE</code>	Client handle is invalid or <a href="#">csx_RequestConfiguration(9F)</a> not done.
	<code>CS_BAD_IRQ</code>	Unable to allocate IRQ resources.
	<code>CS_IN_USE</code>	<code>csx_RequestIRQ()</code> already done or a previous <code>csx_RequestIRQ()</code> has not been done for a corresponding <code>csx_ReleaseIRQ()</code> .
	<code>CS_CONFIGURATION_LOCKED</code>	<a href="#">csx_RequestConfiguration(9F)</a> already done or <a href="#">csx_ReleaseConfiguration(9F)</a> has not been done.
	<code>CS_NO_CARD</code>	No PC Card in socket.
	<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_ReleaseConfiguration\(9F\)](#), [csx\\_RequestConfiguration\(9F\)](#)

*PC Card Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_RequestSocketMask, csx\_ReleaseSocketMask – set or clear the client's client event mask

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RequestSocketMask(client_handle_t ch,
                             request_socket_mask_t *sm);

int32_t csx_ReleaseSocketMask(client_handle_t ch,
                              release_socket_mask_t *rm);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters**

- ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).
- sm* Pointer to a `request_socket_mask_t` structure.
- rm* Pointer to a `release_socket_mask_t` structure.

**Description** The function `csx_RequestSocketMask()` sets the client's client event mask and enables the client to start receiving events at its event callback handler. Once this function returns successfully, the client can start receiving events at its event callback handler. Any pending events generated from the call to [csx\\_RegisterClient\(9F\)](#) will be delivered to the client after this call as well. This allows the client to set up the event handler mutexes before the event handler gets called.

`csx_RequestSocketMask()` must be used before calling [csx\\_GetEventMask\(9F\)](#) or [csx\\_SetEventMask\(9F\)](#) for the client event mask for this socket.

The function `csx_ReleaseSocketMask()` clears the client's client event mask.

**Structure Members** The structure members of `request_socket_mask_t` are:

```
uint32_t Socket;      /* socket number */
uint32_t EventMask;  /* event mask to set or return */
```

The structure members of `release_socket_mask_t` are:

```
uint32_t Socket;      /* socket number */
```

The fields are defined as follows:

Socket	Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
EventMask	This field is bit-mapped. Card Services performs event notification based on this field. See <a href="#">csx_event_handler(9E)</a> for valid event definitions and for additional information about handling events.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.

CS_IN_USE	csx_ReleaseSocketMask() has not been done.
CS_BAD_SOCKET	csx_RequestSocketMask() has not been done.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_event\\_handler\(9E\)](#), [csx\\_GetEventMask\(9F\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_SetEventMask\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_RequestWindow, csx\_ReleaseWindow – request or release window resources

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_RequestWindow(client_handle_t ch, window_handle_t *wh,
                          win_req_t *wr);

int32_t csx_ReleaseWindow(window_handle_t wh);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*wh* Pointer to a `window_handle_t` structure.

*wr* Pointer to a `win_req_t` structure.

**Description** The function `csx_RequestWindow()` requests a block of system address space be assigned to a PC Card in a socket.

The function `csx_ReleaseWindow()` releases window resources which were obtained by a call to `csx_RequestWindow()`. No adapter or socket hardware is modified by this function.

The [csx\\_MapMemPage\(9F\)](#) and [csx\\_ModifyWindow\(9F\)](#) functions use the window handle returned by `csx_RequestWindow()`. This window handle must be freed by calling `csx_ReleaseWindow()` when the client is done using this window.

The PC Card Attribute or Common Memory offset for this window is set by [csx\\_MapMemPage\(9F\)](#).

**Structure Members** The structure members of `win_req_t` are:

```
uint32_t      Socket;                /* socket number */
uint32_t      Attributes;           /* window flags */
uint32_t      Base.base;           /* requested window */
                                     /* base address */
acc_handle_t  Base.handle;         /* returned handle for
                                     /* base of window */

uint32_t      Size;                /* window size requested */
                                     /* or granted */

uint32_t      win_params.AccessSpeed; /* window access speed */
uint32_t      win_params.IOAddrLines; /* IO address lines decoded */
uint32_t      ReqOffset;           /* required window offset */
```

The fields are defined as follows:

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Attributes** This field is bit-mapped and is defined as follows:



`WIN_MEMORY_TYPE_IO`

Points to I/O space.

`WIN_MEMORY_TYPE_CM`

Points to common memory space.

`WIN_MEMORY_TYPE_AM`

These bits select which type of window is being requested. One of these bits must be set.

`WIN_ENABLE`

The client must set this bit to enable the window.

`WIN_ACC_BIG_ENDIAN`

Describes device as big-endian.

`WIN_ACC_LITTLE_ENDIAN`

These bits describe the endian characteristics of the device as big endian or little endian, respectively. Even though most of the devices will have the same endian characteristics as their busses, there are examples of devices with an I/O processor that has opposite endian characteristics of the busses. When either of these bits are set, byte swapping will automatically be performed by the system if the host machine and the device data formats have opposite endian characteristics. The implementation may take advantage of hardware platform byte swapping capabilities.

`WIN_ACC_NEVER_SWAP`

When this is specified, byte swapping will not be invoked in the data access functions.

The ability to specify the order in which the CPU will reference data is provided by the following `Attributes` bits, only one of which may be specified:

`WIN_ACC_STRICT_ORDER`

The data references must be issued by a CPU in program order. Strict ordering is the default behavior.

`WIN_ACC_UNORDERED_OK`

The CPU may re-order the data references. This includes all kinds of re-ordering (that is, a load followed by a store may be replaced by a store followed by a load).

`WIN_ACC_MERGING_OK`

The CPU may merge individual stores to consecutive locations. For example, the CPU may turn two consecutive

byte stores into one halfword store. It may also batch individual loads. For example, the CPU may turn two consecutive byte loads into one halfword load. This bit also implies re-ordering.

#### WIN\_ACC\_LOADCACHING\_OK

The CPU may cache the data it fetches and reuse it until another store occurs. The default behavior is to fetch new data on every load. This bit also implies merging and re-ordering.

#### WIN\_ACC\_STORECACHING\_OK

The CPU may keep the data in the cache and push it to the device (perhaps with other data) at a later time. The default behavior is to push the data right away. This bit also implies load caching, merging, and re-ordering.

These values are advisory, not mandatory. For example, data can be ordered without being merged or cached, even though a driver requests unordered, merged and cached together.

All other bits in the `Attributes` field must be set to 0.

On successful return from `csx_RequestWindow()`, `WIN_OFFSET_SIZE` is set in the `Attributes` field when the client must specify card offsets to [csx\\_MapMemPage\(9F\)](#) that are a multiple of the window size.

<code>Base.base</code>	This field must be set to 0 on calling <code>csx_RequestWindow()</code> .
<code>Base.handle</code>	On successful return from <code>csx_RequestWindow()</code> , the <code>Base.handle</code> field contains an access handle corresponding to the first byte of the allocated memory window which the client must use when accessing the PC Card's memory space via the common access functions. A client must <i>not</i> make any assumptions as to the format of the returned <code>Base.handle</code> field value.
<code>Size</code>	On calling <code>csx_RequestWindow()</code> , the <code>Size</code> field is the size in bytes of the memory window requested. <code>Size</code> may be zero to indicate that Card Services should provide the smallest sized window available. On successful return from <code>csx_RequestWindow()</code> , the <code>Size</code> field contains the actual size of the window allocated.
<code>win_params.AccessSpeed</code>	This field specifies the access speed of the window if the client is requesting a memory window. The <code>AccessSpeed</code> field bit definitions use the format of the extended speed byte of the

Device ID tuple. If the mantissa is 0 (noted as reserved in the *PC Card 95 Standard*), the lower bits are a binary code representing a speed from the following table:

Code	Speed
0	(Reserved - do not use).
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5-7	(Reserved—do not use.)

To request a window that supports the WAIT signal, OR- in the WIN\_USE\_WAIT bit to the AccessSpeed value before calling this function.

It is recommended that clients use the [csx\\_ConvertSpeed\(9F\)](#) function to generate the appropriate AccessSpeed values rather than manually perturbing the AccessSpeed field.

`win_params.IOAddrLines` If the client is requesting an I/O window, the `IOAddrLines` field is the number of I/O address lines decoded by the PC Card in the specified socket. Access to the I/O window is not enabled until [csx\\_RequestConfiguration\(9F\)](#) has been invoked successfully.

`ReqOffset` This field is a Solaris-specific extension that can be used by clients to generate optimum window offsets passed to [csx\\_MapMemPage\(9F\)](#).

Return Values	
<code>CS_SUCCESS</code>	Successful operation.
<code>CS_BAD_ATTRIBUTE</code>	Attributes are invalid.
<code>CS_BAD_SPEED</code>	Speed is invalid.
<code>CS_BAD_HANDLE</code>	Client handle is invalid.
<code>CS_BAD_SIZE</code>	Window size is invalid.
<code>CS_NO_CARD</code>	No PC Card in socket.
<code>CS_OUT_OF_RESOURCE</code>	Unable to allocate window.
<code>CS_UNSUPPORTED_FUNCTION</code>	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_ConvertSpeed\(9F\)](#), [csx\\_MapMemPage\(9F\)](#), [csx\\_ModifyWindow\(9F\)](#),  
[csx\\_RegisterClient\(9F\)](#), [csx\\_RequestConfiguration\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

- 
- Name** csx\_ResetFunction – reset a function on a PC card
- Synopsis** #include <sys/pccard.h>
- ```
int32_t csx_ResetFunction(client_handle_t ch, reset_function_t *rf);
```
- Interface Level** Solaris DDI Specific (Solaris DDI)
- Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).  
*rf* Pointer to a `reset_function_t` structure.
- Description** `csx_ResetFunction()` requests that the specified function on the PC card initiate a reset operation.
- Structure Members** The structure members of `reset_function_t` are:
- ```
uint32_t Socket;          /* socket number */
uint32_t Attributes;     /* reset attributes */
```
- The fields are defined as follows:
- `Socket` Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.
- `Attributes` Must be 0.
- Return Values**
- |                                      |   |
|--------------------------------------|---|
| <code>CS_SUCCESS</code>              | Card Services has noted the reset request.                                      |
| <code>CS_IN_USE</code>               | This Card Services implementation does not permit configured cards to be reset. |
| <code>CS_BAD_HANDLE</code>           | Client handle is invalid.   |
| <code>CS_NO_CARD</code>              | No PC card in socket.   |
| <code>CS_BAD_SOCKET</code>           | Specified socket or function number is invalid.                                 |
| <code>CS_UNSUPPORTED_FUNCTION</code> | No PCMCIA hardware installed.   |
- Context** This function may be called from user or kernel context.
- See Also** [csx\\_event\\_handler\(9E\)](#), [csx\\_RegisterClient\(9F\)](#)  
*PC Card 95 Standard*, PCMCIA/JEIDA
- Notes** `csx_ResetFunction()` has not been implemented in this release and always returns `CS_IN_USE`.

**Name** csx\_SetEventMask, csx\_GetEventMask – set or return the client event mask for the client

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_SetEventMask(client_handle_t ch, sockevent_t *se);
```

```
int32_t csx_GetEventMask(client_handle_t ch, sockevent_t *se);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*se* Pointer to a sockevent\_t structure

**Description** The function csx\_SetEventMask() sets the client or global event mask for the client.

The function csx\_GetEventMask() returns the client or global event mask for the client.

[csx\\_RequestSocketMask\(9F\)](#) must be called before calling csx\_SetEventMask() for the client event mask for this socket.

**Structure Members** The structure members of sockevent\_t are:

```
uint32_t  uint32_t      /* attribute flags for call */
uint32_t  EventMask;   /* event mask to set or return */
uint32_t  Socket;      /* socket number if necessary */
```

The fields are defined as follows:

**Attributes** This is a bit-mapped field that identifies the type of event mask to be returned. The field is defined as follows:

CONF\_EVENT\_MASK\_GLOBAL Client's global event mask. If set, the client's global event mask is returned.

CONF\_EVENT\_MASK\_CLIENT Client's local event mask. If set, the client's local event mask is returned.

**EventMask** This field is bit-mapped. Card Services performs event notification based on this field. See [csx\\_event\\_handler\(9E\)](#) for valid event definitions and for additional information about handling events.

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Return Values**

CS_SUCCESS	Successful operation.
CS_BAD_HANDLE	Client handle is invalid.
CS_BAD_SOCKET	<a href="#">csx_RequestSocketMask(9F)</a> not called for CONF_EVENT_MASK_CLIENT.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** These functions may be called from user or kernel context.

**See Also** [csx\\_event\\_handler\(9E\)](#), [csx\\_RegisterClient\(9F\)](#), [csx\\_ReleaseSocketMask\(9F\)](#),  
[csx\\_RequestSocketMask\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** csx\_SetHandleOffset – set current access handle offset

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_SetHandleOffset(acc_handle_t handle, uint32_t offset);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *handle* Access handle returned by [csx\\_RequestIRQ\(9F\)](#) or [csx\\_RequestIO\(9F\)](#).  
*offset* New access handle offset.

**Description** This function sets the current offset for the access handle, *handle*, to *offset*.

**Return Values** CS\_SUCCESS Successful operation.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetHandleOffset\(9F\)](#), [csx\\_RequestIO\(9F\)](#), [csx\\_RequestIRQ\(9F\)](#)

*PC Card 95 Standard*, PCMCIA/JEIDA



**Name** csx\_ValidateCIS – validate the Card Information Structure (CIS)

**Synopsis** #include <sys/pccard.h>

```
int32_t csx_ValidateCIS(client_handle_t ch, cisinfo_t *ci);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *ch* Client handle returned from [csx\\_RegisterClient\(9F\)](#).

*ci* Pointer to a `cisinfo_t` structure.

**Description** This function validates the Card Information Structure (CIS) on the PC Card in the specified socket.

**Structure Members** The structure members of `cisinfo_t` are:

```
uint32_t Socket; /* socket number to validate CIS on */
uint32_t Chains; /* number of tuple chains in CIS */
uint32_t Tuples; /* total number of tuples in CIS */
```

The fields are defined as follows:

**Socket** Not used in Solaris, but for portability with other Card Services implementations, it should be set to the logical socket number.

**Chains** This field returns the number of valid tuple chains located in the CIS. If 0 is returned, the CIS is not valid.

**Tuples** This field is a Solaris-specific extension and it returns the total number of tuples on all the chains in the PC Card's CIS.

**Return Values**

CS_SUCCESS	Successful operation.
CS_NO_CIS	No CIS on PC Card or CIS is invalid.
CS_NO_CARD	No PC Card in socket.
CS_UNSUPPORTED_FUNCTION	No PCMCIA hardware installed.

**Context** This function may be called from user or kernel context.

**See Also** [csx\\_GetFirstTuple\(9F\)](#), [csx\\_GetTupleData\(9F\)](#), [csx\\_ParseTuple\(9F\)](#), [csx\\_RegisterClient\(9F\)](#)

*PC Card 95 Standard, PCMCIA/JEIDA*

**Name** datamsg – test whether a message is a data message

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

int datamsg(unsigned char type);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *type* The type of message to be tested. The `db_type` field of the [datab\(9S\)](#) structure contains the message type. This field may be accessed through the message block using `mp->b_datap->db_type`.

**Description** The `datamsg()` function tests the type of message to determine if it is a data message type (`M_DATA`, `M_DELAY`, `M_PROTO`, or `M_PCPROTO`).

**Return Values** `datamsg` returns

- 1 if the message is a data message
- 0 otherwise.

**Context** The `datamsg()` function can be called from user, interrupt, or kernel context.

**Examples** The [put\(9E\)](#) routine enqueues all data messages for handling by the [srv\(9E\)](#) (service) routine. All non-data messages are handled in the [put\(9E\)](#) routine.

```
1 xxxput(q, mp)
2     queue_t *q;
3     mblk_t *mp;
4 {
5     if (datamsg(mp->b_datap->db_type)) {
6         putq(q, mp);
7         return;
8     }
9     switch (mp->b_datap->db_type) {
10     case M_FLUSH:
11         ...
12 }
```

**See Also** [put\(9E\)](#), [srv\(9E\)](#), [allocb\(9F\)](#), [datab\(9S\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** DB\_BASE, DB\_LIM, DB\_REF, DB\_TYPE – Data block access macros

**Synopsis** `#include <sys/stream.h>`  
`#include <sys/strsun.h>`

```
uchar_t *DB_BASE(mblk_t *mp);
uchar_t *DB_LIM(mblk_t *mp);
uchar_t DB_TYPE(mblk_t *mp);
uchar_t DB_REF(mblk_t *mp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* Message block to be accessed.

**Description** These macros provide compact access to public members of the [datab\(9S\)](#) structure associated with the specified message block.

In all cases, these macros are equivalent to directly accessing the underlying fields of the [datab\(9S\)](#) associated with the specified message block. Specifically:

DB\_BASE(*mp*) is equivalent to `mp->b_datap->db_base`.

DB\_LIM(*mp*) is equivalent to `mp->b_datap->db_lim`.

DB\_TYPE(*mp*) is equivalent to `mp->b_datap->db_type`.

DB\_REF(*mp*) is equivalent to `mp->b_datap->db_ref`.

**Context** These functions can be called from user, kernel or interrupt context.

**See Also** [msgb\(9S\)](#), [datab\(9S\)](#)

*STREAMS Programming Guide*

**Name** ddi\_add\_event\_handler – add an NDI event service callback handler

**Synopsis** #include <sys/dditypes.h>  
#include <sys/sunddi.h>

```
int ddi_add_event_handler(dev_info_t *dip, ddi_eventcookie_t cookie,  
    void (*handler)(dev_info_t *, ddi_eventcookie_t, void *, void *),  
    void *arg, ddi_registration_id_t *id);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** dev\_info\_t \*dip  
Device node registering the callback.

ddi\_eventcookie\_t cookie  
Cookie returned from call to [ddi\\_get\\_eventcookie\(9F\)](#).

void (\*handler)(dev\_info\_t \*, ddi\_eventcookie\_t, void \*, void \*)  
Callback handler responsible for handling an NDI event service notification.

void \*arg  
Pointer to opaque data supplied by the caller. Typically, this would be a pointer to the driver's soft state structure.

ddi\_registration\_id\_t \*id  
Pointer to registration ID where a unique registration id will be returned. Registration ID must be saved and used when calling [ddi\\_remove\\_event\\_handler\(9F\)](#) to unregister a callback.

**Description** The `ddi_add_event_handler()` function adds a callback handler to be invoked in the face of the event specified by `cookie`. The process of adding a callback handler is also known as subscribing to an event. Upon successful subscription, the handler will be invoked by the system when the event occurs. The handler can be unregistered by using [ddi\\_remove\\_event\\_handler\(9F\)](#).

An instance of a driver can register multiple handlers for an event or a single handler for multiple events. Callback order is not defined and should assumed to be random.

The routine handler will be invoked with the following arguments:

dev_info_t *dip	Device node requesting the notification.
ddi_eventcookie_t cookie	Structure describing event that occurred.
void *arg	Opaque data pointer provided, by the driver, during callback registration.
void *impl_data	Pointer to event specific data defined by the framework which invokes the callback function.

**Return Values** `DDI_SUCCESS` Callback handler registered successfully.

`DDI_FAILURE` Failed to register callback handler. Possible reasons include lack of resources or a bad cookie.

**Context** The `ddi_add_event_handler()` and `handler()` function can be called from user and kernel contexts only.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Committed

**See Also** [attributes\(5\)](#), [ddi\\_get\\_eventcookie\(9F\)](#), [ddi\\_remove\\_event\\_handler\(9F\)](#)

*Writing Device Drivers*

**Notes** Drivers must remove all registered callback handlers for a device instance by calling [ddi\\_remove\\_event\\_handler\(9F\)](#) before detach completes.

**Name** ddi\_add\_intr, ddi\_get\_iblock\_cookie, ddi\_remove\_intr – hardware interrupt handling routines

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_iblock_cookie(dev_info_t *dip, uint_t inumber,
    ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_intr(dev_info_t *dip, uint_t inumber,
    ddi_iblock_cookie_t *iblock_cookiep,
    ddi_idevice_cookie_t *idevice_cookiep,
    uint_t (*int_handler) (caddr_t),
    caddr_t int_handler_arg);

void ddi_remove_intr(dev_info_t *dip,
    uint_t inumber, ddi_iblock_cookie_t iblock_cookie);
```

**Interface Level** Solaris DDI specific (Solaris DDI). These interfaces are obsolete. Use the new interrupt interfaces referenced in [Intro\(9F\)](#). Refer to [Writing Device Drivers](#) for more information.

**Parameters** For ddi\_get\_iblock\_cookie():

*dip* Pointer to dev\_info structure.  
*inumber* Interrupt number.  
*iblock\_cookiep* Pointer to an interrupt block cookie.

For ddi\_add\_intr():

*dip* Pointer to dev\_info structure.  
*inumber* Interrupt number.  
*iblock\_cookiep* Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.  
*idevice\_cookiep* Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored.  
*int\_handler* Pointer to interrupt handler.  
*int\_handler\_arg* Argument for interrupt handler.

For ddi\_remove\_intr():

*dip* Pointer to dev\_info structure.  
*inumber* Interrupt number.  
*iblock\_cookie* Block cookie which identifies the interrupt handler to be removed.

## Description

`ddi_get_iblock_cookie()` `ddi_get_iblock_cookie()` retrieves the interrupt block cookie associated with a particular interrupt specification. This routine should be called before `ddi_add_intr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the interrupt routine. The interrupt number *inumber* determines for which interrupt specification to retrieve the cookie. *inumber* is associated with information provided either by the device (see `sbus(4)`) or the hardware configuration file (see `sysbus(4)`, `isa(4)`, and `driver.conf(4)`). If only one interrupt is associated with the device, *inumber* should be 0.

On a successful return, *\*iblock\_cookiep* contains information needed for initializing locks associated with the interrupt specification corresponding to *inumber* (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize locks acquired by the interrupt routine before calling `ddi_add_intr()` which prevents a possible race condition where the driver's interrupt handler is called immediately *after* the driver has called `ddi_add_intr()` but *before* the driver has initialized the locks. This may happen when an interrupt for a different device occurs on the same interrupt level. If the interrupt routine acquires the lock before the lock has been initialized, undefined behavior may result.

`ddi_add_intr()` `ddi_add_intr()` adds an interrupt handler to the system. The interrupt number *inumber* determines which interrupt the handler will be associated with. (Refer to `ddi_get_iblock_cookie()` above.)

On a successful return, *iblock\_cookiep* contains information used for initializing locks associated with this interrupt specification (see `mutex_init(9F)` and `rw_init(9F)`). Note that the interrupt block cookie is usually obtained using `ddi_get_iblock_cookie()` to avoid the race conditions described above (refer to `ddi_get_iblock_cookie()` above). For this reason, *iblock\_cookiep* is no longer useful and should be set to NULL.

On a successful return, *idevice\_cookiep* contains a pointer to a `ddi_idevice_cookie_t` structure (see `ddi_idevice_cookie(9S)`) containing information useful for some devices that have programmable interrupts. If *idevice\_cookiep* is set to NULL, no value is returned.

The routine *intr\_handler*, with its argument *int\_handler\_arg*, is called upon receipt of the appropriate interrupt. The interrupt handler should return `DDI_INTR_CLAIMED` if the interrupt was claimed, `DDI_INTR_UNCLAIMED` otherwise.

If successful, `ddi_add_intr()` returns `DDI_SUCCESS`. If the interrupt information cannot be found on the sun4u architecture, either `DDI_INTR_NOTFOUND` or `DDI_FAILURE` can be returned. On i86pc and sun4m architectures, if the interrupt information cannot be found, `DDI_INTR_NOTFOUND` is returned.

`ddi_remove_intr()` `ddi_remove_intr()` removes an interrupt handler from the system. Unloadable drivers should call this routine during their `detach(9E)` routine to remove their interrupt handler from the system.

The device interrupt routine for this instance of the device will not execute after `ddi_remove_intr()` returns. `ddi_remove_intr()` may need to wait for the device interrupt routine to complete before returning. Therefore, locks acquired by the interrupt handler should not be held across the call to `ddi_remove_intr()` or deadlock may result.

**For All Three Functions:** For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA and SBus buses. See [sysbus\(4\)](#), [isa\(4\)](#), and [sbus\(4\)](#) for details.

**Return Values** `ddi_add_intr()` and `ddi_get_iblock_cookie()` return:

<code>DDI_SUCCESS</code>	On success.
<code>DDI_INTR_NOTFOUND</code>	On failure to find the interrupt.
<code>DDI_FAILURE</code>	On failure. <code>DDI_FAILURE</code> can also be returned on failure to find interrupt ( <code>sun4u</code> ).

**Context** `ddi_add_intr()`, `ddi_remove_intr()`, and `ddi_get_iblock_cookie()` can be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [driver.conf\(4\)](#), [isa\(4\)](#), [sbus\(4\)](#), [sysbus\(4\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [ddi\\_intr\\_hilevel\(9F\)](#), [Intro\(9F\)](#), [mutex\(9F\)](#), [mutex\\_init\(9F\)](#), [rw\\_init\(9F\)](#), [rwlock\(9F\)](#), [ddi\\_idevice\\_cookie\(9S\)](#)

### *Writing Device Drivers*

**Notes** `ddi_get_iblock_cookie()` must not be called *after* the driver adds an interrupt handler for the interrupt specification corresponding to *inumber*.

All consumers of these interfaces, checking return codes, should verify `return_code != DDI_SUCCESS`. Checking for specific failure codes can result in inconsistent behaviors among platforms.

**Bugs** The *idevice\_cookiep* should really point to a data structure that is specific to the bus architecture that the device operates on. Currently the SBus and PCI buses are supported and a single data structure is used to describe both.



**Name** ddi\_add\_softintr, ddi\_get\_soft\_iblock\_cookie, ddi\_remove\_softintr, ddi\_trigger\_softintr – software interrupt handling routines

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t *dip,
int preference, ddi_iblock_cookie_t *iblock_cookiep);

int ddi_add_softintr(dev_info_t *dip, int preference, ddi_softintr_t *idp,
ddi_iblock_cookie_t *iblock_cookiep, ddi_idevice_cookie_t *
idevice_cookiep,
uint_t(*int_handler) (caddr_t int_handler_arg), caddr_t
int_handler_arg);

void ddi_remove_softintr(ddi_softintr_t id);

void ddi_trigger_softintr(ddi_softintr_t id);
```

**Interface Level** Solaris DDI specific (Solaris DDI). These interfaces are obsolete. Use the new interrupt interfaces referenced in [Intro\(9F\)](#). Refer to *Writing Device Drivers* for more information.

**Parameters**

**ddi\_get\_soft\_iblock\_cookie()**

*dip* Pointer to a dev\_info structure.

*preference* The type of soft interrupt to retrieve the cookie for.

*iblock\_cookiep* Pointer to a location to store the interrupt block cookie.

**ddi\_add\_softintr()**

*dip* Pointer to dev\_info structure.

*preference* A hint value describing the type of soft interrupt to generate.

*idp* Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.

*iblock\_cookiep* Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.

*idevice\_cookiep* Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).

*int\_handler* Pointer to interrupt handler.

*int\_handler\_arg* Argument for interrupt handler.

**ddi\_remove\_softintr()**

*id* The identifier specifying which soft interrupt handler to remove.

`ddi_trigger_softintr()`

*id* The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.

**Description** For `ddi_get_soft_iblock_cookie()`:

`ddi_get_soft_iblock_cookie()` retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before `ddi_add_softintr()` to retrieve the interrupt block cookie needed to initialize locks (`mutex(9F)`, `rwlock(9F)`) used by the software interrupt routine. *preference* determines which type of soft interrupt to retrieve the cookie for. The possible values for *preference* are:

DDI\_SOFTINT\_LOW      Low priority soft interrupt.

DDI\_SOFTINT\_MED      Medium priority soft interrupt.

DDI\_SOFTINT\_HIGH     High priority soft interrupt.

On a successful return, *iblock\_cookiep* contains information needed for initializing locks associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). The driver can then initialize mutexes acquired by the interrupt routine before calling `ddi_add_softintr()` which prevents a possible race condition where the driver's soft interrupt handler is called immediately *after* the driver has called `ddi_add_softintr()` but *before* the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

For `ddi_add_softintr()`:

`ddi_add_softintr()` adds a soft interrupt to the system. The user specified hint *preference* identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for *preference* should be the same as that used in the corresponding call to `ddi_get_soft_iblock_cookie()`. Refer to the description of `ddi_get_soft_iblock_cookie()` above.

The value returned in the location pointed at by *idp* is the soft interrupt identifier. This value is used in later calls to `ddi_remove_softintr()` and `ddi_trigger_softintr()` to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by *iblock\_cookiep* is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see `mutex_init(9F)` and `rw_init(9F)`). Note that the interrupt block cookie is normally obtained using `ddi_get_soft_iblock_cookie()` to avoid the race conditions described above (refer to the description of `ddi_get_soft_iblock_cookie()` above). For this reason, *iblock\_cookiep* is no longer useful and should be set to NULL.

*idevice\_cookiep* is not used and should be set to NULL.

The routine *int\_handler*, with its argument *int\_handler\_arg*, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine *int\_handler* checks this flag, reachable through *int\_handler\_arg*, to determine if it should claim the interrupt and do its work.

The interrupt handler must return DDI\_INTR\_CLAIMED if the interrupt was claimed, DDI\_INTR\_UNCLAIMED otherwise.

If successful, `ddi_add_softintr()` will return DDI\_SUCCESS; if the interrupt information cannot be found, it will return DDI\_FAILURE.

For `ddi_remove_softintr()`:

`ddi_remove_softintr()` removes a soft interrupt from the system. The soft interrupt identifier *id*, which was returned from a call to `ddi_add_softintr()`, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

For `ddi_trigger_softintr()`:

`ddi_trigger_softintr()` triggers a soft interrupt. The soft interrupt identifier *id* is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using `ddi_add_softintr()`.

**Return Values** `ddi_add_softintr()` and `ddi_get_soft_iblock_cookie()` return:

DDI\_SUCCESS     on success  
DDI\_FAILURE     on failure

**Context** These functions can be called from user or kernel context. `ddi_trigger_softintr()` may be called from high-level interrupt context as well.

**Examples** EXAMPLE 1 device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See [ddi\\_intr\\_hilevel\(9F\)](#).

**EXAMPLE 1** device using high-level interrupts *(Continued)*

In the example, the high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt handler. If the soft interrupt handler is not currently running, the high-level interrupt routine triggers a soft interrupt so the soft interrupt handler can process the data. Once running, the soft interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data shared between the high-level interrupt handler and the soft interrupt handler. The low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
    . . .
    ddi_softintr_t      id;
    ddi_iblock_cookie_t high_iblock_cookie;
    kmutex_t           high_mutex;
    ddi_iblock_cookie_t low_iblock_cookie;
    kmutex_t           low_mutex;
    int                 softint_running;
    . . .
};
struct xxstate *xsp;
static uint_t xxsoftintr(caddr_t);
static uint_t xxhighintr(caddr_t);
. . .
```

**EXAMPLE 2** sample attach() routine

The following code fragment would usually appear in the driver's [attach\(9E\)](#) routine. [ddi\\_add\\_intr\(9F\)](#) is used to add the high-level interrupt handler and [ddi\\_add\\_softintr\(\)](#) is used to add the low-level interrupt routine.

```
static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    . . .
    /* get high-level iblock cookie */
    if (ddi_get_iblock_cookie(dip, inumber,
        &xsp->high_iblock_cookie) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize high-level mutex */
    mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
        (void *)xsp->high_iblock_cookie);
```

**EXAMPLE 2** sample attach() routine (Continued)

```

/* add high-level routine - xxhighintr() */
if (ddi_add_intr(dip, inumber, NULL, NULL,
    xxhighintr, (caddr_t) xsp) != DDI_SUCCESS) {
    /* cleanup */
    return (DDI_FAILURE); /* fail attach */
}

/* get soft iblock cookie */
if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
    &xsp->low_iblock_cookie) != DDI_SUCCESS) {
    /* clean up */
    return (DDI_FAILURE); /* fail attach */
}

/* initialize low-level mutex */
mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
    (void *)xsp->low_iblock_cookie);

/* add low level routine - xxsoftintr() */
if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
    NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
    /* cleanup */
    return (DDI_FAILURE); /* fail attach */
}

. . .
}

```

**EXAMPLE 3** High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, `ddi_trigger_softintr()` is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```

static uint_t
xxhighintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    int need_softint;
    . . .
    mutex_enter(&xsp->high_mutex);
    /*
     * Verify this device generated the interrupt

```

**EXAMPLE 3** High-level interrupt routine *(Continued)*

```

    * and disable the device interrupt.
    * Enqueue data for xxsoftintr() processing.
    */

    /* is xxsoftintr() already running ? */
    if (xsp->softint_running)
        need_softint = 0;
    else
        need_softint = 1;
    mutex_exit(&xsp->high_mutex);

    /* read-only access to xsp->id, no mutex needed */
    if (need_softint)
        ddi_trigger_softintr(xsp->id);
    . . .
    return (DDI_INTR_CLAIMED);
}

static uint_t
xxsoftintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *) arg;
    . . .
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);

    /* verify there is work to do */
    if (work_queue_empty || xsp->softint_running) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }

    xsp->softint_running = 1;

    while ( data_on_queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));

        /* de-queue data */

        mutex_exit(&xsp->high_mutex);

        /* Process data on queue */

        mutex_enter(&xsp->high_mutex);

```

**EXAMPLE 3** High-level interrupt routine *(Continued)*

```

    }

    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);

    return (DDI_INTR_CLAIMED);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [ddi\\_add\\_intr\(9F\)](#), [ddi\\_in\\_panic\(9F\)](#), [ddi\\_intr\\_hilevel\(9F\)](#), [ddi\\_remove\\_intr\(9F\)](#), [Intro\(9F\)](#), [mutex\\_init\(9F\)](#)

*Writing Device Drivers*

**Notes** `ddi_add_softintr()` may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for `int_handler_arg` in each of the calls to `ddi_add_softintr()`. Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For example, the argument could be a pointer to the device's soft state structure, which could contain a 'which\_service' field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_trigger_softintr()`.

**Name** ddi\_binding\_name, ddi\_get\_name – return driver binding name

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
char *ddi_binding_name(dev_info_t *dip);  
char *ddi_get_name(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* A pointer to the device's dev\_info structure.

**Description** ddi\_binding\_name() and ddi\_get\_name() return the driver binding name. This is the name used to select a driver for the device. This name is typically derived from the device name property or the device compatible property. The name returned may be a driver alias or the driver name.

**Return Values** ddi\_binding\_name() and ddi\_get\_name() return the name used to bind a driver to a device.

**Context** ddi\_binding\_name() and ddi\_get\_name() can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_node\\_name\(9F\)](#)

*Writing Device Drivers*

**Warnings** The name returned by ddi\_binding\_name() and ddi\_get\_name() is read-only.



- 
- Name** ddi\_btop, ddi\_btopr, ddi\_ptob – page size conversions
- Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned long ddi_btop(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_btopr(dev_info_t *dip, unsigned long bytes);
unsigned long ddi_ptob(dev_info_t *dip, unsigned long pages);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Description** This set of routines use the parent nexus driver to perform conversions in page size units.
- The `ddi_btop()` function converts the given number of bytes to the number of memory pages that it corresponds to, rounding down in the case that the byte count is not a page multiple.
- The `ddi_btopr()` function converts the given number of bytes to the number of memory pages that it corresponds to, rounding up in the case that the byte count is not a page multiple.
- The `ddi_ptob()` function converts the given number of pages to the number of bytes that it corresponds to.
- Because bus nexus may possess their own hardware address translation facilities, these routines should be used in preference to the corresponding DDI/DKI routines [btop\(9F\)](#), [btopr\(9F\)](#), and [ptob\(9F\)](#), which only deal in terms of the pagesize of the main system MMU.
- Return Values** The `ddi_btop()` and `ddi_btopr()` functions return the number of corresponding pages. `ddi_ptob()` returns the corresponding number of bytes. There are no error return values.
- Context** This function can be called from user, interrupt, or kernel context.
- Examples** **EXAMPLE 1** Find the size (in bytes) of one page  

```
pagesize = ddi_ptob(dip, 1L);
```
- See Also** [btop\(9F\)](#), [btopr\(9F\)](#), [ptob\(9F\)](#)  
*Writing Device Drivers*

**Name** ddi\_can\_receive\_sig – Test for ability to receive signals

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
boolean_t ddi_can_receive_sig(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** None.

**Description** The `ddi_can_receive_sig()` function returns a boolean value indicating whether the current thread can receive signals sent by `kill(2)`. If the return value is `B_FALSE`, then the calling thread cannot receive signals, and any call to `qwait_sig(9F)`, `cv_wait_sig(9F)`, or `cv_timedwait_sig(9F)` implicitly becomes `qwait(9F)`, `cv_wait(9F)`, or `cv_timedwait(9F)`, respectively. Drivers that can block indefinitely awaiting an event should use this function to determine if additional means (such as `timeout(9F)`) may be necessary to avoid creating unkillable threads.

**Return Values**

<code>B_FALSE</code>	The calling thread is in a state in which signals cannot be received. For example, the thread is not associated with a user process or is in the midst of <code>exit(2)</code> handling.
<code>B_TRUE</code>	The calling thread may receive a signal while blocked on a condition variable. Note that this function does not check to determine whether signals are blocked (see <code>sigprocmask(2)</code> ).

**Context** The `ddi_can_receive_sig()` function may be called from user, kernel, or interrupt context.

**See Also** `close(9E)`, `cv_wait(9F)`, `qwait(9F)`

**Name** ddi\_cb\_register, ddi\_cb\_unregister – register and unregister a device driver callback handler

**Synopsis** #include <sys/sunddi.h>

```
int ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags,
                  ddi_cb_func_t cbfunc, void *arg1, void *arg2,
                  ddi_cb_handle_t *ret_hdlp);

int ddi_cb_unregister(ddi_cb_handle_t hdl);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_cb\_register()

*dip* Pointer to the dev\_info structure.

*flags* Flags to determine which callback events can be handled.

*cbfunc* Callback handler function.

*arg1* First argument to the callback handler.

*arg2* Second (optional) argument to the callback handler.

*ret\_hdlp* Pointer to return a handle to the registered callback.

ddi\_cb\_unregister()

*hdl* Handle to the registered callback handler that is to be unregistered.

**Description** The ddi\_cb\_register() function installs a callback handler which processes various actions that require the driver's attention while it is attached. The driver specifies which callback actions it can handle through the flags parameter. With each relevant action, the specified callback function passes the arg1 and arg2 arguments along with the description of each callback event to the driver.

The ddi\_cb\_unregister() function removes a previously installed callback handler and prevents future processing of actions.

The flags parameter consists of the following:

DDI\_CB\_FLAG\_INTR

The device driver participates in interrupt resource management. The device driver may receive additional interrupt resources from the system, but only because it can accept callback notices informing it when it has more or less resources available. Callback notices can occur at anytime after the driver is attached. Interrupt availability varies based on the overall needs of the system.

DDI\_CB\_FLAG\_SRIOV

Indicates to the DDI framework that the device driver is IOV capable. Normally IOV device drivers are expected to call pci\_vf\_config(9F) to configure the VFs during attach

and register callbacks with this flag set so that they can be informed through callback notices when VFs are unconfigured by the DDI framework.

If the driver does not explicitly configure the VFs using `pciv_vf_config()` during attach then the PCIE framework will configure the VFs as part of the post-attach processing if this flag is set.

The callback notices while configuring/unconfiguring is performed with two separate callbacks to the driver PRE and POST. This helps the drivers to prepare for the event during PRE and do the necessary initializations/cleanup during POST notices.

#### DDI\_CB\_FLAG\_COMM

The device driver uses `pciv_send(9F)` interfaces to communicate across different domains or within the same domain. The driver may receive callback notices that incoming data has been received.

#### DDI\_CB\_FLAG\_LSR

The device driver supports Live Suspend and Resume (LSR) operations. When LSR operation (Live Suspend or Live Resume) happens, the device driver will be asked to suspend/resume various I/O activities of the device. The LSR operations may also impose various impacts (such as device reset, losing power, etc.) to the device, the driver needs to handle these impacts too. For more information about the activities and impacts, refer to the data type `ddi_cb_lsr_t` described below. Callback can occur at anytime after the driver is attached. Currently, LSR is only supported for PCI and PCIe devices.

The `cdfunc` is a callback handler with the following prototype:

```
typedef int (*ddi_cb_func_t)(dev_info_t *dip,
                             ddi_cb_action_t action, void *cbarg,
                             void *arg1, void *arg2);
```

The `cbfunc` routine with the arguments `dip`, `action`, `cbarg`, `arg1` and `arg2` is called upon receipt of any callbacks for which the driver is registered. The callback handler returns `DDI_SUCCESS` if the callback was handled successfully, `DDI_ENOTSUP` if it received a callback action that it did not know how to process, or `DDI_FAILURE` if it has an internal failure while processing an action.

For interrupt resource management, the `action` parameter can be one of the following:

#### DDI\_CB\_INTR\_ADD

For interrupt resource management, the driver has more available interrupts. The driver can allocate more interrupt vectors and then set up more interrupt handling functions by using `ddi_intr_alloc(9F)`.

#### DDI\_CB\_INTR\_REMOVE

For interrupt resource management, the driver has fewer available interrupts. The driver must release any previously allocated interrupts in excess of what is now available by using `ddi_intr_free(9F)`.

The *cbarg* parameter points to an action-specific argument. Each class of registered actions specifies its own data structure that a callback handler should dereference when it receives those actions.

The *cbarg* parameter is defined as an integer in the case of DDI\_CB\_INTR\_ADD and DDI\_CB\_INTR\_REMOVE actions. The callback handler should cast the *cbarg* parameter to an integer. The integer represents how many interrupts have been added or removed from the total number available to the device driver.

If a driver participates in interrupt resource management, it must register a callback with the DDI\_CB\_FLAG\_INTR flag. The driver then receives the actions DDI\_CB\_INTR\_ADD and DDI\_CB\_INTR\_REMOVE whenever its interrupt availability has changed. The callback handler should use the interrupt functions [ddi\\_intr\\_alloc\(9F\)](#) and [ddi\\_intr\\_free\(9F\)](#) functions to respond accordingly. A driver is not required to allocate all interrupts that are available to it, but it is required to manage its allocations so that it never uses more interrupts than are currently available.

#### DDI\_CB\_PCIV\_CONFIG\_VF

The PF driver is being notified of its VF configuration request. The `pciv_config_vf_t` structure is being passed as *cbarg* to describe the configuration. The *cmd* field in `pciv_config_vf_t` indicates if the VFs are about to or have just been enabled or disabled.

*cbarg* (when action is set to DDI\_CB\_PCIV\_CONFIG\_VF)

```
pciv_config_vf_t

typedef enum {
    PCIV_VFCFG_PARAM,          /* Retrieve VF configuration parameters */
    PCIV_VF_ENABLE,           /* Request to enable VFs synchronously */
    PCIV_VF_DISABLE,          /* Request to disable VFs synchronously */
    PCIV_EVT_VFENABLE_PRE,    /* VFs are just about to be enabled */
    PCIV_EVT_VFENABLE_POST,   /* VFs have just been enabled */
    PCIV_EVT_VFDISABLE_PRE,   /* VFs are just about to be disabled */
    PCIV_EVT_VFDISABLE_POST   /* VFs have just been disabled */
} pciv_vf_config_cmd_t;

typedef struct pciv_config_vf {
    int                version;
    pciv_vf_config_cmd_t cmd; /* pre/post VF enable/disable */
    uint16_t           num_vf; /* number of VFs to be used */
    uint16_t           first_vf_offset; /* offset between 1st VF & PF */
    uint16_t           vf_stride; /* distance between VFs */
    boolean_t          ari_cap; /* ARI capable hierarchy */
    uint32_t           page_size; /* system page size */
} pciv_config_vf_t;
```

The *cmd* field in the `pciv_config_vf_t` informs the driver the reason for the callback execution. The driver can return one of the following codes back to the caller.

**DDI\_SUCCESS**

The request was accepted and resources are properly configured.

**DDI\_NOTAPPLICABLE**

The requested configuration is not applicable.

**DDI\_REQRESET**

The requested configuration cannot be applied until device is reset (for (example, the PF hardware cannot dynamically adjust its internal resources to satisfy the request.)

**DDI\_REQREATTACH**

The requested configuration cannot be applied unless the driver itself is reattached.

**DDI\_CB\_COMM\_RECV**

Drivers are notified of incoming data described by `pciv_recv_event_t`, which is passed as the `cbarg` argument to the callback handler.

`cbarg` (when action is set to `DDI_CB_COMM_RECV`)

`pciv_recv_event_t`

```
typedef enum {
```

```
    PCIV_EVT_READY = 0x1, /* peer side has registered the recv cb */
    PCIV_EVT_NOT_READY, /* peer side has unregistered the recv cb */
    PCIV_EVT_DRV_DATA, /* private driver data event */
    PCIV_EVT_FABRIC /* PCIv framework and fabric admin event */
```

```
} pciv_event_type_t;
```

```
typedef struct pciv_recv_event {
```

```
    pciv_event_type_t event; /* event type */
    caddr_t buf; /* buffer address */
    size_t nbyte; /* size of buffer */
    uint32_t src_func; /* source function */
    dom_id_t src_domain; /* source domain */
```

```
} pciv_recv_event_t;
```

**PCIV\_EVT\_READY**

Both local and remote end has registered its event handler. The `buf` and `nbyte` fields should be ignored

**PCIV\_EVT\_NOT\_READY**

Remote end has not registered its event handler. The `buf` and `nbyte` fields should be ignored

**PCIV\_EVT\_DRV\_DATA**

Data in `buf` with `nbyte` size has been received or sent. Data can only be interpreted by the transmitter and receiver.

**PCIV\_EVT\_FABRIC**

Framework VF administration event, used between framework and PF drivers for VF administration purposes. Administration data is stored in `buf` with `nbyte` size.

*buf, nbyte*

Incoming data is stored in a buffer of address *buf* with length of *nbyte*. Not used for certain event types.

*src\_func*

The source function number of the transmission. It is used by the PF driver to identify the transmission source virtual function number. Besides source VF number, it can also be:

PCIV\_PF      Transmitter is the PF of the receiver.

PCIV\_FRM     Transmitter is the PCI Express framework rather than a driver.

*src\_domain*

The source domain of the transmission. It is used by PF driver to identify the domain the transmission is from.

For Live Suspend & Resume, the action parameter can be one of the following:

DDI\_CB\_LSR\_SUSPEND

Live suspend the device. The detailed I/O activities that will be suspended and impacts that the operation will impose to the device are specified by the argument pointed to by *cbarg*. It is of type `ddi_cb_lsr_t`.

DDI\_CB\_LSR\_RESUME

Live resume the device. The detailed activities and impacts are specified in the argument pointed to by *cbarg*. It will be the same as the previous suspend operation.

DDI\_CB\_LSR\_QUERY\_CAPABILITY

Query the device's LSR capabilities. The detailed activities and impacts that the device driver supports are returned in the argument pointed to by *cbarg*. The type of the argument is `ddi_cb_lsr_t`.

For all the LSR actions, the *cbarg* parameter points to the argument of the same type, `ddi_cb_lsr_t`, which is defined as:

```
typedef struct ddi_cb_lsr {
    uint64_t    activities;
    uint64_t    impacts;
    char        *reason;
} ddi_cb_lsr_t;
```

The *activities* member specifies what I/O activities of the device will be live suspended/resumed (DDI\_CB\_LSR\_SUSPEND and DDI\_CB\_LSR\_RESUME), or the I/O activities that the device driver supports (DDI\_CB\_LSR\_QUERY\_CAPABILITY).

For PCI/PCIe devices, the *activities* member in `ddi_cb_lsr_t` can be a value bit-ORed from the following:

DDI\_CB\_LSR\_ACT\_DMA      DMA activities

DDI\_CB\_LSR\_ACT\_PIO      Programmed I/O activities

DDI\_CB\_LSR\_ACT\_INTR      Interrupt activities

The *impacts* member specifies the side effects the LSR operation will impose on the device. For PCI/PCIe devices, it can be bit-ORed value of the following:

DDI_CB_LSR_IMP_DMA_ADDR_CHANGE	During the LSR operation, the device will undergo DMA address changes. The device driver should unbind and rebind the DMA buffers.
DDI_CB_LSR_IMP_DMA_PROP_CHANGE	During the LSR operation, the platform hardware changes may cause the driver's DMA transaction properties to be invalid, such as the lowest and highest DMA address range, DMA address alignment, DMA segment boundary, etc. The driver needs to destroy and reallocate DMA handles.
DDI_CB_LSR_IMP_DEVICE_RESET	During the LSR operation, the device will undergo device reset. The device driver must restore the hardware context during live resume.
DDI_CB_LSR_IMP_DEVICE_REPLACE	During the LSR operation, the device will be replaced. The device driver needs to check the device identity.
DDI_CB_LSR_IMP_LOSE_POWER	During the LSR operation, the device will lose power. The device driver needs to restore device hardware context during live resume. If necessary, the driver should reload firmware.

The *reason* member is a predefined string to specify the cause of the LSR operation. It is used for printing purposes. There is no predefined string and it may be NULL for DDI\_CB\_LSR\_SUSPEND and DDI\_CB\_LSR\_RESUME. For DDI\_CB\_LSR\_QUERY\_CAPABILITY, the driver must set it to NULL.

The driver's implementation should make the LSR operation transparent to user applications, such that when the driver is suspended, it can block user I/O requests (read/write/ioctl) but should not return failure.

The driver may have grouped activities, such as when one is suspended, the other has to be suspended as well. When the driver is asked to suspend one of the grouped activities, it can choose to suspend all the activities in the group.

The driver's callback function will only be asked to suspend after the former suspension has already been resumed.

The callback function should return as soon as possible. In normal cases, returning in milliseconds should be acceptable while returning in seconds is not.



**Return Values** The `ddi_cb_register()` and `ddi_cb_unregister()` functions return:

<code>DDI_SUCCESS</code>	on success
<code>DDI_EINVAL</code>	An invalid parameter was given when registering a callback handler, or an invalid handle was given when unregistering.
<code>DDI_EALREADY</code>	An attempt was made to register a callback handler while a previous registration still exists.

The *cbfunc* routine must return:

<code>DDI_SUCCESS</code>	on success
<code>DDI_ENOTSUP</code>	The device does not support the operation
<code>DDI_FAILURE</code>	Implementation specific failure

**Context** These functions can be called from kernel, non-interrupt context.

**Examples** EXAMPLE 1 An example using `ddi_cb_register()`.

```

/*
 * attach(9F) routine.
 *
 * Creates soft state, registers callback handler, initializes
 * hardware, and sets up interrupt handling for the driver.
 */
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    xx_state_t          *statep = NULL;
    xx_intr_t           *intrs = NULL;
    ddi_intr_handle_t   *hdl;
    ddi_cb_handle_t     cb_hdl;
    int                 instance;
    int                 type;
    int                 types;
    int                 nintrs;
    int                 nactual;
    int                 inum;

    /* Get device instance */
    instance = ddi_get_instance(dip);

    switch (cmd) {
    case DDI_ATTACH:

        /* Get soft state */
        if (ddi_soft_state_zalloc(state_list, instance) != 0)
            return (DDI_FAILURE);

```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```
statep = ddi_get_soft_state(state_list, instance);
ddi_set_driver_private(dip, (caddr_t)statep);
statep->dip = dip;

/* Initialize hardware */
xx_initialize(statep);

/* Allocate/bind dma handle, etc. */
xx_dma_attach(statep);

/* Initialize Programmed I/O, map register memory, etc. */
xx_pio_attach(statep);

/* Initialize LSR state */
statep->lsr_state = 0;

/* Register callback handler */
if (ddi_cb_register(dip, DDI_CB_FLAG_INTR | DDI_CB_FLAG_LSR,
    xx_cbfunc, statep, NULL, &cb_hdl) != 0) {
    ddi_soft_state_free(state_list, instance);
    return (DDI_FAILURE);
}
statep->cb_hdl = cb_hdl;

/* Select interrupt type */
ddi_intr_get_supported_types(dip, &types);
if (types & DDI_INTR_TYPE_MSIX) {
    type = DDI_INTR_TYPE_MSIX;
} else if (types & DDI_INTR_TYPE_MSI) {
    type = DDI_INTR_TYPE_MSI;
} else {
    type = DDI_INTR_TYPE_FIXED;
}
statep->type = type;

/* Get number of supported interrupts */

ddi_intr_get_nintrs(dip, type, &nintrs);

/* Allocate interrupt handle array */
statep->hdl_size = nintrs * sizeof (ddi_intr_handle_t);
hdl = kmem_zalloc(statep->hdl_size, KMEM_SLEEP);

/* Allocate interrupt setup array */
statep->intrs_size = nintrs * sizeof (xx_intr_t);
```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

statep->intrs = kmem_zalloc(statep->intrs_size, KMEM_SLEEP);

/* Allocate interrupt vectors */
ddi_intr_alloc(dip, hdls, type, 0, nintrs, &nactual, 0);
statep->nactual = nactual;

/* Configure interrupt handling */
xx_setup_interrupts(statep, nactual, statep->intrs);

/* Install and enable interrupt handlers */
for (inum = 0; inum < nactual; inum++) {
    ddi_intr_add_handler(&statep->hdls[inum],
        statep->intrs[inum].inthandler,
        statep->intrs[inum].arg1,
        statep->intrs[inum].arg2);
    ddi_intr_enable(statep->hdls[inum]);
}

break;

case DDI_RESUME:

    /* Get soft state */
    statep = ddi_get_soft_state(state_list, instance);
    if (statep == NULL)
        return (DDI_FAILURE);

    /* Resume hardware */
    xx_resume(statep);

    break;
}

return (DDI_SUCCESS);
}

/*
 * detach(9F) routine.
 *
 * Stops the hardware, disables interrupt handling, unregisters
 * a callback handler, and destroys the soft state for the driver.
 */
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    xx_state_t    *statep = NULL;

```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```
int          instance;
int          inum;

/* Get device instance */
instance = ddi_get_instance(dip);

switch (cmd) {
case DDI_DETACH:

    /* Get soft state */
    statep = ddi_get_soft_state(state_list, instance);
    if (statep == NULL)
        return (DDI_FAILURE);

    /* Stop device */
    xx_uninitialize(statep);

    /* Disable and free interrupts */
    for (inum = 0; inum < statep->nactual; inum++) {
        ddi_intr_disable(statep->hdls[inum]);
        ddi_intr_remove_handler(statep->hdls[inum]);
        ddi_intr_free(statep->hdls[inum]);
    }

    /* Clean up DMA, unbind/free dma handle etc. */
    xx_tear_down_dma(statep);

    /* Clean up PIO, unmap register memory etc. */
    xx_tear_down_pio(statep);

    /* Unregister callback handler */
    ddi_cb_unregister(statep->cb_hdl);

    /* Free interrupt handle array */
    kmem_free(statep->hdls, statep->hdls_size);

    /* Free interrupt setup array */
    kmem_free(statep->intrs, statep->intrs_size);

    /* Free soft state */
    ddi_soft_state_free(state_list, instance);

    break;
```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

case DDI_SUSPEND:

    /* Get soft state */
    statep = ddi_get_soft_state(state_list, instance);
    if (statep == NULL)
        return (DDI_FAILURE);

    /* Suspend hardware */
    xx_quiesce(statep);

    break;
}

return (DDI_SUCCESS);
}

/*
 * (*ddi_cbfunc)() routine.
 *
 * Adapt interrupt usage when availability changes.
 */
int
xx_cbfunc(dev_info_t *dip, ddi_cb_action_t cbaction, void *cbarg,
          void *arg1, void *arg2)
{
    xx_state_t      *statep = (xx_state_t *)arg1;
    int             count;
    int             inum;
    int             nactual;

    switch (cbaction) {
    case DDI_CB_INTR_ADD:
    case DDI_CB_INTR_REMOVE:

        /* Get change in availability */
        count = (int)(uintptr_t)cbarg;

        /* Suspend hardware */
        xx_quiesce(statep);

        /* Tear down previous interrupt handling */
        for (inum = 0; inum < statep->nactual; inum++) {
            ddi_intr_disable(statep->hdls[inum]);
            ddi_intr_remove_handler(statep->hdls[inum]);
        }
    }
}

```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

/* Adjust interrupt vector allocations */
if (cbaction == DDI_CB_INTR_ADD) {

    /* Allocate additional interrupt vectors */
    ddi_intr_alloc(dip, statep->hdls, statep->type,
        statep->nactual, count, &nactual, 0);

    /* Update actual count of available interrupts */
    statep->nactual += nactual;

} else {

    /* Free removed interrupt vectors */
    for (inum = statep->nactual - count;
        inum < statep->nactual; inum++) {
        ddi_intr_free(statep->hdls[inum]);
    }

    /* Update actual count of available interrupts */
    statep->nactual -= count;
}

/* Configure interrupt handling */
xx_setup_interrupts(statep, statep->nactual, statep->intrs);

/* Install and enable interrupt handlers */
for (inum = 0; inum < statep->nactual; inum++) {
    ddi_intr_add_handler(&statep->hdls[inum],
        statep->intrs[inum].inhandler,
        statep->intrs[inum].arg1,
        statep->intrs[inum].arg2);
    ddi_intr_enable(statep->hdls[inum]);
}

/* Resume hardware */
xx_resume(statep);

break;

case DDI_CB_LSR_SUSPEND:

    plsr = (ddi_cb_lsr_t *) cbarg;
    if (plsr->impacts &
        (DDI_CB_LSR_IMP_LOSE_POWER

```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

        | DDI_CB_LSR_IMP_DEVICE_REPLACE))
        return DDI_NOTSUP;

    if (plsr->activities & DDI_CB_LSR_ACT_DMA) {
        xx_stop_dma_engine(statep);
        if (plsr->impacts
            & DDI_CB_LSR_IMP_DMA_ADDR_CHANGE) {
            xx_unbind_dma_buffer(statep);
        }
        if (plsr->impacts
            & DDI_CB_LSR_IMP_DMA_PROP_CHANGE) {
            xx_destroy_dma_handle(statep);
        }
    }

    if (plsr->activities & DDI_CB_LSR_ACT_INTR) {
        xx_disable_interrupts(statep);
    }

    if (plsr->activities & DDI_CB_LSR_ACT_PIO) {
        xx_stop_pio(statep);
    }

    if (plsr->impacts & DDI_CB_LSR_IMP_DEVICE_RESET) {
        /*
         * may save the context to be restored for re-loaded
         * firmware on resume
         */
        xx_save_hardware_context(statep);
    }

    statep->saved_lsr = *plsr;
    statep->lsr_state = XX_LSR_SUSPENDED;

    break;

case DDI_CB_LSR_RESUME:

    if (statep->lsr_state != XX_LSR_SUSPENDED)
        return (DDI_EINVAL);

    plsr = (ddi_cb_lsr_t *) cbarg;

    if (plsr->activities != statep->saved_lsr.activities
        || plsr->impacts != statep->saved_lsr.impacts)

```

EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

        return (DDI_EINVAL);

    if (plsr->impacts & DDI_CB_LSR_IMP_DEVICE_RESET) {
        xx_restore_hardware_context(statep);
    }

    if (plsr->activities & DDI_CB_LSR_ACT_DMA) {
        if (plsr->impacts
            & DDI_CB_LSR_IMP_DMA_PROP_CHANGE) {
            xx_alloc_dma_handle(statep);
        }
        if (plsr->impacts
            & DDI_CB_LSR_IMP_DMA_ADDR_CHANGE) {
            xx_bind_dma_buffer(statep);
        }
        xx_start_dma_engine(statep);
    }

    if (plsr->activities & DDI_CB_LSR_ACT_INTR) {
        xx_enable_interrupts(statep);
    }

    if (plsr->activities & DDI_CB_LSR_ACT_PIO) {
        xx_enable_pio(statep);
    }

    statep->lsr_state = 0;

    break;

case DDI_CB_LSR_QUERY_CAPABILITY:
    /*
     * return the supported activities and impacts
     */
    plsr = (ddi_cb_lsr_t *) cbarg;

    plsr->activities = DDI_CB_LSR_ACT_DMA | DDI_CB_LSR_ACT_PIO
        | DDI_CB_LSR_ACT_INTR;
    plsr->impacts = DDI_CB_LSR_IMP_DEVICE_RESET
        | DDI_CB_LSR_IMP_DMA_ADDR_CHANGE
        | DDI_CB_LSR_IMP_DMA_PROP_CHANGE;

    break;

default:

```



EXAMPLE 1 An example using `ddi_cb_register()`. (Continued)

```

        return (DDI_ENOTSUP);
    }

    return (DDI_SUCCESS);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted
MT-Level	Unsafe

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_free\(9F\)](#), [ddi\\_intr\\_set\\_nreq\(9F\)](#), [pciv\\_send\(9F\)](#), [pciv\\_vf\\_config\(9F\)](#)

**Notes** Users of these interfaces that register for `DDI_CB_FLAG_INTR` become participants in interrupt resource management. With that participation comes a responsibility to properly adjust interrupt usage. In the case of a `DDI_CB_INTR_ADD` action, the system guarantees that a driver can allocate a total number of interrupt resources up to its new number of available interrupts. The total number of interrupt resources is the sum of all resources allocated by the function [ddi\\_intr\\_alloc\(9F\)](#), minus all previously released by the function [ddi\\_intr\\_free\(9F\)](#). In the case of a `DDI_CB_INTR_REMOVE` action, the driver might have more interrupts allocated than are now currently available. It is necessary for the driver to release the excess interrupts, or it will have a negative impact on the interrupt availability for other drivers in the system.

A failure to release interrupts in response to a `DDI_CB_INTR_REMOVE` callback generates the following warning on the system console:

```

WARNING: <driver><instance>: failed to release interrupts for
        IRM (nintrs = ##, navail=##).

```

Participation in interrupt resource management ends when a driver uses the `ddi_cb_unregister()` function to unregister its callback function. The callback function must still operate properly until after the call to the `ddi_cb_unregister()` function completes. If additional interrupts were given to the driver because of its participation, then a final use of the callback function occurs to release the additional interrupts. The call to the `ddi_cb_unregister()` function blocks until the final use of the registered callback function is finished.

**Name** ddi\_check\_acc\_handle, ddi\_check\_dma\_handle – Check data access and DMA handles

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_check_acc_handle(ddi_acc_handle_t acc_handle );  
int ddi_check_dma_handle(ddi_dma_handle_t dma_handle );
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *acc\_handle* Data access handle obtained from a previous call to [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_dma\\_mem\\_alloc\(9F\)](#), or similar function.  
*dma\_handle* DMA handle obtained from a previous call to [ddi\\_dma\\_setup\(9F\)](#) or one of its derivatives.

**Description** The [ddi\\_check\\_acc\\_handle\(\)](#) and [ddi\\_check\\_dma\\_handle\(\)](#) functions check for faults that can interfere with communication between a driver and the device it controls. Each function checks a single handle of a specific type and returns a status value indicating whether faults affecting the resource mapped by the supplied handle have been detected.

If a fault is indicated when checking a data access handle, this implies that the driver is no longer able to access the mapped registers or memory using programmed I/O through that handle. Typically, this might occur after the device has failed to respond to an I/O access (for example, has incurred a bus error or timed out). The effect of programmed I/O accesses made after this happens is undefined; for example, read accesses (for example, [ddi\\_get8\(9F\)](#)) may return random values, and write accesses (for example, [ddi\\_put8\(9F\)](#)) may or may not have any effect. This type of fault is normally fatal to the operation of the device, and the driver should report it via [ddi\\_dev\\_report\\_fault\(9F\)](#) specifying `DDI_SERVICE_LOST` for the impact, and `DDI_DATAPATH_FAULT` for the location.

If a fault is indicated when checking a DMA handle, it implies that a fault has been detected that has (or will) affect DMA transactions between the device and the memory currently bound to the handle (or most recently bound, if the handle is currently unbound). Possible causes include the failure of a component in the DMA data path, or an attempt by the device to make an invalid DMA access. The driver may be able to continue by falling back to a non-DMA mode of operation, but in general, DMA faults are non-recoverable. The contents of the memory currently (or previously) bound to the handle should be regarded as indeterminate. The fault indication associated with the current transaction is lost once the handle is (re-)bound, but because the fault may persist, future DMA operations may not succeed.

Note that some implementations cannot detect all types of failure. If a fault is not indicated, this does not constitute a guarantee that communication is possible. However, if a check fails, this is a positive indication that a problem *does* exist with respect to communication using that handle.

**Return Values** The `ddi_check_acc_handle()` and `ddi_check_dma_handle()` functions return `DDI_SUCCESS` if no faults affecting the supplied handle are detected and `DDI_FAILURE` if any fault affecting the supplied handle is detected.

**Examples**

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    \&...
    /* This driver uses only a single register-access handle */
    status = ddi_regs_map_setup(dip, REGSET_ZERO, &regaddr,
                               0, 0, , &acc_attrs, &acc_hdl);
    if (status != DDI_SUCCESS)
        return (DDI_FAILURE);
    \&...
}

static int
xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    \&...
    if (ddi_check_acc_handle(acc_hdl) != DDI_SUCCESS) {
        ddi_dev_report_fault(dip, DDI_SERVICE_LOST,
                            DDI_DATAPATH_FAULT, "register access fault during read");
        return (EIO);
    }
    \&...
}
```

**Context** The `ddi_check_acc_handle()` and `ddi_check_dma_handle()` functions may be called from user, kernel, or interrupt context.

**See Also** [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dev\\_report\\_fault\(9F\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#)

**Name** ddi\_copyin – copy data to a driver buffer

**Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_copyin(const void *buf, void *driverbuf, size_t cn, int flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>buf</i>	Source address from which data is transferred.
<i>driverbuf</i>	Driver destination address to which data is transferred.
<i>cn</i>	Number of bytes transferred.
<i>flags</i>	Set of flag bits that provide address space information about <i>buf</i> .

**Description** This routine is designed for use in driver [ioctl\(9E\)](#) routines for drivers that support layered ioctls. `ddi_copyin()` copies data from a source address to a driver buffer. The driver developer must ensure that adequate space is allocated for the destination address.

The *flags* argument determines the address space information about *buf*. If the `FKIOCTL` flag is set, this indicates that *buf* is a kernel address, and `ddi_copyin()` behaves like [bcopy\(9F\)](#). Otherwise, *buf* is interpreted as a user buffer address, and `ddi_copyin()` behaves like [copyin\(9F\)](#).

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move according to address alignment.

**Return Values** `ddi_copyin()` returns 0, indicating a successful copy. It returns -1 if one of the following occurs:

- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
- Invalid user address, such as a user area or stack area.
- Invalid address that would have resulted in data being copied into the user block.
- Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.

If -1 is returned to the caller, driver entry point routines should return `EFAULT`.

**Context** `ddi_copyin()` can be called from user or kernel context only.

**Examples** EXAMPLE 1 ddi\_copyin() example

A driver `ioctl(9E)` routine (line 12) can be used to get or set device attributes or registers. For the `XX_SETREGS` condition (line 25), the driver copies the user data in `arg` to the device registers. If the specified argument contains an invalid address, an error code is returned.

```

1 struct device {          /* layout of physical device registers */
2     int     control;     /* physical device control word */
3     int     status;     /* physical device status word */
4     short   recv_char;  /* receive character from device */
5     short   xmit_char   /* transmit character to device */
6 };
7 struct device_state {
8     volatile struct device *regsp; /* pointer to device registers */
9     kmutex_t reg_mutex;           /* protect device registers */
10    . . .
11 };
12 static void *statep; /* for soft state routines */
13
14 xxioctl(dev_t dev, int cmd, int arg, int mode,
15         cred_t *cred_p, int *rval_p)
16 {
17     struct device_state *sp;
18     volatile struct device *rp;
19     struct device reg_buf; /* temporary buffer for registers */
20     int instance;
21
22     instance = getminor(dev);
23     sp = ddi_get_soft_state(statep, instance);
24     if (sp == NULL)
25         return (ENXIO);
26     rp = sp->regsp;
27     . . .
28     switch (cmd) {
29
30     case XX_GETREGS: /* copy data to temp. regs. buf */
31         if (ddi_copyin(arg, &reg_buf,
32             sizeof (struct device), mode) != 0) {
33             return (EFAULT);
34         }
35
36         mutex_enter(&sp->reg_mutex);
37         /*
38          * Copy data from temporary device register
39          * buffer to device registers.
40          * e.g. rp->control = reg_buf.control;
41          */

```

EXAMPLE 1 ddi\_copyin() example (Continued)

```
36         mutex_exit(&sp->reg_mutex);  
  
37         break;  
38     }  
39 }
```

**See Also** [ioctl\(9E\)](#), [bcopy\(9F\)](#), [copyin\(9F\)](#), [copyout\(9F\)](#), [ddi\\_copyout\(9F\)](#), [uiomove\(9F\)](#)

*Writing Device Drivers*

**Notes** The value of the *flags* argument to `ddi_copyin()` should be passed through directly from the *mode* argument of `ioctl()` untranslated.

Driver defined locks should not be held across calls to this function.

`ddi_copyin()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*.

- 
- Name** ddi\_copyout – copy data from a driver
- Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_copyout(const void *driverbuf, void *buf, size_t cn, int flags);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters**
- |                  |  |
|------------------|--|
| <i>driverbuf</i> | Source address in the driver from which the data is transferred.           |
| <i>buf</i>       | Destination address to which the data is transferred.                      |
| <i>cn</i>        | Number of bytes to copy.   |
| <i>flags</i>     | Set of flag bits that provide address space information about <i>buf</i> . |
- Description** This routine is designed for use in driver [ioctl\(9E\)](#) routines for drivers that support layered ioctls. `ddi_copyout()` copies data from a driver buffer to a destination address, *buf*.
- The *flags* argument determines the address space information about *buf*. If the `FKIOCTL` flag is set, this indicates that *buf* is a kernel address, and `ddi_copyout()` behaves like [bcopy\(9F\)](#). Otherwise, *buf* is interpreted as a user buffer address, and `ddi_copyout()` behaves like [copyout\(9F\)](#).
- Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obliged to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.
- Return Values** Under normal conditions, 0 is returned to indicate a successful copy. Otherwise, -1 is returned if one of the following occurs:
- Paging fault; the driver tried to access a page of memory for which it did not have read or write access.
  - Invalid user address, such as a user area or stack area.
  - Invalid address that would have resulted in data being copied into the user block.
  - Hardware fault; a hardware error prevented access to the specified user memory. For example, an uncorrectable parity or ECC error occurred.
- If -1 is returned to the caller, driver entry point routines should return `EFAULT`.
- Context** `ddi_copyout()` can be called from user or kernel context only.

**Examples** EXAMPLE 1 ddi\_copyout() example

A driver `ioctl(9E)` routine (line 12) can be used to get or set device attributes or registers. In the `XX_GETREGS` condition (line 25), the driver copies the current device register values to another data area. If the specified argument contains an invalid address, an error code is returned.

```

1 struct device {          /* layout of physical device registers */
2     int     control;     /* physical device control word */
3     int     status;     /* physical device status word */
4     short   recv_char;   /* receive character from device */
5     short   xmit_char;  /* transmit character to device */
6 };

7 struct device_state {
8     volatile struct device *regsp; /* pointer to device registers */
9     kmutex_t reg_mutex;           /* protect device registers */
10    . . .
11 };

12 static void *statep; /* for soft state routines */

13 xxioctl(dev_t dev, int cmd, int arg, int mode,
14         cred_t *cred_p, int *rval_p)
15 {
16     struct device_state *sp;
17     volatile struct device *rp;
18     struct device reg_buf; /* temporary buffer for registers */
19     int instance;

20     instance = getminor(dev);
21     sp = ddi_get_soft_state(statep, instance);
22     if (sp == NULL)
23         return (ENXIO);
24     rp = sp->regsp;
25     . . .
26     switch (cmd) {

27     case XX_GETREGS: /* copy registers to arg */
28         mutex_enter(&sp->reg_mutex);
29         /*
30          * Copy data from device registers to
31          * temporary device register buffer
32          * e.g. reg_buf.control = rp->control;
33          */
34         mutex_exit(&sp->reg_mutex);
35         if (ddi_copyout(&reg_buf, arg,
36             sizeof (struct device), mode) != 0) {

```



---

EXAMPLE 1 ddi\_copyout() example (Continued)

```
35             return (EFAULT);
36         }

37         break;
38     }
39 }
```

**See Also** [ioctl\(9E\)](#), [bcopy\(9F\)](#), [copyin\(9F\)](#), [copyout\(9F\)](#), [ddi\\_copyin\(9F\)](#), [uiomove\(9F\)](#)

*Writing Device Drivers*

**Notes** The value of the *flags* argument to `ddi_copyout()` should be passed through directly from the *mode* argument of `ioctl()` untranslated.

Driver defined locks should not be held across calls to this function.

`ddi_copyout()` should not be used from a streams driver. See `M_COPYIN` and `M_COPYOUT` in *STREAMS Programming Guide*.

**Name** ddi\_create\_minor\_node – Create a minor node for this device

**Synopsis** #include <sys/stat.h>  
#include <sys/sunddi.h>

```
int ddi_create_minor_node(dev_info_t *dip, char *name, int spec_type,
    minor_t minor_num, char *node_type, int flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dip</i>	A pointer to the device's dev_info structure.
<i>name</i>	The name of this particular minor device.
<i>spec_type</i>	S_IFCHR or S_IFBLK for character or block minor devices respectively.
<i>minor_num</i>	The minor number for this particular minor device.
<i>node_type</i>	Any string literal that uniquely identifies the type of node. The following predefined node types are provided with this release:
DDI_NT_SERIAL	For serial ports
DDI_NT_SERIAL_MB	For on board serial ports
DDI_NT_SERIAL_DO	For dial out ports
DDI_NT_SERIAL_MB_DO	For on board dial out ports
DDI_NT_BLOCK	For hard disks
DDI_NT_BLOCK_CHAN	For hard disks with channel or target numbers
DDI_NT_CD	For CDROM drives
DDI_NT_CD_CHAN	For CDROM drives with channel or target numbers
DDI_NT_TAPE	For tape drives
DDI_NT_NET	For DLPI style 1 or style 2 network devices
DDI_NT_DISPLAY	For display devices
DDI_PSEUDO	For pseudo devices
<i>flag</i>	If the device is a clone device then this flag is set to CLONE_DEV else it is set to 0.

**Description** ddi\_create\_minor\_node() provides the necessary information to enable the system to create the /dev and /devices hierarchies. The *name* is used to create the minor name of the block or character special file under the /devices hierarchy. At-sign (@), slash (/), and space are not allowed. The *spec\_type* specifies whether this is a block or character device. The *minor\_num* is the minor number for the device. The *node\_type* is used to create the names in the /dev

hierarchy that refers to the names in the `/devices` hierarchy. See [disks\(1M\)](#), [ports\(1M\)](#), [tapes\(1M\)](#), [devlinks\(1M\)](#). Finally *flag* determines if this is a clone device or not, and what device class the node belongs to.

**Return Values** `ddi_create_minor_node()` returns:

`DDI_SUCCESS` Was able to allocate memory, create the minor data structure, and place it into the linked list of minor devices for this driver.

`DDI_FAILURE` Minor node creation failed.

**Context** The `ddi_create_minor_node()` function can be called from user context. It is typically called from [attach\(9E\)](#) or [ioctl\(9E\)](#).

**Examples** **EXAMPLE 1** Create Data Structure Describing Minor Device with Minor Number of 0

The following example creates a data structure describing a minor device called *foo* which has a minor number of 0. It is of type `DDI_NT_BLOCK` (a block device) and it is not a clone device.

```
ddi_create_minor_node(dip, "foo", S_IFBLK, 0, DDI_NT_BLOCK, 0);
```

**See Also** [add\\_drv\(1M\)](#), [devlinks\(1M\)](#), [disks\(1M\)](#), [drvconfig\(1M\)](#), [ports\(1M\)](#), [tapes\(1M\)](#), [attach\(9E\)](#), [ddi\\_remove\\_minor\\_node\(9F\)](#)

### *Writing Device Drivers*

**Notes** If the driver is for a network device (*node\_type* `DDI_NT_NET`), note that the driver name will undergo the driver name constraints identified in the NOTES section of [dlpi\(7P\)](#). Additionally, the minor name must match the driver name for a DLPI style 2 provider. If the driver is a DLPI style 1 provider, the minor name must also match the driver name with the exception that the *ppa* is appended to the minor name.

Non-[gld\(7D\)](#)-based DLPI network streams drivers are encouraged to switch to [gld\(7D\)](#). Failing this, a driver that creates DLPI style-2 minor nodes must specify `CLONE_DEV` for its style-2 `ddi_create_minor_node()` nodes and use [qassociate\(9F\)](#). A driver that supports both style-1 and style-2 minor nodes should return `DDI_FAILURE` for `DDI_INFO_DEVT2INSTANCE` and `DDI_INFO_DEVT2DEVINFO` [getinfo\(9E\)](#) calls to style-2 minor nodes. (The correct association is already established by [qassociate\(9F\)](#)). A driver that only supports style-2 minor nodes can use [ddi\\_no\\_info\(9F\)](#) for its [getinfo\(9E\)](#) implementation. For drivers that do not follow these rules, the results of a [modunload\(1M\)](#) of the driver or a [cfgadm\(1M\)](#) remove of hardware controlled by the driver are undefined.

**Warning** Drivers must remove references to `GLOBAL_DEV`, `NODEBOUND_DEV`, `NODESPECIFIC_DEV`, and `ENUMERATED_DEV` to compile under Solaris 10 and later versions.

**Name** ddi\_cred, crgetuid, crgetruid, crgetsuid, crgetgid, crgetrgid, crgetsgid, crgetzoneid, crgetgroups, crgetngroups – access and change parts of the cred\_t structure

**Synopsis** #include <sys/cred.h>

```
uid_t crgetuid(const cred_t *cr);
uid_t crgetruid(const cred_t *cr);
uid_t crgetsuid(const cred_t *cr);
gid_t crgetgid(const cred_t *cr);
gid_t crgetrgid(const cred_t *cr);
gid_t crgetsgid(const cred_t *cr);
zoneid_t crgetzoneid(const cred_t *cr);
const gid_t *crgetgroups(const cred_t *cr);
int crgetngroups(const cred_t *cr);
int crsetresuid(cred_t *cr, uid_t ruid, uid_t euid, uid_t suid);
int crsetresgid(cred_t *cr, gid_t rgid, gid_t egid, gid_t sgid);
int crsetugid(cred_t *cr, uid_t uid, gid_t gid);
int crsetgroups(cred_t *cr, int ngroups, gid_t gids);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

<b>Parameters</b>	<i>cr</i>	pointer to the user credential structure
	<i>uid, ruid, euid, suid</i>	new user id, real, effective and saved user id
	<i>gid, rgid, egid, sgid</i>	new group id, real, effective and saved group id
	<i>ngroups</i>	number of groups in the group array
	<i>gids</i>	pointer to array of new groups

**Description** The user credential is a shared, read-only, ref-counted data structure. Its actual size and layout are subject to change. The functions described in this page allow the programmer to retrieve fields from the structure and to initialize newly allocated credential structures.

crgetuid(), crgetruid(), and crgetsuid() return, respectively, the effective, real, and saved user id from the user credential pointed to by *cr*.

crgetgid(), crgetrgid(), and crgetsgid() return, respectively, the effective, real, and saved group id from the user credential pointed to by *cr*.

crgetzoneid() returns the zone id from the user credential pointed to by *cr*.

`crgetgroups()` returns the group list of the user credential pointed to by *cr*.

`crgetngroups()` returns the number of groups in the user credential pointed to by *cr*.

`crsetresuid()` sets the real, effective and saved user id. All but one can be specified as -1, which causes the original value not to change.

`crsetresgid()` sets the real, effective and saved group id. All but one can be specified as -1, which causes the original value not to change.

`crsetugid()` initializes the real, effective and saved user id all to *uid*. It initializes the real, effective, and saved group id all to *gid*.

`crsetgroups()` sets the number of groups in the user credential to *ngroups* and copies the groups from *gids* to the user credential. If *ngroups* is 0, *gids* need not point to valid storage.

It is an error to call this any of the `crset*()` functions on a user credential structure that was newly allocated.

**Return Values** The `crget*()` functions return the requested information.

The `crset*id()` functions return 0 on success and -1 if any of the specified ids are invalid. The functions might cause a system panic if called on a user credential structure that is referenced by other parts of the system.

**Context** These functions can be called from user and kernel contexts.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	All
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [privileges\(5\)](#), [drv\\_priv\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_device\_copy – copy data from one device register to another device register

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_device_copy(ddi_acc_handle_t src_handle, caddr_t src_addr,
    ssize_t src_advcnt, ddi_acc_handle_t dest_handle,
    caddr_t dest_addr, ssize_t dest_advcnt,
    size_t bytecount, uint_t dev_datsz);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>src_handle</i>	The data access handle of the source device.
<i>src_addr</i>	Base data source address.
<i>src_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.
<i>dest_handle</i>	The data access handle of the destination device.
<i>dest_addr</i>	Base data destination address.
<i>dest_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.
<i>bytecount</i>	Number of bytes to transfer.
<i>dev_datsz</i>	The size of each data word. Possible values are defined as:
DDI_DATA_SZ01_ACC	1 byte data size
DDI_DATA_SZ02_ACC	2 bytes data size
DDI_DATA_SZ04_ACC	4 bytes data size
DDI_DATA_SZ08_ACC	8 bytes data size

**Description** ddi\_device\_copy() copies *bytecount* bytes from the source address, *src\_addr*, to the destination address, *dest\_addr*. The attributes encoded in the access handles, *src\_handle* and *dest\_handle*, govern how data is actually copied from the source to the destination. Only matching data sizes between the source and destination are supported.

Data will automatically be translated to maintain a consistent view between the source and the destination. The translation may involve byte-swapping if the source and the destination devices have incompatible endian characteristics.

The *src\_advcnt* and *dest\_advcnt* arguments specifies the number of *dev\_datsz* units to advance with each access to the device addresses. A value of 0 will use the same source and destination device address on every access. A positive value increments the corresponding device address by certain number of data size units in the next access. On the other hand, a negative value decrements the device address.

The *dev\_datasz* argument determines the size of the data word on each access. The data size must be the same between the source and destination.

**Return Values** `ddi_device_copy()` returns:

`DDI_SUCCESS`      Successfully transferred the data.

`DDI_FAILURE`      The byte count is not a multiple *dev\_datasz*.

**Context** `ddi_device_copy()` can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_device\_zero – zero fill the device

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_device_zero(ddi_acc_handle_t handle, caddr_t dev_addr,
                   size_t bytecount, ssize_t dev_advcnt, uint_t dev_datsz);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>handle</i>	The data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>dev_addr</i>	Beginning of the device address.
<i>bytecount</i>	Number of bytes to zero.
<i>dev_advcnt</i>	Number of <i>dev_datsz</i> units to advance on every access.
<i>dev_datsz</i>	The size of each data word. Possible values are defined as:
DDI_DATA_SZ01_ACC	1 byte data size
DDI_DATA_SZ02_ACC	2 bytes data size
DDI_DATA_SZ04_ACC	4 bytes data size
DDI_DATA_SZ08_ACC	8 bytes data size

**Description** ddi\_device\_zero() function fills the given, *bytecount*, number of byte of zeroes to the device register or memory.

The *dev\_advcnt* argument determines the value of the device address, *dev\_addr*, on each access. A value of 0 will use the same device address, *dev\_addr*, on every access. A positive value increments the device address in the next access while a negative value decrements the address. The device address is incremented and decremented in *dev\_datsz* units.

The *dev\_datsz* argument determines the size of data word on each access.

**Return Values** ddi\_device\_zero() returns:

DDI_SUCCESS	Successfully zeroed the data.
DDI_FAILURE	The byte count is not a multiple of <i>dev_datsz</i> .

**Context** ddi\_device\_zero() can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#)

*Writing Device Drivers*



**Name** ddi\_devid\_compare, ddi\_devid\_free, ddi\_devid\_init, ddi\_devid\_register, ddi\_devid\_sizeof, ddi\_devid\_str\_decode, ddi\_devid\_str\_encode, ddi\_devid\_str\_free, ddi\_devid\_get, ddi\_devid\_unregister, ddi\_devid\_valid – kernel interfaces for device ids

**Synopsis**

```
int ddi_devid_compare(ddi_devid_t devid1, ddi_devid_t devid2);

size_t ddi_devid_sizeof(ddi_devid_t devid);

int ddi_devid_init(dev_info_t *dip, ushort_t devid_type,
    ushort_t nbytes, void *id, ddi_devid_t *retdevid);

void ddi_devid_free(ddi_devid_t devid);

int ddi_devid_register(dev_info_t *dip, ddi_devid_t devid);

int ddi_devid_str_decode(char *devidstr, ddi_devid_t *retdevid,
    char **retminor_name);

int ddi_devid_str_encode(ddi_devid_t devid, char *minor_name);

int ddi_devid_str_free(char *devidstr);

int ddi_devid_get(dev_info_t *dip, ddi_devid_t *retdevid);

void ddi_devid_unregister(dev_info_t *dip);

int ddi_devid_valid(ddi_devid_t devid);
```

**Parameters**

<i>devid</i>	The device id address.
<i>devidstr</i>	The <i>devid</i> and <i>minor_name</i> represented as a string.
<i>devid1</i>	The first of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .
<i>devid2</i>	The second of two device id addresses to be compared calling <code>ddi_devid_compare()</code> .
<i>dip</i>	A <code>dev_info</code> pointer, which identifies the device.
<i>devid_type</i>	The following device id types may be accepted by the <code>ddi_devid_init()</code> function:
DEVID_SCSI3_WWN	World Wide Name associated with SCSI-3 devices.
DEVID_SCSI_SERIAL	Vendor ID and serial number associated with a SCSI device. Note: This may only be used if known to be unique; otherwise a fabricated device id must be used.
DEVID_ENCAP	Device ID of another device. This is for layered device driver usage.
DEVID_FAB	Fabricated device ID.
<i>minor_name</i>	The minor name to be encoded.

<i>nbytes</i>	The length in bytes of device ID.
<i>retdevid</i>	The return address of the device ID.
<i>retminor_name</i>	The return address of a minor name. Free string with <code>ddi_devid_str_free()</code> .

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The following routines are used to provide unique identifiers, device IDs, for devices. Specifically, kernel modules use these interfaces to identify and locate devices, independent of the device's physical connection or its logical device name or number.

The `ddi_devid_compare()` function compares two device IDs byte-by-byte and determines both equality and sort order.

The `ddi_devid_sizeof()` function returns the number of bytes allocated for the passed in device ID (*devid*).

The `ddi_devid_init()` function allocates memory and initializes the opaque device ID structure. This function does not store the *devid*. If the device id is not derived from the device's firmware, it is the driver's responsibility to store the *devid* on some reliable store. When a *devid\_type* of either `DEVID_SCSI3_WWN`, `DEVID_SCSI_SERIAL`, or `DEVID_ENCAP` is accepted, an array of bytes (*id*) must be passed in (*nbytes*).

When the *devid\_type* `DEVID_FAB` is used, the array of bytes (*id*) must be `NULL` and the length (*nbytes*) must be zero. The fabricated device ids, `DEVID_FAB` will be initialized with the machine's host id and a timestamp.

Drivers must free the memory allocated by this function, using the `ddi_devid_free()` function.

The `ddi_devid_free()` function frees the memory allocated for the returned *devid* by the `ddi_devid_init()` and `devid_str_decode()` functions.

The `ddi_devid_register()` function registers the device ID address (*devid*) with the DDI framework, associating it with the `dev_info` passed in (*dip*). The drivers must register device IDs at attach time. See [attach\(9E\)](#).

A driver that calls `ddi_devid_register()` should place the following line in its `driver.conf(4)` file. The presence of this property informs the system that this driver may register a *devid* for a device when device discovery is necessary.

```
ddi-devid-registrant=1;
```

The `ddi_devid_unregister()` function removes the device ID address from the `dev_info` passed in (*dip*). Drivers must use this function to unregister the device ID when devices are

being detached. This function does not free the space allocated for the device ID. The driver must free the space allocated for the device ID, using the `ddi_dev_id_free()` function. See [detach\(9E\)](#).

The `ddi_dev_id_valid()` function validates the device ID (*dev\_id*) passed in. The driver must use this function to validate any fabricated device ID that has been stored on a device.

The `ddi_dev_id_get()` function returns a pointer to the device ID structure through *retdev\_id* if there is already a registered device ID associated with the `dev_info` node. A driver can use this interface to check and get the device ID associated with the `dev_info` node. It returns `DDI_FAILURE` if no device ID is registered for the node.

The `ddi_dev_id_get()` function allocates memory for the opaque device ID structure and initializes it with the associated device ID and returns the pointer to it. The driver must free the memory allocated by this function using the `ddi_dev_id_free()` function.

The `ddi_dev_id_str_encode()` function encodes a *dev\_id* and *minor\_name* into a null-terminated ASCII string, returning a pointer to that string. If both a *dev\_id* and a *minor\_name* are non-null, then a slash (/) is used to separate the *dev\_id* from the *minor\_name* in the encoded string. If *minor\_name* is null, then only the *dev\_id* is encoded. If the *dev\_id* is null, then the special string `id0` is returned. Note that you cannot compare the returned string against another string with `strcmp()` to determine *dev\_id* equality. The returned string must be freed by calling `dev_id_str_free()`.

The `ddi_dev_id_str_decode()` function takes a string previously produced by the [dev\\_id\\_str\\_encode\(3DEVID\)](#) or `ddi_dev_id_str_encode()` function and decodes the contained device ID and *minor\_name*, allocating and returning pointers to the extracted parts through the *retdev\_id* and *retminor\_name* arguments. If the special *dev\_id* `id0` was specified then the returned device ID and *minor\_name* will both be null. A non-null returned *dev\_id* must be freed by the caller through the `ddi_dev_id_free()` function. A non-null returned *minor\_name* must be freed by calling `ddi_dev_id_str_free()`.

The `ddi_dev_id_str_free()` function is used to free all strings returned by the `ddi_dev_id` functions (the `ddi_dev_id_str_encode()` function return value and the returned *retminor\_name* argument).

**Return Values** The `ddi_dev_id_init()` function returns the following values:

- `DDI_SUCCESS`      Success.
- `DDI_FAILURE`      Out of memory. An invalid *dev\_id\_type* was passed in.

The `ddi_dev_id_valid()` function returns the following values:

- `DDI_SUCCESS`      Valid device ID.
- `DDI_FAILURE`      Invalid device ID.

The `ddi_devid_register()` function returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Failure. The device ID is already registered or the device ID is invalid.

The `ddi_devid_valid()` function returns the following values:

- `DDI_SUCCESS` Valid device ID.
- `DDI_FAILURE` Invalid device ID.

The `ddi_devid_get()` function returns the following values:

- `DDI_SUCCESS` Device ID is present and a pointer to it is returned in *retdevid*.
- `DDI_FAILURE` No device ID is defined for this `dev_info` node.

The `ddi_devid_compare()` function returns the following values:

- 1 The first device ID is less than the second device ID.
- 0 The first device ID is equal to the second device ID.
- 1 The first device ID is greater than the second device ID.

The `ddi_devid_sizeof()` function returns the size of the *devid* in bytes. If called with a null, then the number of bytes that must be allocated and initialized to determine the size of a complete device ID is returned.

The `ddi_devid_str_encode()` function returns a value of null to indicate failure. Failure can be caused by attempting to encode an invalid *devid*. If the return value is non-null then the caller must free the returned string by using the `devid_str_free()` function.

The `ddi_devid_str_decode()` function returns the following values:

- `DDI_SUCCESS` Success.
- `DDI_FAILURE` Failure; the *devidstr* string was not valid.

**Context** These functions can be called from a user or kernel context.

**See Also** `devid_get(3DEVID)`, `libdevid(3LIB)`, `driver.conf(4)`, `attach(9E)`, `detach(9E)`, `kmem_free(9F)`

*Writing Device Drivers*

- Name** ddi\_dev\_is\_sid – tell whether a device is self-identifying
- Synopsis**
- ```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_is_sid(dev_info_t *dip);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *dip* A pointer to the device's dev\_info structure.
- Description** The ddi\_dev\_is\_sid() function tells the caller whether the device described by *dip* is self-identifying, that is, a device that can unequivocally tell the system that it exists. This is useful for drivers that support both a self-identifying as well as a non-self-identifying variants of a device (and therefore must be probed).
- Return Values** DDI\_SUCCESS Device is self-identifying.  
DDI\_FAILURE Device is not self-identifying.
- Context** The ddi\_dev\_is\_sid() function can be called from user, interrupt, or kernel context.

**Examples**

```
1  ...
2  int
3  bz_probe(dev_info_t *dip)
4  {
5      ...
6      if (ddi_dev_is_sid(dip) == DDI_SUCCESS) {
7          /*
8           * This is the self-identifying version (OpenBoot).
9           * No need to probe for it because we know it is there.
10         * The existence of dip && ddi_dev_is_sid() proves this.
11         */
12         return (DDI_PROBE_DONTCARE);
13     }
14     /*
15     * Not a self-identifying variant of the device. Now we have to
16     * do some work to see whether it is really attached to the
17     * system.
18     */
19     ...
```

**See Also** [probe\(9E\)](#) *Writing Device Drivers*

**Name** ddi\_dev\_nintrs – return the number of interrupt specifications a device has

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dev_nintrs(dev_info_t *dip, int *resultp);
```

**Interface Level** Solaris DDI specific (Solaris DDI). This interface is obsolete. Use the new interrupt interfaces referenced in [Intro\(9F\)](#). Refer to *Writing Device Drivers* for more information.

**Description** The ddi\_dev\_nintrs() function returns the number of interrupt specifications a device has in \*resultp.

**Return Values** The ddi\_dev\_nintrs() function returns:

DDI\_SUCCESS      A successful return. The number of interrupt specifications that the device has is set in *resultp*.

DDI\_FAILURE      The device has no interrupt specifications.

**Context** The ddi\_dev\_nintrs() function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |

**See Also** [isa\(4\)](#), [sbus\(4\)](#), [ddi\\_add\\_intr\(9F\)](#), [ddi\\_dev\\_nregs\(9F\)](#), [ddi\\_dev\\_regsize\(9F\)](#), [Intro\(9F\)](#)

*Writing Device Drivers*

---

**Name** `ddi_dev_nregs` – return the number of register sets a device has

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_dev_nregs(dev_info_t *dip, int *resultp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

*dip* A pointer to the device's `dev_info` structure.

*resultp* Pointer to an integer that holds the number of register sets on return.

**Description** The `ddi_dev_nregs()` function returns the number of sets of registers the device has.

**Return Values** The `ddi_dev_nregs()` function returns:

`DDI_SUCCESS` A successful return. The number of register sets is returned in *resultp*.

`DDI_FAILURE` The device has no registers.

**Context** The `ddi_dev_nregs()` function can be called from user, interrupt, or kernel context.

**See Also** [ddi\\_dev\\_nintrs\(9F\)](#), [ddi\\_dev\\_regsize\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_dev\_regsz – return the size of a device's register

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dev_regsz(dev_info_t *dip, uint_t rnumber, off_t *resultp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                |                                                                                                                                                                                                                                           |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>dip</i>     | A pointer to the device's dev_info structure.                                                                                                                                                                                             |
| <i>rnumber</i> | The ordinal register number. Device registers are associated with a dev_info and are enumerated in arbitrary sets from 0 on up. The number of registers a device has can be determined from a call to <a href="#">ddi_dev_nregs(9F)</a> . |
| <i>resultp</i> | Pointer to an integer that holds the size, in bytes, of the described register (if it exists).                                                                                                                                            |

**Description** The ddi\_dev\_regsz() function returns the size, in bytes, of the device register specified by *dip* and *rnumber*. This is useful when, for example, one of the registers is a frame buffer with a varying size known only to its proms.

**Return Values** The ddi\_dev\_regsz() function returns:

|             |                                                                                                |
|-------------|------------------------------------------------------------------------------------------------|
| DDI_SUCCESS | A successful return. The size, in bytes, of the specified register, is set in <i>resultp</i> . |
| DDI_FAILURE | An invalid (nonexistent) register number was specified.                                        |

**Context** The ddi\_dev\_regsz() function can be called from user, interrupt, or kernel context.

**See Also** [ddi\\_dev\\_nintrs\(9F\)](#), [ddi\\_dev\\_nregs\(9F\)](#)

*Writing Device Drivers*



**Name** ddi\_dev\_report\_fault – Report a hardware failure

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

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

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

|                 |                                                                                                                                             |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <i>dip</i>      | Pointer to the driver's dev_info structure to which the fault report relates. (Normally the caller's own dev_info pointer).                 |
| <i>impact</i>   | One of a set of enumerated values indicating the impact of the fault on the device's ability to provide normal service.                     |
| <i>location</i> | One of a set of enumerated values indicating the location of the fault, relative to the hardware controlled by the driver specified by dip. |
| <i>message</i>  | Text of the message describing the fault being reported.                                                                                    |

**Description** This function provides a standardized mechanism through which device drivers can report hardware faults. Use of this reporting mechanism enables systems equipped with a fault management system to respond to faults discovered by a driver. On a suitably equipped system, this might include automatic failover to an alternative device and/or scheduling replacement of the faulty hardware.

The driver must indicate the impact of the fault being reported on its ability to provide service by passing one of the following values for the impact parameter:

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_SERVICE_LOST       | Indicates a total loss of service. The driver is unable to implement the normal functions of its hardware.                                                                                                                                                                                                                                                                                                             |
| DDI_SERVICE_DEGRADED   | The driver is unable to provide normal service, but can provide a partial or degraded level of service. The driver may have to make repeated attempts to perform an operation before it succeeds, or it may be running at less than its configured speed. A driver may use this value to indicate that an alternative device should be used if available, but that it can continue operation if no alternative exists. |
| DDI_SERVICE_UNAFFECTED | The service provided by the device is currently unaffected by the reported fault. This value may be used to report recovered errors for predictive failure analysis.                                                                                                                                                                                                                                                   |
| DDI_SERVICE_RESTORED   | The driver has resumed normal service, following a previous report that service was lost or degraded. This message implies that any previously reported fault condition no longer exists.                                                                                                                                                                                                                              |

The location parameter should be one of the following values:

- |                    |                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_DATAPATH_FAULT | The fault lies in the datapath between the driver and the device. The device may be unplugged, or a problem may exist in the bus on which the device resides. This value is appropriate if the device is not responding to accesses, (for example, the device may not be present) or if a call to <a href="#">ddi_check_acc_handle(9F)</a> returns DDI_FAILURE. |
| DDI_DEVICE_FAULT   | The fault lies in the device controlled by the driver. This value is appropriate if the device returns an error from a selftest function, or if the driver is able to determine that device is present and accessible, but is not functioning correctly.                                                                                                        |
| DDI_EXTERNAL_FAULT | The fault is external to the device. For example, an Ethernet driver would use this value when reporting a cable fault.                                                                                                                                                                                                                                         |

If a device returns detectably bad data during normal operation (an "impossible" value in a register or DMA status area, for example), the driver should check the associated handle using [ddi\\_check\\_acc\\_handle\(9F\)](#) or [ddi\\_check\\_dma\\_handle\(9F\)](#) before reporting the fault. If the fault is associated with the handle, the driver should specify DDI\_DATAPATH\_FAULT rather than DDI\_DEVICE\_FAULT. As a consequence of this call, the device's state may be updated to reflect the level of service currently available. See [ddi\\_get\\_devstate\(9F\)](#).

Note that if a driver calls [ddi\\_get\\_devstate\(9F\)](#) and discovers that its device is down, a fault should not be reported- the device is down as the result of a fault that has already been reported. Additionally, a driver should avoid incurring or reporting additional faults when the device is already known to be unusable. The `ddi_dev_report_fault()` call should only be used to report hardware (device) problems and should not be used to report purely software problems such as memory (or other resource) exhaustion.

**Examples** An Ethernet driver receives an error interrupt from its device if various fault conditions occur. The driver must read an error status register to determine the nature of the fault, and report it appropriately:

```
static int
xx_error_intr(xx_soft_state *ssp)
{
    ...
    error_status = ddi_get32(ssp->handle, &ssp->regs->xx_err_status);
    if (ddi_check_acc_handle(ssp->handle) != DDI_SUCCESS) {
```

```
        ddi_dev_report_fault(ssp->dip, DDI_SERVICE_LOST,  
            DDI_DATAPATH_FAULT, "register access fault");  
        return DDI_INTR_UNCLAIMED;  
    }  
    if (ssp->error_status & XX_CABLE_FAULT) {  
        ddi_dev_report_fault(ssp->dip, DDI_SERVICE_LOST,  
            DDI_EXTERNAL_FAULT, "cable fault")  
        return DDI_INTR_CLAIMED;  
    }  
    if (ssp->error_status & XX_JABBER) {  
        ddi_dev_report_fault(ssp->dip, DDI_SERVICE_DEGRADED,  
            DDI_EXTERNAL_FAULT, "jabbering detected")  
        return DDI_INTR_CLAIMED;  
    }  
    ...  
}
```

**Context** The `ddi_dev_report_fault()` function may be called from user, kernel, or interrupt context.

**See Also** [ddi\\_check\\_acc\\_handle\(9F\)](#), [ddi\\_check\\_dma\\_handle\(9F\)](#), [ddi\\_get\\_devstate\(9F\)](#)

**Name** ddi\_dma\_addr\_bind\_handle – binds an address to a DMA handle

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, struct as *as,
    caddr_t addr, size_t len, uint_t flags, int (*callback) (caddr_t) ,
    caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------------------------|------------------|---------------------------------------------------------------------------|--------------|----------------------|-----------------|------------------------------------------------|-----------------|------------------------------|--------------------|---------------------------------------------------|-------------------|-----------------------------------------------------------------------|
| <i>handle</i>      | The DMA handle previously allocated by a call to <a href="#">ddi_dma_alloc_handle(9F)</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>as</i>          | A pointer to an address space structure. This parameter should be set to NULL, which implies kernel address space.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>addr</i>        | Virtual address of the memory object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>len</i>         | Length of the memory object in bytes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>flags</i>       | Valid flags include: <table> <tr> <td>DDI_DMA_WRITE</td> <td>Transfer direction is from memory to I/O.</td> </tr> <tr> <td>DDI_DMA_READ</td> <td>Transfer direction is from I/O to memory.</td> </tr> <tr> <td>DDI_DMA_RDWR</td> <td>Both read and write.</td> </tr> <tr> <td>DDI_DMA_REDZONE</td> <td>Establish an MMU redzone at end of the object.</td> </tr> <tr> <td>DDI_DMA_PARTIAL</td> <td>Partial resource allocation.</td> </tr> <tr> <td>DDI_DMA_CONSISTENT</td> <td>Nonsequential, random, and small block transfers.</td> </tr> <tr> <td>DDI_DMA_STREAMING</td> <td>Sequential, unidirectional, block-sized, and block-aligned transfers.</td> </tr> </table> | DDI_DMA_WRITE | Transfer direction is from memory to I/O. | DDI_DMA_READ     | Transfer direction is from I/O to memory.                                 | DDI_DMA_RDWR | Both read and write. | DDI_DMA_REDZONE | Establish an MMU redzone at end of the object. | DDI_DMA_PARTIAL | Partial resource allocation. | DDI_DMA_CONSISTENT | Nonsequential, random, and small block transfers. | DDI_DMA_STREAMING | Sequential, unidirectional, block-sized, and block-aligned transfers. |
| DDI_DMA_WRITE      | Transfer direction is from memory to I/O.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_READ       | Transfer direction is from I/O to memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_RDWR       | Both read and write.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_REDZONE    | Establish an MMU redzone at end of the object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_PARTIAL    | Partial resource allocation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_CONSISTENT | Nonsequential, random, and small block transfers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_STREAMING  | Sequential, unidirectional, block-sized, and block-aligned transfers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>callback</i>    | The address of a function to call back later if resources are not currently available. The following special function addresses may also be used. <table> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table>                                                                                                                                                                                                                                                                                                  | DDI_DMA_SLEEP | Wait until resources are available.       | DDI_DMA_DONTWAIT | Do not wait until resources are available and do not schedule a callback. |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_SLEEP      | Wait until resources are available.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| DDI_DMA_DONTWAIT   | Do not wait until resources are available and do not schedule a callback.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>arg</i>         | Argument to be passed to the callback function, <i>callback</i> , if such a function is specified.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>cookiep</i>     | A pointer to the first <a href="#">ddi_dma_cookie(9S)</a> structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |
| <i>ccountp</i>     | Upon a successful return, <i>ccountp</i> points to a value representing the number of cookies for this DMA object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                           |                  |                                                                           |              |                      |                 |                                                |                 |                              |                    |                                                   |                   |                                                                       |

**Description** `ddi_dma_addr_bind_handle()` allocates DMA resources for a memory object such that a device can perform DMA to or from the object. DMA resources are allocated considering the device's DMA attributes as expressed by `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`).

`ddi_dma_addr_bind_handle()` fills in the first DMA cookie pointed to by `cookiep` with the appropriate address, length, and bus type. `*ccountp` is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved by calling `ddi_dma_nextcookie(9F)` the number of times specified by `*countp-1`.

When a DMA transfer completes, the driver frees up system DMA resources by calling `ddi_dma_unbind_handle(9F)`.

The `flags` argument contains information for mapping routines.

|                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_DMA_WRITE, DDI_DMA_READ, DDI_DMA_RDWR</code> | These flags describe the intended direction of the DMA transfer.                                                                                                                                                                                                                                                                                                                                                                              |
| <code>DDI_DMA_STREAMING</code>                         | This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the <code>minxfer</code> and <code>burstsizes</code> fields in the DMA attribute structure, <code>ddi_dma_attr(9S)</code> (see <code>ddi_dma_alloc_handle(9F)</code> ) is used to allocate the most effective hardware support for large transfers. |
| <code>DDI_DMA_CONSISTENT</code>                        | This flag should be set if the device accesses memory randomly, or if synchronization steps using <code>ddi_dma_sync(9F)</code> need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using <code>DDI_DMA_CONSISTENT</code> .                                                                                                                                    |
| <code>DDI_DMA_REDZONE</code>                           | If this flag is set, the system attempts to establish a protected red zone after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support a red zone.                                                                                                                                                                              |

**DDI\_DMA\_PARTIAL**

Setting this flag indicates the caller can accept resources for part of the object. That is, if the size of the object exceeds the resources available, only resources for a portion of the object are allocated. The system indicates this condition by returning status `DDI_DMA_PARTIAL_MAP`. At a later point, the caller can use [ddi\\_dma\\_getwin\(9F\)](#) to change the valid portion of the object for which resources are allocated. If resources were allocated for only part of the object, `ddi_dma_addr_bind_handle()` returns resources for the first DMA window. Even when `DDI_DMA_PARTIAL` is set, the system may decide to allocate resources for the entire object (less overhead) in which case `DDI_DMA_MAPPED` is returned.

The callback function *callback* indicates how a caller wants to handle the possibility of resources not being available. If *callback* is set to `DDI_DMA_DONTWAIT`, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If *callback* is set to `DDI_DMA_SLEEP`, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set and a DMA resource allocation fails, this value is assumed to be the address of a function to be called when resources become available. When the specified function is called, *arg* is passed to it as an argument. The specified callback function must return either `DDI_DMA_CALLBACK_RUNOUT` or `DDI_DMA_CALLBACK_DONE`.

`DDI_DMA_CALLBACK_RUNOUT` indicates that the callback function attempted to allocate DMA resources but failed. In this case, the callback function is put back on a list to be called again later. `DDI_DMA_CALLBACK_DONE` indicates that either the allocation of DMA resources was successful or the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are available. The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.

**Return Values** `ddi_dma_addr_bind_handle()` returns:

|                                  |                                                                                                                                                                                       |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_DMA_MAPPED</code>      | Successfully allocated resources for the entire object.                                                                                                                               |
| <code>DDI_DMA_PARTIAL_MAP</code> | Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the <code>DDI_DMA_PARTIAL</code> flag in <i>flags</i> . |
| <code>DDI_DMA_INUSE</code>       | Another I/O transaction is using the DMA handle.                                                                                                                                      |

---

|                     |                                                                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_DMA_NORESOURCES | No resources are available at the present time.                                                                                                                 |
| DDI_DMA_NOMAPPING   | The object cannot be reached by the device requesting the resources.                                                                                            |
| DDI_DMA_TOOBIG      | The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration. |

**Context** `ddi_dma_addr_bind_handle()` can be called from user, kernel, or interrupt context, except when *callback* is set to `DDI_DMA_SLEEP`, in which case it can only be called from user or kernel context.

**See Also** `ddi_dma_alloc_handle(9F)`, `ddi_dma_free_handle(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_mem_alloc(9F)`, `ddi_dma_mem_free(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_unbind_handle(9F)`, `ddi_umem_iosetup(9F)`, `ddi_dma_attr(9S)`, `ddi_dma_cookie(9S)`

#### *Writing Device Drivers*

**Notes** If the driver permits partial mapping with the `DDI_DMA_PARTIAL` flag, the number of cookies in each window may exceed the size of the device's scatter/gather list as specified in the `dma_attr_sgl len` field in the `ddi_dma_attr(9S)` structure. In this case, each set of cookies comprising a DMA window will satisfy the DMA attributes as described in the `ddi_dma_attr(9S)` structure in all aspects. The driver should set up its DMA engine and perform one transfer for each set of cookies sufficient for its scatter/gather list, up to the number of cookies for this window, before advancing to the next window using `ddi_dma_getwin(9F)`.

**Name** ddi\_dma\_addr\_setup – easier DMA setup for use with virtual addresses

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_addr_setup(dev_info_t *dip, struct as *as, caddr_t addr,
    size_t len, uint_t flags, int (*waitfp) (caddr_t), caddr_t arg,
    ddi_dma_lim_t * lim, ddi_dma_handle_t *handlep);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#) should be used instead.

**Parameters**

*dip* A pointer to the device's dev\_info structure.

*as* A pointer to an address space structure. Should be set to NULL, which implies kernel address space.

*addr* Virtual address of the memory object.

*len* Length of the memory object in bytes.

*flags* Flags that would go into the ddi\_dma\_req structure (see [ddi\\_dma\\_req\(9S\)](#)).

*waitfp* The address of a function to call back later if resources aren't available now. The special function addresses DDI\_DMA\_SLEEP and DDI\_DMA\_DONTWAIT (see [ddi\\_dma\\_req\(9S\)](#)) are taken to mean, respectively, wait until resources are available or, do not wait at all and do not schedule a callback.

*arg* Argument to be passed to a callback function, if such a function is specified.

*lim* A pointer to a DMA limits structure for this device (see [ddi\\_dma\\_lim\\_sparc\(9S\)](#) or [ddi\\_dma\\_lim\\_x86\(9S\)](#)). If this pointer is NULL, a default set of DMA limits is assumed.

*handlep* Pointer to a DMA handle. See [ddi\\_dma\\_setup\(9F\)](#) for a discussion of handle.

**Description** The ddi\_dma\_addr\_setup() function is an interface to [ddi\\_dma\\_setup\(9F\)](#). It uses its arguments to construct an appropriate ddi\_dma\_req structure and calls [ddi\\_dma\\_setup\(9F\)](#) with it.

**Return Values** See [ddi\\_dma\\_setup\(9F\)](#) for the possible return values for this function.

**Context** The ddi\_dma\_addr\_setup() can be called from user, interrupt, or kernel context, except when *waitfp* is set to DDI\_DMA\_SLEEP, in which case it cannot be called from interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |



**See Also** [attributes\(5\)](#), [ddi\\_dma\\_buf\\_setup\(9F\)](#), [ddi\\_dma\\_free\(9F\)](#), [ddi\\_dma\\_htoc\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_iopb\\_alloc\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_alloc\_handle – allocate DMA handle

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_alloc_handle(dev_info_t *dip, ddi_dma_attr_t *attr,
    int (*callback) (caddr_t), caddr_t arg, ddi_dma_handle_t *handlep);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                  |                                                                                                                                            |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <i>dip</i>       | Pointer to the device's dev_info structure.                                                                                                |
| <i>attr</i>      | Pointer to a DMA attribute structure for this device (see <a href="#">ddi_dma_attr(9S)</a> ).                                              |
| <i>callback</i>  | The address of a function to call back later if resources aren't available now. The following special function addresses may also be used. |
| DDI_DMA_SLEEP    | Wait until resources are available.                                                                                                        |
| DDI_DMA_DONTWAIT | Do not wait until resources are available and do not schedule a callback.                                                                  |
| <i>arg</i>       | Argument to be passed to a callback function, if such a function is specified.                                                             |
| <i>handlep</i>   | Pointer to the DMA handle to be initialized.                                                                                               |

**Description** ddi\_dma\_alloc\_handle() allocates a new DMA handle. A DMA handle is an opaque object used as a reference to subsequently allocated DMA resources. ddi\_dma\_alloc\_handle() accepts as parameters the device information referred to by *dip* and the device's DMA attributes described by a [ddi\\_dma\\_attr\(9S\)](#) structure. A successful call to ddi\_dma\_alloc\_handle() fills in the value pointed to by *handlep*. A DMA handle must only be used by the device for which it was allocated and is only valid for one I/O transaction at a time.

The callback function, *callback*, indicates how a caller wants to handle the possibility of resources not being available. If *callback* is set to DDI\_DMA\_DONTWAIT, then the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If *callback* is set to DDI\_DMA\_SLEEP, then the caller wishes to have the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be a function to call at a later time when resources may become available. When the specified function is called, it is passed *arg* as an argument. The specified callback function must return either DDI\_DMA\_CALLBACK\_RUNOUT or DDI\_DMA\_CALLBACK\_DONE. DDI\_DMA\_CALLBACK\_RUNOUT indicates that the callback routine attempted to allocate DMA resources but failed to do so, in which case the callback function is put back on a list to be called again later. DDI\_DMA\_CALLBACK\_DONE indicates either success at allocating DMA resources or the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions that are accessible from interrupt context is available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, and so forth.

When a DMA handle is no longer needed, [ddi\\_dma\\_free\\_handle\(9F\)](#) must be called to free the handle.

**Return Values** `ddi_dma_alloc_handle()` returns:

|                                  |                                                                                                                                                   |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_SUCCESS</code>         | Successfully allocated a new DMA handle.                                                                                                          |
| <code>DDI_DMA_BADATTR</code>     | The attributes specified in the <a href="#">ddi_dma_attr(9S)</a> structure make it impossible for the system to allocate potential DMA resources. |
| <code>DDI_DMA_NORESOURCES</code> | No resources are available.                                                                                                                       |

**Context** `ddi_dma_alloc_handle()` can be called from user, kernel, or interrupt context, except when *callback* is set to `DDI_DMA_SLEEP`, in which case it can be called from user or kernel context only.

**See Also** [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_burstsizes\(9F\)](#), [ddi\\_dma\\_free\\_handle\(9F\)](#), [ddi\\_dma\\_unbind\\_handle\(9F\)](#), [ddi\\_dma\\_attr\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_buf\_bind\_handle – binds a system buffer to a DMA handle

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle, struct buf *bp,
    uint_t flags, int (*callback)(caddr_t), caddr_t
    arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------------------------------------|------------------|---------------------------------------------------------------------------|--------------|---------------------|-----------------|------------------------------------------------|-----------------|-----------------------------|--------------------|---------------------------------------------------|-------------------|-----------------------------------------------------------------------|
| <i>handle</i>      | The DMA handle previously allocated by a call to <a href="#">ddi_dma_alloc_handle(9F)</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>bp</i>          | A pointer to a system buffer structure (see <a href="#">buf(9S)</a> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>flags</i>       | Valid flags include: <table> <tr> <td>DDI_DMA_WRITE</td> <td>Transfer direction is from memory to I/O</td> </tr> <tr> <td>DDI_DMA_READ</td> <td>Transfer direction is from I/O to memory</td> </tr> <tr> <td>DDI_DMA_RDWR</td> <td>Both read and write</td> </tr> <tr> <td>DDI_DMA_REDZONE</td> <td>Establish an MMU redzone at end of the object.</td> </tr> <tr> <td>DDI_DMA_PARTIAL</td> <td>Partial resource allocation</td> </tr> <tr> <td>DDI_DMA_CONSISTENT</td> <td>Nonsequential, random, and small block transfers.</td> </tr> <tr> <td>DDI_DMA_STREAMING</td> <td>Sequential, unidirectional, block-sized, and block-aligned transfers.</td> </tr> </table> | DDI_DMA_WRITE | Transfer direction is from memory to I/O | DDI_DMA_READ     | Transfer direction is from I/O to memory                                  | DDI_DMA_RDWR | Both read and write | DDI_DMA_REDZONE | Establish an MMU redzone at end of the object. | DDI_DMA_PARTIAL | Partial resource allocation | DDI_DMA_CONSISTENT | Nonsequential, random, and small block transfers. | DDI_DMA_STREAMING | Sequential, unidirectional, block-sized, and block-aligned transfers. |
| DDI_DMA_WRITE      | Transfer direction is from memory to I/O                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_READ       | Transfer direction is from I/O to memory                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_RDWR       | Both read and write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_REDZONE    | Establish an MMU redzone at end of the object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_PARTIAL    | Partial resource allocation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_CONSISTENT | Nonsequential, random, and small block transfers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_STREAMING  | Sequential, unidirectional, block-sized, and block-aligned transfers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>callback</i>    | The address of a function to call back later if resources are not available now. The following special function addresses may also be used. <table> <tr> <td>DDI_DMA_SLEEP</td> <td>Wait until resources are available.</td> </tr> <tr> <td>DDI_DMA_DONTWAIT</td> <td>Do not wait until resources are available and do not schedule a callback.</td> </tr> </table>                                                                                                                                                                                                                                                                                                    | DDI_DMA_SLEEP | Wait until resources are available.      | DDI_DMA_DONTWAIT | Do not wait until resources are available and do not schedule a callback. |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_SLEEP      | Wait until resources are available.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| DDI_DMA_DONTWAIT   | Do not wait until resources are available and do not schedule a callback.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>arg</i>         | Argument to be passed to the callback function, <i>callback</i> , if such a function is specified.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>cookiep</i>     | A pointer to the first <a href="#">ddi_dma_cookie(9S)</a> structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |
| <i>ccountp</i>     | Upon a successful return, <i>ccountp</i> points to a value representing the number of cookies for this DMA object.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |               |                                          |                  |                                                                           |              |                     |                 |                                                |                 |                             |                    |                                                   |                   |                                                                       |

**Description** ddi\_dma\_buf\_bind\_handle() allocates DMA resources for a system buffer such that a device can perform DMA to or from the buffer. DMA resources are allocated considering the device's DMA attributes as expressed by [ddi\\_dma\\_attr\(9S\)](#) (see [ddi\\_dma\\_alloc\\_handle\(9F\)](#)).

`ddi_dma_buf_bind_handle()` fills in the first DMA cookie pointed to by *cookiep* with the appropriate address, length, and bus type. *\*countp* is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved by calling `ddi_dma_nextcookie(9F)` *\*countp*-1 times.

When a DMA transfer completes, the driver should free up system DMA resources by calling `ddi_dma_unbind_handle(9F)`.

The *flags* argument contains information for mapping routines.

|                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_DMA_WRITE, DDI_DMA_READ, DDI_DMA_RDWR</code> | These flags describe the intended direction of the DMA transfer.                                                                                                                                                                                                                                                                                                                                                                              |
| <code>DDI_DMA_STREAMING</code>                         | This flag should be set if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the <code>minxfer</code> and <code>burstsizes</code> fields in the DMA attribute structure, <code>ddi_dma_attr(9S)</code> (see <code>ddi_dma_alloc_handle(9F)</code> ) is used to allocate the most effective hardware support for large transfers. |
| <code>DDI_DMA_CONSISTENT</code>                        | This flag should be set if the device accesses memory randomly, or if synchronization steps using <code>ddi_dma_sync(9F)</code> need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using <code>DDI_DMA_CONSISTENT</code> .                                                                                                                                    |
| <code>DDI_DMA_REDZONE</code>                           | If this flag is set, the system attempts to establish a protected red zone after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support a red zone.                                                                                                                                                                              |
| <code>DDI_DMA_PARTIAL</code>                           | Setting this flag indicates the caller can accept resources for part of the object. That is, if the size of the object exceeds the resources available, only resources for a                                                                                                                                                                                                                                                                  |

portion of the object are allocated. The system indicates this condition returning status `DDI_DMA_PARTIAL_MAP`. At a later point, the caller can use [ddi\\_dma\\_getwin\(9F\)](#) to change the valid portion of the object for which resources are allocated. If resources were allocated for only part of the object, `ddi_dma_addr_bind_handle()` returns resources for the first DMA window. Even when `DDI_DMA_PARTIAL` is set, the system may decide to allocate resources for the entire object (less overhead) in which case `DDI_DMA_MAPPED` is returned.

The callback function, *callback*, indicates how a caller wants to handle the possibility of resources not being available. If *callback* is set to `DDI_DMA_DONTWAIT`, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If *callback* is set to `DDI_DMA_SLEEP`, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be the address of a function to call at a later time when resources may become available. When the specified function is called, it is passed *arg* as an argument. The specified callback function must return either `DDI_DMA_CALLBACK_RUNOUT` or `DDI_DMA_CALLBACK_DONE`. `DDI_DMA_CALLBACK_RUNOUT` indicates that the callback function attempted to allocate DMA resources but failed to do so. In this case the callback function is put back on a list to be called again later. `DDI_DMA_CALLBACK_DONE` indicates either a successful allocation of DMA resources or that the driver no longer wishes to retry.

The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, etc.

**Return Values** `ddi_dma_buf_bind_handle()` returns:

|                                  |                                                                                                                                                                                       |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_DMA_MAPPED</code>      | Successfully allocated resources for the entire object.                                                                                                                               |
| <code>DDI_DMA_PARTIAL_MAP</code> | Successfully allocated resources for a part of the object. This is acceptable when partial transfers are permitted by setting the <code>DDI_DMA_PARTIAL</code> flag in <i>flags</i> . |
| <code>DDI_DMA_INUSE</code>       | Another I/O transaction is using the DMA handle.                                                                                                                                      |
| <code>DDI_DMA_NORESOURCES</code> | No resources are available at the present time.                                                                                                                                       |
| <code>DDI_DMA_NOMAPPING</code>   | The object cannot be reached by the device requesting the resources.                                                                                                                  |

---

**DDI\_DMA\_TOOBIG** The object is too big. A request of this size can never be satisfied on this particular system. The maximum size varies depending on machine and configuration.

**Context** `ddi_dma_buf_bind_handle()` can be called from user, kernel, or interrupt context, except when *callback* is set to `DDI_DMA_SLEEP`, in which case it can be called from user or kernel context only.

**See Also** `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`, `ddi_dma_free_handle(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_unbind_handle(9F)`, `buf(9S)`, `ddi_dma_attr(9S)`, `ddi_dma_cookie(9S)`

#### *Writing Device Drivers*

**Notes** If the driver permits partial mapping with the `DDI_DMA_PARTIAL` flag, the number of cookies in each window may exceed the size of the device's scatter/gather list as specified in the `dma_attr_sgl len` field in the `ddi_dma_attr(9S)` structure. In this case, each set of cookies comprising a DMA window will satisfy the DMA attributes as described in the `ddi_dma_attr(9S)` structure in all aspects. The driver should set up its DMA engine and perform one transfer for each set of cookies sufficient for its scatter/gather list, up to the number of cookies for this window, before advancing to the next window using `ddi_dma_getwin(9F)`.

**Name** ddi\_dma\_buf\_setup – easier DMA setup for use with buffer structures

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_buf_setup(dev_info_t *dip, struct buf *bp, uint_t flags,
    int (*waitfp) (caddr_t),, caddr_t arg, ddi_dma_lim_t *lim,
    ddi_dma_handle_t *handlep);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#) should be used instead.

**Parameters**

- dip* A pointer to the device's dev\_info structure.
- bp* A pointer to a system buffer structure (see [buf\(9S\)](#)).
- flags* Flags that go into a ddi\_dma\_req structure (see [ddi\\_dma\\_req\(9S\)](#)).
- waitfp* The address of a function to call back later if resources aren't available now. The special function addresses DDI\_DMA\_SLEEP and DDI\_DMA\_DONTWAIT (see [ddi\\_dma\\_req\(9S\)](#)) are taken to mean, respectively, wait until resources are available, or do not wait at all and do not schedule a callback.
- arg* Argument to be passed to a callback function, if such a function is specified.
- lim* A pointer to a DMA limits structure for this device (see [ddi\\_dma\\_lim\\_sparc\(9S\)](#) or [ddi\\_dma\\_lim\\_x86\(9S\)](#)). If this pointer is NULL, a default set of DMA limits is assumed.
- handlep* Pointer to a DMA handle. See [ddi\\_dma\\_setup\(9F\)](#) for a discussion of handle.

**Description** The ddi\_dma\_buf\_setup() function is an interface to [ddi\\_dma\\_setup\(9F\)](#). It uses its arguments to construct an appropriate ddi\_dma\_req structure and calls ddi\_dma\_setup() with it.

**Return Values** See [ddi\\_dma\\_setup\(9F\)](#) for the possible return values for this function.

**Context** The ddi\_dma\_buf\_setup() function can be called from user, interrupt, or kernel context, except when *waitfp* is set to DDI\_DMA\_SLEEP, in which case it cannot be called from interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_addr\\_setup\(9F\)](#), [ddi\\_dma\\_free\(9F\)](#), [ddi\\_dma\\_htoc\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [physio\(9F\)](#), [buf\(9S\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)



*Writing Device Drivers*

**Name** ddi\_dma\_burstsizes – find out the allowed burst sizes for a DMA mapping

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_burstsizes(ddi_dma_handle_t handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *handle* A DMA handle that was filled in by a successful call to [ddi\\_dma\\_setup\(9F\)](#).

**Description** `ddi_dma_burstsizes()` returns the allowed burst sizes for a DMA mapping. This value is derived from the `dlim_burstsizes` member of the [ddi\\_dma\\_lim\\_sparc\(9S\)](#) structure, but it shows the allowable burstsizes *after* imposing on it the limitations of other device layers in addition to device's own limitations.

**Return Values** `ddi_dma_burstsizes()` returns a binary encoded value of the allowable DMA burst sizes. See [ddi\\_dma\\_lim\\_sparc\(9S\)](#) for a discussion of DMA burst sizes.

**Context** This function can be called from user or interrupt context.

**See Also** [ddi\\_dma\\_devalign\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** `ddi_dma_coff` – convert a DMA cookie to an offset within a DMA handle.

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_coff(ddi_dma_handle_t handle, ddi_dma_cookie_t *cookiep,
                off_t *offp);
```

**Interface Level** Solaris SPARC DDI (Solaris SPARC DDI). This interface is obsolete.

**Parameters**

*handle*      The *handle* filled in by a call to `ddi_dma_setup(9F)`.

*cookiep*      A pointer to a DMA cookie (see `ddi_dma_cookie(9S)`) that contains the appropriate address, length and bus type to be used in programming the DMA engine.

*offp*          A pointer to an offset to be filled in.

**Description** The `ddi_dma_coff()` function converts the values in DMA cookie pointed to by *cookiep* to an offset (in bytes) from the beginning of the object that the DMA handle has mapped.

The `ddi_dma_coff()` function allows a driver to update a DMA cookie with values it reads from its device's DMA engine after a transfer completes and convert that value into an offset into the object that is mapped for DMA.

**Return Values** The `ddi_dma_coff()` function returns:

`DDI_SUCCESS`      Successfully filled in *offp*.

`DDI_FAILURE`      Failed to successfully fill in *offp*.

**Context** The `ddi_dma_coff()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |

**See Also** `ddi_dma_setup(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_cookie(9S)`

*Writing Device Drivers*

**Name** ddi\_dma\_curwin – report current DMA window offset and size

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_curwin(ddi_dma_handle_t handle, off_t *offp, uint_t *lenp);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_getwin\(9F\)](#) should be used instead.

**Parameters**

*handle* The DMA handle filled in by a call to [ddi\\_dma\\_setup\(9F\)](#).

*offp* A pointer to a value which will be filled in with the current offset from the beginning of the object that is mapped for DMA.

*lenp* A pointer to a value which will be filled in with the size, in bytes, of the current window onto the object that is mapped for DMA.

**Description** The `ddi_dma_curwin()` function reports the current DMA window offset and size. If a DMA mapping allows partial mapping, that is if the `DDI_DMA_PARTIAL` flag in the [ddi\\_dma\\_req\(9S\)](#) structure is set, its current (effective) DMA window offset and size can be obtained by a call to `ddi_dma_curwin()`.

**Return Values** The `ddi_dma_curwin()` function returns:

`DDI_SUCCESS` The current length and offset can be established.

`DDI_FAILURE` Otherwise.

**Context** The `ddi_dma_curwin()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_getwin\(9F\)](#), [ddi\\_dma\\_movwin\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_devalign – find DMA mapping alignment and minimum transfer size

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_devalign(ddi_dma_handle_t handle, uint_t *alignment,
    uint_t *minxfr);
```

**Interface Level** Solaris DDI specific (Solaris DDI). This interface is obsolete.

**Parameters**

- handle* The DMA handle filled in by a successful call to [ddi\\_dma\\_setup\(9F\)](#).
- alignment* A pointer to an unsigned integer to be filled in with the minimum required alignment for DMA. The alignment is guaranteed to be a power of two.
- minxfr* A pointer to an unsigned integer to be filled in with the minimum effective transfer size (see [ddi\\_iomin\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#) and [ddi\\_dma\\_lim\\_x86\(9S\)](#)). This also is guaranteed to be a power of two.

**Description** The `ddi_dma_devalign()` function determines after a successful DMA mapping (see [ddi\\_dma\\_setup\(9F\)](#)) the minimum required data alignment and minimum DMA transfer size.

**Return Values** The `ddi_dma_devalign()` function returns:

`DDI_SUCCESS` The *alignment* and *minxfr* values have been filled.  
`DDI_FAILURE` The handle was illegal.

**Context** The `ddi_dma_devalign()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |

**See Also** [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_iomin\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dmae, ddi\_dmae\_alloc, ddi\_dmae\_release, ddi\_dmae\_prog, ddi\_dmae\_disable, ddi\_dmae\_enable, ddi\_dmae\_stop, ddi\_dmae\_getcnt, ddi\_dmae\_1stparty, ddi\_dmae\_getlim, ddi\_dmae\_getattr – system DMA engine functions

**Synopsis**

```
int ddi_dmae_alloc(dev_info_t *dip, int chnl, int (*callback) (caddr_t),
    caddr_t arg);

int ddi_dmae_release(dev_info_t *dip, int chnl);

int ddi_dmae_prog(dev_info_t *dip, struct ddi_dmae_req *dmaereqp,
    ddi_dma_cookie_t *cookiep, int chnl);

int ddi_dmae_disable(dev_info_t *dip, int chnl);

int ddi_dmae_enable(dev_info_t *dip, int chnl);

int ddi_dmae_stop(dev_info_t *dip, int chnl);

int ddi_dmae_getcnt(dev_info_t *dip, int chnl, int *countp);

int ddi_dmae_1stparty(dev_info_t *dip, int chnl);

int ddi_dmae_getlim(dev_info_t *dip, ddi_dma_lim_t *limitsp);

int ddi_dmae_getattr(dev_info_t *dip, ddi_dma_attr_t *attrp);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The `ddi_dmae_getlim()` interface, described below, is obsolete. Use `ddi_dmae_getattr()`, also described below, to replace it.

**Parameters**

*dip* A `dev_info` pointer that identifies the device.

*chnl* A DMA channel number. On ISA buses this number must be 0, 1, 2, 3, 5, 6, or 7.

*callback* The address of a function to call back later if resources are not currently available. The following special function addresses may also be used:

|                  |                                                                           |
|------------------|---------------------------------------------------------------------------|
| DDI_DMA_SLEEP    | Wait until resources are available.                                       |
| DDI_DMA_DONTWAIT | Do not wait until resources are available and do not schedule a callback. |

*arg* Argument to be passed to the callback function, if specified.

*dmaereqp* A pointer to a DMA engine request structure. See [ddi\\_dmae\\_req\(9S\)](#).

*cookiep* A pointer to a [ddi\\_dma\\_cookie\(9S\)](#) object, obtained from [ddi\\_dma\\_segtocookie\(9F\)](#), which contains the address and count.

*countp* A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.

*limitsp* A pointer to a DMA limit structure. See [ddi\\_dma\\_lim\\_x86\(9S\)](#).

*attrp* A pointer to a DMA attribute structure. See [ddi\\_dma\\_attr\(9S\)](#).

**Description** There are three possible ways that a device can perform DMA engine functions:

- Bus master DMA      If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from [ddi\\_dma\\_segtocookie\(9F\)](#). See [ddi\\_dma\\_cookie\(9S\)](#) for a description of a DMA cookie.
- Third-party DMA      This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except `ddi_dmae_1stparty()`, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.
- First-party DMA      Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the `ddi_dmae_1stparty()` function may be used to perform whatever configuration is necessary to enable this mode.

`ddi_dmae_alloc()` The `ddi_dmae_alloc()` function is used to acquire a DMA channel of the system DMA engine. `ddi_dmae_alloc()` allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using `ddi_dmae_release()` after completion of the DMA operation. In any case, the channel must be released before the driver successfully detaches. See [detach\(9E\)](#). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of *callback* determines what action will be taken. If the value of *callback* is `DDI_DMA_DONTWAIT`, `ddi_dmae_alloc()` will return immediately. The value `DDI_DMA_SLEEP` will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, `ddi_dmae_alloc()` returns immediately, and when resources might have become available, the callback function is called (with the argument *arg*) from interrupt context. When the callback function is called, it should attempt to allocate the DMA channel again. If it succeeds or no longer needs the channel, it must return the value `DDI_DMA_CALLBACK_DONE`. If it tries to allocate the channel but fails to do so, it must return the value `DDI_DMA_CALLBACK_RUNOUT`. In this case, the callback function is put back on a list to be called again later.

`ddi_dmae_prog()` The `ddi_dmae_prog()` function programs the DMA channel for a DMA transfer. The `ddi_dmae_req` structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to `ddi_dmae_prog()` may specify a value of `NULL` for *dmaereqp* if no changes to the

programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing `ddi_dmae_prog()` a cookie obtained from `ddi_dma_segtocookie(9F)`. Other DMA engine parameters are specified by the DMA engine request structure passed in through `dmaereqp`. The fields of that structure are documented in `ddi_dmae_req(9S)`.

Before using `ddi_dmae_prog()`, you must allocate system DMA resources using DMA setup functions such as `ddi_dma_buf_setup(9F)`. `ddi_dma_segtocookie(9F)` can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to `ddi_dmae_prog()`.

`ddi_dmae_disable()` The `ddi_dmae_disable()` function disables the DMA channel so that it no longer responds to a device's DMA service requests.

`ddi_dmae_enable()` The `ddi_dmae_enable()` function enables the DMA channel for operation. This may be used to re-enable the channel after a call to `ddi_dmae_disable()`. The channel is automatically enabled after successful programming by `ddi_dmae_prog()`.

`ddi_dmae_stop()` The `ddi_dmae_stop()` function disables the channel and terminates any active operation.

`ddi_dmae_getcnt()` The `ddi_dmae_getcnt()` function examines the count register of the DMA channel and sets `*countp` to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

`ddi_dmae_1stparty()` In the case of ISA buses, `ddi_dmae_1stparty()` configures a channel in the system's DMA engine to operate in a "slave" ("cascade") mode.

When operating in `ddi_dmae_1stparty()` mode, the DMA channel must first be allocated using `ddi_dmae_alloc()` and then configured using `ddi_dmae_1stparty()`. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from `ddi_dma_segtocookie(9F)`.

`ddi_dmae_getlim()` This function is obsolete. Use `ddi_dmae_getattr()`, described below, instead.

The `ddi_dmae_getlim()` function fills in the DMA limit structure, pointed to by `limitsp`, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use `ddi_dmae_getlim()`. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see `ddi_dma_nextseg(9F)` and `ddi_dma_nextwin(9F)`). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by `ddi_dmae_getlim()`. After calling `ddi_dmae_getlim()`, a driver must examine, and possibly set, the size of the DMA engine's scatter/gather list to determine



whether DMA chaining will be used. See [ddi\\_dma\\_lim\\_x86\(9S\)](#) and [ddi\\_dmae\\_req\(9S\)](#) for additional information on scatter/gather DMA.

`ddi_dmae_getattr()` The `ddi_dmae_getattr()` function fills in the DMA attribute structure, pointed to by *attrp*, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use `ddi_dmae_getattr()`. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies. See [ddi\\_dma\\_nextcookie\(9F\)](#) and [ddi\\_dma\\_getwin\(9F\)](#).

**Return Values**

|                     |                                                                                                                                                         |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_SUCCESS         | Upon success, for all of these routines.                                                                                                                |
| DDI_FAILURE         | May be returned due to invalid arguments.                                                                                                               |
| DDI_DMA_NORESOURCES | May be returned by <code>ddi_dmae_alloc()</code> if the requested resources are not available and the value of <i>dmae_waitfp</i> is not DDI_DMA_SLEEP. |

**Context** If `ddi_dmae_alloc()` is called from interrupt context, then its *dmae\_waitfp* argument and the callback function must not have the value DDI\_DMA\_SLEEP. Otherwise, all these routines can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTETYPE | ATTRIBUTEVALUE |
|---------------|----------------|
| Architecture  | x86            |

**See Also** [isa\(4\)](#), [attributes\(5\)](#), [ddi\\_dma\\_buf\\_setup\(9F\)](#), [ddi\\_dma\\_getwin\(9F\)](#), [ddi\\_dma\\_nextcookie\(9F\)](#), [ddi\\_dma\\_nextseg\(9F\)](#), [ddi\\_dma\\_nextwin\(9F\)](#), [ddi\\_dma\\_segtocookie\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_attr\(9S\)](#), [ddi\\_dma\\_cookie\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#), [ddi\\_dmae\\_req\(9S\)](#)

**Name** ddi\_dma\_free – release system DMA resources

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_free(ddi_dma_handle_t handle);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_free\\_handle\(9F\)](#) should be used instead.

**Parameters** *handle* The handle filled in by a call to [ddi\\_dma\\_setup\(9F\)](#).

**Description** The `ddi_dma_free()` function releases system DMA resources set up by [ddi\\_dma\\_setup\(9F\)](#). When a DMA transfer completes, the driver should free up system DMA resources established by a call to [ddi\\_dma\\_setup\(9F\)](#). This is done by a call to `ddi_dma_free()`. `ddi_dma_free()` does an implicit [ddi\\_dma\\_sync\(9F\)](#) for you so any further synchronization steps are not necessary.

**Return Values** The `ddi_dma_free()` function returns:

DDI\_SUCCESS Successfully released resources

DDI\_FAILURE Failed to free resources

**Context** The `ddi_dma_free()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_addr\\_setup\(9F\)](#), [ddi\\_dma\\_buf\\_setup\(9F\)](#), [ddi\\_dma\\_free\\_handle\(9F\)](#), [ddi\\_dma\\_htoc\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_free\_handle – free DMA handle

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_dma_free_handle(ddi_dma_handle_t *handle);
```

**Parameters** *handle* A pointer to the DMA handle previously allocated by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** `ddi_dma_free_handle()` destroys the DMA handle pointed to by *handle*. Any further references to the DMA handle will have undefined results. Note that [ddi\\_dma\\_unbind\\_handle\(9F\)](#) must be called prior to `ddi_dma_free_handle()` to free any resources the system may be caching on the handle.

**Context** `ddi_dma_free_handle()` can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_dma\\_alloc\\_handle\(9F\)](#), [ddi\\_dma\\_unbind\\_handle\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_get\_attr – get the device DMA attribute structure from a DMA handle

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_get_attr(ddi_dma_handle_t handle, ddi_dma_attr_t *attrp);
```

**Interface Level** Solaris DDI specific (Solaris DDI). This interface is obsolete.

**Parameters** *handle* The handle filled in by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).  
*attrp* Pointer to a buffer suitable for holding a DMA attribute structure. See [ddi\\_dma\\_attr\(9S\)](#).

**Description** `ddi_dma_get_attr()` is used to get a [ddi\\_dma\\_attr\(9S\)](#) structure. This structure describes the attributes of the DMA data path to which any memory object bound to the given handle will be subject.

**Return Values** `DDI_SUCCESS` Successfully passed back attribute structure in buffer pointed to by *attrp*.  
`DDI_DMA_BADATTR` A valid attribute structure could not be passed back.

**Context** `ddi_dma_get_attr()` can be called from any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |

**See Also** [ddi\\_dma\\_alloc\\_handle\(9F\)](#), [ddi\\_dma\\_attr\(9S\)](#)

**Name** ddi\_dma\_getwin – activate a new DMA window

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_getwin(ddi_dma_handle_t handle, uint_t win,
                  off_t *offp, size_t *lenp, ddi_dma_cookie_t *cookiep,
                  uint_t *ccountp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                |                                                                                                                                                   |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>handle</i>  | The DMA handle previously allocated by a call to <a href="#">ddi_dma_alloc_handle(9F)</a> .                                                       |
| <i>win</i>     | Number of the window to activate.                                                                                                                 |
| <i>offp</i>    | Pointer to an offset. Upon a successful return, <i>offp</i> will contain the new offset indicating the beginning of the window within the object. |
| <i>lenp</i>    | Upon a successful return, <i>lenp</i> will contain the size, in bytes, of the current window.                                                     |
| <i>cookiep</i> | A pointer to the first <a href="#">ddi_dma_cookie(9S)</a> structure.                                                                              |
| <i>ccountp</i> | Upon a successful return, <i>ccountp</i> will contain the number of cookies for this DMA window.                                                  |

**Description** `ddi_dma_getwin()` activates a new DMA window. If a DMA resource allocation request returns `DDI_DMA_PARTIAL_MAP` indicating that resources for less than the entire object were allocated, the current DMA window can be changed by a call to `ddi_dma_getwin()`.

The caller must first determine the number of DMA windows, *N*, using [ddi\\_dma\\_numwin\(9F\)](#). `ddi_dma_getwin()` takes a DMA window number from the range  $[0..N-1]$  as the parameter *win* and makes it the current DMA window.

`ddi_dma_getwin()` fills in the first DMA cookie pointed to by *cookiep* with the appropriate address, length, and bus type. *\*ccountp* is set to the number of DMA cookies representing this DMA object. Subsequent DMA cookies must be retrieved using [ddi\\_dma\\_nextcookie\(9F\)](#).

`ddi_dma_getwin()` takes care of underlying resource synchronizations required to shift the window. However accessing the data prior to or after moving the window requires further synchronization steps using [ddi\\_dma\\_sync\(9F\)](#).

`ddi_dma_getwin()` is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, the interrupt routine returns without invoking another DMA transfer. Otherwise, it calls `ddi_dma_getwin()` to shift the current window and start another DMA transfer.

**Return Values** `ddi_dma_getwin()` returns:

`DDI_SUCCESS` Resources for the specified DMA window are allocated.

`DDI_FAILURE` *win* is not a valid window index.

**Context** `ddi_dma_getwin()` can be called from user, kernel, or interrupt context.

**See Also** `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`,  
`ddi_dma_buf_bind_handle(9F)`, `ddi_dma_nextcookie(9F)`, `ddi_dma_numwin(9F)`,  
`ddi_dma_sync(9F)`, `ddi_dma_unbind_handle(9F)`, `ddi_dma_cookie(9S)`

*Writing Device Drivers*

**Name** ddi\_dma\_htoc – convert a DMA handle to a DMA address cookie

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_htoc(ddi_dma_handle_t handle, off_t off,
                ddi_dma_cookie_t *cookiep);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#) or [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#) should be used instead.

**Parameters**

- handle* The handle filled in by a call to [ddi\\_dma\\_setup\(9F\)](#).
- off* An offset into the object that *handle* maps.
- cookiep* A pointer to a [ddi\\_dma\\_cookie\(9S\)](#) structure.

**Description** The `ddi_dma_htoc()` function takes a DMA handle (established by [ddi\\_dma\\_setup\(9F\)](#)), and fills in the cookie pointed to by *cookiep* with the appropriate address, length, and bus type to be used to program the DMA engine.

**Return Values** The `ddi_dma_htoc()` function returns:

`DDI_SUCCESS` Successfully filled in the cookie pointed to by *cookiep*.  
`DDI_FAILURE` Failed to successfully fill in the cookie.

**Context** The `ddi_dma_htoc()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_addr\\_setup\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_buf\\_setup\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_cookie\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_mem\_alloc – allocate memory for DMA transfer

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
    ddi_device_acc_attr_t *accattrp, uint_t flags,
    int (*waitfp) (caddr_t), caddr_t arg, caddr_t *kaddrp,
    size_t *real_length, ddi_acc_handle_t *handlep);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *handle* The DMA handle previously allocated by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).

*length* The length in bytes of the desired allocation.

*accattrp* Pointer to a [ddi\\_device\\_acc\\_attr\(\)](#) structure of the device. See [ddi\\_device\\_acc\\_attr\(9S\)](#). The value in `devacc_attr_dataorder` is ignored in the current release. The value in `devacc_attr_endian_flags` is meaningful on the SPARC architecture only.

*flags* Used to determine the data transfer mode and/or the cache attribute.

Possible values of the data transfer are:

|                    |                                                                       |
|--------------------|-----------------------------------------------------------------------|
| DDI_DMA_STREAMING  | Sequential, unidirectional, block-sized, and block-aligned transfers. |
| DDI_DMA_CONSISTENT | Nonsequential transfers of small objects.                             |

Possible values of the cache attribute are:

|                          |                                                                                                                                                                                                                                               |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IOMEM_DATA_CACHED        | The CPU can cache the data it fetches and push it to memory at a later time. This is the default attribute that is used if no cache attributes are specified.                                                                                 |
| IOMEM_DATA_UC_WR_COMBINE | The CPU never caches the data, but writes can occur out of order or can be combined. Reordering is implied.<br><br>If <code>IOMEM_DATA_UC_WR_COMBINE</code> is specified but not supported, <code>IOMEM_DATA_UNCACHED</code> is used instead. |
| IOMEM_DATA_UNCACHED      | The CPU never caches data, but has uncacheable access to memory. Strict ordering is implied.                                                                                                                                                  |



The cache attributes are mutually exclusive. Any combination of the values leads to a failure. On the SPARC architecture, only IOMEM\_DATA\_CACHED is meaningful. Others lead to a failure.

*waitfp* The address of a function to call back later if resources are not available now. The callback function indicates how a caller wants to handle the possibility of resources not being available. If callback is set to DDI\_DMA\_DONTWAIT, the caller does not care if the allocation fails, and can handle an allocation failure appropriately. If callback is set to DDI\_DMA\_SLEEP, the caller wishes to have the allocation routines wait for resources to become available. If any other value is set and a DMA resource allocation fails, this value is assumed to be the address of a function to be called when resources become available. When the specified function is called, *arg* is passed to it as an argument. The specified callback function must return either DDI\_DMA\_CALLBACK\_RUNOUT or DDI\_DMA\_CALLBACK\_DONE. DDI\_DMA\_CALLBACK\_RUNOUT indicates that the callback function attempted to allocate DMA resources but failed. In this case, the callback function is put back on a list to be called again later. DDI\_DMA\_CALLBACK\_DONE indicates that either the allocation of DMA resources was successful or the driver no longer wishes to retry. The callback function is called in interrupt context. Therefore, only system functions accessible from interrupt context are available.

The callback function must take whatever steps are necessary to protect its critical resources, data structures, queues, and so on.

*arg* Argument to be passed to the callback function, if such a function is specified.

*kaddrp* On successful return, *kaddrp* points to the allocated memory.

*real\_length* The amount of memory, in bytes, allocated. Alignment and padding requirements may require `ddi_dma_mem_alloc()` to allocate more memory than requested in *length*.

*handlep* Pointer to a data access handle.

**Description** The `ddi_dma_mem_alloc()` function allocates memory for DMA transfers to or from a device. The allocation will obey the alignment, padding constraints and device granularity as specified by the DMA attributes (see `ddi_dma_attr(9S)`) passed to `ddi_dma_alloc_handle(9F)` and the more restrictive attributes imposed by the system.

The *flags* parameter should be set to DDI\_DMA\_STREAMING if the device is doing sequential, unidirectional, block-sized, and block-aligned transfers to or from memory. The alignment and padding constraints specified by the *minxfer* and *burstsizes* fields in the DMA attribute structure, `ddi_dma_attr(9S)` (see `ddi_dma_alloc_handle(9F)`) will be used to allocate the most effective hardware support for large transfers. For example, if an I/O transfer can be sped

up by using an I/O cache, which has a minimum transfer of one cache line, `ddi_dma_mem_alloc()` will align the memory at a cache line boundary and it will round up *real\_length* to a multiple of the cache line size.

The *flags* parameter should be set to `DDI_DMA_CONSISTENT` if the device accesses memory randomly, or if synchronization steps using `ddi_dma_sync(9F)` need to be as efficient as possible. I/O parameter blocks used for communication between a device and a driver should be allocated using `DDI_DMA_CONSISTENT`.

The device access attributes are specified in the location pointed by the *accattrp* argument (see `ddi_device_acc_attr(9S)`).

The data access handle is returned in *handlep*. *handlep* is opaque – drivers may not attempt to interpret its value. To access the data content, the driver must invoke `ddi_get8(9F)` or `ddi_put8(9F)` (depending on the data transfer direction) with the data access handle.

DMA resources must be established before performing a DMA transfer by passing *kaddrp* and *real\_length* as returned from `ddi_dma_mem_alloc()` and the flag `DDI_DMA_STREAMING` or `DDI_DMA_CONSISTENT` to `ddi_dma_addr_bind_handle(9F)`. In addition, to ensure the consistency of a memory object shared between the CPU and the device after a DMA transfer, explicit synchronization steps using `ddi_dma_sync(9F)` or `ddi_dma_unbind_handle(9F)` are required.

**Return Values** The `ddi_dma_mem_alloc()` function returns:

`DDI_SUCCESS`      Memory successfully allocated.  
`DDI_FAILURE`      Memory allocation failed.

**Context** The `ddi_dma_mem_alloc()` function can be called from user, interrupt, or kernel context except when *waitfp* is set to `DDI_DMA_SLEEP`, in which case it cannot be called from interrupt context.

**See Also** `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`, `ddi_dma_mem_free(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_unbind_handle(9F)`, `ddi_get8(9F)`, `ddi_put8(9F)`, `ddi_device_acc_attr(9S)`, `ddi_dma_attr(9S)`

### *Writing Device Drivers*

**Warnings** If `DDI_NEVERSWAP_ACC` is specified, memory can be used for any purpose; but if either endian mode is specified, you must use `ddi_get/put*` and never anything else.

**Name** ddi\_dma\_mem\_free – free previously allocated memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_dma_mem_free(ddi_acc_handle_t *handlep);
```

**Parameters** *handlep* Pointer to the data access handle previously allocated by a call to [ddi\\_dma\\_mem\\_alloc\(9F\)](#).

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** `ddi_dma_mem_free()` deallocates the memory acquired by [ddi\\_dma\\_mem\\_alloc\(9F\)](#). In addition, it destroys the data access handle *handlep* associated with the memory.

**Context** `ddi_dma_mem_free()` can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_dma\\_mem\\_alloc\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_movwin – shift current DMA window

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_dma_movwin(ddi_dma_handle_t handle, off_t *offp,
    uint_t *lenp, ddi_dma_cookie_t *cookiep);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_getwin\(9F\)](#) should be used instead.

**Parameters**

*handle* The DMA handle filled in by a call to [ddi\\_dma\\_setup\(9F\)](#).

*offp* A pointer to an offset to set the DMA window to. Upon a successful return, it will be filled in with the new offset from the beginning of the object resources are allocated for.

*lenp* A pointer to a value which must either be the current size of the DMA window (as known from a call to [ddi\\_dma\\_curwin\(9F\)](#) or from a previous call to [ddi\\_dma\\_movwin\(\)](#)). Upon a successful return, it will be filled in with the size, in bytes, of the current window.

*cookiep* A pointer to a DMA cookie (see [ddi\\_dma\\_cookie\(9S\)](#)). Upon a successful return, cookie is filled in just as if an implicit [ddi\\_dma\\_htoc\(9F\)](#) had been made.

**Description** The [ddi\\_dma\\_movwin\(\)](#) function shifts the current DMA window. If a DMA request allows the system to allocate resources for less than the entire object by setting the `DDI_DMA_PARTIAL` flag in the [ddi\\_dma\\_req\(9S\)](#) structure, the current DMA window can be shifted by a call to [ddi\\_dma\\_movwin\(\)](#).

The caller must first determine the current DMA window size by a call to [ddi\\_dma\\_curwin\(9F\)](#). Using the current offset and size of the window thus retrieved, the caller of [ddi\\_dma\\_movwin\(\)](#) may change the window onto the object by changing the offset by a value which is some multiple of the size of the DMA window.

The [ddi\\_dma\\_movwin\(\)](#) function takes care of underlying resource synchronizations required to `shift` the window. However, if you want to *access* the data prior to or after moving the window, further synchronizations using [ddi\\_dma\\_sync\(9F\)](#) are required.

This function is normally called from an interrupt routine. The first invocation of the DMA engine is done from the driver. All subsequent invocations of the DMA engine are done from the interrupt routine. The interrupt routine checks to see if the request has been completed. If it has, it returns without invoking another DMA transfer. Otherwise it calls [ddi\\_dma\\_movwin\(\)](#) to shift the current window and starts another DMA transfer.

**Return Values** The [ddi\\_dma\\_movwin\(\)](#) function returns:

`DDI_SUCCESS` The current length and offset are legal and have been set.

`DDI_FAILURE` Otherwise.

**Context** The `ddi_dma_movwin()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_curwin\(9F\)](#), [ddi\\_dma\\_getwin\(9F\)](#), [ddi\\_dma\\_htoc\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_cookie\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Warnings** The caller must guarantee that the resources used by the object are inactive prior to calling this function.

**Name** ddi\_dma\_nextcookie – retrieve subsequent DMA cookie

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_dma_nextcookie(ddi_dma_handle_t handle,
    ddi_dma_cookie_t *cookiep);
```

**Parameters** *handle* The handle previously allocated by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).  
*cookiep* A pointer to a [ddi\\_dma\\_cookie\(9S\)](#) structure.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** `ddi_dma_nextcookie()` retrieves subsequent DMA cookies for a DMA object. `ddi_dma_nextcookie()` fills in the [ddi\\_dma\\_cookie\(9S\)](#) structure pointed to by *cookiep*. The [ddi\\_dma\\_cookie\(9S\)](#) structure must be allocated prior to calling `ddi_dma_nextcookie()`.

The DMA cookie count returned by [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), or [ddi\\_dma\\_getwin\(9F\)](#) indicates the number of DMA cookies a DMA object consists of. If the resulting cookie count, *N*, is larger than 1, `ddi_dma_nextcookie()` must be called *N*-1 times to retrieve all DMA cookies.

**Context** `ddi_dma_nextcookie()` can be called from user, kernel, or interrupt context.

**Examples** EXAMPLE 1 Process a scatter-gather list of I/O requests.

This example demonstrates the use of `ddi_dma_nextcookie()` to process a scatter-gather list of I/O requests.

```
/* setup scatter-gather list with multiple DMA cookies */
ddi_dma_cookie_t dmacookie;
uint_t          ccount;
...

status = ddi_dma_buf_bind_handle(handle, bp, DDI_DMA_READ,
    NULL, NULL, &dmacookie, &ccount);

if (status == DDI_DMA_MAPPED) {

    /* program DMA engine with first cookie */

    while (--ccount > 0) {
        ddi_dma_nextcookie(handle, &dmacookie);
        /* program DMA engine with next cookie */
    }
}
...
```

**See Also** [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#),  
[ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_unbind\\_handle\(9F\)](#), [ddi\\_dma\\_cookie\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_nextseg – get next DMA segment

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_nextseg(ddi_dma_win_t win, ddi_dma_seg_t seg,
    ddi_dma_seg_t *nseg);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_nextcookie\(9F\)](#) should be used instead.

**Parameters** *win* A DMA window.  
*seg* The current DMA segment or NULL.  
*nseg* A pointer to the next DMA segment to be filled in. If *seg* is NULL, a pointer to the first segment within the specified window is returned.

**Description** The `ddi_dma_nextseg()` function gets the next DMA segment within the specified window *win*. If the current segment is NULL, the first DMA segment within the window is returned.

A DMA segment is always required for a DMA window. A DMA segment is a contiguous portion of a DMA window (see [ddi\\_dma\\_nextwin\(9F\)](#)) which is entirely addressable by the device for a data transfer operation.

An example where multiple DMA segments are allocated is where the system does not contain DVMA capabilities and the object may be non-contiguous. In this example the object will be broken into smaller contiguous DMA segments. Another example is where the device has an upper limit on its transfer size (for example an 8-bit address register) and has expressed this in the DMA limit structure (see [ddi\\_dma\\_lim\\_sparc\(9S\)](#) or [ddi\\_dma\\_lim\\_x86\(9S\)](#)). In this example the object will be broken into smaller addressable DMA segments.

**Return Values** The `ddi_dma_nextseg()` function returns:

DDI\_SUCCESS      Successfully filled in the next segment pointer.  
DDI\_DMA\_DONE      There is no next segment. The current segment is the final segment within the specified window.  
DDI\_DMA\_STALE      *win* does not refer to the currently active window.

**Context** The `ddi_dma_nextseg()` function can be called from user, interrupt, or kernel context.

**Examples** For an example, see [ddi\\_dma\\_segtocookie\(9F\)](#).

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |



**See Also** `attributes(5)`, `ddi_dma_addr_setup(9F)`, `ddi_dma_buf_setup(9F)`,  
`ddi_dma_nextcookie(9F)`, `ddi_dma_nextwin(9F)`, `ddi_dma_segtocookie(9F)`,  
`ddi_dma_sync(9F)`, `ddi_dma_lim_sparc(9S)`, `ddi_dma_lim_x86(9S)`, `ddi_dma_req(9S)`

*Writing Device Drivers*

**Name** ddi\_dma\_nextwin – get next DMA window

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_nextwin(ddi_dma_handle_t handle, ddi_dma_win_t win,
    ddi_dma_win_t *nwin);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_getwin\(9F\)](#) should be used instead.

**Parameters**

|               |                                                                                                                                       |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <i>handle</i> | A DMA handle.                                                                                                                         |
| <i>win</i>    | The current DMA window or NULL.                                                                                                       |
| <i>nwin</i>   | A pointer to the next DMA window to be filled in. If <i>win</i> is NULL, a pointer to the first window within the object is returned. |

**Description** The `ddi_dma_nextwin()` function shifts the current DMA window *win* within the object referred to by *handle* to the next DMA window *nwin*. If the current window is NULL, the first window within the object is returned. A DMA window is a portion of a DMA object or might be the entire object. A DMA window has system resources allocated to it and is prepared to accept data transfers. Examples of system resources are DVMA mapping resources and intermediate transfer buffer resources.

All DMA objects require a window. If the DMA window represents the whole DMA object it has system resources allocated for the entire data transfer. However, if the system is unable to setup the entire DMA object due to system resource limitations, the driver writer may allow the system to allocate system resources for less than the entire DMA object. This can be accomplished by specifying the `DDI_DMA_PARTIAL` flag as a parameter to [ddi\\_dma\\_buf\\_setup\(9F\)](#) or [ddi\\_dma\\_addr\\_setup\(9F\)](#) or as part of a [ddi\\_dma\\_req\(9S\)](#) structure in a call to [ddi\\_dma\\_setup\(9F\)](#).

Only the window that has resources allocated is valid per object at any one time. The currently valid window is the one that was most recently returned from `ddi_dma_nextwin()`. Furthermore, because a call to `ddi_dma_nextwin()` will reallocate system resources to the new window, the previous window will become invalid. It is a *severe* error to call `ddi_dma_nextwin()` before any transfers into the current window are complete.

The `ddi_dma_nextwin()` function takes care of underlying memory synchronizations required to shift the window. However, if you want to access the data before or after moving the window, further synchronizations using [ddi\\_dma\\_sync\(9F\)](#) are required.

**Return Values** The `ddi_dma_nextwin()` function returns:

|                           |                                                                                              |
|---------------------------|----------------------------------------------------------------------------------------------|
| <code>DDI_SUCCESS</code>  | Successfully filled in the next window pointer.                                              |
| <code>DDI_DMA_DONE</code> | There is no next window. The current window is the final window within the specified object. |

DDI\_DMA\_STALE *win* does not refer to the currently active window.

**Context** The `ddi_dma_nextwin()` function can be called from user, interrupt, or kernel context.

**Examples** For an example see `ddi_dma_segtocookie(9F)`.

**Attributes** See `attributes(5)` for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** `attributes(5)`, `ddi_dma_addr_setup(9F)`, `ddi_dma_buf_setup(9F)`, `ddi_dma_getwin(9F)`, `ddi_dma_nextseg(9F)`, `ddi_dma_segtocookie(9F)`, `ddi_dma_sync(9F)`, `ddi_dma_req(9S)`

*Writing Device Drivers*

**Name** ddi\_dma\_numwin – retrieve number of DMA windows

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_numwin(ddi_dma_handle_t handle, uint_t *nwinp);
```

**Parameters** *handle* The DMA handle previously allocated by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).  
*nwinp* Upon a successful return, *nwinp* will contain the number of DMA windows for this object.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_dma\_numwin() returns the number of DMA windows for a DMA object if partial resource allocation was permitted.

**Return Values** ddi\_dma\_numwin() returns:

DDI\_SUCCESS Successfully filled in the number of DMA windows.

DDI\_FAILURE DMA windows are not activated.

**Context** ddi\_dma\_numwin() can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#),  
[ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_unbind\\_handle\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_segtocookie – convert a DMA segment to a DMA address cookie

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_segtocookie(ddi_dma_seg_t seg, off_t *offp, off_t *lenp,
    ddi_dma_cookie_t *cookiep);
```

**Interface Level** This interface is obsolete. [ddi\\_dma\\_nextcookie\(9F\)](#) should be used instead.

**Parameters**

- seg* A DMA segment.
- offp* A pointer to an `off_t`. Upon a successful return, it is filled in with the offset. This segment is addressing within the object.
- lenp* The byte length. This segment is addressing within the object.
- cookiep* A pointer to a DMA cookie (see [ddi\\_dma\\_cookie\(9S\)](#)).

**Description** The `ddi_dma_segtocookie()` function takes a DMA segment and fills in the cookie pointed to by *cookiep* with the appropriate address, length, and bus type to be used to program the DMA engine. `ddi_dma_segtocookie()` also fills in *\*offp* and *\*lenp*, which specify the range within the object.

**Return Values** The `ddi_dma_segtocookie()` function returns:

DDI\_SUCCESS Successfully filled in all values.  
DDI\_FAILURE Failed to successfully fill in all values.

**Context** The `ddi_dma_segtocookie()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 `ddi_dma_segtocookie()` example

```
for (win = NULL; (retw = ddi_dma_nextwin(handle, win, &nwin)) !=
    DDI_DMA_DONE; win = nwin) {
    if (retw != DDI_SUCCESS) {
        /* do error handling */
    } else {
        for (seg = NULL; (rets = ddi_dma_nextseg(nwin, seg, &nseg)) !=
            DDI_DMA_DONE; seg = nseg) {
            if (rets != DDI_SUCCESS) {

                /* do error handling */
            } else {
                ddi_dma_segtocookie(nseg, &off, &len, &cookie);

                /* program DMA engine */
            }
        }
    }
}
```

EXAMPLE 1 ddi\_dma\_segtocookie() example (Continued)

```
    }  
}
```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_nextcookie\(9F\)](#), [ddi\\_dma\\_nextseg\(9F\)](#), [ddi\\_dma\\_nextwin\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_cookie\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_dma\_set\_sbus64 – allow 64-bit transfers on SBus

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_set_sbus64(ddi_dma_handle_t handle, uint_t burstsizes);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *handle* The handle filled in by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).  
*burstsizes* The possible burst sizes the device's DMA engine can accept in 64-bit mode.

**Description** `ddi_dma_set_sbus64()` informs the system that the device wishes to perform 64-bit data transfers on the SBus. The driver must first allocate a DMA handle using [ddi\\_dma\\_alloc\\_handle\(9F\)](#) with a [ddi\\_dma\\_attr\(9S\)](#) structure describing the DMA attributes for a 32-bit transfer mode.

*burstsizes* describes the possible burst sizes the device's DMA engine can accept in 64-bit mode. It may be distinct from the burst sizes for 32-bit mode set in the [ddi\\_dma\\_attr\(9S\)](#) structure. The system will activate 64-bit SBus transfers if the SBus supports them. Otherwise, the SBus will operate in 32-bit mode.

After DMA resources have been allocated (see [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#) or [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#)), the driver should retrieve the available burst sizes by calling [ddi\\_dma\\_burstsizes\(9F\)](#). This function will return the burst sizes in 64-bit mode if the system was able to activate 64-bit transfers. Otherwise burst sizes will be returned in 32-bit mode.

**Return Values** `ddi_dma_set_sbus64()` returns:

DDI\_SUCCESS Successfully set the SBus to 64-bit mode.  
DDI\_FAILURE 64-bit mode could not be set.

**Context** `ddi_dma_set_sbus64()` can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|----------------|-----------------|
| Architecture   | SBus            |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_burstsizes\(9F\)](#), [ddi\\_dma\\_attr\(9S\)](#)

**Notes** 64-bit SBus mode is activated on a per SBus slot basis. If there are multiple SBus cards in one slot, they all must operate in 64-bit mode or they all must operate in 32-bit mode.

**Name** ddi\_dma\_setup – setup DMA resources

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_setup(dev_info_t *dip, ddi_dma_req_t *dmareqp,
                 ddi_dma_handle_t *handlep);
```

**Interface Level** This interface is obsolete. The functions [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_free\\_handle\(9F\)](#), and [ddi\\_dma\\_unbind\\_handle\(9F\)](#) should be used instead.

**Parameters**

*dip* A pointer to the device's dev\_info structure.

*dmareqp* A pointer to a DMA request structure (see [ddi\\_dma\\_req\(9S\)](#)).

*handlep* A pointer to a DMA handle to be filled in. See below for a discussion of a handle. If *handlep* is NULL, the call to `ddi_dma_setup()` is considered an advisory call, in which case no resources are allocated, but a value indicating the legality and the feasibility of the request is returned.

**Description** The `ddi_dma_setup()` function allocates resources for a memory object such that a device can perform DMA to or from that object.

A call to `ddi_dma_setup()` informs the system that device referred to by *dip* wishes to perform DMA to or from a memory object. The memory object, the device's DMA capabilities, the device driver's policy on whether to wait for resources, are all specified in the `ddi_dma_req` structure pointed to by *dmareqp*.

A successful call to `ddi_dma_setup()` fills in the value pointed to by *handlep*. This is an opaque object called a DMA handle. This handle is then used in subsequent DMA calls, until [ddi\\_dma\\_free\(9F\)](#) is called.

Again a DMA handle is opaque—drivers may *not* attempt to interpret its value. When a driver wants to enable its DMA engine, it must retrieve the appropriate address to supply to its DMA engine using a call to [ddi\\_dma\\_htoc\(9F\)](#), which takes a pointer to a DMA handle and returns the appropriate DMA address.

When DMA transfer completes, the driver should free up the allocated DMA resources by calling `ddi_dma_free()`

**Return Values** The `ddi_dma_setup()` function returns:

DDI\_DMA\_MAPPED                      Successfully allocated resources for the object. In the case of an *advisory* call, this indicates that the request is legal.



|                     |                                                                                                                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_DMA_PARTIAL_MAP | Successfully allocated resources for a <i>part</i> of the object. This is acceptable when partial transfers are allowed using a flag setting in the <code>ddi_dma_req</code> structure (see <a href="#">ddi_dma_req(9S)</a> and <a href="#">ddi_dma_movwin(9F)</a> ). |
| DDI_DMA_NORESOURCES | When no resources are available.                                                                                                                                                                                                                                      |
| DDI_DMA_NOMAPPING   | The object cannot be reached by the device requesting the resources.                                                                                                                                                                                                  |
| DDI_DMA_TOOBIG      | The object is too big and exceeds the available resources. The maximum size varies depending on machine and configuration.                                                                                                                                            |

**Context** The `ddi_dma_setup()` function can be called from user, interrupt, or kernel context, except when the `dmr_fp` member of the `ddi_dma_req` structure pointed to by *dmareqp* is set to `DDI_DMA_SLEEP`, in which case it cannot be called from interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_free\\_handle\(9F\)](#), [ddi\\_dma\\_unbind\\_handle\(9F\)](#), [ddi\\_dma\\_addr\\_setup\(9F\)](#), [ddi\\_dma\\_buf\\_setup\(9F\)](#), [ddi\\_dma\\_free\(9F\)](#), [ddi\\_dma\\_htoc\(9F\)](#), [ddi\\_dma\\_movwin\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_req\(9S\)](#)

### *Writing Device Drivers*

**Notes** The construction of the `ddi_dma_req` structure is complicated. Use of the provided interface functions such as [ddi\\_dma\\_buf\\_setup\(9F\)](#) simplifies this task.

**Name** ddi\_dma\_sync – synchronize CPU and I/O views of memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_sync(ddi_dma_handle_t handle, off_t offset,
                size_t length, uint_t type);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- handle* The *handle* filled in by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).
- offset* The offset into the object described by the *handle*.
- length* The length, in bytes, of the area to synchronize. When *length* is zero, the entire range starting from *offset* to the end of the object has the requested operation applied to it.
- type* Indicates the caller's desire about what view of the memory object to synchronize. The possible values are DDI\_DMA\_SYNC\_FORDEV, DDI\_DMA\_SYNC\_FORCPU and DDI\_DMA\_SYNC\_FORKERNEL.

**Description** The `ddi_dma_sync()` function is used to selectively synchronize either a DMA device's or a CPU's view of a memory object that has DMA resources allocated for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When resources are allocated for DMA using `ddi_dma_addr_bind_handle()` or `ddi_dma_buf_bind_handle()`, an implicit `ddi_dma_sync()` is done. When DMA resources are deallocated using [ddi\\_dma\\_unbind\\_handle\(9F\)](#), an implicit `ddi_dma_sync()` is done. However, at any time between DMA resource allocation and deallocation, if the memory object has been modified by either the DMA device or a CPU and you wish to ensure that the change is noticed by the party that did *not* do the modifying, a call to `ddi_dma_sync()` is required. This is true independent of any attributes of the memory object including, but not limited to, whether or not the memory was allocated for consistent mode I/O (see [ddi\\_dma\\_mem\\_alloc\(9F\)](#)) or whether or not DMA resources have been allocated for consistent mode I/O (see [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#) or [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#)).

If a consistent view of the memory object must be ensured between the time DMA resources are allocated for the object and the time they are deallocated, you *must* call `ddi_dma_sync()` to ensure that either a CPU or a DMA device has such a consistent view.

What to set *type* to depends on the view you are trying to ensure consistency for. If the memory object is modified by a CPU, and the object is going to be read by the DMA engine of the device, use `DDI_DMA_SYNC_FORDEV`. This ensures that the device's DMA engine sees any changes that a CPU has made to the memory object. If the DMA engine for the device has *written* to the memory object, and you are going to *read* (with a CPU) the object (using an

---

extant virtual address mapping that you have to the memory object), use `DDI_DMA_SYNC_FORCPU`. This ensures that a CPU's view of the memory object includes any changes made to the object by the device's DMA engine. If you are only interested in the kernel's view (kernel-space part of the CPU's view) you may use `DDI_DMA_SYNC_FORKERNEL`. This gives a hint to the system—that is, if it is more economical to synchronize the kernel's view only, then do so; otherwise, synchronize for CPU.

**Return Values** The `ddi_dma_sync()` function returns:

`DDI_SUCCESS` Caches are successfully flushed.

`DDI_FAILURE` The address range to be flushed is out of the address range established by `ddi_dma_addr_bind_handle(9F)` or `ddi_dma_buf_bind_handle(9F)`.

**Context** The `ddi_dma_sync()` function can be called from user, interrupt, or kernel context.

**See Also** `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_alloc_handle(9F)`,  
`ddi_dma_buf_bind_handle(9F)`, `ddi_dma_mem_alloc(9F)`, `ddi_dma_unbind_handle(9F)`

*Writing Device Drivers*

**Name** ddi\_dma\_unbind\_handle – unbinds the address in a DMA handle

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_dma_unbind_handle(ddi_dma_handle_t handle);
```

**Parameters** *handle* The DMA handle previously allocated by a call to [ddi\\_dma\\_alloc\\_handle\(9F\)](#).

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_dma\_unbind\_handle() frees all DMA resources associated with an existing DMA handle. When a DMA transfer completes, the driver should call ddi\_dma\_unbind\_handle() to free system DMA resources established by a call to [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#) or [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#). ddi\_dma\_unbind\_handle() does an implicit [ddi\\_dma\\_sync\(9F\)](#) making further synchronization steps unnecessary.

**Return Values** DDI\_SUCCESS on success  
DDI\_FAILURE on failure

**Context** ddi\_dma\_unbind\_handle() can be called from user, kernel, or interrupt context.

**See Also** [ddi\\_dma\\_addr\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_alloc\\_handle\(9F\)](#),  
[ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [ddi\\_dma\\_free\\_handle\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_driver\_major – return driver's major device number

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
major_t ddi_driver_major(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** ddi\_driver\_major() returns the major device number for the driver associated with the supplied dev\_info node. This value can then be used as an argument to [makedevice\(9F\)](#) to construct a complete dev\_t.

**Parameters** *dip* A pointer to the device's dev\_info structure.

**Return Values** ddi\_driver\_major() returns the major number of the driver bound to a device, if any, or DDI\_MAJOR\_T\_NONE otherwise.

**Context** ddi\_driver\_major() can be called from kernel or interrupt context.

**See Also** [ddi\\_driver\\_name\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_driver\_name – return normalized driver name

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
const char *ddi_driver_name(dev_info_t *devi);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** dip     A pointer to the device's dev\_info structure.

**Description** ddi\_driver\_name() returns the normalized driver name. This name is typically derived from the device name property or the device compatible property. If this name is a driver alias, the corresponding driver name is returned.

**Return Values** ddi\_driver\_name() returns the actual name of the driver bound to a device.

**Context** ddi\_driver\_name() can be called from kernel, or interrupt context.

**See Also** [ddi\\_get\\_name\(9F\)](#)

*Writing Device Drivers*

**Warnings** The name returned by ddi\_driver\_name() is read-only.

- 
- Name** ddi\_enter\_critical, ddi\_exit\_critical – enter and exit a critical region of control
- Synopsis**
- ```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned int ddi_enter_critical(void);

void ddi_exit_critical(unsignedint ddic);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *ddic* The returned value from the call to ddi\_enter\_critical() must be passed to ddi\_exit\_critical().
- Description** Nearly all driver operations can be done without any special synchronization and protection mechanisms beyond those provided by, for example, mutexes (see [mutex\(9F\)](#)). However, for certain devices there can exist a very short critical region of code which *must* be allowed to run uninterrupted. The function ddi\_enter\_critical() provides a mechanism by which a driver can ask the system to guarantee to the best of its ability that the current thread of execution will neither be preempted nor interrupted. This stays in effect until a bracketing call to ddi\_exit\_critical() is made (with an argument which was the returned value from ddi\_enter\_critical()).
- The driver may not call any functions external to itself in between the time it calls ddi\_enter\_critical() and the time it calls ddi\_exit\_critical().
- Return Values** The ddi\_enter\_critical() function returns an opaque unsigned integer which must be used in the subsequent call to ddi\_exit\_critical().
- Context** This function can be called from user, interrupt, or kernel context.
- Warnings** Driver writers should note that in a multiple processor system this function does not temporarily suspend other processors from executing. This function also cannot guarantee to actually block the hardware from doing such things as interrupt acknowledge cycles. What it *can* do is guarantee that the currently executing thread will not be preempted.
- Do not write code bracketed by ddi\_enter\_critical() and ddi\_exit\_critical() that can get caught in an infinite loop, as the machine may crash if you do.
- See Also** [mutex\(9F\)](#)
- Writing Device Drivers*

**Name** ddi\_ffs, ddi\_fls – find first (last) bit set in a long integer

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_ffs(long mask);
int ddi_fls(long mask);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mask* A 32-bit argument value to search through.

**Description** The function `ddi_ffs()` takes its argument and returns the shift count that the first (least significant) bit set in the argument corresponds to. The function `ddi_fls()` does the same, only it returns the shift count for the last (most significant) bit set in the argument.

**Return Values** 0 No bits are set in mask.

*N* Bit *N* is the least significant (`ddi_ffs`) or most significant (`ddi_fls`) bit set in mask. Bits are numbered from 1 to 32, with bit 1 being the least significant bit position and bit 32 the most significant position.

**Context** This function can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)



**Name** `ddi_fm_acc_err_clear`, `ddi_fm_dma_err_clear` – clear the error status for an access or DMA handle

**Synopsis** `#include <sys/ndifma.h>`

```
void ddi_fm_acc_err_clear(ddi_acc_handle_t acc_handle,
    int version);
```

```
void ddi_fm_dma_err_clear(ddi_dma_handle_t dma_handle,
    int version);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *acc\_handle* Data access handle obtained from a previous call to `ddi_regs_map_setup(9F)`, `ddi_dma_mem_alloc(9F)`, or to a similar function.

*dma\_handle* DMA handle obtained from a previous call to `ddi_dma_mem_alloc(9F)` or one of its derivatives.

*version* Version number of `ddi_fm_error_t`.

**Description** The `ddi_fm_dma_err_clear()` and `ddi_fm_acc_err_clear()` functions clear the error status of a DMA or access handle respectively.

Once cleared, the driver is again able to access the mapped registers or memory using programmed I/O through the handle.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_mem\\_alloc\(9F\)](#), [ddi\\_fm\\_acc\\_err\\_get\(9F\)](#), [ddi\\_fm\\_dma\\_err\\_get\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_fm\_acc\_err\_get, ddi\_fm\_dma\_err\_get – get the error status for an access or DMA handle

**Synopsis** #include <sys/ndifma.h>

```
void ddi_fm_acc_err_get(ddi_acc_handle_t acc_handle,
    ddi_fm_error_t *error_status, int version);

void ddi_fm_dma_err_get(ddi_dma_handle_t dma_handle,
    ddi_fm_error_t *error_status, int version);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>acc_handle</i>	Data access handle obtained from a previous call to <a href="#">ddi_regs_map_setup(9F)</a> , <a href="#">ddi_dma_mem_alloc(9F)</a> , or to a similar function.
<i>dma_handle</i>	DMA handle obtained from a previous call to <a href="#">ddi_dma_mem_alloc(9F)</a> or one of its derivatives.
<i>error_status</i>	Pointer to where the error status for the access or DMA handle should be returned.
<i>version</i>	Version number of <code>ddi_fm_error_t</code> . The driver should always set this to <code>DDI_FME_VERSION</code> .

**Description** The `ddi_fm_dma_err_get()` and `ddi_fm_acc_err_get()` functions return the error status for a DMA or access handle, respectively. If a fault has occurred that affects the resource mapped by the supplied handle, the supplied *error\_status* structure is updated to reflect error information captured during error handling by a bus or other device driver in the I/O data path. When no error has occurred, *error\_status.fme\_status* is set to `DDI_FM_OK`.

If an error is indicated for an access handle, the driver might no longer be able to access the mapped registers or memory using programmed I/O through the handle. Typically, this might occur after the device has failed to respond to an I/O access, such as in the case of a bus error or a timeout. The effect of programmed I/O access made at the time of a fault is undefined. Read access via [ddi\\_get8\(9F\)](#), for example, can return random values, while write access via [ddi\\_put8\(9F\)](#) might or might not have an effect. It is possible, however, that the error might be transient. In that case, the driver can attempt to recover by calling `ddi_fm_acc_err_clear()`, resetting the device to return it to a known state, then retrying any potentially failed transactions.

If an error is indicated for a DMA handle, it implies that an error has been detected that has or will affect DMA transactions between the device and the memory currently bound to the handle, or to the memory most recently bound if the handle is currently unbound. Possible causes include the failure of a component in the DMA data path or an attempt by the device to make an invalid DMA access. The contents of any memory currently or previously bound to the handle should be considered indeterminate. The driver might be able to continue by freeing memory that is bound to the handle back to the system, resetting the device to return it to a known state, then retrying any potentially failed transactions.

If the driver is unable to recover, the operating state should be changed by a call to `ddi_fm_service_impact()` that specifies `DDI_SERVICE_LOST` for the impacted device instance. If the recovery and retry succeed, a call should still be made to `ddi_fm_service_impact()` but `DDI_SERVICE_UNAFFECTED` should be specified.

**Context** The `ddi_fm_acc_err_get()` and `ddi_fm_dma_err_get()` functions can be called from user, kernel, or high-level interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_mem\\_alloc\(9F\)](#), [ddi\\_fm\\_acc\\_err\\_clear\(9F\)](#), [ddi\\_fm\\_service\\_impact\(9F\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_fm\\_error\(9S\)](#),

*Writing Device Drivers*

**Name** ddi\_fm\_ereport\_post – post an FMA Protocol Error Report Event

**Synopsis** #include <sys/ddifm.h>

```
void ddi_fm_ereport_post(dev_info_t *dip, char *ereport_class,
    uint64_t ena, int *sflag, ... /* name-value pair args */);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the dev_info structure
<i>ereport_class</i>	FMA Event Protocol error class
<i>ena</i>	Error Numeric Association
<i>sflag</i>	Determines whether caller can sleep for memory or other event resources.

**Description** The ddi\_fm\_ereport\_post() function causes an encoded fault management error report name-value pair list to be queued for delivery to the Fault Manager daemon, fmd(1M). The sflag parameter indicates whether or not the caller is willing to wait for system memory and event channel resources to become available.

The following ereport\_class strings are available for use by any leaf device driver:

device.inval_state	A leaf driver discovers that the device is in an invalid or inconsistent state. For example, the driver might detect that receive or send ring descriptor indices are corrupted. It might also find an invalid value in a register or a driver-to-device protocol violation.
device.no_response	A leaf driver times out waiting for a response from the device. For example, timeouts can occur when no confirmation is seen after resetting, enabling, or disabling part of the device.
device.badint_limit	A leaf device sends too many consecutive interrupts with no work to do.
device.intern_corr	A leaf device reports to the driver that it has itself detected an internal correctable error.
device.intern_uncorr	A leaf device reports to the driver that it has itself detected an internal uncorrectable error.
device.stall	A leaf driver determines that data transmission has stalled indefinitely.

The ena indicates the Format 1 Error Numeric Association for this error report. It might have already been initialized by another error-detecting software module. For example, if ddi\_fm\_ereport\_post() is called from an error handler callback function, the fme\_ena field from the passed-in ddi\_fm\_error argument should be used. Otherwise it should be set to 0 and will be initialized by ddi\_fm\_ereport\_post().

The name-value pair *args* variable argument list contains one or more (names, type, value pointer) nvpair tuples for non-array `data_type_t` types or one or more (name, type, number of elements, value pointer) tuples for `data_type_t` array types. There is one mandatory tuple to describe the ereport version. This should contain the following values:

```
name - FM_VERSION
type - DATA_TYPE_UINT8
value - FM_EREPORT_VERS0
```

Additional nvpair tuples can describe error conditions for logging purposes, but are not interpreted by the I/O framework or fault manager. The end of the argument list is specified by `NULL`.

**Context** The `ddi_fm_ereport_post()` function can be called from user, kernel, or high-level interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [fmd\(1M\)](#), [attributes\(5\)](#), [ddi\\_fm\\_service\\_impact\(9F\)](#)

**Name** ddi\_fm\_handler\_register, ddi\_fm\_handler\_unregister – register or unregister an error handling callback

**Synopsis** #include <sys/ddifm.h>

```
void ddi_fm_handler_register(dev_info_t *dip,  
                             ddi_err_func_t error_handler, void *impl_data);  
  
void ddi_fm_handler_unregister(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the dev_info structure
<i>error_handler</i>	Pointer to an error handler callback function
<i>impl_data</i>	Pointer to private data for use by the caller

**Description** The ddi\_fm\_handler\_register() function registers an error handler callback routine with the I/O Fault Management framework. The error handler callback, *error\_handler*, is called to process error conditions detected by the system. In addition to its device instance, *dip*, the error handler is called with a pointer to a fault management error status structure, ddi\_fm\_error\_t. For example:

```
int (*ddi_err_func_t)(dev_info_t *dip, ddi_fm_error_t *error_status);
```

A driver error handling callback is passed the following arguments:

- a pointer to the device instance registered for this callback.
- a data structure containing common fault management data and status for error handling.

The primary responsibilities of the error handler include:

- to check for outstanding hardware or software errors.
- where possible, to isolate the device that might have caused the errors.
- to report errors that were detected.

During the invocation of an error handler, a device driver might need to quiesce or suspend all I/O activities in order to check for error conditions or status in:

- hardware control and status registers.
- outstanding I/O transactions.
- access or DMA handles.

For each error detected, the driver must formulate and post an error report via ddi\_fm\_ereport\_post() for problem analysis by the Solaris Fault Manager [fmd\(1M\)](#).

For a PCI, PCI/X, or PCI Express leaf device, the pci\_ereport\_post() function is provided to carry out reporting responsibilities on behalf of the driver. In many cases, an error handler callback function of the following form can be used:

```
xxx_err_cb(dev_info_t *dip, ddi_fm_error_t *errp) {
    pci_ereport_post(dip, errp, NULL);
    return (errp->fme_status);
}
```

In addition, the driver might be able to carry out further device specific checks within the error handler.

Error handlers can be called from kernel, interrupt, or high-level interrupt context. The interrupt block cookie returned from `ddi_fm_init()` should be used to allocate and initialize any synchronization variables and locks that might be used within the error handler callback function. Such locks may not be held by the driver when a device register is accessed with functions such as [ddi\\_get8\(9F\)](#) and [ddi\\_put8\(9F\)](#).

The data structure, `ddi_fm_error_t`, contains an FMA protocol (format 1) ENA for the current error propagation chain, the status of the error handler callback, an error expectation flag, and any potential access or DMA handles associated with an error detected by the parent nexus.

The `ddi_fm_handler_unregister()` function removes a previously registered error handling callback for the device instance specified by the *dip*.

**Context** The `ddi_fm_handler_register()` and `ddi_fm_handler_unregister()` functions must be called from kernel context in an [attach\(9E\)](#) or [detach\(9E\)](#) entry point. The registered error handler, *error\_handler*, callback can be called from kernel, interrupt, or high level interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [fmd\(1M\)](#), [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [ddi\\_fm\\_ereport\\_post\(9F\)](#), [ddi\\_fm\\_init\(9F\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#), [pci\\_ereport\\_post\(9F\)](#), [ddi\\_fm\\_error\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_fm\_init, ddi\_fm\_fini, ddi\_fm\_capable – initialize and get the FM capabilities for a device instance

**Synopsis** #include <sys/ddifm.h>

```
void ddi_fm_init(dev_info_t *dip, int *fm_capability,
                ddi_iblock_cookie_t *ibcp);

void ddi_fm_fini(dev_info_t *dip);

int ddi_fm_capable(dev_info_t *dip, int *fm_capability);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** ddi\_fm\_init()

*dip* Pointer to the dev\_info structure

*fm\_capability* Fault Management capability bit mask

*ibcp* Pointer to where the interrupt block cookie should be returned.

**Description** A device driver can declare its fault management capabilities to the I/O Fault Management framework by calling ddi\_fm\_init(). The ddi\_fm\_init() function allocates and initializes resources according to the bitwise-inclusive-OR of the fault management capabilities, described in the following and supported by the driver's immediate nexus parent.

DDI_FM_NOT_CAPABLE	The driver does not support any FMA features. This is the default value assigned to device drivers.
DDI_FM_EREPOR_T_CAPABLE	The driver generates FMA protocol error events (ereports) upon the detection of an error condition.
DDI_FM_ACCCHK_CAPABLE	The driver checks for errors upon the completion of one or more access I/O transactions.
DDI_FM_DMACHK_CAPABLE	The driver checks for errors upon the completion of one or more DMA I/O transactions.
DDI_FM_ERRCB_CAPABLE	The driver is capable of error handler callback registration.

If the parent nexus is not capable of supporting any one of the requested capabilities, the associated bit will not be set and returned as such to the driver. Before returning from ddi\_fm\_init(), the I/O Fault Management framework creates a set of fault management capability properties: fm-ereport-capable, fm-errcb-capable, fm-accchk-capable, and fm-dmachk-capable. The current supported fault management capability levels are observable via prtconf(1M).



A driver can support the administrative selection of fault management capabilities by exporting and setting a fault management capability level property in its `driver.conf(4)` file to the values described above. The `fm_capable` properties must be set and read prior to calling `ddi_fm_init()` with the desired capability list.

`ddi_fm_fini()` This function cleans up resources allocated to support fault management for the *dip* structure.

`ddi_fm_capable()` This function returns the capability bit mask currently set for the *dip* structure.

**Context** These functions can be called from kernel context in a driver `attach(9E)` or `detach(9E)` operation.

**Attributes** See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** `prtconf(1M)`, `driver.conf(4)`, `attributes(5)`, `attach(9E)`, `detach(9E)`

*Writing Device Drivers*

**Name** ddi\_fm\_service\_impact – report the impact of an error

**Synopsis** #include <sys/ddifm.h>

```
void ddi_fm_service_impact(dev_info_t *dip, int *impact);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the dev\_info structure

*impact* Impact of error

**Description** The following service impact values are accepted by ddi\_fm\_service\_impact():

DDI\_SERVICE\_LOST The service provided by the device is unavailable due to an error. The operational state of the device will transition to DEVI\_DEVICE\_DOWN.

DDI\_SERVICE\_DEGRADED The driver is unable to provide normal service, but it can provide a partial or degraded level of service. The driver might have to make repeated attempts to perform an operation before it succeeds, or it may be running at less than its configured speed. A driver may use this value to indicate that an alternative device should be used if available, but that it can continue operation if no alternative exists. The operational state of the device will transition to DEVI\_DEVICE\_DEGRADED.

DDI\_SERVICE\_RESTORED The service provided by the device has been restored. The operational state of the device will transition to its pre-error condition state and DEVI\_DEVICE\_DOWN or DEVI\_DEVICE\_DEGRADED is removed.

DDI\_SERVICE\_UNAFFECTED The service provided by the device was unaffected by the error.

**Context** The ddi\_fm\_service\_impact() function can be called from user, kernel, or high-level interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Committed

**See Also** [fmd\(1M\)](#), [attributes\(5\)](#), [ddi\\_fm\\_ereport\\_post\(9F\)](#), [pci\\_ereport\\_post\(9F\)](#)

**Name** `ddi_get8`, `ddi_get16`, `ddi_get32`, `ddi_get64`, `ddi_getb`, `ddi_getw`, `ddi_getl`, `ddi_getll` – read data from the mapped memory address, device register or allocated DMA memory address

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
uint8_t ddi_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The `ddi_getb()`, `ddi_getl()`, `ddi_getll()`, and `ddi_getw()` functions are obsolete. The `ddi_get8()` function replaces `ddi_getb()`. The `ddi_get32()` function replaces `ddi_getl()`. The `ddi_get64()` function replaces `ddi_getll()`. The `ddi_get16()` function replaces `ddi_getw()`.

**Parameters** *handle* The data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

*dev\_addr* Base device address.

**Description** The `ddi_get8()`, `ddi_get16()`, `ddi_get32()`, and `ddi_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, *dev\_addr*.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA and SBus buses. See [sysbus\(4\)](#), [isa\(4\)](#), and [sbus\(4\)](#) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See [pci\(4\)](#).

**Return Values** These functions return the value read from the mapped address.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	ddi_getb(), ddi_getl(), ddi_getll(), and ddi_getw() are Obsolete;  ddi_get8(), ddi_get16(), ddi_get32(), ddi_get64() are Committed

**See Also** [ddi\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#)

**Name** ddi\_get\_cred – returns a pointer to the credential structure of the caller

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
cred_t *ddi_get_cred(void)
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_get\_cred() returns a pointer to the user credential structure of the caller.

**Return Values** ddi\_get\_cred() returns a pointer to the caller's credential structure.

**Context** ddi\_get\_cred() can be called from user context only.

**See Also** [Writing Device Drivers](#)

**Name** ddi\_get\_devstate – Check device state

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

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

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's dev\_info structure

**Description** The ddi\_get\_devstate() function returns a value indicating the state of the device specified by *dip*, as derived from the configuration operations that have been performed on it (or on the bus on which it resides) and any fault reports relating to it.

<b>Return Values</b> DDI_DEVSTATE_OFFLINE	The device is offline. In this state, the device driver is not attached, nor will it be attached automatically. The device cannot be used until it is brought online.
DDI_DEVSTATE_DOWN	The device is online but unusable due to a fault.
DDI_DEVSTATE_QUIESCED	The bus on which the device resides has been quiesced. This is not a fault, but no operations on the device should be performed while the bus remains quiesced.
DDI_DEVSTATE_DEGRADED	The device is online but only able to provide a partial or degraded service, due to a fault.
DDI_DEVSTATE_UP	The device is online and fully operational.

**Context** The ddi\_get\_devstate() function may be called from user, kernel, or interrupt context.

**Notes** A device driver should call this function to check its own state at each major entry point, and before committing resources to a requested operation. If a driver discovers that its device is already down, it should perform required cleanup actions and return as soon as possible. If appropriate, it should return an error to its caller, indicating that the device has failed (for example, a driver's [read\(9E\)](#) routine would return EIO).

Depending on the driver, some non-I/O operations (for example, calls to the driver's [ioctl\(9E\)](#) routine) may still succeed; only functions which would require fully accessible and operational hardware will necessarily fail. If the bus on which the device resides is quiesced, the driver may return a value indicating the operation should be retried later (for example, EAGAIN). Alternatively, for some classes of device, it may be appropriate for the driver to enqueue the operation and service it once the bus has been unquiesced. Note that not all busses support the quiesce/unquiesce operations, so this value may never be seen by some drivers.

**See Also** `attach(9E)`, `ioctl(9E)`, `open(9E)`, `read(9E)`, `strategy(9E)`, `write(9E)`,  
`ddi_dev_report_fault(9F)`

**Name** ddi\_get\_driver\_private, ddi\_set\_driver\_private – get or set the address of the device's private data area

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_set_driver_private(dev_info_t *dip, void *data);  
void *ddi_get_driver_private(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_get\_driver\_private()  
*dip* Pointer to device information structure to get from.  
ddi\_set\_driver\_private()  
*dip* Pointer to device information structure to set.  
*data* Data area address to set.

**Description** The ddi\_get\_driver\_private() function returns the address of the device's private data area from the device information structure pointed to by *dip*.

The ddi\_set\_driver\_private() function sets the address of the device's private data area in the device information structure pointed to by *dip* with the value of *data*.

**Return Values** The ddi\_get\_driver\_private() function returns the contents of *devi\_driver\_data*. If ddi\_set\_driver\_private() has not been previously called with *dip*, an unpredictable value is returned.

**Context** These functions can be called from user , interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)



**Name** ddi\_get\_eventcookie – retrieve a NDI event service cookie handle

**Synopsis** #include <sys/dditypes.h>  
#include <sys/sunddi.h>

```
int ddi_get_eventcookie(dev_info_t *dip, char *name,
    ddi_eventcookie_t *event_cookiep);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

dev_info_t *dip	Child device node requesting the cookie.
char *name	NULL-terminated string containing the name of the event.
ddi_eventcookie_t *event_cookiep	Pointer to cookie where event cookie will be returned.

**Description** The ddi\_get\_eventcookie() function queries the device tree for a cookie matching the given event name and returns a reference to that cookie. The search is performed by a calling up the device tree hierarchy until the request is satisfied by a bus nexus driver, or the top of the dev\_info tree is reached.

The cookie returned by this function can be used to register a callback handler, unregister a callback handler, or post an event.

**Return Values**

DDI_SUCCESS	Cookie handle is returned.
DDI_FAILURE	Request was not serviceable by any nexus driver in the driver's ancestral device tree hierarchy.

**Context** The ddi\_get\_eventcookie() function can be called from user and kernel contexts only.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_add\\_event\\_handler\(9F\)](#), [ddi\\_remove\\_event\\_handler\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_getminor – get kernel internal minor number from an external dev\_t

**Synopsis** #include <sys/types.h>  
#include <sys/mkdev.h>  
#include <sys/ddi.h>

```
minor_t ddi_getminor(dev_t dev);
```

**Interface Level** This interface is obsolete. [getminor\(9F\)](#) should be used instead.

**Parameters** The following parameters are supported:

*dev* Device number.

**Description** ddi\_getminor() extracts the minor number from a device number. This call should be used only for device numbers that have been passed to the kernel from the user space through opaque interfaces such as the contents of [ioctl\(9E\)](#) and [putmsg\(2\)](#). The device numbers passed in using standard device entry points must continue to be interpreted using the [getminor\(9F\)](#) interface. This new interface is used to translate between user visible device numbers and in kernel device numbers. The two numbers may differ in a clustered system.

For certain bus types, you can call this DDI function from a high-interrupt context. These types include ISA and SBus buses. See [sysbus\(4\)](#), [isa\(4\)](#), and [sbus\(4\)](#) for details.

**Context** ddi\_getminor() can be called from user context only.

**Return Values** The minor number or EMINOR\_UNKNOWN if the minor number of the device is invalid.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**See Also** [attributes\(5\)](#), [getmajor\(9F\)](#), [getminor\(9F\)](#), [makedevice\(9F\)](#)

*Writing Device Drivers*

**Warnings** Drivers are required to replace calls to ddi\_getminor.9f by [getminor\(9F\)](#) in order to compile under Solaris 10 and later versions.

- 
- Name** ddi\_get\_instance – get device instance number
- Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_instance(dev_info_t *dip);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *dip* Pointer to dev\_info structure.
- Description** The ddi\_get\_instance() function returns the instance number of the device corresponding to *dip*.
- The system assigns an instance number to every device. Instance numbers for devices attached to the same driver are unique. This provides a way for the system and the driver to uniquely identify one or more devices of the same type. The instance number is derived by the system from different properties for different device types in an implementation specific manner.
- Once an instance number has been assigned to a device, it will remain the same even across reconfigurations and reboots. Therefore, instance numbers seen by a driver may not appear to be in consecutive order. For example, if device foo0 has been assigned an instance number of 0 and device foo1 has been assigned an instance number of 1, if foo0 is removed, foo1 will continue to be associated with instance number 1 (even though foo1 is now the only device of its type on the system).
- Return Values** The ddi\_get\_instance() function returns the instance number of the device corresponding to *dip*.
- Context** The ddi\_get\_instance() function can be called from user, interrupt, or kernel context.
- See Also** [path\\_to\\_inst\(4\)](#)
- Writing Device Drivers*

**Name** ddi\_get\_kt\_did – get identifier of current thread

**Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
kt_did_t ddi_get_kt_did(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The ddi\_get\_kt\_did() function returns a unique 64-bit identifier for the currently running thread.

**Context** This routine can be called from user, kernel, or interrupt context. This routine cannot be called from a high-level interrupt context.

**Return Values** ddi\_get\_kt\_did() always returns the identifier for the current thread. There are no error conditions.

**See Also** *Writing Device Drivers*

**Notes** The value returned by this function can also be seen in adb or mdb as the did field displayed when using the thread macro.

This interface is intended for tracing and debugging purposes.

**Name** ddi\_get\_lbolt, ddi\_get\_lbolt64 – return the number of clock ticks since boot

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
clock_t ddi_get_lbolt(void);  
int64_t ddi_get_lbolt64(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The `ddi_get_lbolt()` function returns a value that represents the number of clock ticks since the system booted. This value is used as a counter or timer inside the system kernel. The tick frequency can be determined by using [drv\\_usec2hz\(9F\)](#), which converts microseconds into clock ticks.

The `ddi_get_lbolt64()` behaves essentially the same as `ddi_get_lbolt()`, except the value is returned in a longer data type (`int64_t`) that will not wrap for 2.9 billion years.

**Return Values** The `ddi_get_lbolt()` function returns the number of clock ticks since boot in a `clock_t` type.

The `ddi_get_lbolt64()` function returns the number of clock ticks since boot in a `int64_t` type.

**Context** These routines can be called from any context.

**See Also** [ddi\\_get\\_time\(9F\)](#), [drv\\_getparm\(9F\)](#), [drv\\_usec2hz\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** ddi\_get\_parent – find the parent of a device information structure

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
dev_info_t *ddi_get_parent(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* Pointer to a device information structure.

**Description** The `ddi_get_parent()` function returns a pointer to the device information structure which is the parent of the one pointed to by *dip*.

**Return Values** The `ddi_get_parent()` function returns a pointer to a device information structure.

**Context** The `ddi_get_parent()` function can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)

**Name** ddi\_get\_pid – returns the process ID

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
pid_t ddi_get_pid(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_get\_pid() obtains the process ID of the current process. This value can be used to allow only a select process to perform a certain operation. It can also be used to determine whether a device context belongs to the current process.

**Return Values** ddi\_get\_pid() returns the process ID.

**Context** This routine can be called from user context only.

**See Also** [drv\\_getparm\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** ddi\_get\_time – returns the current time in seconds

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
time_t ddi_get_time(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_get\_time() returns the current time in seconds since 00:00:00 UTC, January 1, 1970. This value can be used to set of wait or expiration intervals.

**Return Values** ddi\_get\_time() returns the time in seconds.

**Context** This routine can be called from any context.

**See Also** ddi\_get\_lbolt(9F), drv\_getparm(9F), drv\_usectohz(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*



**Name** ddi\_in\_panic – determine if system is in panic state

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_in_panic(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** Drivers controlling devices on which the system may write a kernel crash dump in the event of a panic can call `ddi_in_panic()` to determine if the system is panicking.

When the system is panicking, the calls of functions scheduled by [timeout\(9F\)](#) and [ddi\\_trigger\\_softintr\(9F\)](#) will never occur. Neither can [delay\(9F\)](#) be relied upon, since it is implemented via [timeout\(9F\)](#).

Drivers that need to enforce a time delay such as SCSI bus reset delay time must busy-wait when the system is panicking.

**Return Values** `ddi_in_panic()` returns 1 if the system is in panic, or 0 otherwise.

**Context** `ddi_in_panic()` may be called from any context.

**See Also** [dump\(9E\)](#), [delay\(9F\)](#), [ddi\\_trigger\\_softintr\(9F\)](#), [timeout\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_intr\_add\_handler, ddi\_intr\_remove\_handler – add or remove interrupt handler

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_intr_add_handler(ddi_intr_handle_t *h,
    ddi_intr_handler_t inthandler, void *arg1,
    void *arg2);

int ddi_intr_remove_handler(ddi_intr_handle_t h);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_intr\_add\_handler()

*h* Pointer to the DDI interrupt handle

*inthandler* Pointer to interrupt handler

*arg1* First argument for the interrupt handler

*arg2* Second, optional, argument for the interrupt handler

ddi\_intr\_remove\_handler()

*h* DDI interrupt handle

**Description** The `ddi_intr_add_handler()` function adds an interrupt handler given by the *inthandler* argument to the system with the handler arguments *arg1* and *arg2* for the previously allocated interrupt handle specified by the *h* pointer. The arguments *arg1* and *arg2* are passed as the first and second arguments, respectively, to the interrupt handler *inthandler*. See `<sys/ddi_intr.h>` for the definition of the interrupt handler.

The routine *inthandler* with the arguments *arg1* and *arg2* is called upon receipt of the appropriate interrupt. The interrupt handler should return `DDI_INTR_CLAIMED` if the interrupt is claimed and `DDI_INTR_UNCLAIMED` otherwise.

The `ddi_intr_add_handler()` function must be called after `ddi_intr_alloc()`, but before `ddi_intr_enable()` is called. The interrupt must be enabled through `ddi_intr_enable()` or `ddi_intr_block_enable()` before it can be used.

The `ddi_intr_remove_handler()` function removes the handler association, added previously with `ddi_intr_add_handler()`, for the interrupt identified by the interrupt handle *h* argument. Unloadable drivers should call this routine during their [detach\(9E\)](#) routine to remove the interrupt handler from the system.

The `ddi_intr_remove_handler()` function is used to disassociate the handler after the interrupt is disabled to remove *dup-ed* interrupt handles. See [ddi\\_intr\\_dup\\_handler\(9F\)](#) for

*dup-ed* interrupt handles. If a handler is duplicated with the `ddi_intr_dup_handler()` function, all added and duplicated instances of the handler must be removed with `ddi_intr_remove_handler()` in order for the handler to be completely removed.

**Return Values** The `ddi_intr_add_handler()` and `ddi_intr_remove_handler()` functions return:

DDI\_SUCCESS     On success.

DDI\_EINVAL     On encountering invalid input parameters.

DDI\_FAILURE     On any implementation specific failure.

**Context** The `ddi_intr_add_handler()` and `ddi_intr_remove_handler()` functions can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_block\\_enable\(9F\)](#), [ddi\\_intr\\_disable\(9F\)](#), [ddi\\_intr\\_dup\\_handler\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_free\(9F\)](#), [ddi\\_intr\\_get\\_supported\\_types\(9F\)](#), [mutex\(9F\)](#), [mutex\\_init\(9F\)](#), [rw\\_init\(9F\)](#), [rwlock\(9F\)](#)

### *Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses MSI and MSI-X interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any MSI and MSI-X interrupt usage settings that have been applied.

The second argument, *arg2*, is optional. Device drivers are free to use the two arguments however they see fit. There is no officially recommended model or restrictions. For example, an interrupt handler may wish to use the first argument as the pointer to its softstate and the second argument as the value of the MSI vector.

**Name** ddi\_intr\_add\_softint, ddi\_intr\_remove\_softint, ddi\_intr\_trigger\_softint, ddi\_intr\_get\_softint\_pri, ddi\_intr\_set\_softint\_pri – software interrupt handling routines

**Synopsis** #include <sys/types.h>  
#include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_intr_add_softint(dev_info_t *dip,  
    ddi_softint_handle_t *h, int soft_pri,  
    ddi_intr_handler_t handler, void *arg1);  
  
int ddi_intr_trigger_softint(ddi_softint_handle_t h,  
    void *arg2);  
  
int ddi_intr_remove_softint(ddi_softint_handle_t h);  
  
int ddi_intr_get_softint_pri(ddi_softint_handle_t h,  
    uint *soft_prip);  
  
int ddi_intr_set_softint_pri(ddi_softint_handle_t h,  
    uint soft_pri);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_intr\_add\_softint()  
*dip* Pointer to a dev\_info structure  
*h* Pointer to the DDI soft interrupt handle  
*soft\_pri* Priority to associate with a soft interrupt  
*handler* Pointer to soft interrupt handler  
*arg1* Argument for the soft interrupt handler  
  
ddi\_intr\_trigger\_softint()  
*h* DDI soft interrupt handle  
*arg2* Additional argument for the soft interrupt handler  
  
ddi\_intr\_remove\_softint()  
*h* DDI soft interrupt handle  
  
ddi\_intr\_get\_softint\_pri()  
*h* DDI soft interrupt handle  
*soft\_prip* Soft interrupt priority of the handle  
  
ddi\_intr\_set\_softint\_pri()

*h*            DDI soft interrupt handle  
*soft\_prip*    Soft interrupt priority of the handle

**Description** The `ddi_intr_add_softint()` function adds the soft interrupt handler given by the *handler* argument *arg1*. The *handler* runs at the soft interrupt priority given by the *soft\_prip* argument.

The value returned in the location pointed at by *h* is the soft interrupt handle. This value is used in later calls to `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()` and `ddi_intr_set_softint_pri()`.

The software priority argument *soft\_prip* is a relative priority value within the range of `DDI_INTR_SOFTPRI_MIN` and `DDI_INTR_SOFTPRI_MAX`. If the driver does not know what priority to use, the default *soft\_prip* value of `DDI_INTR_SOFTPRI_DEFAULT` could be specified. The default value is the lowest possible soft interrupt priority value.

The *soft\_prip* argument contains the value needed to initialize the lock associated with a soft interrupt. See `mutex_init(9F)` and `rw_init(9F)`. The handler cannot be triggered until the lock is initialized.

The `ddi_intr_remove_softint()` function removes the handler for the soft interrupt identified by the interrupt handle *h* argument. Once removed, the soft interrupt can no longer be triggered, although any trigger calls in progress can still be delivered to the handler.

Drivers must remove any soft interrupt handlers before allowing the system to unload the driver. Otherwise, kernel resource leaks might occur.

The `ddi_intr_trigger_softint()` function triggers the soft interrupt specified by the interrupt handler *h* argument. A driver may optionally specify an additional argument *arg2* that is passed to the soft interrupt handler. Subsequent `ddi_intr_trigger_softint()` events, along with *arg2*, will be dropped until the one pending is serviced and returns the error code `DDI_EPENDING`.

The routine *handler*, with the *arg1* and *arg2* arguments, is called upon the receipt of a software interrupt. These were registered through a prior call to `ddi_intr_add_softint()`. Software interrupt handlers must not assume that they have work to do when they run. Like hardware interrupt handlers, they may run because a soft interrupt has occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. Before triggering the soft interrupt, the driver must indicate to the soft interrupt handler that it has work to do. This is usually done by setting a flag in the state structure. The routine *handler* checks this flag, reached through *arg1* and *arg2*, to determine if it should claim the interrupt and do its work.

The interrupt handler must return `DDI_INTR_CLAIMED` if the interrupt was claimed and `DDI_INTR_UNCLAIMED` otherwise.

The `ddi_intr_get_softint_pri()` function retrieves the soft interrupt priority, a small integer value, associated with the soft interrupt handle. The handle is defined by the *h* argument, and the priority returned is in the value of the integer pointed to by the *soft\_prip* argument.

**Return Values** The `ddi_intr_add_softint()`, `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()`, `ddi_intr_get_softint_pri()`, `ddi_intr_set_softint_pri()` functions return:

<code>DDI_SUCCESS</code>	On success.
<code>DDI_EAGAIN</code>	On encountering internal error regarding currently unavailable resources.
<code>DDI_EINVAL</code>	On encountering invalid input parameters.
<code>DDI_FAILURE</code>	On any implementation specific failure.
<code>DDI_EPENDING</code>	On encountering a previously triggered softint event that is pending.

**Context** The `ddi_intr_add_softint()`, `ddi_intr_remove_softint()`, `ddi_intr_trigger_softint()`, `ddi_intr_get_softint_pri()`, `ddi_intr_set_softint_pri()` functions can be called from either user or kernel non-interrupt context.

**Examples** **EXAMPLE 1** Device using high-level interrupts

In the following example, the device uses high-level interrupts. High-level interrupts are those that interrupt at the level of the scheduler and above. High-level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high-level interrupt handlers must take care to do a minimum of work because they are not preemptable. See [ddi\\_intr\\_get\\_hilevel\\_pri\(9F\)](#).

In the example, the high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt handler. If the soft interrupt handler is not currently running, the high-level interrupt routine triggers a soft interrupt so the soft interrupt handler can process the data. Once running, the soft interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data shared between the high-level interrupt handler and the soft interrupt handler. The low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
    ...
    ddi_intr_handle_t    int_hdl;
    int                 high_pri;
    kmutex_t            high_mutex;
    ddi_softint_handle_t soft_hdl;
```

**EXAMPLE 1** Device using high-level interrupts (Continued)

```

    int                low_soft_pri;
    kmutex_t          low_mutex;
    int                softint_running;
    ...
};

struct xxstate *xsp;
static uint_t xxsoftint_handler(void *, void *);
static uint_t xxhighintr(void *, void *);
...

```

**EXAMPLE 2** Sample attach() routine

The following code fragment would usually appear in the driver's [attach\(9E\)](#) routine. [ddi\\_intr\\_add\\_handler\(9F\)](#) is used to add the high-level interrupt handler and [ddi\\_intr\\_add\\_softint\(\)](#) is used to add the low-level interrupt routine.

```

static uint_t
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int                types;
    int                *actual;
    int                nintrs;
    struct xxstate    *xsp;
    ...

    (void) ddi_intr_get_supported_types(dip, &types);
    (void) ddi_intr_get_nintrs(dip < DDI_INTR_TYPE_FIXED, *nintrs);
    (void) ddi_intr_alloc(dip, &xsp->int_hdl, DDI_INTR_TYPE_FIXED,
        1, nintrs, *actual, 0);

    /* initialize high-level mutex */
    (void) ddi_intr_get_pri(xsp->int_hdl, &high_pri);
    mutex_init(&xsp->high_mutex, NULL, MUTEX_DRIVER,
        DDI_INTR_PRI(xsp->high_pri));

    /* Ensure that this is a hi-level interrupt */
    if (ddi_intr_get_hilevel_pri(h) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }

    /* add high-level routine - xxhighintr() */
    if (ddi_intr_add_handler(xsp->int_hdl, xxhighintr,
        arg1, NULL) != DDI_SUCCESS) {
        /* cleanup */
    }
}

```

**EXAMPLE 2** Sample attach() routine (Continued)

```

        return (DDI_FAILURE); /* fail attach */
    }

    /* Enable high-level routine - xxhighintr() */
    if (ddi_intr_enable(xsp->int_hdl) != DDI_SUCCESS) {
        /* cleanup */
        return (DDI_FAILURE); /* fail attach */
    }

    /* Enable soft interrupts */
    xsp->low_soft_pri = DDI_INTR_SOFTPRI_MIN;
    if (ddi_intr_add_softint(dip, &xsp->soft_hdl,
        xsp->low_soft_pri, xxsoftint_handler, arg1) != DDI_SUCCESS) {
        /* clean up */
        return (DDI_FAILURE); /* fail attach */
    }

    /* initialize low-level mutex */
    mutex_init(&xsp->low_mutex, NULL, MUTEX_DRIVER,
        DDI_INTR_PRI(xsp->low_soft_pri));

    ...
}

```

**EXAMPLE 3** High-level interrupt routine

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, `ddi_intr_trigger_softint()` is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```

static uint_t
xxhighintr(void *arg1, void *arg2)
{
    struct xxstate *xsp = (struct xxstate *)arg1;
    int need_softint;
    ...
    mutex_enter(&xsp->high_mutex);
    /*
     * Verify this device generated the interrupt
     * and disable the device interrupt.
     * Enqueue data for xxsoftint_handler() processing.
     */

    /* is xxsoftint_handler() already running ? */

```



**EXAMPLE 3** High-level interrupt routine *(Continued)*

```

    need_softint = (xsp->softint_running) ? 0 : 1;
    mutex_exit(&xsp->high_mutex);

    /* read-only access to xsp->id, no mutex needed */
    if (xsp->soft_hdl && need_softint)
        ddi_intr_trigger_softint(xsp->soft_hdl, arg2);
    ...
    return (DDI_INTR_CLAIMED);
}

static uint_t
xxsoftint_handler(void *arg1, void *arg2)
{
    struct xxstate *xsp = (struct xxstate *)arg1;
    ...
    mutex_enter(&xsp->low_mutex);
    mutex_enter(&xsp->high_mutex);

    /* verify there is work to do */
    if (work queue empty || xsp->softint_running ) {
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);
        return (DDI_INTR_UNCLAIMED);
    }

    xsp->softint_running = 1;

    while ( data on queue ) {
        ASSERT(mutex_owned(&xsp->high_mutex));
        /* de-queue data */
        mutex_exit(&xsp->high_mutex);

        /* Process data on queue */
        mutex_enter(&xsp->high_mutex);
    }

    xsp->softint_running = 0;
    mutex_exit(&xsp->high_mutex);
    mutex_exit(&xsp->low_mutex);
    return (DDI_INTR_CLAIMED);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_free\(9F\)](#), [ddi\\_intr\\_get\\_hilevel\\_pri\(9F\)](#), [mutex\\_init\(9F\)](#), [rw\\_init\(9F\)](#), [rwlock\(9F\)](#)

### *Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

The `ddi_intr_add_softint()` may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for *arg1* in each of the calls to `ddi_intr_add_softint()`. Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For example, the argument could be a pointer to the soft state structure of the device that could contain a `which_service` field that the handler examines. The driver must set this field to the appropriate value before calling `ddi_intr_trigger_softint()`.

Every time a modifiable valid second argument, *arg2*, is provided when `ddi_intr_trigger_softint()` is invoked, the DDI framework saves *arg2* internally and passes it to the interrupt handler *handler*.

A call to `ddi_intr_set_softint_pri()` could fail if a previously scheduled soft interrupt trigger is still pending.

**Name** ddi\_intr\_alloc, ddi\_intr\_free – allocate or free interrupts for a given interrupt type

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_alloc(dev_info_t *dip, ddi_intr_handle_t *h_array, int type,
                  int inum, int count, int *actualp, int behavior);

int ddi_intr_free(ddi_intr_handle_t h);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

**ddi\_intr\_alloc()**

*dip* Pointer to the dev\_info structure

*h\_array* Pointer to an array of DDI interrupt handles

*type* Interrupt type

*inum* Interrupt number

*count* Number of interrupts requested. The *count* should not exceed the total number of interrupts supported by the device, as returned by a call to [ddi\\_intr\\_get\\_nintrs\(9F\)](#).

*actualp* Pointer to the number of interrupts actually allocated

*behavior* Flag to determine the allocation algorithm

**ddi\_intr\_free()**

*h* DDI interrupt handle

**Description** The `ddi_intr_alloc()` function allocates interrupts of the interrupt type given by the *type* argument beginning at the interrupt number *inum*. If `ddi_intr_alloc()` allocates any interrupts, it returns the actual number of interrupts allocated in the integer pointed to by the *actualp* argument and returns the number of interrupt handles in the interrupt handle array pointed to by the *h\_array* argument.

Specific interrupts are always specified by the combination of interrupt *type* and *inum*. For legacy devices, *inum* refers to the *n*th interrupt, typically as defined by the device's `interrupts` property. For PCI fixed interrupts, *inum* refers to the interrupt number. The *inum* is the relative interrupt vector number, from 0 to 31 for MSI, from 0 to 2047 for MSI-X. The first interrupt vector is 0. The last relative vector is 31 for MSI or 2047 for MSI-X.

The *h\_array* must be pre-allocated by the caller as a *count* sized array of `ddi_intr_handle_t`'s.

If MSI interrupts are being allocated, the *count* argument passed should be a number between 1 and 32, specified as a power of two. If *count* is not specified as a power of two, the error `DDI_EINVAL` is returned.

The behavior flag controls the interrupt allocation algorithm. It takes one of two input values: `DDI_INTR_ALLOC_NORMAL` or `DDI_INTR_ALLOC_STRICT`. If the *count* value used is greater than `NINTRs`, then the call fails with `DDI_EINVAL` unconditionally. When set to `DDI_INTR_ALLOC_STRICT`, the call succeeds if and only if *count* interrupts are allocated. Otherwise, the call fails, and the number of available interrupts is returned in *actualp*. When set to `DDI_INTR_ALLOC_NORMAL`, the call succeeds if at least one interrupt is allocated, and the number of allocated interrupts is returned in *actualp*.

The handle for each allocated interrupt, if any, is returned in the array of handles given by the *h\_array* argument.

The `ddi_intr_free()` function releases the system resources and interrupt vectors associated with the `ddi_intr_handle_t h`, including any resources associated with the handle *h* itself. Once freed, the handle *h* should not be used in any further calls.

The `ddi_intr_free()` function should be called once for each handle in the handle array.

**Return Values** The `ddi_intr_alloc()` and `ddi_intr_free()` functions return:

<code>DDI_SUCCESS</code>	On success.
<code>DDI_EAGAIN</code>	Not enough interrupt resources.
<code>DDI_EINVAL</code>	On encountering invalid input parameters.
<code>DDI_INTR_NOTFOUND</code>	On failure to find the interrupt.
<code>DDI_FAILURE</code>	On any implementation specific failure.

**Context** The `ddi_intr_alloc()` and `ddi_intr_free()` functions can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_block\\_enable\(9F\)](#), [ddi\\_intr\\_disable\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_get\\_cap\(9F\)](#), [ddi\\_intr\\_get\\_nintrs\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_intr\\_get\\_supported\\_types\(9F\)](#), [ddi\\_intr\\_remove\\_handler\(9F\)](#)

*Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses MSI and MSI-X interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any MSI and MSI-X interrupt usage settings that have been applied.

**Name** ddi\_intr\_dup\_handler – reuse interrupt handler and arguments for MSI-X interrupts

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_intr_dup_handler(ddi_intr_handle_t primary, int vector,
                        ddi_intr_handle_t *new);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *primary* Original DDI interrupt handle  
*vector* Interrupt number to duplicate  
*new* Pointer to new DDI interrupt handle

**Description** The `ddi_intr_dup_handler()` function is a feature for MSI-X interrupts that allows an unallocated interrupt vector of a device to use a previously initialized or added primary MSI-X interrupt vector in order to share the same vector address, vector data, interrupt handler, and handler arguments. This feature allows a driver to alias the resources provided by the Solaris Operating System to the unallocated interrupt vectors on an associated device. For example, if 2 MSI-X interrupts were allocated to a driver and 32 interrupts were supported on the device, the driver could alias the 2 interrupts it received to the 30 remaining on the device.

The `ddi_intr_dup_handler()` function must be called after the primary interrupt handle has been added to the system or enabled by `ddi_intr_add_handler(9F)` and `ddi_intr_enable(9F)` calls, respectively. If successful, the function returns the new interrupt handle for a given vector in the *new* argument passed to the function. The new interrupt handle must not have been previously allocated with `ddi_intr_alloc(9F)`. Otherwise, the `ddi_intr_dup_handler()` call will fail.

The only supported calls on *dup-ed* interrupt handles are `ddi_intr_set_mask(9F)`, `ddi_intr_clr_mask(9F)`, `ddi_intr_get_pending(9F)`, `ddi_intr_enable(9F)`, `ddi_intr_disable(9F)`, and `ddi_intr_free(9F)`.

A call to `ddi_intr_dup_handler()` does not imply that the interrupt source is automatically enabled. Initially, the dup-ed handle is in the disabled state and must be enabled before it can be used by calling `ddi_intr_enable()`. Likewise, `ddi_intr_disable()` must be called to disable the enabled dup-ed interrupt source.

A dup-ed interrupt is removed by calling `ddi_intr_free()` after it has been disabled. The `ddi_intr_remove_handler(9F)` call is not required for a dup-ed handle.

Before removing the original MSI-X interrupt handler, all dup-ed interrupt handlers associated with this MSI-X interrupt must have been disabled and freed. Otherwise, calls to `ddi_intr_remove_handler()` will fail with `DDI_FAILURE`.

See the EXAMPLES section for code that illustrates the use of the `ddi_intr_dup_handler()` function.

**Return Values** The `ddi_intr_dup_handler()` function returns:

`DDI_SUCCESS` On success.

Note that the interface should be verified to ensure that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

`DDI_EINVAL` On encountering invalid input parameters. `DDI_EINVAL` is also returned if a dup is attempted from a dup-ed interrupt or if the hardware device is found not to support MSI-X interrupts.

`DDI_FAILURE` On any implementation specific failure.

**Examples** EXAMPLE 1 Using the `ddi_intr_dup_handler()` function

```
int
add_msix_interrupts(intr_state_t *state)
{
    int x, y;

    /*
     * For this example, assume the device supports multiple
     * interrupt vectors, but only request to be allocated
     * 1 MSI-X to use and then dup the rest.
     */
    if (ddi_intr_get_nintrs(state->dip, DDI_INTR_TYPE_MSIX,
        &state->intr_count) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "Failed to retrieve the MSI-X interrupt count");
        return (DDI_FAILURE);
    }

    state->intr_size = state->intr_count * sizeof (ddi_intr_handle_t);
    state->intr_htable = kmem_zalloc(state->intr_size, KM_SLEEP);

    /* Allocate one MSI-X interrupt handle */
    if (ddi_intr_alloc(state->dip, state->intr_htable,
        DDI_INTR_TYPE_MSIX, state->inum, 1, &state->actual,
        DDI_INTR_ALLOC_STRICT) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "Failed to allocate MSI-X interrupt");
        kmem_free(state->intr_htable, state->intr_size);
        return (DDI_FAILURE);
    }

    /* Get the count of how many MSI-X interrupts we dup */
    state->dup_cnt = state->intr_count - state->actual;
}
```

EXAMPLE 1 Using the `ddi_intr_dup_handler()` function (Continued)

```

if (ddi_intr_get_pri(state->intr_htable[0],
    &state->intr_pri) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Failed to get interrupt priority");
    goto error1;
}

/* Make sure the MSI-X priority is below 'high level' */
if (state->intr_pri >= ddi_intr_get_hilevel_pri()) {
    cmn_err(CE_WARN, "Interrupt PRI is too high");
    goto error1;
}

/*
 * Add the handler for the interrupt
 */
if (ddi_intr_add_handler(state->intr_htable[0],
    (ddi_intr_handler_t *)intr_isr, (caddr_t)state,
    NULL) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Failed to add interrupt handler");
    goto error1;
}

/* Enable the main MSI-X handle first */
if (ddi_intr_enable(state->intr_htable[0]) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Failed to enable interrupt");
    goto error2;
}

/*
 * Create and enable dups of the original MSI-X handler, note
 * that the inum we are using starts at 0.
 */
for (x = 1; x < state->dup_cnt; x++) {
    if (ddi_intr_dup_handler(state->intr_htable[0],
        state->inum + x, &state->intr_htable[x]) != DDI_SUCCESS) {
        for (y = x - 1; y > 0; y--) {
            (void) ddi_intr_disable(state->intr_htable[y]);
            (void) ddi_intr_free(state->intr_htable[y]);
        }

        goto error2;
    }
    if (ddi_intr_enable(state->intr_htable[x]) != DDI_SUCCESS) {
        for (y = x; y > 0; y--) {

```



EXAMPLE 1 Using the `ddi_intr_dup_handler()` function (Continued)

```

        (void) ddi_intr_disable(state->intr_htable[y]);
        (void) ddi_intr_free(state->intr_htable[y]);
    }

    goto error2;
}

return (DDI_SUCCESS);

error2:
    (void) ddi_intr_remove_handler(state->intr_htable[0]);
error1:
    (void) ddi_intr_free(state->intr_htable[0]);

    kmem_free(state->intr_htable, state->intr_size);
    return (DDI_FAILURE);
}

void
remove_msix_interrupts(intr_state_t *state)
{
    int x;

    /*
     * Disable all the handles and free the dup-ed handles
     * before we can remove the main MSI-X interrupt handle.
     */
    for (x = 1; x < state->dup_cnt; x++) {
        (void) ddi_intr_disable(state->intr_htable[x]);
        (void) ddi_intr_free(state->intr_htable[x]);
    }

    /*
     * We can remove and free the main MSI-X handler now
     * that all the dups have been freed.
     */
    (void) ddi_intr_disable(state->intr_htable[0]);
    (void) ddi_intr_remove_handler(state->intr_htable[0]);
    (void) ddi_intr_free(state->intr_htable[0]);

    kmem_free(state->intr_htable, state->intr_size);
}

```

**Context** The `ddi_intr_dup_handler()` function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_clr\\_mask\(9F\)](#), [ddi\\_intr\\_disable\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_free\(9F\)](#), [ddi\\_intr\\_get\\_pending\(9F\)](#), [ddi\\_intr\\_get\\_supported\\_types\(9F\)](#), [ddi\\_intr\\_set\\_mask\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_intr\_enable, ddi\_intr\_block\_enable, ddi\_intr\_disable, ddi\_intr\_block\_disable – enable or disable a given interrupt or range of interrupts

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_intr_enable(ddi_intr_handle_t h);

int ddi_intr_block_enable(ddi_intr_handle_t *h_array, int count);

int ddi_intr_disable(ddi_intr_handle_t h);

int ddi_intr_block_disable(ddi_intr_handle_t *h_array, int count);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

```
ddi_intr_enable()
    h      DDI interrupt handle

ddi_intr_block_enable()
    h_array  Pointer to an array of DDI interrupt handles
    count    Number of interrupts

ddi_intr_disable()
    h      DDI interrupt handle

ddi_intr_block_disable()
    h_array  Pointer to an array of DDI interrupt handles
    count    Number of interrupts
```

**Description** The `ddi_intr_enable()` function enables the interrupt given by the interrupt handle *h*.

The `ddi_intr_block_enable()` function enables a range of interrupts given by the *count* and *h\_array* arguments, where *count* must be at least 1 and *h\_array* is pointer to a count-sized array of interrupt handles.

The `ddi_intr_block_enable()` function can be used only if the device or host bridge supports the block enable/disable feature. The `ddi_intr_get_cap()` function returns the RO flag `DDI_INTR_FLAG_BLOCK` if the device or host bridge supports the interrupt block enable/disable feature for the given interrupt type. The `ddi_intr_block_enable()` function is useful for enabling MSI interrupts when the optional per-vector masking capability is not supported.

The `ddi_intr_enable()` or `ddi_intr_block_enable()` functions must be called after the required interrupt resources are allocated with `ddi_intr_alloc()`, the interrupt handlers are added through `ddi_intr_add_handler()`, and the required locks are initialized by [mutex\(9F\)](#) or [rwlock\(9F\)](#).

Once enabled by either of the enable calls, the interrupt can be taken and passed to the driver's interrupt service routine. Enabling an interrupt implies clearing any system or device mask bits associated with the interrupt.

The `ddi_intr_disable()` function disables the interrupt given by the interrupt handle *h*.

The `ddi_intr_block_disable()` function disables a range of interrupts given by the *count* and *h\_array* arguments, where *count* must be at least 1 and *h\_array* is pointer to a count-sized array of interrupt handles.

The `ddi_intr_block_disable()` function can be used only if the device or host bridge supports the block enable/disable feature. The `ddi_intr_get_cap()` function returns the RO flag `DDI_INTR_FLAG_BLOCK` if the device or host bridge supports the interrupt block enable/disable feature for the given interrupt type. The `ddi_intr_block_disable()` function is useful for disabling MSI interrupts when the optional per-vector masking capability is not supported.

The `ddi_intr_disable()` or `ddi_intr_block_disable()` functions must be called before removing the interrupt handler and freeing the corresponding interrupt with `ddi_intr_remove_handler()` and `ddi_intr_free()`, respectively. The `ddi_intr_block_disable()` function should be called if the `ddi_intr_block_enable()` function was used to enable the interrupts.

**Return Values** The `ddi_intr_enable()`, `ddi_intr_block_enable()`, `ddi_intr_disable()`, and `ddi_intr_block_disable()` functions return:

- `DDI_SUCCESS`      On success.
- `DDI_EINVAL`      On encountering invalid input parameters.
- `DDI_FAILURE`      On any implementation specific failure.

**Context** The `ddi_intr_enable()`, `ddi_intr_block_enable()`, `ddi_intr_disable()`, and `ddi_intr_block_disable()` functions can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_dup\\_handler\(9F\)](#), [ddi\\_intr\\_free\(9F\)](#), [ddi\\_intr\\_get\\_cap\(9F\)](#), [ddi\\_intr\\_remove\\_handler\(9F\)](#), [mutex\(9F\)](#), [rwlock\(9F\)](#)

*Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to DDI\_SUCCESS. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

If a device driver that uses MSI and MSI-X interrupts resets the device, the device might reset its configuration space modifications. Such a reset could cause a device driver to lose any MSI and MSI-X interrupt usage settings that have been applied.

**Name** ddi\_intr\_get\_cap, ddi\_intr\_set\_cap – get or set interrupt capabilities for a given interrupt type

**Synopsis** #include <sys/types.h>  
#include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_intr_get_cap(ddi_intr_handle_t h, int *flagsp);
```

```
int ddi_intr_set_cap(ddi_intr_handle_t h, int flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_intr\_get\_cap()

*h* DDI interrupt handle

*flagsp* Pointer to the capability flags returned for this handle

ddi\_intr\_set\_cap()

*h* DDI interrupt handle

*flags* Contains the capability flag to be set

**Description** The ddi\_intr\_get\_cap() function returns the interrupt capability flags for the interrupt handle *h*. Upon a successful return, the flags are returned in the integer pointed to by the *flagsp* argument.

These flags are typically combinations of the following:

DDI\_INTR\_FLAG\_EDGE For discrete interrupts, the host supports edge type of trigger. This flag is not returned for DDI\_INTR\_TYPE\_MSI or DDI\_INTR\_TYPE\_MSIX interrupt types. This is a read-write (RW) flag.

DDI\_INTR\_FLAG\_LEVEL For discrete interrupts the host supports level, edge, or both types of triggers. This flag is not returned for DDI\_INTR\_TYPE\_MSI or DDI\_INTR\_TYPE\_MSIX interrupt types.

DDI\_INTR\_FLAG\_MASKABLE The interrupt can be masked either by the device or by the host bridge, or optionally by the host. This is a read-only (RO) flag.

DDI\_INTR\_FLAG\_PENDING The interrupt supports an interrupt pending bit. This is a read-only (RO) flag.

DDI\_INTR\_FLAG\_BLOCK All interrupts of the given type must be block-enabled and are not individually maskable. This is a read-only (RO) flag.

The ddi\_intr\_set\_cap() function allows a driver to specify the capability flags for the interrupt handle *h*. Only DDI\_INTR\_FLAG\_LEVEL and DDI\_INTR\_FLAG\_EDGE flags can be set.

Some devices can support both level and edge capability and either can be set by using the `ddi_intr_set_cap()` function. Setting the capability flags is device and platform dependent.

The `ddi_intr_set_cap()` function can be called after interrupts are allocated and prior to adding the interrupt handler. For all other times it returns failure.

**Return Values** The `ddi_intr_get_cap()` and `ddi_intr_set_cap()` functions return:

`DDI_SUCCESS` On success.

`DDI_EINVAL` On encountering invalid input parameters.

`DDI_FAILURE` On any implementation specific failure.

`DDI_ENOTSUP` On device not supporting operation.

**Context** The `ddi_intr_get_cap()` and `ddi_intr_set_cap()` functions can be called from either user or kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_block\\_enable\(9F\)](#), [ddi\\_intr\\_get\\_nintrs\(9F\)](#), [ddi\\_intr\\_get\\_pending\(9F\)](#), [ddi\\_intr\\_get\\_supported\\_types\(9F\)](#), [ddi\\_intr\\_set\\_mask\(9F\)](#)

*Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_get\_hilevel\_pri – get minimum priority level for a high-level interrupt

**Synopsis** `#include <sys/types.h>`  
`#include <sys/conf.h>`  
`#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
int ddi_intr_get_hilevel_pri(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** Upon a successful return, the `ddi_intr_get_hilevel_pri()` function returns the minimum priority level for a high-level interrupt. The return priority value can be used to compare to other priority values, such as those returned from [ddi\\_intr\\_get\\_pri\(9F\)](#), to determine if a given interrupt priority is a high-level interrupt.

High-level interrupts must be handled without using system services that manipulate thread or process states, because such interrupts are not blocked by the scheduler.

In addition, high-level interrupt handlers must take care to do a minimum of work because they cannot be preempted.

A typical high-level interrupt handler puts data into a circular buffer and schedule a soft interrupt by calling `ddi_intr_trigger_softint()`. The circular buffer can be protected by using a mutex that is properly initialized for the interrupt handler.

The `ddi_intr_get_hilevel_pri()` function can be used before calling `ddi_intr_add_handler()` to help determine which type of interrupt handler can be used. Most device drivers are designed with the knowledge that supported devices always generate low level interrupts. On some machines, however, interrupts are high-level above the scheduler level and on other machines they are not. Devices such as those those using SBus interrupts or VME bus level 6 or 7 interrupts must use the `ddi_intr_get_hilevel_pri()` function to test the type of interrupt handler that can be used.

**Return Values** The `ddi_intr_get_hilevel_pri()` function returns the priority value for a high-level interrupt.

**Context** The `ddi_intr_get_hilevel_pri()` function can be called from either user or kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed



**See Also** [attributes\(5\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_intr\\_trigger\\_softint\(9F\)](#), [mutex\(9F\)](#)

*Writing Device Drivers*

**Name** `ddi_intr_get_nintrs`, `ddi_intr_get_navail` – return number of interrupts supported or available for a given interrupt type

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_intr_get_nintrs(dev_info_t *dip, int type, int *nintrsp);
int ddi_intr_get_navail(dev_info_t *dip, int type, int *navailp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** `ddi_intr_get_nintrs()`

*dip* Pointer to `dev_info` structure

*type* Interrupt type

*nintrsp* Pointer to number of interrupts of the given type that are supported by the system

`ddi_intr_get_navail()`

*dip* Pointer to `dev_info` structure

*type* Interrupt type

*navailp* Pointer to number of interrupts of the given type that are currently available from the system

**Description** The `ddi_intr_get_nintrs()` function returns the number of interrupts of the given *type* supported by a particular hardware device. On a successful return, the number of supported interrupts is returned as an integer pointed to by the *nintrsp* argument.

If the hardware device is not found to support any interrupts of the given *type*, the `DDI_INTR_NOTFOUND` failure is returned rather than a zero in *nintrsp*.

The `ddi_intr_get_navail()` function returns the number of interrupts of a given *type* that is available to a particular hardware device. On a successful return, the number of available interrupts is returned as an integer pointed to by *navailp*.

The hardware device may support more than one interrupt and can request that all interrupts be allocated. The host software can then use policy-based decisions to determine how many interrupts are made available to the device. Based on the determination, a value is returned that should be used to allocate interrupts with the `ddi_int_alloc()` function.

The `ddi_intr_get_supported_types(9F)` function returns a list of valid supported types for the given hardware device. It must be called prior to calling either the `ddi_intr_get_nintrs()` or `ddi_intr_get_navail()`.

**Return Values** The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions return:

<code>DDI_SUCCESS</code>	On success.
<code>DDI_EINVAL</code>	On encountering invalid input parameters.
<code>DDI_INTR_NOTFOUND</code>	On not finding any interrupts for the given interrupt type.
<code>DDI_FAILURE</code>	On any implementation specific failure.

**Context** The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions can be called from either user or kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_get\\_supported\\_types\(9F\)](#)

#### *Writing Device Drivers*

**Notes** The `ddi_intr_get_nintrs()` and `ddi_intr_get_navail()` functions can be called at any time, even if the driver has added an interrupt handler for a given interrupt specification.

On x86 platforms, the number of interrupts returned by the `ddi_intr_get_navail()` function might need to be further reduced by the number of interrupts available for each interrupt priority on the system. In that case, drivers should use different priorities for some of the interrupts.

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_get\_pending – get pending bit for a given interrupt

**Synopsis** `#include <sys/types.h>`  
`#include <sys/conf.h>`  
`#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
int ddi_intr_get_pending(ddi_intr_handle_t h, int *pendingp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *h* DDI interrupt handle  
*pendingp* Pointer to the pending capability returned for this handle

**Description** The `ddi_intr_get_pending()` function returns non-zero as the integer pointed to by the *pendingp* argument if a corresponding interrupt is pending. The corresponding interrupt handle *h* must already be enabled. The call succeeds if the device or host bridge supports the ability to read the interrupt pending bits of its interrupts. The driver should use `ddi_intr_get_cap()` function to see if the `DDI_INTR_FLAG_PENDING` flag is returned to indicate that interrupts support interrupt pending bits.

If the `DDI_INTR_FLAG_PENDING` capability is not supported, `ddi_intr_get_pending()` returns `DDI_ENOTSUP` and zero in *pendingp*.

**Return Values** The `ddi_intr_get_pending()` function returns:

`DDI_SUCCESS` On success.  
`DDI_EINVAL` On encountering invalid input parameters.  
`DDI_FAILURE` On any implementation specific failure.  
`DDI_ENOTSUP` On device not supporting operation.

**Context** The `ddi_intr_get_pending()` function can be called from either user or kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_block\\_enable\(9F\)](#), [ddi\\_intr\\_block\\_disable\(9F\)](#), [ddi\\_intr\\_clr\\_mask\(9F\)](#), [ddi\\_intr\\_disable\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_set\\_mask\(9F\)](#)

*Writing Device Drivers*

**Notes** Any consumer of this interface should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_get\_pri, ddi\_intr\_set\_pri – get or set priority of a given interrupt

**Synopsis** #include <sys/types.h>  
#include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_intr_get_pri(ddi_intr_handle_t h, uint_t *prip);  
int ddi_intr_set_pri(ddi_intr_handle_t h, uint_t pri);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_intr\_get\_pri()

*h* DDI interrupt handle

*prip* Pointer to the priority returned for this handle

ddi\_intr\_set\_pri()

*h* DDI interrupt handle

*pri* Contains the priority to be set

**Description** The ddi\_intr\_get\_pri() function returns the current priority of the interrupt handle *h* of a given device. Upon a successful return, *prip* points to a small integer value, typically in the DDI\_INTR\_PRI\_MIN...DDI\_INTR\_PRI\_MAX range, that represents the current software priority setting for the interrupt. See <sys/ddi\_intr.h> for values of DDI\_INTR\_PRI\_MIN or DDI\_INTR\_PRI\_MAX.

The ddi\_intr\_get\_pri() function can be called any time, even if the driver adds an interrupt handler for the interrupt specification.

The software priority returned from ddi\_intr\_get\_pri() can be used in calls to mutex\_init() and rw\_init().

The ddi\_intr\_set\_pri() function sets the priority *pri* of the interrupt handle *h* of a given device. The function validates that the argument is within the supported range.

The ddi\_intr\_set\_pri() function can only be called prior to adding the interrupt handler or when an interrupt handler is unassigned. DDI\_FAILURE is returned in all other cases.

**Return Values** The ddi\_intr\_get\_pri() and ddi\_intr\_set\_pri() functions return:

DDI\_SUCCESS On success.

DDI\_EINVAL On encountering invalid input parameters.

DDI\_FAILURE On any implementation specific failure.

DDI\_ENOTSUP On device not supporting operation.

**Context** The `ddi_intr_get_pri()` and `ddi_intr_set_pri()` functions can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [mutex\\_init\(9F\)](#), [rw\\_init\(9F\)](#)

*Writing Device Drivers*

**Notes** The priority returned from `ddi_intr_get_pri()` should be typecast by calling the `DDI_INTR_PRI` macro before passing it onto [mutex\\_init\(9F\)](#).

Consumers of these interfaces should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_get\_supported\_types – return information on supported hardware interrupt types

**Synopsis**

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_intr_get_supported_types(dev_info_t *dip, int *typesp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

*dip* Pointer to dev\_info structure

*typesp* Pointer to supported interrupt types

**Description** The ddi\_intr\_get\_supported\_types() function retrieves the interrupt types supported by a particular hardware device and by the system software. Upon successful return, the supported types are returned as a bit mask in the integer pointed to by the *typesp* argument. See <sys/ddi\_intr.h> for a list of interrupts that can be returned by a hardware device.

For PCI devices that support MSI and/or MSI-X based hardware, this interface returns only the interrupt types that are supported by all the hardware in the path to the hardware device.

An interrupt type is usable by the hardware device if it is returned by the ddi\_intr\_get\_supported\_types() function. The device driver can be programmed to use one of the returned interrupt types to receive hardware interrupts.

**Return Values** The ddi\_intr\_get\_supported\_types() function returns:

DDI_SUCCESS	On success.
DDI_EINVAL	On encountering invalid input parameters.
DDI_INTR_NOTFOUND	Returned when the hardware device is found not to support any hardware interrupts.

**Context** The ddi\_intr\_get\_supported\_types() function can be called from user or kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [pci\(4\)](#), [attributes\(5\)](#), [pcmcia\(7D\)](#), [sysbus\(4\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#)

*Writing Device Drivers*



**Notes** The `ddi_intr_get_supported_types()` function can be called by the device driver even at any time if the driver has added an interrupt handler for a given interrupt type.

Soft interrupts are always usable and are not returned by this interface.

Any consumer of this interface should verify that the return value is not equal to `DDI_SUCCESS`. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_hilevel – indicate interrupt handler type

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_intr_hilevel(dev_info_t *dip, uint_t inumber);
```

**Interface Level** Solaris DDI specific (Solaris DDI). This interface is obsolete. Use the new interrupt interfaces referenced in [Intro\(9F\)](#). Refer to *Writing Device Drivers* for more information.

**Parameters** *dip* Pointer to dev\_info structure.

*inumber* Interrupt number.

**Description** The ddi\_intr\_hilevel() function returns non-zero if the specified interrupt is a “high level” interrupt.

High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler.

In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable.

A typical high level interrupt handler would put data into a circular buffer and schedule a soft interrupt by calling ddi\_trigger\_softintr(). The circular buffer could be protected by using a mutex that was properly initialized for the interrupt handler.

The ddi\_intr\_hilevel() function can be used before calling ddi\_add\_intr() to decide which type of interrupt handler should be used. Most device drivers are designed with the knowledge that the devices they support will always generate low level interrupts, however some devices, for example those using SBus or VME bus level 6 or 7 interrupts must use this test because on some machines those interrupts are high level (above the scheduler level) and on other machines they are not.

**Return Values** non-zero indicates a high-level interrupt.

**Context** These functions can be called from useruser, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [ddi\\_add\\_intr\(9F\)](#), [Intro\(9F\)](#), [mutex\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_intr\_set\_mask, ddi\_intr\_clr\_mask – set or clear mask for a given interrupt

**Synopsis** #include <sys/types.h>  
#include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_intr_set_mask(ddi_intr_handle_t h);
```

```
int ddi_intr_clr_mask(ddi_intr_handle_t h);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *h* DDI interrupt handle

**Description** The `ddi_intr_set_mask()` function masks the given interrupt pointed to by the device's interrupt handle *h* if the device or host bridge supports the masking operation. The `ddi_intr_get_cap()` function returns the RO flag `DDI_INTR_FLAG_MASKABLE` if the device or host bridge supports interrupt mask bits for the given interrupt type. In flight interrupts can still be taken and delivered to the driver.

The `ddi_intr_clr_mask()` function unmask the given interrupt pointed by the device's interrupt handle *h* if the device or host bridge supports the masking operation.

The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions should be called only if an interrupt is enabled. Otherwise the framework will return `DDI_EINVAL` to such calls.

The mask cannot be cleared directly if the OS implementation has also temporarily masked the interrupt. A call to `ddi_intr_clr_mask()` must be preceded by a call to `ddi_intr_set_mask()`. It is not necessary to call `ddi_intr_clr_mask()` when adding and enabling the interrupt.

**Return Values** The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions return:

`DDI_SUCCESS` On success.

`DDI_EINVAL` On encountering invalid input parameters or when an interrupt is not enabled.

`DDI_FAILURE` On any implementation specific failure.

`DDI_ENOTSUP` On device not supporting operation.

**Context** The `ddi_intr_set_mask()` and `ddi_intr_clr_mask()` functions can be called from any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [ddi\\_intr\\_block\\_disable\(9F\)](#), [ddi\\_intr\\_block\\_enable\(9F\)](#), [ddi\\_intr\\_disable\(9F\)](#), [ddi\\_intr\\_enable\(9F\)](#), [ddi\\_intr\\_get\\_pending\(9F\)](#)

*Writing Device Drivers*

**Notes** Consumers of these interfaces should verify that the return value is not equal to DDI\_SUCCESS. Incomplete checking for failure codes could result in inconsistent behavior among platforms.

**Name** ddi\_intr\_set\_nreq – set the number of interrupts requested for a device driver instance

**Synopsis** #include <sys/ddi\_intr.h>

```
int ddi_intr_set_nreq(dev_info_t *dip, int nreq);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* Pointer to the dev\_info structure.

*nreq* Number of interrupts requested.

**Description** The ddi\_intr\_set\_nreq() function changes the number of interrupts requested by a device driver instance.

The *nreq* parameter is the total number of interrupt resources that this instance of the device driver would like to have available. The *nreq* parameter includes any interrupt resources already allocated by the driver. For example, if the driver instance already has two MSI-X vectors and it wants two more, it should call this function with an *nreq* parameter set to four.

The *nreq* parameter can be any value between one and the maximum number of interrupts supported by the device hardware, as reported by a call to the [ddi\\_intr\\_get\\_nintrs\(9F\)](#) function. The driver receives a callback notifying it in cases when it must release any previously allocated interrupts, or when it is allowed to allocate more interrupts as a result of its new *nreq* parameter.

The ddi\_intr\_set\_nreq() function is not supported unless a driver is already consuming interrupts, and if it has a registered callback handler that can process actions related to changes in interrupt availability. See [ddi\\_cb\\_register\(9F\)](#) for an explanation on how to enable this functionality.

**Return Values** The ddi\_intr\_set\_nreq() function returns:

DDI\_SUCCESS on success

DDI\_EINVAL The operation is invalid because the *nreq* parameter is not a legal value

DDI\_ENOTSUP The operation is not supported. The driver must have a registered callback, and the system must have interrupt pools implemented.

DDI\_FAILURE Implementation specific failure

**Context** These functions can be called from kernel, non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [ddi\\_cb\\_register\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_get\\_nintrs\(9F\)](#)

**Notes** The Interrupt Resource Management feature is limited to device driver instances that are using MSI-X interrupts (interrupt type `DDI_INTR_TYPE_MSIX`). Attempts to use this function for any other type of interrupts fails with `DDI_ENOTSUP`.

The total number of interrupts requested by the driver is initially defined by the *count* parameter provided by the driver's first call to the [ddi\\_intr\\_alloc\(9F\)](#) function, specifically during the driver instance's [attach\(9E\)](#) routine. The `ddi_intr_set_nreq()` function is only used if the driver instance experiences changes in its I/O load. In response to increased I/O load, the driver may want to request additional interrupt resources. In response to diminished I/O load, the driver may volunteer to return extra interrupt resources back to the system.

**Name** ddi\_io\_get8, ddi\_io\_get16, ddi\_io\_get32, ddi\_io\_getb, ddi\_io\_getl, ddi\_io\_getw – read data from the mapped device register in I/O space

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
uint8_t ddi_io_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_io_get16(ddi_acc_handle_t handle, uint16_t *dev_addr);
uint32_t ddi_io_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_io\_get8(), ddi\_io\_get16(), ddi\_io\_get32(), ddi\_io\_getb(), ddi\_io\_getl(), and ddi\_io\_getw() functions are obsolete. The [ddi\\_get8\(9F\)](#) function replaces ddi\_io\_get8(). The [ddi\\_get16\(9F\)](#) function replaces ddi\_io\_get16(). The [ddi\\_get32\(9F\)](#) function replaces ddi\_io\_get32(). The [ddi\\_get8\(9F\)](#) function replaces ddi\_io\_getb(). The [ddi\\_get32\(9F\)](#) function replaces ddi\_io\_getl(). The [ddi\\_get16\(9F\)](#) function replaces ddi\_io\_getw().

**Parameters** *handle* Data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).  
*dev\_addr* Device address.

**Description** These routines generate a read of various sizes from the device address, *dev\_addr*, in I/O space. The ddi\_io\_get8(), ddi\_io\_get16(), and ddi\_io\_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address, *dev\_addr*.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Obsolete

**See Also** [isa\(4\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_get16\(9F\)](#), [ddi\\_get32\(9F\)](#), [ddi\\_io\\_put8\(9F\)](#), [ddi\\_io\\_rep\\_get8\(9F\)](#), [ddi\\_io\\_rep\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_iomin – find minimum alignment and transfer size for DMA

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_iomin(dev_info_t *dip, int initial, int streaming);
```

**Interface Level** Solaris DDI specific (Solaris DDI). This interface is obsolete.

**Parameters**

*dip* A pointer to the device's `dev_info` structure.

*initial* The initial minimum DMA transfer size in bytes. This may be zero or an appropriate `dlim_minxfer` value for device's `ddi_dma_lim` structure (see [ddi\\_dma\\_lim\\_sparc\(9S\)](#) or [ddi\\_dma\\_lim\\_x86\(9S\)](#)). This value must be a power of two.

*streaming* This argument, if non-zero, indicates that the returned value should be modified to account for *streaming* mode accesses (see [ddi\\_dma\\_req\(9S\)](#) for a discussion of streaming versus non-streaming access mode).

**Description** The `ddi_iomin()` function, finds out the minimum DMA transfer size for the device pointed to by *dip*. This provides a mechanism by which a driver can determine the effects of underlying caches as well as intervening bus adapters on the granularity of a DMA transfer.

**Return Values** The `ddi_iomin()` function returns the minimum DMA transfer size for the calling device, or it returns zero, which means that you cannot get there from here.

**Context** This function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [ddi\\_dma\\_devalign\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*



**Name** ddi\_iopb\_alloc, ddi\_iopb\_free – allocate and free non-sequentially accessed memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_iopb_alloc(dev_info_t *dip, ddi_dma_lim_t *limits,
                 uint_t length, caddr_t *iopbp);

void ddi_iopb_free(caddr_t iopb);
```

**Interface Level** These interfaces are obsolete. Use [ddi\\_dma\\_mem\\_alloc\(9F\)](#) instead of `ddi_iopb_alloc()`. Use [ddi\\_dma\\_mem\\_free\(9F\)](#) instead of `ddi_iopb_free()`.

### Parameters

<code>ddi_iopb_alloc()</code>	<i>dip</i>	A pointer to the device's <code>dev_info</code> structure.
	<i>limits</i>	A pointer to a DMA limits structure for this device (see <a href="#">ddi_dma_lim_sparc(9S)</a> or <a href="#">ddi_dma_lim_x86(9S)</a> ). If this pointer is NULL, a default set of DMA limits is assumed.
	<i>length</i>	The length in bytes of the desired allocation.
	<i>iopbp</i>	A pointer to a <code>caddr_t</code> . On a successful return, <i>iopbp</i> points to the allocated storage.
<code>ddi_iopb_free()</code>	<i>iopb</i>	The <i>iopb</i> returned from a successful call to <code>ddi_iopb_alloc()</code> .

**Description** The `ddi_iopb_alloc()` function allocates memory for DMA transfers and should be used if the device accesses memory in a non-sequential fashion, or if synchronization steps using [ddi\\_dma\\_sync\(9F\)](#) should be as lightweight as possible, due to frequent use on small objects. This type of access is commonly known as *consistent* access. The allocation will obey the alignment and padding constraints as specified in the *limits* argument and other limits imposed by the system.

Note that you still must use DMA resource allocation functions (see [ddi\\_dma\\_setup\(9F\)](#)) to establish DMA resources for the memory allocated using `ddi_iopb_alloc()`.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using [ddi\\_dma\\_sync\(9F\)](#) or [ddi\\_dma\\_free\(9F\)](#) are still required. The DMA resources will be allocated so that these synchronization steps are as efficient as possible.

The `ddi_iopb_free()` function frees up memory allocated by `ddi_iopb_alloc()`.

**Return Values** The `ddi_iopb_alloc()` function returns:

DDI_SUCCESS	Memory successfully allocated.
DDI_FAILURE	Allocation failed.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_free\(9F\)](#), [ddi\\_dma\\_mem\\_alloc\(9F\)](#), [ddi\\_dma\\_mem\\_free\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_mem\\_alloc\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Notes** This function uses scarce system resources. Use it selectively.

**Name** ddi\_io\_put8, ddi\_io\_put16, ddi\_io\_put32, ddi\_io\_putb, ddi\_io\_putl, ddi\_io\_putw – write data to the mapped device register in I/O space

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_io_put8(ddi_acc_handle_t handle, uint8_t *dev_addr,
                uint8_t value);

void ddi_io_put16(ddi_acc_handle_t handle, uint16_t *dev_addr,
                 uint16_t value);

void ddi_io_put32(ddi_acc_handle_t handle, uint32_t *dev_addr,
                 uint32_t value);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_io\_put8(), ddi\_io\_put16(), ddi\_io\_put32(), ddi\_io\_putb(), ddi\_io\_putl(), and ddi\_io\_putw() functions are obsolete. The [ddi\\_put8\(9F\)](#) function replaces ddi\_io\_put8(). The [ddi\\_put16\(9F\)](#) function replaces ddi\_io\_put16(). The [ddi\\_put32\(9F\)](#) function replaces ddi\_io\_put32(). The [ddi\\_put8\(9F\)](#) function replaces ddi\_io\_putb(). The [ddi\\_put32\(9F\)](#) function replaces ddi\_io\_putl(). The [ddi\\_put16\(9F\)](#) function replaces ddi\_io\_putw().

**Parameters** *handle* Data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).  
*dev\_addr* Base device address.  
*value* Data to be written to the device.

**Description** These routines generate a write of various sizes to the device address, *dev\_addr*, in I/O space. The ddi\_io\_put8(), ddi\_io\_put16(), and ddi\_io\_put32() functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address, *dev\_addr*.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [isa\(4\)](#), [ddi\\_io\\_get8\(9F\)](#), [ddi\\_io\\_rep\\_get8\(9F\)](#), [ddi\\_io\\_rep\\_put8\(9F\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_put16\(9F\)](#), [ddi\\_put32\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_io\_rep\_get8, ddi\_io\_rep\_get16, ddi\_io\_rep\_get32, ddi\_io\_rep\_getb, ddi\_io\_rep\_getl, ddi\_io\_rep\_getw – read multiple data from the mapped device register in I/O space

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_io_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
    uint8_t *dev_addr, size_t repcount);

void ddi_io_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
    uint16_t *dev_addr, size_t repcount);

void ddi_io_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
    uint32_t *dev_addr, size_t repcount);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_io\_rep\_get8(), ddi\_io\_rep\_get16(), ddi\_io\_rep\_get32(), ddi\_io\_rep\_getb(), ddi\_io\_rep\_getl(), and ddi\_io\_rep\_getw() functions are obsolete. The [ddi\\_rep\\_get8\(9F\)](#) function replaces ddi\_io\_rep\_get8(). The [ddi\\_rep\\_get16\(9F\)](#) function replaces ddi\_io\_rep\_get16(). The [ddi\\_rep\\_get32\(9F\)](#) function replaces ddi\_io\_rep\_get32(). The [ddi\\_rep\\_get8\(9F\)](#) function replaces ddi\_io\_rep\_getb(). The [ddi\\_rep\\_get32\(9F\)](#) function replaces ddi\_io\_rep\_getl(). The [ddi\\_rep\\_get16\(9F\)](#) function replaces ddi\_io\_rep\_getw().

**Parameters** *handle* The data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

*host\_addr* Base host address.

*dev\_addr* Base device address.

*repcount* Number of data accesses to perform.

**Description** These routines generate multiple reads from the device address, *dev\_addr*, in I/O space. *repcount* data is copied from the device address, *dev\_addr*, to the host address, *host\_addr*. For each input datum, the ddi\_io\_rep\_get8(), ddi\_io\_rep\_get16(), and ddi\_io\_rep\_get32() functions read 8 bits, 16 bits, and 32 bits of data, respectively, from the device address. *host\_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [isa\(4\)](#), [ddi\\_io\\_get8\(9F\)](#), [ddi\\_io\\_put8\(9F\)](#), [ddi\\_io\\_rep\\_put8\(9F\)](#),  
[ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_get16\(9F\)](#),  
[ddi\\_rep\\_get32\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_io\_rep\_put8, ddi\_io\_rep\_put16, ddi\_io\_rep\_put32, ddi\_io\_rep\_putb, ddi\_io\_rep\_putl, ddi\_io\_rep\_putw – write multiple data to the mapped device register in I/O space

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_io_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
    uint8_t *dev_addr, size_t recount);
```

```
void ddi_io_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
    uint16_t *dev_addr, size_t recount);
```

```
void ddi_io_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
    uint32_t *dev_addr, size_t recount);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_io\_rep\_put8(), ddi\_io\_rep\_put16(), ddi\_io\_rep\_put32(), ddi\_io\_rep\_putb(), ddi\_io\_rep\_putl(), and ddi\_io\_rep\_putw() functions are obsolete. The [ddi\\_rep\\_put8\(9F\)](#) function replaces ddi\_io\_rep\_put8(). The [ddi\\_rep\\_put16\(9F\)](#) function replaces ddi\_io\_rep\_put16(). The [ddi\\_rep\\_put32\(9F\)](#) function replaces ddi\_io\_rep\_put32(). The [ddi\\_rep\\_put8\(9F\)](#) function replaces ddi\_io\_rep\_putb(). The [ddi\\_rep\\_put32\(9F\)](#) function replaces ddi\_io\_rep\_putl(). The [ddi\\_rep\\_put16\(9F\)](#) function replaces ddi\_io\_rep\_putw().

**Parameters**

<i>handle</i>	Data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>host_addr</i>	Base host address.
<i>dev_addr</i>	Base device address.
<i>recount</i>	Number of data accesses to perform.

**Description** These routines generate multiple writes to the device address, *dev\_address*, in I/O space. *recount* data is copied from the host address, *host\_addr*, to the device address, *dev\_addr*. For each input datum, the ddi\_io\_rep\_put8(), ddi\_io\_rep\_put16(), and ddi\_io\_rep\_put32() functions write 8 bits, 16 bits, and 32 bits of data, respectively, to the device address. *host\_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [isa\(4\)](#), [ddi\\_io\\_get8\(9F\)](#), [ddi\\_io\\_put8\(9F\)](#), [ddi\\_io\\_rep\\_get8\(9F\)](#),  
[ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#), [ddi\\_rep\\_put16\(9F\)](#), [ddi\\_rep\\_put32\(9F\)](#),  
[ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_log\_sysevent – log system event for drivers

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_log_sysevent(dev_info_t *dip, char *vendor,
    char *class, char *subclass, nvlist_t *attr_list,
    sysevent_id_t *eidp, int sleep_flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dip</i>	A pointer to the <code>dev_info</code> node for this driver.
<i>vendor</i>	A pointer to a string defining the vendor. Third-party drivers should use their company's stock symbol (or similarly enduring identifier). Sun-supplied drivers should use <code>DDI_VENDOR_SUNW</code> .
<i>class</i>	A pointer to a string defining the event class.
<i>subclass</i>	A pointer to a string defining the event subclass.
<i>attr_list</i>	A pointer to an <code>nvlist_t</code> , listing the name-value attributes associated with the event or <code>NULL</code> if there are no such attributes for this event.
<i>eidp</i>	The address of a <code>sysevent_id_t</code> structure in which the event's sequence number and timestamp are returned if the event is successfully queued. May be <code>NULL</code> if this information is not of interest. See below for the definition of <code>sysevent_id_t</code> .
<i>sleep_flag</i>	Indicates how a caller wants to handle the possibility of resources not being available. If <i>sleep_flag</i> is <code>DDI_NOSLEEP</code> , the caller does not care if the allocation fails or the queue is full and can handle a failure appropriately. If <i>sleep_flag</i> is <code>DDI_SLEEP</code> , the caller wishes to have the allocation and queuing routines wait for resources to become available.

**Description** The `ddi_log_sysevent()` function causes a system event, of the specified class and subclass, to be generated on behalf of the driver and queued for delivery to `syseventd`, the user-land `syseventd` daemon.

The publisher string for the event is constructed using the vendor name and driver name, with the format:

```
"<vendor>:kern:<driver-name>"
```

The two fields of `eidp`, `eid_seq` and `eid_ts`, are sufficient to uniquely identify an event.

**Structure Members** The structure members of `sysevent_id_t` are:

```
uint64_t    eid_seq;        /* sysevent sequence number */
hrtime_t    eid_ts;        /* sysevent timestamp */
```



**Return Values** The `ddi_log_sysevent()` function returns:

<code>DDI_SUCCESS</code>	The event has been queued for delivery successfully.
<code>DDI_ENOMEM</code>	There is not enough memory to queue the system event at this time. <code>DDI_ENOMEM</code> cannot be returned when <i>sleep_flag</i> is <code>DDI_SLEEP</code> .
<code>DDI_EBUSY</code>	The system event queue is full at this time. <code>DDI_EBUSY</code> cannot be returned when <i>sleep_flag</i> is <code>DDI_SLEEP</code> .
<code>DDI_ETRANSPORT</code>	The syseventd daemon is not responding and events cannot be queued or delivered at this time. <code>DDI_ETRANSPORT</code> can be returned even when <i>sleep_flag</i> is <code>DDI_SLEEP</code> .
<code>DDI_ECONTEXT</code>	<i>sleep_flag</i> is <code>DDI_SLEEP</code> and the driver is running in interrupt context.

`ddi_log_sysevent` supports the following data types:

`DATA_TYPE_BYTE`

`DATA_TYPE_INT16`

`DATA_TYPE_UINT16`

`DATA_TYPE_INT32`

`DATA_TYPE_UINT32`

`DATA_TYPE_INT64`

`DATA_TYPE_UINT64`

`DATA_TYPE_STRING`

`DATA_TYPE_BYTE_ARRAY`

`DATA_TYPE_INT16_ARRAY`

`DATA_TYPE_UINT16_ARRAY`

DATA\_TYPE\_INT32\_ARRAY

DATA\_TYPE\_UINT32\_ARRAY

DATA\_TYPE\_INT64\_ARRAY

DATA\_TYPE\_UINT64\_ARRAY

**Context** The `ddi_log_sysevent()` function can be called from user, interrupt, or kernel context, except when *sleep\_flag* is `DDI_SLEEP`, in which case it cannot be called from interrupt context.

**Examples** EXAMPLE 1 Logging System Event with No Attributes

```
if (ddi_log_sysevent(dip, DDI_VENDOR_SUNW, "class", "subclass",
    NULL, NULL, DDI_SLEEP) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "error logging system event\n");
}
```

EXAMPLE 2 Logging System Event with Two Name/Value Attributes, an Integer and a String

```
nvlist_t    *attr_list;
sysevent_id_t  eid;

if (nvlist_alloc(&attr_list, NV_UNIQUE_NAME_TYPE, KM_SLEEP) == 0)
{
    err = nvlist_add_uint32(attr_list, int_name, int_value);
    if (err == 0)
        err = nvlist_add_string(attr_list, str_name, str_value);
    if (err == 0)
        err = ddi_log_sysevent(dip, DDI_VENDOR_SUNW,
            "class", "subclass", attr_list, &eid, DDI_SLEEP);
    if (err != DDI_SUCCESS)
        cmn_err(CE_WARN, "error logging system event\n");
    nvlist_free(attr_list);
}
```

EXAMPLE 3 Use Timeout to Handle nvlist and System Event Resource Allocation Failures

Since no blocking calls are made, this example would be useable from a driver needing to generate an event from interrupt context.

```
static int
xx_se_timeout_handler(xx_state_t *xx)
{
    xx->xx_timeoutid = (xx_generate_event(xx) ?
        timeout(xx_se_timeout_handler, xx, 4) : 0);
}
```

**EXAMPLE 3** Use Timeout to Handle nvlist and System Event Resource Allocation Failures  
(Continued)

```

static int
xx_generate_event(xx_state_t *xx)
{
    int err;

    err = nvlist_alloc(&xx->xx_ev_attrlist, NV_UNIQUE_NAME_TYPE, 0);
    if (err != 0)
        return (1);
    err = nvlist_add_uint32(&xx->xx_ev_attrlist,
        xx->xx_ev_name, xx->xx_ev_value);
    if (err != 0) {
        nvlist_free(xx->xx_ev_attrlist);
        return(1);
    }

    err = ddi_log_sysevent(xx->xx_dip, DDI_VENDOR_SUNW,
        xx->xx_ev_class, xx->xx_ev_sbclass,
        xx->xx_ev_attrlist, NULL, DDI_NOSLEEP);
    nvlist_free(xx->xx_ev_attrlist);
    if (err == DDI_SUCCESS || err == DDI_ETRANSPORT) {
        if (err == DDI_ETRANSPORT)
            cmn_err(CE_WARN, "cannot log system event\n");
        return (0);
    }
    return (1);
}

```

**See Also** [syseventd\(1M\)](#), [attributes\(5\)](#), [nvlist\\_add\\_boolean\(9F\)](#), [nvlist\\_alloc\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_map\_regs, ddi\_unmap\_regs – map or unmap registers

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_map_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp,
                off_t offset, off_t len);
```

```
void ddi_unmap_regs(dev_info_t *dip, uint_t rnumber, caddr_t *kaddrp,
                   off_t offset, off_t len);
```

**Interface Level** These interfaces are obsolete. Use [ddi\\_regs\\_map\\_setup\(9F\)](#) instead of `ddi_map_regs()`. Use [ddi\\_regs\\_map\\_free\(9F\)](#) instead of `ddi_unmap_regs()`.

### Parameters

`ddi_map_regs()` *dip* Pointer to the device's `dev_info` structure.  
*rnumber* Register set number.  
*kaddrp* Pointer to the base kernel address of the mapped region (set on return).  
*offset* Offset into register space.  
*len* Length to be mapped.

`ddi_unmap_regs()` *dip* Pointer to the device's `dev_info` structure.  
*rnumber* Register set number.  
*kaddrp* Pointer to the base kernel address of the region to be unmapped.  
*offset* Offset into register space.  
*len* Length to be unmapped.

**Description** The `ddi_map_regs()` function maps in the register set given by *rnumber*. The register number determines which register set will be mapped if more than one exists. The base kernel virtual address of the mapped register set is returned in *kaddrp*. *offset* specifies an offset into the register space to start from and *len* indicates the size of the area to be mapped. If *len* is non-zero, it overrides the length given in the register set description. See the discussion of the `reg` property in [sbus\(4\)](#) and for more information on register set descriptions. If *len* and *offset* are 0, the entire space is mapped.

The `ddi_unmap_regs()` function undoes mappings set up by `ddi_map_regs()`. This is provided for drivers preparing to detach themselves from the system, allowing them to release allocated mappings. Mappings must be released in the same way they were mapped (a call to `ddi_unmap_regs()` must correspond to a previous call to `ddi_map_regs()`). Releasing portions of previous mappings is not allowed. *rnumber* determines which register set will be

unmapped if more than one exists. The *kaddrp*, *offset* and *len* specify the area to be unmapped. *kaddrp* is a pointer to the address returned from `ddi_map_regs()`; *offset* and *len* should match what `ddi_map_regs()` was called with.

**Return Values** The `ddi_map_regs()` function returns:

`DDI_SUCCESS` on success.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [attributes\(5\)](#), [sbus\(4\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_mem\_alloc, ddi\_mem\_free – allocate and free sequentially accessed memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_mem_alloc(dev_info_t *dip, ddi_dma_lim_t *limits,
                 uint_t length, uint_t flags, caddr_t *kaddrp,
                 uint_t *real_length);

void ddi_mem_free(caddr_t kaddr);
```

**Interface Level** These interfaces are obsolete. [ddi\\_dma\\_mem\\_alloc\(9F\)](#) and [ddi\\_dma\\_mem\\_free\(9F\)](#) should be used instead.

### Parameters

ddi_mem_alloc()	<i>dip</i>	A pointer to the device's <code>dev_info</code> structure.
	<i>limits</i>	A pointer to a DMA limits structure for this device (see <a href="#">ddi_dma_lim_sparc(9S)</a> or <a href="#">ddi_dma_lim_x86(9S)</a> ). If this pointer is NULL, a default set of DMA limits is assumed.
	<i>length</i>	The length in bytes of the desired allocation.
	<i>flags</i>	The possible flags 1 and 0 are taken to mean, respectively, wait until memory is available, or do not wait.
	<i>kaddrp</i>	On a successful return, <i>*kaddrp</i> points to the allocated memory.
	<i>real_length</i>	The length in bytes that was allocated. Alignment and padding requirements may cause <code>ddi_mem_alloc()</code> to allocate more memory than requested in <i>length</i> .
ddi_mem_free()	<i>kaddr</i>	The memory returned from a successful call to <code>ddi_mem_alloc()</code> .

**Description** The `ddi_mem_alloc()` function allocates memory for DMA transfers and should be used if the device is performing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as *streaming* access. The allocation will obey the alignment and padding constraints as specified by the *limits* argument and other limits imposed by the system.

Note that you must still use DMA resource allocation functions (see [ddi\\_dma\\_setup\(9F\)](#)) to establish DMA resources for the memory allocated using `ddi_mem_alloc()`.

`ddi_mem_alloc()` returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size.

[ddi\\_dma\\_setup\(9F\)](#) requires the actual length.

In order to make the view of a memory object shared between a CPU and a DMA device consistent, explicit synchronization steps using [ddi\\_dma\\_sync\(9F\)](#) or [ddi\\_dma\\_free\(9F\)](#) are required.

The `ddi_mem_free()` function frees up memory allocated by `ddi_mem_alloc()`.

**Return Values** The `ddi_mem_alloc()` function returns:

`DDI_SUCCESS` Memory successfully allocated.

`DDI_FAILURE` Allocation failed.

**Context** The `ddi_mem_alloc()` function can be called from user, interrupt, or kernel context, except when `flags` is set to 1, in which case it cannot be called from interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Obsolete

**See Also** [attributes\(5\)](#), [ddi\\_dma\\_free\(9F\)](#), [ddi\\_dma\\_mem\\_alloc\(9F\)](#), [ddi\\_dma\\_mem\\_free\(9F\)](#), [ddi\\_dma\\_setup\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [ddi\\_iopb\\_alloc\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_mem\_get8, ddi\_mem\_get16, ddi\_mem\_get32, ddi\_mem\_get64, ddi\_mem\_getb, ddi\_mem\_getl, ddi\_mem\_getll, ddi\_mem\_getw – read data from mapped device in the memory space or allocated DMA memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
uint8_t ddi_mem_get8(ddi_acc_handle_t handle, uint8_t *dev_addr);
uint16_t ddi_mem_get16(ddi_acc_handle_t handle, uint16_t * dev_addr);
uint32_t ddi_mem_get32(ddi_acc_handle_t handle, uint32_t *dev_addr);
uint64_t ddi_mem_get64(ddi_acc_handle_t handle, uint64_t *dev_addr);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_mem\_get8(), ddi\_mem\_get16(), ddi\_mem\_get32(), ddi\_mem\_get64(), ddi\_mem\_getb(), ddi\_mem\_getl(), ddi\_mem\_getll(), and ddi\_mem\_getw() functions are obsolete. The [ddi\\_get8\(9F\)](#) function replaces ddi\_mem\_get8(). The [ddi\\_get16\(9F\)](#) function replaces ddi\_mem\_get16(). The [ddi\\_get32\(9F\)](#) function replaces ddi\_mem\_get32(). The [ddi\\_get64\(9F\)](#) function replaces ddi\_mem\_get64(). The [ddi\\_get8\(9F\)](#) function replaces ddi\_mem\_getb(). The [ddi\\_get32\(9F\)](#) function replaces ddi\_mem\_getl(). The [ddi\\_get64\(9F\)](#) function replaces ddi\_mem\_getll(). The [ddi\\_get16\(9F\)](#) function replaces ddi\_mem\_getw().

**Parameters** *handle* The data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

*dev\_addr* Base device address.

**Description** These routines generate a read of various sizes from memory space or allocated DMA memory. The ddi\_mem\_get8(), ddi\_mem\_get16(), ddi\_mem\_get32(), and ddi\_mem\_get64() functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, *dev\_addr*, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete



**See Also** [ddi\\_get8\(9F\)](#), [ddi\\_get16\(9F\)](#), [ddi\\_get32\(9F\)](#), [ddi\\_get64\(9F\)](#), [ddi\\_mem\\_put8\(9F\)](#), [ddi\\_mem\\_rep\\_get8\(9F\)](#), [ddi\\_mem\\_rep\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_mem\_put8, ddi\_mem\_put16, ddi\_mem\_put32, ddi\_mem\_put64, ddi\_mem\_putb, ddi\_mem\_putw, ddi\_mem\_putl, ddi\_mem\_putll – write data to mapped device in the memory space or allocated DMA memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_mem_put8(ddi_acc_handle_t handle, uint8_t *dev_addr,
                 uint8_t value);

void ddi_mem_put16(ddi_acc_handle_t handle, uint16_t *dev_addr,
                  uint16_t value);

void ddi_mem_put32(ddi_acc_handle_t handle, uint32_t *dev_addr,
                  uint32_t value);

void ddi_mem_put64(ddi_acc_handle_t handle, uint64_t *dev_addr,
                  uint64_t value);
```

**Parameters** *handle*      The data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

*dev\_addr*      Base device address.

*value*          The data to be written to the device.

**Interface Level** Solaris DDI specific (Solaris DDI). The `ddi_mem_put8()`, `ddi_mem_put16()`, `ddi_mem_put32()`, `ddi_mem_put64()`, `ddi_mem_putb()`, `ddi_mem_putl()`, `ddi_mem_putll()`, and `ddi_mem_putw()` functions are obsolete. The [ddi\\_put8\(9F\)](#) function replaces `ddi_mem_put8()`. The [ddi\\_put16\(9F\)](#) function replaces `ddi_mem_put16()`. The [ddi\\_put32\(9F\)](#) function replaces `ddi_mem_put32()`. The [ddi\\_put64\(9F\)](#) function replaces `ddi_mem_put64()`. The [ddi\\_put8\(9F\)](#) function replaces `ddi_mem_putb()`. The [ddi\\_put32\(9F\)](#) function replaces `ddi_mem_putl()`. The [ddi\\_put64\(9F\)](#) function replaces `ddi_mem_putll()`. The [ddi\\_put16\(9F\)](#) function replaces `ddi_mem_putw()`.

**Description** These routines generate a write of various sizes to memory space or allocated DMA memory. The `ddi_mem_put8()`, `ddi_mem_put16()`, `ddi_mem_put32()`, and `ddi_mem_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, *dev\_addr*, in memory space.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [ddi\\_mem\\_get8\(9F\)](#), [ddi\\_mem\\_rep\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_put16\(9F\)](#), [ddi\\_put32\(9F\)](#), [ddi\\_put64\(9F\)](#), [ddi\\_mem\\_get8\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_mem\_rep\_get8, ddi\_mem\_rep\_get16, ddi\_mem\_rep\_get32, ddi\_mem\_rep\_get64, ddi\_mem\_rep\_getb, ddi\_mem\_rep\_getl, ddi\_mem\_rep\_getll, ddi\_mem\_rep\_getw – read multiple data from mapped device in the memory space or allocated DMA memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_mem_rep_get8(ddi_acc_handle_t handle,
    uint8_t *host_addr, uint8_t *dev_addr, size_t recount,
    uint_t flags);

void ddi_mem_rep_get16(ddi_acc_handle_t handle,
    uint16_t *host_addr, uint16_t *dev_addr, size_t recount,
    uint_t flags);

void ddi_mem_rep_get32(ddi_acc_handle_t handle,
    uint32_t *host_addr, uint32_t *dev_addr, size_t recount,
    uint_t flags);

void ddi_mem_rep_get64(ddi_acc_handle_t handle,
    uint64_t *host_addr, uint64_t *dev_addr, size_t recount,
    uint_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_mem\_rep\_get8(), ddi\_mem\_rep\_get16(), ddi\_mem\_rep\_get32(), ddi\_mem\_rep\_get64(), ddi\_mem\_rep\_getb(), ddi\_mem\_rep\_getl(), ddi\_mem\_rep\_getll(), and ddi\_mem\_rep\_getw() functions are obsolete. The [ddi\\_rep\\_get8\(9F\)](#) function replaces ddi\_mem\_rep\_get8(). The [ddi\\_rep\\_get16\(9F\)](#) function replaces ddi\_mem\_rep\_get16(). The [ddi\\_rep\\_get32\(9F\)](#) function replaces ddi\_mem\_rep\_get32(). The [ddi\\_rep\\_get64\(9F\)](#) function replaces ddi\_mem\_rep\_get64(). The [ddi\\_rep\\_get8\(9F\)](#) function replaces ddi\_mem\_rep\_getb(). The [ddi\\_rep\\_get32\(9F\)](#) function replaces ddi\_mem\_rep\_getl(). The [ddi\\_rep\\_get64\(9F\)](#) function replaces ddi\_mem\_rep\_getll(). The [ddi\\_rep\\_get16\(9F\)](#) function replaces ddi\_mem\_rep\_getw().

**Parameters**

<i>handle</i>	The data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>host_addr</i>	Base host address.
<i>dev_addr</i>	Base device address.
<i>recount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags:
DDI_DEV_AUTOINCR	Automatically increment the device address, <i>dev_addr</i> , during data accesses.
DDI_DEV_NO_AUTOINCR	Do not advance the device address, <i>dev_addr</i> , during data accesses.

**Description** These routines generate multiple reads from memory space or allocated DMA memory. *repcount* data is copied from the device address, *dev\_addr*, in memory space to the host address, *host\_addr*. For each input datum, the `ddi_mem_rep_get8()`, `ddi_mem_rep_get16()`, `ddi_mem_rep_get32()`, and `ddi_mem_rep_get64()` functions read 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, from the device address, *dev\_addr*. *dev\_addr* and *host\_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, *dev\_addr*, as a memory buffer location on the device and increments its address on the next input datum. However, when the *flags* argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** `ddi_mem_get8(9F)`, `ddi_mem_put8(9F)`, `ddi_mem_rep_put8(9F)`, `ddi_regs_map_setup(9F)`, `ddi_rep_get8(9F)`, `ddi_rep_get16(9F)`, `ddi_rep_get32(9F)`, `ddi_rep_get64(9F)`, `ddi_device_acc_attr(9S)`

**Name** ddi\_mem\_rep\_put8, ddi\_mem\_rep\_put16, ddi\_mem\_rep\_put32, ddi\_mem\_rep\_put64, ddi\_mem\_rep\_putb, ddi\_mem\_rep\_putl, ddi\_mem\_rep\_putll, ddi\_mem\_rep\_putw – write multiple data to mapped device in the memory space or allocated DMA memory

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_mem_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
    uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
    uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
    uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_mem_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
    uint64_t *dev_addr, size_t repcount, uint_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_mem\_rep\_put8(), ddi\_mem\_rep\_put16(), ddi\_mem\_rep\_put32(), ddi\_mem\_rep\_put64(), ddi\_mem\_rep\_putb(), ddi\_mem\_rep\_putl(), ddi\_mem\_rep\_putll(), and ddi\_mem\_rep\_putw() functions are obsolete. The [ddi\\_rep\\_put8\(9F\)](#) function replaces ddi\_mem\_rep\_put8(). The [ddi\\_rep\\_put16\(9F\)](#) function replaces ddi\_mem\_rep\_put16(). The [ddi\\_rep\\_put32\(9F\)](#) function replaces ddi\_mem\_rep\_put32(). The [ddi\\_rep\\_put64\(9F\)](#) function replaces ddi\_mem\_rep\_put64(). The [ddi\\_rep\\_put8\(9F\)](#) function replaces ddi\_mem\_rep\_putb(). The [ddi\\_rep\\_put32\(9F\)](#) function replaces ddi\_mem\_rep\_putl(). The [ddi\\_rep\\_put64\(9F\)](#) function replaces ddi\_mem\_rep\_putll(). The [ddi\\_rep\\_put16\(9F\)](#) function replaces ddi\_mem\_rep\_putw().

**Parameters**

<i>handle</i>	The data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>host_addr</i>	Base host address.
<i>dev_addr</i>	Base device address.
<i>repcount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags:
DDI_DEV_AUTOINCR	Automatically increment the device address, <i>dev_addr</i> , during data accesses.
DDI_DEV_NO_AUTOINCR	Do not advance the device address, <i>dev_addr</i> , during data accesses.

**Description** These routines generate multiple writes to memory space or allocated DMA memory. *repcount* data is copied from the host address, *host\_addr*, to the device address, *dev\_addr*, in memory space. For each input datum, the ddi\_mem\_rep\_put8(), ddi\_mem\_rep\_put16(),

`ddi_mem_rep_put32()`, and `ddi_mem_rep_put64()` functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address. `dev_addr` and `host_addr` must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

When the `flags` argument is set to `DDI_DEV_AUTOINCR`, these functions will treat the device address, `dev_addr`, as a memory buffer location on the device and increments its address on the next input datum. However, when the `flags` argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing from a data register.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete

**See Also** [ddi\\_mem\\_get8\(9F\)](#), [ddi\\_mem\\_put8\(9F\)](#), [ddi\\_mem\\_rep\\_get8\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#), [ddi\\_rep\\_put16\(9F\)](#), [ddi\\_rep\\_put32\(9F\)](#), [ddi\\_rep\\_put64\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_mmap\_get\_model – return data model type of current thread

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
uint_t ddi_mmap_get_model(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** ddi\_mmap\_get\_model() returns the C Language Type Model which the current thread expects. ddi\_mmap\_get\_model() is used in combination with ddi\_model\_convert\_from(9F) in the mmap(9E) driver entry point to determine whether there is a data model mismatch between the current thread and the device driver. The device driver might have to adjust the shape of data structures before exporting them to a user thread which supports a different data model.

**Return Values**

DDI_MODEL_ILP32	Current thread expects 32-bit ( <i>ILP32</i> ) semantics.
DDI_MODEL_LP64	Current thread expects 64-bit ( <i>LP64</i> ) semantics.
DDI_FAILURE	The ddi_mmap_get_model() function was not called from the mmap(9E) entry point.

**Context** The ddi\_mmap\_get\_model() function can only be called from the mmap(9E) driver entry point.

**Examples** EXAMPLE 1 : Using ddi\_mmap\_get\_model()

The following is an example of the mmap(9E) entry point and how to support 32-bit and 64-bit applications with the same device driver.

```
struct data32 {
    int len;
    caddr32_t addr;
};

struct data {
    int len;
    caddr_t addr;
};

xxmmap(dev_t dev, off_t off, int prot) {
    struct data dtc; /* a local copy for clash resolution */
    struct data *dp = (struct data *)shared_area;

    switch (ddi_model_convert_from(ddi_mmap_get_model())) {
    case DDI_MODEL_ILP32:
    {
        struct data32 *da32p;

        da32p = (struct data32 *)shared_area;
        dp = &dtc;
```



EXAMPLE 1 : Using `ddi_mmap_get_model()` (Continued)

```
        dp->len = da32p->len;
        dp->address = da32->address;
        break;
    }
    case DDI_MODEL_NONE:
        break;
    }
    /* continues along using dp */
    ...
}
```

**See Also** [mmap\(9E\)](#), [ddi\\_model\\_convert\\_from\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_model\_convert\_from – determine data model type mismatch

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
uint_t ddi_model_convert_from(uint_t model);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *model* The data model type of the current thread.

**Description** ddi\_model\_convert\_from() is used to determine if the current thread uses a different C Language Type Model than the device driver. The 64-bit version of Solaris will require a 64-bit kernel to support both 64-bit and 32-bit user mode programs. The difference between a 32-bit program and a 64-bit program is in its C Language Type Model: a 32-bit program is ILP32 (integer, longs, and pointers are 32-bit) and a 64-bit program is LP64 (longs and pointers are 64-bit). There are a number of driver entry points such as [ioctl\(9E\)](#) and [mmap\(9E\)](#) where it is necessary to identify the C Language Type Model of the user-mode originator of a kernel event. For example any data which flows between programs and the device driver or vice versa need to be identical in format. A 64-bit device driver may need to modify the format of the data before sending it to a 32-bit application. ddi\_model\_convert\_from() is used to determine if data that is passed between the device driver and the application requires reformatting to any non-native data model.

**Return Values** DDI\_MODEL\_ILP32 A conversion to/from ILP32 is necessary.  
DDI\_MODEL\_NONE No conversion is necessary. Current thread and driver use the same data model.

**Context** ddi\_model\_convert\_from() can be called from any context.

**Examples** **EXAMPLE 1** : Using ddi\_model\_convert\_from() in the ioctl() entry point to support both 32-bit and 64-bit applications.

The following is an example how to use ddi\_model\_convert\_from() in the ioctl() entry point to support both 32-bit and 64-bit applications.

```
struct passargs32 {
    int len;
    caddr32_t addr;
};

struct passargs {
    int len;
    caddr_t addr;
};

xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *credp, int *rvalp) {
    struct passargs pa;
```

**EXAMPLE 1** : Using `ddi_model_convert_from()` in the `ioctl()` entry point to support both 32-bit and 64-bit applications. *(Continued)*

```

switch (ddi_model_convert_from(mode & FMODELS)) {
    case DDI_MODEL_ILP32:
    {
        struct passargs32 pa32;

        ddi_copyin(arg, &pa32, sizeof (struct passargs32), mode);
        pa.len = pa32.len;
        pa.address = pa32.address;
        break;
    }
    case DDI_MODEL_NONE:
        ddi_copyin(arg, &pa, sizeof (struct passargs), mode);
        break;
}

do_ioctl(&pa);
. . . .
}

```

**See Also** [ioctl\(9E\)](#), [mmap\(9E\)](#), [ddi\\_mmap\\_get\\_model\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_modopen, ddi\_modsym, ddi\_modclose – dynamically-loaded kernel module functions

**Synopsis** #include <sys/modctl.h>

```
ddi_modhandle_t ddi_modopen(const char*modname, int mode,
                           int *errnop);

void *ddi_modsym(ddi_modhandle_t handle, const char *symname,
                int *errnop);

int ddi_modclose(ddi_modhandle_t handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *modname* The name of the dynamically-loaded kernel module (file) to be opened. The *modname* string is of the form:

"[*namespace*/[*dirspace*/]]*modulename*"

Each "*namespace*" directory along the standard kernel moddir/*module-path* path ([system\(4\)](#)) is searched to locate the module. If "*namespace*" is not specified, "misc" is assumed. If "*dirspace*" is specified, then "*namespace*" must be explicitly provided.

*mode* Currently, KRTLD\_MODE\_FIRST.

*errnop* Pointer to errno returned on error, if NULL then no additional error information is available.

*handle* Opaque handle returned from ddi\_modopen(), invalidated by ddi\_modclose().

*symname* Symbol's name as a character string.

**Description** The function prototypes for ddi\_modopen(), ddi\_modsym(), and ddi\_modclose() are modeled after the userland [libdl\(3LIB\)](#), [dlopen\(3C\)](#), [dlsym\(3C\)](#), and [dlclose\(3C\)](#) interfaces, however not all user features are available and the kernel symbol resolution is different. The [dlerror\(3C\)](#) interface is not appropriate for the kernel environment, so the new *errnop* return argument was added for ddi\_modopen() and ddi\_modsym().

The ddi\_modopen() function makes a dynamically-loaded kernel module named by "modname" available to a running kernel. ddi\_modopen() returns a handle that the caller can use on subsequent calls to ddi\_modsym() and ddi\_modclose(). The value of this handle should not be interpreted in any way by the caller.

The ddi\_modopen() interface works best as a dynamic component/object plug-in mechanism when targeting kernel "misc" modules that contain a single "struct modlmisc" module linkage, however non-"misc" modules and modules with multiple linkage structures can also be targeted.

There are two different symbol resolution search orders associated with the `ddi_modopen()` function: one search order to resolve symbols during the load of the targeted module, another search order to resolve `ddi_modsym()` calls against the handle returned by `ddi_modopen()`. To resolve symbols during module load, the standard kernel module load search order is used; to resolve symbols during module “A” load, the order is as follows:

```
A -> A's dependencies -> unix -> unix's dependencies
```

A single-level, left-to-right search in “ld -N” (or the `_depends_on` alternative that is retained for backward compatibility) modules occurs. For UNIX on Sparc, `_depends_on` is similar to “genunix misc/platmod cpu/SUNW,UltraSPARC-III+ dtracestubs” for Intel, it is “genunix dtracestubs”. The `ddi_modsym()` search is limited to the module directly associated with the handle.

The `ddi_modopen()` function increments the reference count on the named kernel module. Upon the first load of a module, the `_init(9E)` initialization code in the module is called; `ddi_modopen()` does not return until `_init` completes.

The `ddi_modsym()` function allows a caller to obtain the address of a symbol that is defined within a module. The *handle* argument is a valid `ddi_modhandle_t` as returned by `ddi_modopen()`, the *symname* argument is the symbol's name as a character string. The special handle values supported by `ddi_modsym(3C)` are not supported.

The `ddi_modclose()` function decrements the reference count of the kernel module associated with the specified handle. After the `ddi_modclose()` function is called, all `ddi_modsym()` resolutions obtained (either directly or indirectly) using the now closed *handle* are invalid; further use of these resolutions can cause undefined behavior (that is, may lead to a panic). When the last `ddi_modclose()` of a module occurs, and there are no further references to the module, the module `_fini(9E)` entry point may be called. If `_fini` returns success then the module may be unloaded.

**Return Values** The `ddi_modopen()` function returns a handle to the dynamically-loaded kernel module. The `ddi_modopen()` function returns NULL if the module cannot be found, the object cannot be relocated, or an error occurs during the process of resolving and relocating its symbolic references.

The `ddi_modsym()` function returns NULL if the *symname* symbol cannot be found directly within the module associated with the *handle*.

If the *handle* was not referenced, `ddi_modclose()` returns 0. If the *handle* is invalid, `ddi_modclose()` may return a non-zero value.

When either `ddi_modopen()` or `ddi_modsym()` return NULL, additional *errno* information related to the failure is returned in *\*errnop* if it is not NULL.

**Context** `ddi_modopen()` can be called from user context only.

**Examples** EXAMPLE 1 Coding a Dynamically Loaded Kernel Module

The following example shows code to dynamically load and call a “test” interface in a module called “dltest”. The “test” interface then adds one to its integer argument.

```

ddi_modhandle_t modh;
int             (*test)(int);
int             i = 0;
int             errno;
---%<---
/* dynamically load "dltest" kernel 'misc' module */
modh = ddi_modopen("dltest", KRTLD_MODE_FIRST, &errno);
if (modh == NULL)
    goto fail;      /* failed to open dltest module */

test = (int (*)())ddi_modsym(modh, "test", &errno);
if (test == NULL) {
    (void) ddi_modclose(modh);
    goto fail;      /* failed to find "test" interface */
}

/* invoke test interface and verify result */
i = (*test)(0);
ASSERT(i == 1);

(void) ddi_modclose(modh);
---%<---

```

The implementation of the “dltest” “misc” module is as follows:

```

#include <sys/modctl.h>
static dltest_add = 0;

/* define the module linkage */
static struct modlmisc      modlmisc = {&mod_miscops, "dltest"};
static struct modlinkage    modlinkage = {
    MODREV_1, (void *)&modmisc, NULL
};
int
_init(void)
{
    int    i;

    dltest_add = 1;          /* initialization */
    if ((i = mod_install(&modlinkage)) != 0)
        dltest_add = -1;    /* un-initialization */
    return (i);
}

```

**EXAMPLE 1** Coding a Dynamically Loaded Kernel Module *(Continued)*

```

}
int
_fini()
{
    int    i;

    if ((i = mod_remove(&modlinkage)) == 0)
        dltest_add = -1;      /* un-initialization */
    return (i);
}
int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

/* "test" interface */
int
test(int i)
{
    return (i + dltest_add);
}

```

**EXAMPLE 2** Dynamically Accessing a Kernel Module within a Drive

The following example shows driver code to dynamically load into the kernel a module constructed via the `elfwrap(1)` utility and containing firmware intended for download to a device. The “start” and “end” pointers provide the addresses of the beginning of the data and first byte beyond the data.

```

ddi_modhandle_t modp;
char *data_startp, *data_endp;
size_t nbytes;
int rv;

modp = ddi_modopen("firmware-rev1.2a", KRTLD_MODE_FIRST, &rv);
data_startp = (char *)ddi_modsym(modp, "fw-rev1.2a_start", &rv);
data_endp = (char *)ddi_modsym(modp, "fw-rev1.2a_end", &rv);
nbytes = data_endp - data_startp;
rv = ddi_modclose(modp);

```

**See Also** `dlclose(3C)`, `dlopen(3C)`, `dlsym(3C)`, `libdl(3LIB)`, `boot(1M)`, `elfwrap(1)`, `modload(1M)`, `system(4)`, `_fini(9E)`, `_info(9E)`, `_init(9E)`

*Writing Device Drivers*

**Warnings** A [system\(4\)](#)forceload must be established for modules targeted by `ddi_modopen()` by code involved in the mount of root on “bootdev” during machine [boot\(1M\)](#).



- Name** ddi\_node\_name – return the devinfo node name
- Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

char *ddi_node_name(dev_info_t *dip);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *dip* A pointer the device's dev\_info structure.
- Description** The ddi\_node\_name() function returns the device node name contained in the dev\_info node pointed to by *dip*.
- Return Values** The ddi\_node\_name() function returns the device node name contained in the dev\_info structure.
- Context** The ddi\_node\_name() function can be called from user, interrupt, or kernel context.
- See Also** [ddi\\_binding\\_name\(9F\)](#)  
*Writing Device Drivers*

**Name** ddi\_no\_info – stub for getinfo(9E)

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_no_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,  
              void **result)
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<code>dev_info_t *dip</code>	Pointer to dev_info structure.
<code>ddi_info_cmd_t infocmd</code>	Command argument. Valid command values are: DDI_INFO_DEVT2DEVINFO and DDI_INFO_DEVT2INSTANCE.
<code>void *arg</code>	Command-specific argument.
<code>void **result</code>	Pointer to where the requested information is stored.

**Description** The ddi\_no\_info() function always returns DDI\_FAILURE. It is provided as a convenience routine for drivers not providing a [cb\\_ops\(9S\)](#) or for network drivers only providing DLPI-2 services. Such drivers can use ddi\_no\_info() in the devo\_getinfo entry point (see [getinfo\(9E\)](#)) of the [dev\\_ops\(9S\)](#) structure.

**Return Values** The ddi\_no\_info() function always returns DDI\_FAILURE.

**See Also** [getinfo\(9E\)](#), [qassociate\(9F\)](#), [cb\\_ops\(9S\)](#), [dev\\_ops\(9S\)](#)

**Name** ddi\_peek, ddi\_peek8, ddi\_peek16, ddi\_peek32, ddi\_peek64, ddi\_peekc, ddi\_peekcs, ddi\_peekl, ddi\_peekd – read a value from a location

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_peek8(dev_info_t *dip, int8_t *addr, int8_t *valuep);
int ddi_peek16(dev_info_t *dip, int16_t *addr, int16_t *valuep);
int ddi_peek32(dev_info_t *dip, int32_t *addr, int32_t *valuep);
int ddi_peek64(dev_info_t *dip, int64_t *addr, int64_t *valuep);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_peekc(), ddi\_peekcs(), ddi\_peekl(), and ddi\_peekd() functions are obsolete. Use, respectively, ddi\_peek8(), ddi\_peek16(), ddi\_peek32(), and ddi\_peek64(), instead.

**Parameters**

*dip* A pointer to the device's dev\_info structure.

*addr* Virtual address of the location to be examined.

*valuep* Pointer to a location to hold the result. If a null pointer is specified, then the value read from the location will simply be discarded.

**Description** These routines cautiously attempt to read a value from a specified virtual address, and return the value to the caller, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be read without an error occurring, an error code is returned.

The routines are most useful when first trying to establish the presence of a device on the system in a driver's [probe\(9E\)](#) or [attach\(9E\)](#) routines.

**Return Values**

DDI\_SUCCESS The value at the given virtual address was successfully read, and if *valuep* is non-null, *valuep* will have been updated.

DDI\_FAILURE An error occurred while trying to read the location. *valuep* is unchanged.

**Context** These functions can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Checking to see that the status register of a device is mapped into the kernel address space:

```
if (ddi_peek8(dip, csr, (int8_t *)0) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "Status register not mapped");
    return (DDI_FAILURE);
}
```

EXAMPLE 2 Reading and logging the device type of a particular device:

```
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
```

**EXAMPLE 2** Reading and logging the device type of a particular device: *(Continued)*

```

{
    ...
    /* map device registers */
    ...

    if (ddi_peek32(dip, id_addr, &id_value) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "%s%d: cannot read device identifier",
            ddi_get_name(dip), ddi_get_instance(dip));
        goto failure;
    } else
        cmn_err(CE_CONT, "!%s%d: device type 0x%x\n",
            ddi_get_name(dip), ddi_get_instance(dip), id_value);
    ...
    ...

    ddi_report_dev(dip);
    return (DDI_SUCCESS);

failure:
    /* free any resources allocated */
    ...
    return (DDI_FAILURE);
}

```

**See Also** [attach\(9E\)](#), [probe\(9E\)](#), [ddi\\_poke\(9F\)](#)

### *Writing Device Drivers*

**Notes** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

Previous Name	New Name
ddi_peekc	ddi_peek8
ddi_peeks	ddi_peek16
ddi_peekl	ddi_peek32
ddi_peekd	ddi_peek64

**Name** ddi\_periodic\_add – issue nanosecond periodic timeout requests

**Synopsis** #include <sys/dditypes.h>  
#include <sys/sunddi.h>

```
ddi_periodic_t ddi_periodic_add(void (*func)(void *), void arg,
                               hrtime_t interval, int level);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *func* The callback function is invoked periodically in the specified interval. If the argument level is zero, the function is invoked in kernel context. Otherwise, it's invoked in interrupt context at the specified level.

*arg* The argument passed to the callback function.

*interval* Interval time in nanoseconds.

*level* Callback interrupt level. If the value is zero, the callback function is invoked in kernel context. If the value is more than zero, but less than or equal to ten, the callback function is invoked in interrupt context at the specified interrupt level, which may be used for real time applications.

This value must be in range of 0-10, which can be either a numeric number, a pre-defined macro (DDI\_IPL\_0, ... , DDI\_IPL\_10), or the DDI\_INTR\_PRI macro with the interrupt priority.

**Description** The ddi\_periodic\_add() function schedules the specified function to be periodically invoked in the nanosecond interval time.

As with [timeout\(9F\)](#), the exact time interval over which the function takes effect cannot be guaranteed, but the value given is a close approximation.

**Return Values** ddi\_periodic\_add() returns the non-zero opaque value (ddi\_periodic\_t), which might be used for [ddi\\_periodic\\_delete\(9F\)](#) to specify the request.

**Context** The ddi\_periodic\_add() function may be called from user or kernel context.

**Examples** EXAMPLE 1 Using ddi\_periodic\_add() for a periodic callback function

In the following example, the device driver registers a periodic callback function invoked in kernel context.

```
static void
my_periodic_func(void *arg)
{
    /*
     * This handler is invoked periodically.
     */
    struct my_state *statep = (struct my_state *)arg;
```

EXAMPLE 1 Using `ddi_periodic_add()` for a periodic callback function *(Continued)*

```

        mutex_enter(&statep->lock);
        if (load_unbalanced(statep)) {
            balance_tasks(statep);
        }
        mutex_exit(&statep->lock);
    }

static void
start_periodic_timer(struct my_state *statep)
{
    hrttime_t interval = CHECK_INTERVAL;

    mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
              (void *)DDI_IPL_0);

    /*
     * Register my_callback which is invoked periodically
     * in CHECK_INTERVAL in kernel context.
     */
    statep->periodic_id = ddi_periodic_add(my_periodic_func,
                                          statep, interval, DDI_IPL_0);
}

```

In the following example, the device driver registers a callback function invoked in interrupt context at level 7.

```

/*
 * This handler is invoked periodically in interrupt context.
 */
static void
my_periodic_int7_func(void *arg)
{
    struct my_state *statep = (struct my_state *)arg;
    mutex_enter(&statep->lock);
    monitor_device(statep);
    mutex_exit(&statep->lock);
}

static void
start_monitor_device(struct my_state *statep)
{
    hrttime_t interval = MONITOR_INTERVAL;

    mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
              (void *)DDI_IPL_7);
}

```

```
    /*  
     * Register the callback function invoked periodically  
     * at interrupt level 7.  
     */  
    statep->periodic_id = ddi_periodic_add(my_periodic_int7_func,  
        statep, interval, DDI_IPL_7);  
}
```

**See Also** [cv\\_timedwait\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_periodic\\_delete\(9F\)](#),  
[ddi\\_intr\\_get\\_softint\\_pri\(9F\)](#), [delay\(9F\)](#), [drv\\_usectohz\(9F\)](#), [qtimeout\(9F\)](#),  
[quntimeout\(9F\)](#), [timeout\(9F\)](#), [untimeout\(9F\)](#)

**Notes** A caller can only specify an interval in an integral multiple of 10ms. No other values are supported at this time. The interval specified is a lower bound on the interval on which the callback occurs.

**Name** ddi\_periodic\_delete – cancel nanosecond periodic timeout requests

**Synopsis** #include <sys/dditypes.h>  
#include <sys/sunddi.h>

```
void ddi_periodic_delete(ddi_periodic_t req);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *req* ddi\_periodic\_t opaque value returned by [ddi\\_periodic\\_add\(9F\)](#)

**Description** The `ddi_periodic_delete()` function cancels the [ddi\\_periodic\\_add\(9F\)](#) request that was previously issued.

As with [untimeout\(9F\)](#), calling `ddi_periodic_delete()` against a periodic *timeout* request which is either running on another CPU, or has already been canceled causes no problems. Unlike [untimeout\(9F\)](#), there are no restrictions on the lock which might be held across the call to `ddi_periodic_delete()`.

**Context** The `ddi_periodic_delete()` function may be called from user or kernel context.

**Examples** EXAMPLE 1 Cancelling a timeout request

In the following example, the device driver cancels the *timeout* request by calling `ddi_periodic_delete()` against the request that was previously issued.

```
/*
 * Stop the periodic timer
 */
static void
stop_periodic_timer(struct my_state *statep)
{
    ddi_periodic_delete(statep->periodic_id);
    delay(1); /* wait for one tick */
    mutex_destory(&statep->lock);
}

static void
start_periodic_timer(struct my_state *statep)
{
    hrtime_t interval = CHECK_INTERVAL;

    mutex_init(&statep->lock, NULL, MUTEX_DRIVER,
              (void *)DDI_IPL_0);

    /*
     * Register my_callback which is invoked periodically
     * in CHECK_INTERVAL in kernel context.
     */
}
```



**EXAMPLE 1** Cancelling a timeout request *(Continued)*

```

        statep->periodic_id = ddi_periodic_add(my_periodic_func,
        statep, interval, DDI_IPL_0);
    }

    static void
    my_periodic_func(void *arg)
    {
        /*
         * This handler is invoked periodically.
         */
        struct my_state *statep = (struct my_state *)arg;

        mutex_enter(&statep->lock);
        if (load_unbalanced(statep)) {
            balance_tasks(statep);
        }
        mutex_exit(&statep->lock);
    }

```

**See Also** [cv\\_timedwait\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_periodic\\_add\(9F\)](#), [delay\(9F\)](#), [drv\\_usecstohz\(9F\)](#), [qtimeout\(9F\)](#), [quntimeout\(9F\)](#), [timeout\(9F\)](#), [untimeout\(9F\)](#)

**Notes** There might be a race between a callback invocation and `ddi_periodic_delete()`. A device driver should take a responsibility for this avoidance if needed by using the kernel synchronization such as a mutex lock or calling [delay\(9F\)](#) as in the example above.

**Name** ddi\_poke, ddi\_poke8, ddi\_poke16, ddi\_poke32, ddi\_poke64, ddi\_pokec, ddi\_pokes, ddi\_pokel, ddi\_poked – write a value to a location

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_poke8(dev_info_t *dip, int8_t *addr, int8_t value);
int ddi_poke16(dev_info_t *dip, int16_t *addr, int16_t value);
int ddi_poke32(dev_info_t *dip, int32_t *addr, int32_t value);
int ddi_poke64(dev_info_t *dip, int64_t *addr, int64_t value);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The `ddi_pokec()`, `ddi_pokes()`, `ddi_pokel()`, and `ddi_poked()` functions are obsolete. Use, respectively, `ddi_poke8()`, `ddi_poke16()`, `ddi_poke32()`, and `ddi_poke64()`, instead.

**Parameters**

*dip*      A pointer to the device's `dev_info` structure.

*addr*      Virtual address of the location to be written to.

*value*     Value to be written to the location.

**Description** These routines cautiously attempt to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary.

If the address is not valid, or the value cannot be written without an error occurring, an error code is returned.

These routines are most useful when first trying to establish the presence of a given device on the system in a driver's [probe\(9E\)](#) or [attach\(9E\)](#) routines.

On multiprocessing machines these routines can be extremely heavy-weight, so use the [ddi\\_peek\(9F\)](#) routines instead if possible.

**Return Values**

DDI\_SUCCESS      The value was successfully written to the given virtual address.

DDI\_FAILURE      An error occurred while trying to write to the location.

**Context** These functions can be called from user, interrupt, or kernel context.

**See Also** [attach\(9E\)](#), [probe\(9E\)](#), [ddi\\_peek\(9F\)](#)

*Writing Device Drivers*

**Notes** The functions described in this manual page previously used symbolic names which specified their data access size; the function names have been changed so they now specify a fixed-width data size. See the following table for the new name equivalents:

Previous Name	New Name
ddi_pokec	ddi_poke8
ddi_pokes	ddi_poke16
ddi_pokel	ddi_poke32
ddi_poked	ddi_poke64

**Name** ddi\_prop\_create, ddi\_prop\_modify, ddi\_prop\_remove, ddi\_prop\_remove\_all, ddi\_prop\_undefine – create, remove, or modify properties for leaf device drivers

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_create(dev_t dev, dev_info_t *dip, int flags,
                  char *name, caddr_t valuep, int length);

int ddi_prop_undefine(dev_t dev, dev_info_t *dip, int flags,
                    char *name);

int ddi_prop_modify(dev_t dev, dev_info_t *dip, int flags,
                  char *name, caddr_t valuep, int length);

int ddi_prop_remove(dev_t dev, dev_info_t *dip, char *name);

void ddi_prop_remove_all(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_prop\_create() and ddi\_prop\_modify() functions are obsolete. Use [ddi\\_prop\\_update\(9F\)](#) instead of these functions.

**Parameters** ddi\_prop\_create()

*dev* dev\_t of the device.

*dip* dev\_info\_t pointer of the device.

*flags* flag modifiers. The only possible flag value is DDI\_PROP\_CANSLEEP: Memory allocation may sleep.

*name* name of property.

*valuep* pointer to property value.

*length* property length.

ddi\_prop\_undefine()

*dev* dev\_t of the device.

*dip* dev\_info\_t pointer of the device.

*flags* flag modifiers. The only possible flag value is DDI\_PROP\_CANSLEEP: Memory allocation may sleep.

*name* name of property.

ddi\_prop\_modify()

*dev* dev\_t of the device.

*dip* dev\_info\_t pointer of the device.

*flags* flag modifiers. The only possible flag value is DDI\_PROP\_CANSLEEP: Memory allocation may sleep.

*name* name of property.

*valuep* pointer to property value.

*length* property length.

`ddi_prop_remove()`

*dev* `dev_t` of the device.

*dip* `dev_info_t` pointer of the device.

*name* name of property.

`ddi_prop_remove_all()`

*dip* `dev_info_t` pointer of the device.

**Description** Device drivers have the ability to create and manage their own properties as well as gain access to properties that the system creates on behalf of the driver. A driver uses [ddi\\_getprop\(9F\)](#) to query whether or not a specific property exists.

Property creation is done by creating a new property definition in the driver's property list associated with *dip*.

Property definitions are stacked; they are added to the beginning of the driver's property list when created. Thus, when searched for, the most recent matching property definition will be found and its value will be return to the caller.

The individual functions are described as follows:

`ddi_prop_create()` `ddi_prop_create()` adds a property to the device's property list. If the property is not associated with any particular *dev* but is associated with the physical device itself, then the argument *dev* should be the special device DDI\_DEV\_T\_NONE. If you do not have a *dev* for your device (for example during [attach\(9E\)](#) time), you can create one using [makedevice\(9F\)](#) with a major number of DDI\_MAJOR\_T\_UNKNOWN. `ddi_prop_create()` will then make the correct *dev* for your device.

For boolean properties, you must set *length* to 0. For all other properties, the *length* argument must be set to the number of bytes used by the data structure representing the property being created.

Note that creating a property involves allocating memory for the property list, the property name and the property value. If *flags* does not contain `DDI_PROP_CANSLEEP`, `ddi_prop_create()` returns `DDI_PROP_NO_MEMORY` on memory allocation failure or `DDI_PROP_SUCCESS` if the allocation succeeded. If `DDI_PROP_CANSLEEP` was set, the caller may sleep until memory becomes available.

`ddi_prop_undefine()` `ddi_prop_undefine()` is a special case of property creation where the value of the property is set to undefined. This property has the effect of terminating a property search at the current devinfo node, rather than allowing the search to proceed up to ancestor devinfo nodes. However, `ddi_prop_undefine()` will not terminate a search when the [ddi\\_prop\\_get\\_int\(9F\)](#) or [ddi\\_prop\\_lookup\(9F\)](#) routines are used for lookup of 64-bit property value. See [ddi\\_prop\\_op\(9F\)](#).

Note that undefining properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.

`ddi_prop_modify()` `ddi_prop_modify()` modifies the length and the value of a property. If `ddi_prop_modify()` finds the property in the driver's property list, allocates memory for the property value and returns `DDI_PROP_SUCCESS`. If the property was not found, the function returns `DDI_PROP_NOT_FOUND`.

Note that modifying properties does involve memory allocation, and therefore, is subject to the same memory allocation constraints as `ddi_prop_create()`.

`ddi_prop_remove()` `ddi_prop_remove()` unlinks a property from the device's property list. If `ddi_prop_remove()` finds the property (an exact match of both *name* and *dev*), it unlinks the property, frees its memory, and returns `DDI_PROP_SUCCESS`, otherwise, it returns `DDI_PROP_NOT_FOUND`.

`ddi_prop_remove_all()` `ddi_prop_remove_all()` removes the properties of all the *dev\_t*'s associated with the *dip*. It is called before unloading a driver.

**Return Values** The `ddi_prop_create()` function returns the following values:

<code>DDI_PROP_SUCCESS</code>	On success.
<code>DDI_PROP_NO_MEMORY</code>	On memory allocation failure.

DDI\_PROP\_INVAL\_ARG If an attempt is made to create a property with *dev* equal to DDI\_DEV\_T\_ANY or if *name* is NULL or *name* is the NULL string.

The `ddi_prop_undefine()` function returns the following values:

DDI\_PROP\_SUCCESS On success.

DDI\_PROP\_NO\_MEMORY On memory allocation failure.

DDI\_PROP\_INVAL\_ARG If an attempt is made to create a property with *dev* DDI\_DEV\_T\_ANY or if *name* is NULL or *name* is the NULL string.

The `ddi_prop_modify()` function returns the following values:

DDI\_PROP\_SUCCESS On success.

DDI\_PROP\_NO\_MEMORY On memory allocation failure.

DDI\_PROP\_INVAL\_ARG If an attempt is made to create a property with *dev* equal to DDI\_DEV\_T\_ANY or if *name* is NULL or *name* is the NULL string.

DDI\_PROP\_NOT\_FOUND On property search failure.

The `ddi_prop_remove()` function returns the following values:

DDI\_PROP\_SUCCESS On success.

DDI\_PROP\_INVAL\_ARG If an attempt is made to create a property with *dev* equal to DDI\_DEV\_T\_ANY or if *name* is NULL or *name* is the NULL string.

DDI\_PROP\_NOT\_FOUND On property search failure.

**Context** If DDI\_PROP\_CANSLEEP is set, these functions cannot be called from interrupt context. Otherwise, they can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Creating a Property

The following example creates a property called *nblocks* for each partition on a disk.

```
int propval = 8192;

for (minor = 0; minor < 8; minor++) {
    (void) ddi_prop_create(makedevice(DDI_MAJOR_T_UNKNOWN, minor),
        dev, DDI_PROP_CANSLEEP, "nblocks", (caddr_t) &propval,
        sizeof (int));
    ...
}
```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	ddi_prop_create() and ddi_prop_modify() are Obsolete

**See Also** [driver.conf\(4\)](#), [attributes\(5\)](#), [attach\(9E\)](#), [ddi\\_getpropflen\(9F\)](#), [ddi\\_prop\\_op\(9F\)](#), [ddi\\_prop\\_update\(9F\)](#), [makedevice\(9F\)](#)

*Writing Device Drivers*



**Name** ddi\_prop\_exists – check for the existence of a property

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_exists(dev_t match_dev, dev_info_t *dip, uint_t flags,
    char *name);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.				
<i>dip</i>	Pointer to the device info node of device whose property list should be searched.				
<i>flags</i>	Possible flag values are some combination of: <table> <tr> <td>DDI_PROP_DONTPASS</td> <td>Do not pass request to parent device information node if the property is not found.</td> </tr> <tr> <td>DDI_PROP_NOTPROM</td> <td>Do not look at PROM properties (ignored on platforms that do not support PROM properties).</td> </tr> </table>	DDI_PROP_DONTPASS	Do not pass request to parent device information node if the property is not found.	DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).
DDI_PROP_DONTPASS	Do not pass request to parent device information node if the property is not found.				
DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).				
<i>name</i>	String containing the name of the property.				

**Description** ddi\_prop\_exists() checks for the existence of a property regardless of the property value data type.

Properties are searched for based on the *dip*, *name*, and *match\_dev*. The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI\_PROP\_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI\_PROP\_DONTPASS is not set, pass this request to the parent device information node.
6. Return 0 if not found and 1 if found.

Usually, the *match\_dev* argument should be set to the actual device number that this property is associated with. However, if the *match\_dev* argument is DDI\_DEV\_T\_ANY, then ddi\_prop\_exists() will match the request regardless of the *match\_dev* the property was created with. That is the first property whose name matches *name* will be returned. If a property was created with *match\_dev* set to DDI\_DEV\_T\_NONE then the only way to look up this property is with a *match\_dev* set to DDI\_DEV\_T\_ANY. PROM properties are always created with *match\_dev* set to DDI\_DEV\_T\_NONE.

*name* must always be set to the name of the property being looked up.

**Return Values** `ddi_prop_exists()` returns 1 if the property exists and 0 otherwise.

**Context** These functions can be called from user or kernel context.

**Examples** `EXAMPLE 1` : Using `ddi_prop_exists()`

The following example demonstrates the use of `ddi_prop_exists()`.

```
/*
 * Enable "whizzy" mode if the "whizzy-mode" property exists
 */
if (ddi_prop_exists(xx_dev, xx_dip, DDI_PROP_NOTPROM,
    "whizzy-mode") == 1) {
    xx_enable_whizzy_mode(xx_dip);
} else {
    xx_disable_whizzy_mode(xx_dip);
}
```

**See Also** `ddi_prop_get_int(9F)`, `ddi_prop_lookup(9F)`, `ddi_prop_remove(9F)`,  
`ddi_prop_update(9F)`

*Writing Device Drivers*

**Name** ddi\_prop\_get\_int, ddi\_prop\_get\_int64 – lookup integer property

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_get_int(dev_t match_dev, dev_info_t *dip, uint_t flags,
    char *name, int defvalue);

int64_t ddi_prop_get_int64(dev_t match_dev, dev_info_t *dip, uint_t flags,
    char *name, int64_t defvalue);
```

**Parameters**

<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.				
<i>dip</i>	Pointer to the device info node of device whose property list should be searched.				
<i>flags</i>	Possible flag values are some combination of: <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 10px;">DDI_PROP_DONTPASS</td> <td>Do not pass request to parent device information node if property not found.</td> </tr> <tr> <td style="padding-right: 10px;">DDI_PROP_NOTPROM</td> <td>Do not look at PROM properties (ignored on platforms that do not support PROM properties).</td> </tr> </table>	DDI_PROP_DONTPASS	Do not pass request to parent device information node if property not found.	DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).
DDI_PROP_DONTPASS	Do not pass request to parent device information node if property not found.				
DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).				
<i>name</i>	String containing the name of the property.				
<i>defvalue</i>	An integer value that is returned if the property cannot be found.				

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The `ddi_prop_get_int()` and `ddi_prop_get_int64()` functions search for an integer property and, if found, returns the value of the property.

Properties are searched for based on the *dip*, *name*, *match\_dev*, and the type of the data (integer).

The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If `DDI_PROP_NOTPROM` is not set, search the PROM properties (if they exist).
5. If `DDI_PROP_DONTPASS` is not set, pass this request to the parent device information node.
6. Return *defvalue*.

Usually, the *match\_dev* argument should be set to the actual device number that this property is associated with. However, if the *match\_dev* argument is `DDI_DEV_T_ANY`, then `ddi_prop_get_int()` and `ddi_prop_get_int64()` will match the request regardless of the

*match\_dev* the property was created with. If a property was created with *match\_dev* set to `DDI_DEV_T_NONE`, then the only way to look up this property is with a *match\_dev* set to `DDI_DEV_T_ANY`. PROM properties are always created with *match\_dev* set to `DDI_DEV_T_NONE`.

*name* must always be set to the name of the property being looked up.

The return value of the routine is the value of the property. If the property is not found, the argument *defvalue* is returned as the value of the property.

`ddi_prop_get_int64()` will not search the PROM for 64-bit property values.

**Return Values** `ddi_prop_get_int()` and `ddi_prop_get_int64()` return the value of the property. If the property is not found, the argument *defvalue* is returned. If the property is found, but cannot be decoded into an `int` or an `int64`, then `DDI_PROP_NOT_FOUND` is returned.

**Context** `ddi_prop_get_int()` and `ddi_prop_get_int64()` can be called from user or kernel context.

**Examples** EXAMPLE 1 Using `ddi_prop_get_int()`

The following example demonstrates the use of `ddi_prop_get_int()`.

```
/*
 * Get the value of the integer "width" property, using
 * our own default if no such property exists
 */
width = ddi_prop_get_int(xx_dev, xx_dip, 0, "width",
                        XX_DEFAULT_WIDTH);
```

**See Also** [ddi\\_prop\\_exists\(9F\)](#), [ddi\\_prop\\_lookup\(9F\)](#), [ddi\\_prop\\_remove\(9F\)](#), [ddi\\_prop\\_update\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_prop\_lookup, ddi\_prop\_lookup\_int\_array, ddi\_prop\_lookup\_int64\_array, ddi\_prop\_lookup\_string\_array, ddi\_prop\_lookup\_string, ddi\_prop\_lookup\_byte\_array, ddi\_prop\_free – look up property information

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_lookup_int_array(dev_t match_dev, dev_info_t *dip,
    uint_t flags, char *name, int **datap, uint_t *nelementsp);

int ddi_prop_lookup_int64_array(dev_t match_dev, dev_info_t *dip,
    uint_t flags, char *name, int64_t **datap, uint_t *nelementsp);

int ddi_prop_lookup_string_array(dev_t match_dev, dev_info_t *dip,
    uint_t flags, char *name, char ***datap, uint_t *nelementsp);

int ddi_prop_lookup_string(dev_t match_dev, dev_info_t *dip, uint_t flags,
    char *name, char **datap);

int ddi_prop_lookup_byte_array(dev_t match_dev, dev_info_t *dip,
    uint_t flags, char *name, uchar_t **datap, uint_t *nelementsp);

void ddi_prop_free(void *data);
```

**Parameters**

<i>match_dev</i>	Device number associated with property or DDI_DEV_T_ANY.				
<i>dip</i>	Pointer to the device info node of device whose property list should be searched.				
<i>flags</i>	Possible flag values are some combination of: <table> <tr> <td>DDI_PROP_DONTPASS</td> <td>Do not pass request to parent device information node if the property is not found.</td> </tr> <tr> <td>DDI_PROP_NOTPROM</td> <td>Do not look at PROM properties (ignored on platforms that do not support PROM properties).</td> </tr> </table>	DDI_PROP_DONTPASS	Do not pass request to parent device information node if the property is not found.	DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).
DDI_PROP_DONTPASS	Do not pass request to parent device information node if the property is not found.				
DDI_PROP_NOTPROM	Do not look at PROM properties (ignored on platforms that do not support PROM properties).				
<i>name</i>	String containing the name of the property.				
<i>nelementsp</i>	The address of an unsigned integer which, upon successful return, will contain the number of elements accounted for in the memory pointed at by <i>datap</i> . The elements are either integers, strings or bytes depending on the interface used.				
<i>datap</i>	<table> <tr> <td>ddi_prop_lookup_int_array()</td> <td>The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.</td> </tr> <tr> <td>ddi_prop_lookup_int64_array()</td> <td>The address of a pointer to an array of 64-bit integers which, upon</td> </tr> </table>	ddi_prop_lookup_int_array()	The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.	ddi_prop_lookup_int64_array()	The address of a pointer to an array of 64-bit integers which, upon
ddi_prop_lookup_int_array()	The address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array property value.				
ddi_prop_lookup_int64_array()	The address of a pointer to an array of 64-bit integers which, upon				

	successful return, will point to memory containing the integer array property value.
<code>ddi_prop_lookup_string_array()</code>	The address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the <i>argv</i> argument to <code>execve(2)</code> .
<code>ddi_prop_lookup_string()</code>	The address of a pointer to a string which, upon successful return, will point to memory containing the NULL terminated string value of the property.
<code>ddi_prop_lookup_byte_array()</code>	The address of pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The property look up routines search for and, if found, return the value of a given property. Properties are searched for based on the *dip*, *name*, *match\_dev*, and the type of the data (integer, string, or byte). The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If `DDI_PROP_NOTPROM` is not set, search the PROM properties (if they exist).
5. If `DDI_PROP_DONTPASS` is not set, pass this request to the parent device information node.
6. Return `DDI_PROP_NOT_FOUND`.

Usually, the *match\_dev* argument should be set to the actual device number that this property is associated with. However, if the *match\_dev* argument is `DDI_DEV_T_ANY`, the property look up routines will match the request regardless of the actual *match\_dev* the property was created with. If a property was created with *match\_dev* set to `DDI_DEV_T_NONE`, then the only way to look up this property is with a *match\_dev* set to `DDI_DEV_T_ANY`. PROM properties are always created with *match\_dev* set to `DDI_DEV_T_NONE`.

*name* must always be set to the name of the property being looked up.

For the routines `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()`, *datap* is the address of a pointer which, upon successful return, will point to memory containing the value of the property. In each case *\*datap* points to a different type of property value. See the individual descriptions of the routines below for details on the different return values. *nelementsp* is the address of an unsigned integer which, upon successful return, will contain the number of integer, string or byte elements accounted for in the memory pointed at by *\*datap*.

All of the property look up routines may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up routine and is finished with that property, it must be freed by calling `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if one called `ddi_prop_lookup_int_array()` with *datap* set to the address of a pointer to an integer, *&my\_int\_ptr*, then the companion free call would be `ddi_prop_free(my_int_ptr)`.

<code>ddi_prop_lookup_int_array()</code>	This routine searches for and returns an array of integer property values. An array of integers is defined to <i>*nelementsp</i> number of 4 byte long integer elements. <i>datap</i> should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.
<code>ddi_prop_lookup_int64_array()</code>	This routine searches for and returns an array of 64-bit integer property values. The array is defined to be <i>*nelementsp</i> number of <code>int64_t</code> elements. <i>datap</i> should be set to the address of a pointer to an array of <code>int64_t</code> 's which, upon successful return, will point to memory containing the integer array value of the property. This routine will not search the PROM for 64-bit property values.
<code>ddi_prop_lookup_string_array()</code>	This routine searches for and returns a property that is an array of strings. <i>datap</i> should be set to address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the <i>argv</i> argument to <code>execve(2)</code> .

<code>ddi_prop_lookup_string()</code>	This routine searches for and returns a property that is a null-terminated string. <i>datap</i> should be set to the address of a pointer to string which, upon successful return, will point to memory containing the string value of the property.
<code>ddi_prop_lookup_byte_array()</code>	This routine searches for and returns a property that is an array of bytes. <i>datap</i> should be set to the address of a pointer to an array of bytes which, upon successful return, will point to memory containing the byte array value of the property.
<code>ddi_prop_free()</code>	Frees the resources associated with a property previously allocated using <code>ddi_prop_lookup_int_array()</code> , <code>ddi_prop_lookup_int64_array()</code> , <code>ddi_prop_lookup_string_array()</code> , <code>ddi_prop_lookup_string()</code> , or <code>ddi_prop_lookup_byte_array()</code> .

**Return Values** The functions `ddi_prop_lookup_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_lookup_string_array()`, `ddi_prop_lookup_string()`, and `ddi_prop_lookup_byte_array()` return the following values:

<code>DDI_PROP_SUCCESS</code>	Upon success.
<code>DDI_PROP_INVALID_ARG</code>	If an attempt is made to look up a property with <i>match_dev</i> equal to <code>DDI_DEV_T_NONE</code> , <i>name</i> is NULL or <i>name</i> is the null string.
<code>DDI_PROP_NOT_FOUND</code>	Property not found.
<code>DDI_PROP_UNDEFINED</code>	Property explicitly not defined (see <a href="#">ddi_prop_undefine(9F)</a> ).
<code>DDI_PROP_CANNOT_DECODE</code>	The value of the property cannot be decoded.

**Context** These functions can be called from user or kernel context.

**Examples** `EXAMPLE1` Using `ddi_prop_lookup_int_array()`

The following example demonstrates the use of `ddi_prop_lookup_int_array()`.

```
int    *options;
int    noptions;

/*
 * Get the data associated with the integer "options" property
 * array, along with the number of option integers
```



---

**EXAMPLE 1** Using `ddi_prop_lookup_int_array()` (Continued)

```
*/
if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, xx_dip, 0,
    "options", &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
     * Do "our thing" with the options data from the property
     */
    xx_process_options(options, noptions);

    /*
     * Free the memory allocated for the property data
     */
    ddi_prop_free(options);
}
```

**See Also** [execve\(2\)](#), [ddi\\_prop\\_exists\(9F\)](#), [ddi\\_prop\\_get\\_int\(9F\)](#), [ddi\\_prop\\_remove\(9F\)](#),  
[ddi\\_prop\\_undefine\(9F\)](#), [ddi\\_prop\\_update\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_prop\_op, ddi\_getprop, ddi\_getlongprop, ddi\_getlongprop\_buf, ddi\_getproplen – get property information for leaf device drivers

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
               int flags, char *name, caddr_t valuep, int *lengthp);
```

```
int ddi_getprop(dev_t dev, dev_info_t *dip, int flags, char *name,
               int defvalue);
```

```
int ddi_getlongprop(dev_t dev, dev_info_t *dip, int flags, char *name,
                   caddr_t valuep, int *lengthp);
```

```
int ddi_getlongprop_buf(dev_t dev, dev_info_t *dip, int flags, char *name,
                       caddr_t valuep, int *lengthp);
```

```
int ddi_getproplen(dev_t dev, dev_info_t *dip, int flags, char *name,
                  int *lengthp);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_getlongprop(), ddi\_getlongprop\_buf(), ddi\_getprop(), and ddi\_getproplen() functions are obsolete. Use [ddi\\_prop\\_lookup\(9F\)](#) instead of ddi\_getlongprop(), ddi\_getlongprop\_buf(), and ddi\_getproplen(). Use [ddi\\_prop\\_get\\_int\(9F\)](#) instead of ddi\_getprop()

**Parameters**

<i>dev</i>	Device number associated with property or DDI_DEV_T_ANY as the <i>wildcard</i> device number.
<i>dip</i>	Pointer to a device info node.
<i>prop_op</i>	Property operator.
<i>flags</i>	Possible flag values are some combination of: <ul style="list-style-type: none"> <li>DDI_PROP_DONTPASS      do not pass request to parent device information node if property not found</li> <li>DDI_PROP_CANSLEEP      the routine may sleep while allocating memory</li> <li>DDI_PROP_NOTPROM      do not look at PROM properties (ignored on architectures that do not support PROM properties)</li> </ul>
<i>name</i>	String containing the name of the property.
<i>valuep</i>	If <i>prop_op</i> is PROP_LEN_AND_VAL_BUF, this should be a pointer to the users buffer. If <i>prop_op</i> is PROP_LEN_AND_VAL_ALLOC, this should be the <i>address</i> of a pointer.
<i>lengthp</i>	On exit, <i>lengthp</i> will contain the property length. If <i>prop_op</i> is PROP_LEN_AND_VAL_BUF then before calling ddi_prop_op(), <i>lengthp</i> should point to an int that contains the length of callers buffer.

*defvalue* The value that `ddi_getprop()` returns if the property is not found.

**Description** The `ddi_prop_op()` function gets arbitrary-size properties for leaf devices. The routine searches the device's property list. If it does not find the property at the device level, it examines the *flags* argument, and if `DDI_PROP_DONTPASS` is set, then `ddi_prop_op()` returns `DDI_PROP_NOT_FOUND`. Otherwise, it passes the request to the next level of the device info tree. If it does find the property, but the property has been explicitly undefined, it returns `DDI_PROP_UNDEFINED`. Otherwise it returns either the property length, or both the length and value of the property to the caller via the *valuep* and *lengthp* pointers, depending on the value of *prop\_op*, as described below, and returns `DDI_PROP_SUCCESS`. If a property cannot be found at all, `DDI_PROP_NOT_FOUND` is returned.

Usually, the *dev* argument should be set to the actual device number that this property applies to. However, if the *dev* argument is `DDI_DEV_T_ANY`, the *wildcard dev*, then `ddi_prop_op()` will match the request based on *name* only (regardless of the actual *dev* the property was created with). This property/dev match is done according to the property search order which is to first search software properties created by the driver in *last-in, first-out* (LIFO) order, next search software properties created by the *system* in LIFO order, then search PROM properties if they exist in the system architecture.

Property operations are specified by the *prop\_op* argument. If *prop\_op* is `PROP_LEN`, then `ddi_prop_op()` just sets the callers length, *\*lengthp*, to the property length and returns the value `DDI_PROP_SUCCESS` to the caller. The *valuep* argument is not used in this case. Property lengths are 0 for boolean properties, `sizeof (int)` for integer properties, and size in bytes for long (variable size) properties.

If *prop\_op* is `PROP_LEN_AND_VAL_BUF`, then *valuep* should be a pointer to a user-supplied buffer whose length should be given in *\*lengthp* by the caller. If the requested property exists, `ddi_prop_op()` first sets *\*lengthp* to the property length. It then examines the size of the buffer supplied by the caller, and if it is large enough, copies the property value into that buffer, and returns `DDI_PROP_SUCCESS`. If the named property exists but the buffer supplied is too small to hold it, it returns `DDI_PROP_BUF_TOO_SMALL`.

If *prop\_op* is `PROP_LEN_AND_VAL_ALLOC`, and the property is found, `ddi_prop_op()` sets *\*lengthp* to the property length. It then attempts to allocate a buffer to return to the caller using the `kmem_alloc(9F)` routine, so that memory can be later recycled using `kmem_free(9F)`. The driver is expected to call `kmem_free()` with the returned address and size when it is done using the allocated buffer. If the allocation is successful, it sets *\*valuep* to point to the allocated buffer, copies the property value into the buffer and returns `DDI_PROP_SUCCESS`. Otherwise, it returns `DDI_PROP_NO_MEMORY`. Note that the *flags* argument may affect the behavior of memory allocation in `ddi_prop_op()`. In particular, if `DDI_PROP_CANSLEEP` is set, then the routine will wait until memory is available to copy the requested property.

The `ddi_getprop()` function returns boolean and integer-size properties. It is a convenience wrapper for `ddi_prop_op()` with `prop_op` set to `PROP_LEN_AND_VAL_BUF`, and the buffer is provided by the wrapper. By convention, this function returns a 1 for boolean (zero-length) properties.

The `ddi_getlongprop()` function returns arbitrary-size properties. It is a convenience wrapper for `ddi_prop_op()` with `prop_op` set to `PROP_LEN_AND_VAL_ALLOC`, so that the routine will allocate space to hold the buffer that will be returned to the caller via `*valuep`.

The `ddi_getlongprop_buf()` function returns arbitrary-size properties. It is a convenience wrapper for `ddi_prop_op()` with `prop_op` set to `PROP_LEN_AND_VAL_BUF` so the user must supply a buffer.

The `ddi_getproplen()` function returns the length of a given property. It is a convenience wrapper for `ddi_prop_op()` with `prop_op` set to `PROP_LEN`.

**Return Values** The `ddi_prop_op()`, `ddi_getlongprop()`, `ddi_getlongprop_buf()`, and `ddi_getproplen()` functions return:

<code>DDI_PROP_SUCCESS</code>	Property found and returned.
<code>DDI_PROP_NOT_FOUND</code>	Property not found.
<code>DDI_PROP_UNDEFINED</code>	Property already explicitly undefined.
<code>DDI_PROP_NO_MEMORY</code>	Property found, but unable to allocate memory. <i>lengthp</i> points to the correct property length.
<code>DDI_PROP_BUF_TOO_SMALL</code>	Property found, but the supplied buffer is too small. <i>lengthp</i> points to the correct property length.

The `ddi_getprop()` function returns:

The value of the property or the value passed into the routine as *defvalue* if the property is not found. By convention, the value of zero length properties (boolean properties) are returned as the integer value 1.

**Context** These functions can be called from user, interrupt, or kernel context, provided `DDI_PROP_CANSLEEP` is not set; if it is set, they cannot be called from interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	<code>ddi_getlongprop()</code> , <code>ddi_getlongprop_buf()</code> , <code>ddi_getprop()</code> , and <code>ddi_getproplen()</code> functions are Obsolete

**See Also** `attributes(5)`, `ddi_prop_create(9F)`, `ddi_prop_get_int(9F)`, `ddi_prop_lookup(9F)`, `kmem_alloc(9F)`, `kmem_free(9F)`

*Writing Device Drivers*

**Name** ddi\_prop\_update, ddi\_prop\_update\_int\_array, ddi\_prop\_update\_int, ddi\_prop\_update\_string\_array, ddi\_prop\_update\_int64, ddi\_prop\_update\_int64\_array, ddi\_prop\_update\_string, ddi\_prop\_update\_byte\_array – update properties

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_prop_update_int_array(dev_t dev, dev_info_t *dip,
    char *name, int *data, uint_t nelements);

int ddi_prop_update_int(dev_t dev, dev_info_t *dip, char *name,
    int data);

int ddi_prop_update_int64_array(dev_t dev, dev_info_t *dip, char *name,
    int64_t *data, uint_t nelements);

int ddi_prop_update_int64(dev_t dev, dev_info_t *dip, char *name,
    int64_t data);

int ddi_prop_update_string_array(dev_t dev, dev_info_t *dip, char *name,
    char **data, uint_t nelements);

int ddi_prop_update_string(dev_t dev, dev_info_t *dip, char *name,
    char *data);

int ddi_prop_update_byte_array(dev_t dev, dev_info_t *dip, char *name,
    uchar_t *data, uint_t nelements);
```

**Parameters** *dev* Device number associated with the device.  
*dip* Pointer to the device info node of device whose property list should be updated.  
*name* String containing the name of the property to be updated.  
*nelements* The number of elements contained in the memory pointed at by *data*.

ddi\_prop\_update\_int\_array()

*data* A pointer an integer array with which to update the property.

ddi\_prop\_update\_int()

*data* An integer value with which to update the property.

ddi\_prop\_update\_int64\_array()

*data* An pointer to a 64-bit integer array with which to update the property.

ddi\_prop\_update\_int64()

*data* A 64-bit integer value with which to update the property.

ddi\_prop\_update\_string\_array()

*data* A pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULL terminated strings, much like the *argv* argument to `execve(2)`.

`ddi_prop_update_string()`

*data* A pointer to a string value with which to update the property.

`ddi_prop_update_byte_array()`

*data* A pointer to a byte array with which to update the property.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The property update routines search for and, if found, modify the value of a given property. Properties are searched for based on the *dip*, *name*, *dev*, and the type of the data (integer, string, or byte). The driver software properties list is searched. If the property is found, it is updated with the supplied value. If the property is not found on this list, a new property is created with the value supplied. For example, if a driver attempts to update the "foo" property, a property named "foo" is searched for on the driver's software property list. If "foo" is found, the value is updated. If "foo" is not found, a new property named "foo" is created on the driver's software property list with the supplied value even if a "foo" property exists on another property list (such as a PROM property list).

Every property value has a data type associated with it: byte, integer, or string. A property should be updated using a function with the same corresponding data type as the property value. For example, an integer property must be updated using either `ddi_prop_update_int_array()` or `ddi_prop_update_int()`. For a 64-bit integer, you must use `ddi_prop_update_int64_array()` or `ddi_prop_update_int64()`. Attempts to update a property with a function that does not correspond to the property data type that was used to create it results in an undefined state.

Usually, the *dev* argument should be set to the actual device number that this property is associated with. If the property is not associated with any particular *dev*, then the argument *dev* should be set to `DDI_DEV_T_NONE`. This property will then match a look up request (see [ddi\\_prop\\_lookup\(9F\)](#)) with the *match\_dev* argument set to `DDI_DEV_T_ANY`. If no *dev* is available for the device (for example during [attach\(9E\)](#) time), one can be created using [makedevice\(9F\)](#) with a major number of `DDI_MAJOR_T_UNKNOWN`. The update routines will then generate the correct *dev* when creating or updating the property.

*name* must always be set to the name of the property being updated.

For the routines `ddi_prop_update_int_array()`, `ddi_prop_lookup_int64_array()`, `ddi_prop_update_string_array()`, `ddi_prop_update_string()`, and `ddi_prop_update_byte_array()`, *data* is a pointer which points to memory containing the value of the property. In each case *\*data* points to a different type of property value. See the

individual descriptions of the routines below for details concerning the different values. *nelements* is an unsigned integer which contains the number of integer, string, or byte elements accounted for in the memory pointed at by *\*data*.

For the routines `ddi_prop_update_int()` and `ddi_prop_update_int64()`, *data* is the new value of the property.

`ddi_prop_update_int_array()`

Updates or creates an array of integer property values. An array of integers is defined to be *nelements* of 4 byte long integer elements. *data* must be a pointer to an integer array with which to update the property.

`ddi_prop_update_int()`

Update or creates a single integer value of a property. *data* must be an integer value with which to update the property.

`ddi_prop_update_int64_array()`

Updates or creates an array of 64-bit integer property values. An array of integers is defined to be *nelements* of `int64_t` integer elements. *data* must be a pointer to a 64-bit integer array with which to update the property.

`ddi_prop_update_int64()`

Updates or creates a single 64-bit integer value of a property. *data* must be an `int64_t` value with which to update the property.

`ddi_prop_update_string_array()`

Updates or creates a property that is an array of strings. *data* must be a pointer to a string array with which to update the property. The array of strings is formatted as an array of pointers to NULLterminated strings, much like the *argv* argument to `execve(2)`.

`ddi_prop_update_string()`

Updates or creates a property that is a single string value. *data* must be a pointer to a string with which to update the property.

`ddi_prop_update_byte_array()`

Updates or creates a property that is an array of bytes. *data* should be a pointer to a byte array with which to update the property.

The property update routines may block to allocate memory needed to hold the value of the property.



**Return Values** All of the property update routines return:

DDI_PROP_SUCCESS	On success.
DDI_PROP_INVALID_ARG	If an attempt is made to update a property with <i>name</i> set to NULL or <i>name</i> set to the null string.
DDI_PROP_CANNOT_ENCODE	If the bytes of the property cannot be encoded.

**Context** These functions can only be called from user or kernel context.

**Examples** EXAMPLE 1 Updating Properties

The following example demonstrates the use of `ddi_prop_update_int_array()`.

```
int    options[4];

/*
 * Create the "options" integer array with
 * our default values for these parameters
 */
options[0] = XX_OPTIONS0;
options[1] = XX_OPTIONS1;
options[2] = XX_OPTIONS2;
options[3] = XX_OPTIONS3;
i = ddi_prop_update_int_array(xx_dev, xx_dip, "options",
    &options, sizeof (options) / sizeof (int));
```

**See Also** [execve\(2\)](#), [attach\(9E\)](#), [ddi\\_prop\\_lookup\(9F\)](#), [ddi\\_prop\\_remove\(9F\)](#), [makedevice\(9F\)](#)

*Writing Device Drivers*

**Name** ddi\_put8, ddi\_put16, ddi\_put32, ddi\_put64, ddi\_putb, ddi\_putl, ddi\_putll, ddi\_putw – write data to the mapped memory address, device register or allocated DMA memory address

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_put8(ddi_acc_handle_t handle, uint8_t *dev_addr,
             uint8_t value);

void ddi_put16(ddi_acc_handle_t handle, uint16_t *dev_addr,
              uint16_t value);

void ddi_put32(ddi_acc_handle_t handle, uint32_t *dev_addr,
              uint32_t value);

void ddi_put64(ddi_acc_handle_t handle, uint64_t *dev_addr,
              uint64_t value);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_putb(), ddi\_putl(), ddi\_putll(), and ddi\_putw() functions are obsolete. The ddi\_put8() function replaces ddi\_putb(). The ddi\_put32() function replaces ddi\_putl(). The ddi\_put64() function replaces ddi\_putll(). The ddi\_put16() function replaces ddi\_putw().

**Parameters** *handle* The data access handle returned from setup calls, such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

*value* The data to be written to the device.

*dev\_addr* Base device address.

**Description** These routines generate a write of various sizes to the mapped memory or device register. The ddi\_put8(), ddi\_put16(), ddi\_put32(), and ddi\_put64() functions write 8 bits, 16 bits, 32 bits and 64 bits of data, respectively, to the device address, *dev\_addr*.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

For certain bus types, you can call these DDI functions from a high-interrupt context. These types include ISA and SBus buses. See [sysbus\(4\)](#), [isa\(4\)](#), and [sbus\(4\)](#) for details. For the PCI bus, you can, under certain conditions, call these DDI functions from a high-interrupt context. See [pci\(4\)](#).

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	ddi_putb(), ddi_putl(), ddi_putll(), and ddi_putw() are Obsolete  ddi_put8(), ddi_put16(), ddi_put32(), ddi_put64() are Committed

**See Also** [ddi\\_get8\(9F\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_regs\_map\_free – free a previously mapped register address space

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_regs_map_free(ddi_acc_handle_t *handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *handle* Pointer to a data access handle previously allocated by a call to a setup routine such as [ddi\\_regs\\_map\\_setup\(9F\)](#).

**Description** `ddi_regs_map_free()` frees the mapping represented by the data access handle *handle*. This function is provided for drivers preparing to detach themselves from the system, allowing them to release allocated system resources represented in the handle.

**Context** `ddi_regs_map_free()` must be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus, SBus, ISA

**See Also** [attributes\(5\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#)

*Writing Device Drivers*

**Name** `ddi_regs_map_setup` – set up a mapping for a register address space

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
int ddi_regs_map_setup(dev_info_t *dip, uint_t rnumber, caddr_t *addrp,
    offset_t offset, offset_t len, ddi_device_acc_attr_t *accattrp,
    ddi_acc_handle_t *handlep);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dip</i>	Pointer to the device's <code>dev_info</code> structure.
<i>rnumber</i>	Index number to the register address space set. A value of 0 indicates PCI configuration space, while a value of 1 indicates the real start of device register sets.
<i>addrp</i>	A platform-dependent value that, when added to an offset that is less than or equal to the <i>len</i> parameter (see below), is used for the <code>dev_addr</code> argument to the <code>ddi_get</code> , <code>ddi_mem_get</code> , and <code>ddi_io_get/put</code> routines.
<i>offset</i>	Offset into the register address space.
<i>len</i>	Length to be mapped.
<i>accattrp</i>	Pointer to a device access attribute structure of this mapping (see <a href="#">ddi_device_acc_attr(9S)</a> ).
<i>handlep</i>	Pointer to a data access handle.

**Description** `ddi_regs_map_setup()` maps in the register set given by *rnumber*. The register number determines which register set is mapped if more than one exists.

*offset* specifies the starting location within the register space and *len* indicates the size of the area to be mapped. If *len* is non-zero, it overrides the length given in the register set description. If both *len* and *offset* are 0, the entire space is mapped. The base of the mapped register space is returned in *addrp*.

The device access attributes are specified in the location pointed by the *accattrp* argument (see [ddi\\_device\\_acc\\_attr\(9S\)](#) for details).

The data access handle is returned in *handlep*. *handlep* is opaque; drivers should not attempt to interpret its value. The handle is used by the system to encode information for subsequent data access function calls to maintain a consistent view between the host and the device.

**Return Values** `ddi_regs_map_setup()` returns:

<code>DDI_SUCCESS</code>	Successfully set up the mapping for data access.
<code>DDI_FAILURE</code>	Invalid register number <i>rnumber</i> , offset <i>offset</i> , or length <i>len</i> .

- DDI\_ME\_RNUMBER\_RANGE** Invalid register number *rnumber* or unable to find *reg* property.
- DDI\_REGS\_ACC\_CONFLICT** Cannot enable the register mapping due to access conflicts with other enabled mappings.

Note that the return value **DDI\_ME\_RNUMBER\_RANGE** is not supported on all platforms. Also, there is potential overlap between **DDI\_ME\_RNUMBER\_RANGE** and **DDI\_FAILURE**. Drivers should check for **!=DDI\_SUCCESS** rather than checking for a specific failure value.

**Context** `ddi_regs_map_setup()` must be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI Local Bus, SBus, ISA

**See Also** [attributes\(5\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_remove\_event\_handler – remove an NDI event service callback handler

**Synopsis** #include <sys/dditypes.h>  
#include <sys/sunddi.h>

```
int ddi_remove_event_handler(ddi_registration_id_t id);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** ddi\_registration\_id\_t *id* Unique system wide registration ID return by [ddi\\_add\\_event\\_handler\(9F\)](#) upon successful registration.

**Description** The `ddi_remove_event_handler()` function removes the callback handler specified by the registration *id* (`ddi_registration_id_t`). Upon successful removal, the callback handler is removed from the system and will not be invoked in the face of the event.

**Return Values** DDI\_SUCCESS      Callback handler removed successfully.  
DDI\_FAILURE      Failed to remove callback handler.

**Context** The `ddi_remove_event_handler()` function can be called from user and kernel contexts only.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Committed

**See Also** [attributes\(5\)](#), [ddi\\_add\\_event\\_handler\(9F\)](#), [ddi\\_get\\_eventcookie\(9F\)](#)

*Writing Device Drivers*

**Notes** Device drivers must remove all registered callback handlers before [detach\(9E\)](#) processing for that device instance is complete.

**Name** ddi\_remove\_minor\_node – remove a minor node for this dev\_info

**Synopsis** void ddi\_remove\_minor\_node(dev\_info\_t \*dip, char \*name);

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* A pointer to the device's dev\_info structure.  
*name* The name of this minor device. If *name* is NULL, then remove all minor data structures from this dev\_info.

**Description** ddi\_remove\_minor\_node() removes a data structure from the linked list of minor data structures that is pointed to by the dev\_info structure for this driver.

**Examples** EXAMPLE 1 Removing a minor node

This will remove a data structure describing a minor device called dev1 which is linked into the dev\_info structure pointed to by dip:

```
ddi_remove_minor_node(dip, "dev1");
```

**See Also** [attach\(9E\)](#), [detach\(9E\)](#), [ddi\\_create\\_minor\\_node\(9F\)](#)

*Writing Device Drivers*



**Name** ddi\_removing\_power – check whether DDI\_SUSPEND might result in power being removed from a device

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_removing_power(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The ddi\_removing\_power() function indicates whether a currently pending call into a driver's detach(9E) entry point with a command of DDI\_SUSPEND is likely to result in power being removed from the device.

ddi\_removing\_power() can return true and power still not be removed from the device due to a failure to suspend and power off the system.

**Parameters** The ddi\_removing\_power() function supports the following parameter:

*dip* pointer to the device's dev\_info structure

**Return Values** The ddi\_removing\_power() function returns:

- 1 Power might be removed by the framework as a result of the pending DDI\_SUSPEND call.
- 0 Power will not be removed by the framework as a result of the pending DDI\_SUSPEND call.

**Examples** EXAMPLE 1 Protecting a Tape from Abrupt Power Removal

A tape driver that has hardware that would damage the tape if power is removed might include this code in its detach(9E) code:

```
int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    ...
    case DDI_SUSPEND:
        /*
         * We do not allow DDI_SUSPEND if power will be removed and
         * we have a device that damages tape when power is removed
         * We do support DDI_SUSPEND for Device Reconfiguration,
         * however.
         */
        if (ddi_removing_power(dip) && xxdamages_tape(dip))
            return (DDI_FAILURE);
    ...
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [cpr\(7\)](#), [attach\(9E\)](#), [detach\(9E\)](#)

*Writing Device Drivers*

**Name** ddi\_rep\_get8, ddi\_rep\_get16, ddi\_rep\_get32, ddi\_rep\_get64, ddi\_rep\_getw, ddi\_rep\_getl, ddi\_rep\_getll, ddi\_rep\_getb – read data from the mapped memory address, device register or allocated DMA memory address

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_rep_get8(ddi_acc_handle_t handle, uint8_t *host_addr,
                uint8_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get16(ddi_acc_handle_t handle, uint16_t *host_addr,
                 uint16_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get32(ddi_acc_handle_t handle, uint32_t *host_addr,
                 uint32_t *dev_addr, size_t repcount, uint_t flags);

void ddi_rep_get64(ddi_acc_handle_t handle, uint64_t *host_addr,
                 uint64_t *dev_addr, size_t repcount, uint_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_rep\_getb(), ddi\_rep\_getl(), ddi\_rep\_getll(), and ddi\_rep\_getw() functions are obsolete. The ddi\_rep\_get8() function replaces ddi\_rep\_getb(). The ddi\_rep\_get32() function replaces ddi\_rep\_getl(). The ddi\_rep\_get64() function replaces ddi\_rep\_getll(). The ddi\_rep\_get16() function replaces ddi\_rep\_getw().

**Parameters**

<i>handle</i>	The data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>host_addr</i>	Base host address.
<i>dev_addr</i>	Base device address.
<i>repcount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags:
DDI_DEV_AUTOINCR	Automatically increment the device address, <i>dev_addr</i> , during data accesses.
DDI_DEV_NO_AUTOINCR	Do not advance the device address, <i>dev_addr</i> , during data accesses.

**Description** These routines generate multiple reads from the mapped memory or device register. *repcount* data is copied from the device address, *dev\_addr*, to the host address, *host\_addr*. For each input datum, the ddi\_rep\_get8(), ddi\_rep\_get16(), ddi\_rep\_get32(), and ddi\_rep\_get64() functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, from the device address, *dev\_addr*. *dev\_addr* and *host\_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these function treat the device address, *dev\_addr*, as a memory buffer location on the device and increment its address on the next input datum. However, when the *flags* argument is to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when reading from a data register.

**Return Values** These functions return the value read from the mapped address.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	<p><code>ddi_rep_getb()</code>, <code>ddi_rep_getl()</code>, <code>ddi_rep_getll()</code>, and <code>ddi_rep_getw()</code> are Obsolete</p> <p><code>ddi_rep_get8()</code>, <code>ddi_rep_get16()</code>, <code>ddi_rep_get32()</code>, <code>ddi_rep_get64()</code> are Committed</p>

**See Also** [ddi\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#)

**Name** ddi\_report\_dev – announce a device

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
void ddi_report_dev(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* a pointer the device's dev\_info structure.

**Description** ddi\_report\_dev() prints a banner at boot time, announcing the device pointed to by *dip*. The banner is always placed in the system logfile (displayed by [dmesg\(1M\)](#)), but is only displayed on the console if the system was booted with the verbose (-v) argument.

**Context** ddi\_report\_dev() can be called from user context.

**See Also** [dmesg\(1M\)](#), [kernel\(1M\)](#)

*Writing Device Drivers*

**Name** ddi\_rep\_put8, ddi\_rep\_put16, ddi\_rep\_put32, ddi\_rep\_put64, ddi\_rep\_putb, ddi\_rep\_putw, ddi\_rep\_putl, ddi\_rep\_putll – write data to the mapped memory address, device register or allocated DMA memory address

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
void ddi_rep_put8(ddi_acc_handle_t handle, uint8_t *host_addr,
                uint8_t *dev_addr, size_t recount, uint_t flags);

void ddi_rep_put16(ddi_acc_handle_t handle, uint16_t *host_addr,
                 uint16_t *dev_addr, size_t recount, uint_t flags);

void ddi_rep_put32(ddi_acc_handle_t handle, uint32_t *host_addr,
                 uint32_t *dev_addr, size_t recount, uint_t flags);

void ddi_rep_put64(ddi_acc_handle_t handle, uint64_t *host_addr,
                 uint64_t *dev_addr, size_t recount, uint_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The ddi\_rep\_putb(), ddi\_rep\_putl(), ddi\_rep\_putll(), and ddi\_rep\_putw() functions are obsolete. The ddi\_rep\_put8() function replaces ddi\_rep\_putb(). The ddi\_rep\_put32() function replaces ddi\_rep\_putl(). The ddi\_rep\_put64() function replaces ddi\_rep\_putll(). The ddi\_rep\_put16() function replaces ddi\_rep\_putw().

**Parameters**

<i>handle</i>	The data access handle returned from setup calls, such as <a href="#">ddi_regs_map_setup(9F)</a> .
<i>host_addr</i>	Base host address.
<i>dev_addr</i>	Base device address.
<i>recount</i>	Number of data accesses to perform.
<i>flags</i>	Device address flags:
DDI_DEV_AUTOINCR	Automatically increment the device address, <i>dev_addr</i> , during data accesses.
DDI_DEV_NO_AUTOINCR	Do not advance the device address, <i>dev_addr</i> , during data accesses.

**Description** These routines generate multiple writes to the mapped memory or device register. *recount* data is copied from the host address, *host\_addr*, to the device address, *dev\_addr*. For each input datum, the ddi\_rep\_put8(), ddi\_rep\_put16(), ddi\_rep\_put32(), and ddi\_rep\_put64() functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively, to the device address, *dev\_addr*. *dev\_addr* and *host\_addr* must be aligned to the datum boundary described by the function.

Each individual datum will automatically be translated to maintain a consistent view between the host and the device based on the encoded information in the data access handle. The translation may involve byte-swapping if the host and the device have incompatible endian characteristics.

When the *flags* argument is set to `DDI_DEV_AUTOINCR`, these functions treat the device address, *dev\_addr*, as a memory buffer location on the device and increment its address on the next input datum. However, when the *flags* argument is set to `DDI_DEV_NO_AUTOINCR`, the same device address will be used for every datum access. For example, this flag may be useful when writing to a data register.

**Context** These functions can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	ddi_rep_putb(), ddi_rep_putl(), ddi_rep_putll(), and ddi_rep_putw() are Obsolete  ddi_rep_put8(), ddi_rep_put16(), ddi_rep_put32(), ddi_rep_put64() are Committed

**See Also** [ddi\\_get8\(9F\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_regs\\_map\\_free\(9F\)](#), [ddi\\_regs\\_map\\_setup\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

**Name** ddi\_root\_node – get the root of the dev\_info tree

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
dev_info_t *ddi_root_node(void);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The ddi\_root\_node() function returns a pointer to the root node of the device information tree.

**Return Values** The ddi\_root\_node() function returns a pointer to a device information structure.

**Context** The ddi\_root\_node() function can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)



**Name** ddi\_segmap, ddi\_segmap\_setup – set up a user mapping using seg\_dev

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int ddi_segmap(dev_t dev, off_t offset, struct as *asp,
               caddr_t *addrp, off_t len, uint_t prot,
               uint_t maxprot, uint_t flags, cred_t *credp);

int ddi_segmap_setup(dev_t dev, off_t offset, struct as *asp,
                    caddr_t *addrp, off_t len, uint_t prot,
                    uint_t maxprot, uint_t flags, cred_t *credp,
                    ddi_device_acc_attr_t *accattrp, uint_t rnumber);
```

**Interface Level** These interfaces are obsolete. See [devmap\(9E\)](#) for an alternative to `ddi_segmap()`. Use [devmap\\_setup\(9F\)](#) instead of `ddi_segmap_setup()`.

**Parameters**

<i>dev</i>	The device whose memory is to be mapped.										
<i>offset</i>	The offset within device memory at which the mapping begins.										
<i>asp</i>	An opaque pointer to the user address space into which the device memory should be mapped.										
<i>addrp</i>	Pointer to the starting address within the user address space to which the device memory should be mapped.										
<i>len</i>	Length (in bytes) of the memory to be mapped.										
<i>prot</i>	A bit field that specifies the protections. Some combinations of possible settings are: <table> <tr> <td>PROT_READ</td> <td>Read access is desired.</td> </tr> <tr> <td>PROT_WRITE</td> <td>Write access is desired.</td> </tr> <tr> <td>PROT_EXEC</td> <td>Execute access is desired.</td> </tr> <tr> <td>PROT_USER</td> <td>User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).</td> </tr> <tr> <td>PROT_ALL</td> <td>All access is desired.</td> </tr> </table>	PROT_READ	Read access is desired.	PROT_WRITE	Write access is desired.	PROT_EXEC	Execute access is desired.	PROT_USER	User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).	PROT_ALL	All access is desired.
PROT_READ	Read access is desired.										
PROT_WRITE	Write access is desired.										
PROT_EXEC	Execute access is desired.										
PROT_USER	User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).										
PROT_ALL	All access is desired.										
<i>maxprot</i>	Maximum protection flag possible for attempted mapping (the PROT_WRITE bit may be masked out if the user opened the special file read-only). If ( <code>maxprot &amp; prot</code> ) != <code>prot</code> then there is an access violation.										
<i>flags</i>	Flags indicating type of mapping. Possible values are (other bits may be set): <table> <tr> <td>MAP_PRIVATE</td> <td>Changes are private.</td> </tr> <tr> <td>MAP_SHARED</td> <td>Changes should be shared.</td> </tr> </table>	MAP_PRIVATE	Changes are private.	MAP_SHARED	Changes should be shared.						
MAP_PRIVATE	Changes are private.										
MAP_SHARED	Changes should be shared.										

---

	MAP_FIXED	The user specified an address in <i>*addrp</i> rather than letting the system pick an address.
	<i>credp</i>	Pointer to user credential structure.
ddi_segmap_setup()	<i>dev_acc_attr</i>	Pointer to a <a href="#">ddi_device_acc_attr(9S)</a> structure which contains the device access attributes to apply to this mapping.
	<i>rnumber</i>	Index number to the register address space set.

**Description** Future releases of Solaris will provide this function for binary and source compatibility. However, for increased functionality, use [ddi\\_devmap\\_segmap\(9F\)](#) instead. See [ddi\\_devmap\\_segmap\(9F\)](#) for details.

`ddi_segmap()` and `ddi_segmap_setup()` set up user mappings to device space. When setting up the mapping, the `ddi_segmap()` and `ddi_segmap_setup()` routines call the [mmap\(9E\)](#) entry point to validate the range to be mapped. When a user process accesses the mapping, the driver's [mmap\(9E\)](#) entry point is again called to retrieve the page frame number that needs to be loaded. The mapping translations for that page are then loaded on behalf of the driver by the DDI framework.

`ddi_segmap()` is typically used as the [segmap\(9E\)](#) entry in the [cb\\_ops\(9S\)](#) structure for those devices that do not choose to provide their own [segmap\(9E\)](#) entry point. However, some drivers may have their own [segmap\(9E\)](#) entry point to do some initial processing on the parameters and then call `ddi_segmap()` to establish the default memory mapping.

`ddi_segmap_setup()` is used in the driver's [segmap\(9E\)](#) entry point to set up the mapping and assign device access attributes to that mapping. *rnumber* specifies the register set representing the range of device memory being mapped. See [ddi\\_device\\_acc\\_attr\(9S\)](#) for details regarding what device access attributes are available.

`ddi_segmap_setup()` cannot be used directly in the [cb\\_ops\(9S\)](#) structure and requires a driver to have a [segmap\(9E\)](#) entry point.

**Return Values** `ddi_segmap()` and `ddi_segmap_setup()` return the following values:

0	Successful completion.
Non-zero	An error occurred. In particular, they return ENXIO if the range to be mapped is invalid.

**Context** `ddi_segmap()` and `ddi_segmap_setup()` can be called from user or kernel context only.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**See Also** [mmap\(2\)](#), [attributes\(5\)](#), [devmap\(9E\)](#), [mmap\(9E\)](#), [segmap\(9E\)](#), [devmap\\_setup\(9F\)](#), [cb\\_ops\(9S\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#)

*Writing Device Drivers*

**Name** ddi\_slaveonly – tell if a device is installed in a slave access only location

**Synopsis**

```
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_slaveonly(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* A pointer to the device's dev\_info structure.

**Description** The ddi\_slaveonly() function tells the caller if the bus, or part of the bus that the device is installed on, does not permit the device to become a DMA master, that is, whether the device has been installed in a slave access only slot.

**Return Values** DDI\_SUCCESS The device has been installed in a slave access only location.  
DDI\_FAILURE The device has not been installed in a slave access only location.

**Context** The ddi\_slaveonly() function can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)

**Name** ddi\_soft\_state, ddi\_get\_soft\_state, ddi\_soft\_state\_fini, ddi\_soft\_state\_free, ddi\_soft\_state\_init, ddi\_soft\_state\_zalloc – driver soft state utility routines

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

void *ddi_get_soft_state(void *state, int item);
void ddi_soft_state_fini(void **state_p);
void ddi_soft_state_free(void *state, int item);
int ddi_soft_state_init(void **state_p, size_t size, size_t n_items);
int ddi_soft_state_zalloc(void *state, int item);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>state_p</i>	Address of the opaque state pointer which will be initialized by <code>ddi_soft_state_init()</code> to point to implementation dependent data.
<i>size</i>	Size of the item which will be allocated by subsequent calls to <code>ddi_soft_state_zalloc()</code> .
<i>n_items</i>	A hint of the number of items which will be preallocated; zero is allowed.
<i>state</i>	An opaque pointer to implementation-dependent data that describes the soft state.
<i>item</i>	The item number for the state structure; usually the instance number of the associated devinfo node.

**Description** Most device drivers maintain state information with each instance of the device they control; for example, a soft copy of a device control register, a mutex that must be held while accessing a piece of hardware, a partition table, or a unit structure. These utility routines are intended to help device drivers manage the space used by the driver to hold such state information.

For example, if the driver holds the state of each instance in a single state structure, these routines can be used to dynamically allocate and deallocate a separate structure for each instance of the driver as the instance is attached and detached.

To use the routines, the driver writer needs to declare a state pointer, *state\_p*, which the implementation uses as a place to hang a set of per-driver structures; everything else is managed by these routines.

The routine `ddi_soft_state_init()` is usually called in the driver's [\\_init\(9E\)](#) routine to initialize the state pointer, set the size of the soft state structure, and to allow the driver to pre-allocate a given number of such structures if required.

The routine `ddi_soft_state_zalloc()` is usually called in the driver's [attach\(9E\)](#) routine. The routine is passed an item number which is used to refer to the structure in subsequent

calls to `ddi_get_soft_state()` and `ddi_soft_state_free()`. The item number is usually just the instance number of the `devinfo` node, obtained with `ddi_get_instance(9F)`. The routine attempts to allocate space for the new structure, and if the space allocation was successful, `DDI_SUCCESS` is returned to the caller. Returned memory is zeroed.

A pointer to the space previously allocated for a soft state structure can be obtained by calling `ddi_get_soft_state()` with the appropriate item number.

The space used by a given soft state structure can be returned to the system using `ddi_soft_state_free()`. This routine is usually called from the driver's `detach(9E)` entry point.

The space used by all the soft state structures allocated on a given state pointer, together with the housekeeping information used by the implementation can be returned to the system using `ddi_soft_state_fini()`. This routine can be called from the driver's `_fini(9E)` routine.

The `ddi_soft_state_zalloc()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines coordinate access to the underlying data structures in an MT-safe fashion, thus no additional locks should be necessary.

**Return Values** `ddi_get_soft_state()`

`NULL`        The requested state structure was not allocated at the time of the call.

*pointer*     The pointer to the state structure.

`ddi_soft_state_init()`

`0`            The allocation was successful.

`EINVAL`     Either the size parameter was zero, or the *state\_p* parameter was invalid.

`ddi_soft_state_zalloc()`

`DDI_SUCCESS`    The allocation was successful.

`DDI_FAILURE`    The routine failed to allocate the storage required; either the *state* parameter was invalid, the item number was negative, or an attempt was made to allocate an item number that was already allocated.

**Context**    The `ddi_soft_state_init()` and `ddi_soft_state_alloc()` functions can be called from user or kernel context only, since they may internally call `kmem_zalloc(9F)` with the `KM_SLEEP` flag.

The `ddi_soft_state_fini()`, `ddi_soft_state_free()` and `ddi_get_soft_state()` routines can be called from any driver context.

**Examples** EXAMPLE 1 Creating and Removing Data Structures

The following example shows how the routines described above can be used in terms of the driver entry points of a character-only driver. The example concentrates on the portions of the code that deal with creating and removing the driver's data structures.

```
typedef struct {
    volatile caddr_t *csr;      /* device registers */
    kmutex_t      csr_mutex;    /* protects 'csr' field */
    unsigned int  state;
    dev_info_t    *dip;         /* back pointer to devinfo */
} devstate_t;
static void *statep;

int
_init(void)
{
    int error;

    error = ddi_soft_state_init(&statep, sizeof (devstate_t), 0);
    if (error != 0)
        return (error);
    if ((error = mod_install(&modlinkage)) != 0)
        ddi_soft_state_fini(&statep);
    return (error);
}

int
_fini(void)
{
    int error;

    if ((error = mod_remove(&modlinkage)) != 0)
        return (error);
    ddi_soft_state_fini(&statep);
    return (0);
}

static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    devstate_t *softc;

    switch (cmd) {
    case DDI_ATTACH:
        instance = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
```

**EXAMPLE 1** Creating and Removing Data Structures *(Continued)*

```

        return (DDI_FAILURE);
    softc = ddi_get_soft_state(statep, instance);
    softc->dip = dip;
    ...
    return (DDI_SUCCESS);
default:
    return (DDI_FAILURE);
}
}

static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;

    switch (cmd) {

    case DDI_DETACH:
        instance = ddi_get_instance(dip);
        ...
        ddi_soft_state_free(statep, instance);
        return (DDI_SUCCESS);

    default:
        return (DDI_FAILURE);
    }
}

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *cred_p)
{
    devstate_t *softc;
    int instance;

    instance = getminor(*devp);
    if ((softc = ddi_get_soft_state(statep, instance)) == NULL)
        return (ENXIO);
    ...
    softc->state |= XX_IN_USE;
    ...
    return (0);
}

```

**See Also** [\\_fini\(9E\)](#), [\\_init\(9E\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [ddi\\_get\\_instance\(9F\)](#), [getminor\(9F\)](#), [kmem\\_zalloc\(9F\)](#)



---

### *Writing Device Drivers*

**Warnings** There is no attempt to validate the `item` parameter given to `ddi_soft_state_zalloc()` other than it must be a positive signed integer. Therefore very large item numbers may cause the driver to hang forever waiting for virtual memory resources that can never be satisfied.

**Notes** If necessary, a hierarchy of state structures can be constructed by embedding state pointers in higher order state structures.

**Diagnostics** All of the messages described below usually indicate bugs in the driver and should not appear in normal operation of the system.

WARNING: ddi\_soft\_state\_zalloc: bad handle

WARNING: ddi\_soft\_state\_free: bad handle

WARNING: ddi\_soft\_state\_fini: bad handle

The implementation-dependent information kept in the state variable is corrupt.

WARNING: ddi\_soft\_state\_free: null handle

WARNING: ddi\_soft\_state\_fini: null handle

The routine has been passed a null or corrupt state pointer. Check that `ddi_soft_state_init()` has been called.

WARNING: ddi\_soft\_state\_free: item %d not in range [0..%d]

The routine has been asked to free an item which was never allocated. The message prints out the invalid item number and the acceptable range.

**Name** ddi\_strtol – String conversion routines

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_strtol(const char *str, char **endptr, int base,
              long *result);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>str</i>	Pointer to a character string to be converted.
<i>endptr</i>	Post-conversion final string of unrecognized characters.
<i>base</i>	Radix used for conversion.
<i>result</i>	Pointer to variable which contains the converted value.

**Description** The `ddi_strtol()` function converts the initial portion of the string pointed to by *str* to a type long int representation and stores the converted value in *result*.

The function first decomposes the input string into three parts:

1. An initial (possibly empty) sequence of white-space characters (' ', '\t', '\n', '\r', '\f')
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The `ddi_strtol()` function then attempts to convert the subject sequence to an integer and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a plus (“+”) or minus (“-”) sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35 and only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign, if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence

contains no characters if the input string is empty or consists entirely of white-space characters or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed and the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

**Return Values** Upon successful completion, `ddi_strtol()` returns 0 and stores the converted value in *result*. If no conversion is performed due to invalid *base*, `ddi_strtol()` returns EINVAL and the variable pointed by *result* is not changed.

If the correct value is outside the range of representable values, `ddi_strtol()` returns ERANGE and the value pointed to by *result* is not changed.

**Context** The `ddi_strtol()` function may be called from user, kernel or interrupt context.

**See Also** *Writing Device Drivers*

**Name** ddi\_strtoll, ddi\_strtoull – string conversion functions

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_strtoll(const char *str, char **endptr, int base,
               longlong_t *result);
```

```
int ddi_strtoull(const char *str, char **endptr, int base,
                 u_longlong_t *result);
```

**Parameters** *str* pointer to a character string to be converted  
*endptr* post-conversion final string of unrecognized characters  
*base* radix used for conversion  
*result* pointer to variable which contains the converted value

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The `ddi_strtoll()` function converts the initial portion of the string pointed to by *str* to a type `longlong_t` representation and stores the converted value in *result*.

The `ddi_strtoull()` function converts the initial portion of the string pointed to by *str* to a type `u_longlong_t` representation and stores the converted value in *result*.

These functions first decomposes the input string into three parts:

1. An initial (possibly empty) sequence of white-space characters (“”, “\t”, “\n”, “\r”, “\f”)
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The `ddi_strtoll()` function then attempts to convert the subject sequence to an integer and returns the result. The `ddi_strtoull()` function attempts to convert the subject sequence to an unsigned integer and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a plus (“+”) or minus (“-”) sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign. The letters from a (or A) to z (or Z) inclusive are ascribed the

values 10 to 35 and only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed and the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

**Return Values** Upon successful completion, these functions return 0 and store the converted value in *result*. If no conversion is performed due to an invalid base, these functions return EINVAL and the variable pointed by *result* is not changed.

**Context** These functions may be called from user, kernel or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [Writing Device Drivers](#)

**Name** ddi\_strtoul – String conversion functions

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_strtoul(const char *str, char **endptr, int base,
               unsigned long *result);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>str</i>	Pointer to a character string to be converted.
<i>endptr</i>	Post-conversion final string of unrecognized characters.
<i>base</i>	Radix used for conversion.
<i>result</i>	Pointer to variable which contains the converted value.

**Description** The `ddi_strtoul()` function converts the initial portion of the string pointed to by *str* to a type `unsigned long int` representation and stores the converted value in *result*.

The function first decomposes the input string into three parts:

1. An initial (possibly empty) sequence of white-space characters (' ', '\t', '\n', '\r', '\f')
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The `ddi_strtoul()` function then attempts to convert the subject sequence to an `unsigned integer` and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a plus (“+”) or minus (“-”) sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35 and only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence

contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed and the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

**Return Values** Upon successful completion, `ddi_strtol()` returns 0 and stores the converted value in *result*. If no conversion is performed due to invalid *base*, `ddi_strtol()` returns EINVAL and the variable pointed by *result* is not changed.

If the correct value is outside the range of representable values, `ddi_strtol()` returns ERANGE and the value pointed to by *result* is not changed.

**Context** The `ddi_strtol()` function may be called from user, kernel or interrupt context.

**See Also** *Writing Device Drivers*

**Name** ddi\_uem\_alloc, ddi\_uem\_free – allocate and free page-aligned kernel memory

**Synopsis**

```
#include <sys/types.h>
#include <sys/sunddi.h>
```

```
void *ddi_uem_alloc(size_t size, int flag,
    ddi_uem_cookie_t *cookiep);

void ddi_uem_free(ddi_uem_cookie_t cookie);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

### Parameters

ddi\_uem\_alloc() *size*        Number of bytes to allocate.

*flag*                    Used to determine the sleep and pageable conditions.

Possible sleep flags are DDI\_UMEM\_SLEEP, which allows sleeping until memory is available, and DDI\_UMEM\_NOSLEEP, which returns NULL immediately if memory is not available.

The default condition is to allocate locked memory; this can be changed to allocate pageable memory using the DDI\_UMEM\_PAGEABLE flag.

*cookiep*                Pointer to a kernel memory cookie.

ddi\_uem\_free() *cookie*     A kernel memory cookie allocated in ddi\_uem\_alloc().

**Description** ddi\_uem\_alloc() allocates page-aligned kernel memory and returns a pointer to the allocated memory. The number of bytes allocated is a multiple of the system page size (roundup of *size*). The allocated memory can be used in the kernel and can be exported to user space. See [devmap\(9E\)](#) and [devmap\\_uem\\_setup\(9F\)](#) for further information.

*flag* determines whether the caller can sleep for memory and whether the allocated memory is locked or not. DDI\_UMEM\_SLEEP allocations may sleep but are guaranteed to succeed. DDI\_UMEM\_NOSLEEP allocations do not sleep but may fail (return NULL) if memory is currently unavailable. If DDI\_UMEM\_PAGEABLE is set, pageable memory will be allocated. These pages can be swapped out to secondary memory devices. The initial contents of memory allocated using ddi\_uem\_alloc() is zero-filled.

\**cookiep* is a pointer to the kernel memory cookie that describes the kernel memory being allocated. A typical use of *cookiep* is in [devmap\\_uem\\_setup\(9F\)](#) when the drivers want to export the kernel memory to a user application.

To free the allocated memory, a driver calls ddi\_uem\_free() with the cookie obtained from ddi\_uem\_alloc(). ddi\_uem\_free() releases the entire buffer.



**Return Values** Non-null Successful completion. `ddi_umem_alloc()` returns a pointer to the allocated memory.

NULL Memory cannot be allocated by `ddi_umem_alloc()` because `DDI_UMEM_NOSLEEP` is set and the system is out of resources.

**Context** `ddi_umem_alloc()` can be called from any context if *flag* is set to `DDI_UMEM_NOSLEEP`. If `DDI_UMEM_SLEEP` is set, `ddi_umem_alloc()` can be called from user and kernel context only. `ddi_umem_free()` can be called from any context.

**See Also** [devmap\(9E\)](#), [condvar\(9F\)](#), [devmap\\_umem\\_setup\(9F\)](#), [kmem\\_alloc\(9F\)](#), [mutex\(9F\)](#), [rwlock\(9F\)](#), [semaphore\(9F\)](#)

### *Writing Device Drivers*

**Warnings** Setting the `DDI_UMEM_PAGEABLE` flag in `ddi_umem_alloc()` will result in an allocation of pageable memory. Because these pages can be swapped out to secondary memory devices, drivers should use this flag with care. This memory must not be used for the following purposes:

- For synchronization objects such as locks and condition variables. See [mutex\(9F\)](#), [semaphore\(9F\)](#), [rwlock\(9F\)](#), and [condvar\(9F\)](#).
- For driver interrupt routines.

Memory allocated using `ddi_umem_alloc()` without setting `DDI_UMEM_PAGEABLE` flag cannot be paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.

Excessive use of kernel memory is likely to effect overall system performance. Over-commitment of kernel memory may cause unpredictable consequences.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing an invalid pointer, will cause the system to corrupt data or panic.

Do not call `ddi_umem_alloc()` within `DDI_SUSPEND` and `DDI_RESUME` operations. Memory acquired at these times is not reliable. In some cases, such a call can cause a system to hang.

**Notes** `ddi_umem_alloc(0, flag, cookiep)` always returns NULL. `ddi_umem_free(NULL)` has no effects on system.

**Name** ddi\_umem\_iosetup – Setup I/O requests to application memory

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
struct buf *ddi_umem_iosetup(ddi_umem_cookie_t cookie, off_t off,
    size_t len, int direction, dev_t dev, daddr_t blkno,
    int (*iodone) (struct buf *), int sleepflag);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>cookie</i>	The kernel memory cookie allocated by <a href="#">ddi_umem_lock(9F)</a> .
<i>off</i>	Offset from the start of the cookie.
<i>len</i>	Length of the I/O request in bytes.
<i>direction</i>	Must be set to B_READ for reads from the device or B_WRITE for writes to the device.
<i>dev</i>	Device number
<i>blkno</i>	Block number on device.
<i>iodone</i>	Specific <a href="#">biodone(9F)</a> routine.
<i>sleepflag</i>	Determines whether caller can sleep for memory. Possible flags are DDI_UMEM_SLEEP to allow sleeping until memory is available, or DDI_UMEM_NOSLEEP to return NULL immediately if memory is not available.

**Description** The `ddi_umem_iosetup(9F)` function is used by drivers to setup I/O requests to application memory which has been locked down using [ddi\\_umem\\_lock\(9F\)](#).

The `ddi_umem_iosetup(9F)` function returns a pointer to a [buf\(9S\)](#) structure corresponding to the memory cookie *cookie*. Drivers can setup multiple buffer structures simultaneously active using the same memory cookie. The [buf\(9S\)](#) structures can span all or part of the region represented by the cookie and can overlap each other. The [buf\(9S\)](#) structure can be passed to [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#) to initiate DMA transfers to or from the locked down memory.

The *off* parameter specifies the offset from the start of the cookie. The *len* parameter represents the length of region to be mapped by the buffer. The *direction* parameter must be set to either B\_READ or B\_WRITE, to indicate the action that will be performed by the device. (Note that this direction is in the opposite sense of the VM system's direction of DDI\_UMEMLOCK\_READ and DDI\_UMEMLOCK\_WRITE.) The direction must be compatible with the flags used to create the memory cookie in [ddi\\_umem\\_lock\(9F\)](#). For example, if `ddi_umem_lock()` is called with the *flags* parameter set to DDI\_UMEMLOCK\_READ, the *direction* parameter in `ddi_umem_iosetup()` should be set to B\_WRITE.

The *dev* parameter specifies the device to which the buffer is to perform I/O. The *blkno* parameter represents the block number on the device. It will be assigned to the `b_blkno` field of the returned buffer structure. The *iodone* parameter enables the driver to identify a specific [biodone\(9F\)](#) routine to be called by the driver when the I/O is complete. The *sleepflag* parameter determines if the caller can sleep for memory. `DDI_UMEM_SLEEP` allocations may sleep but are guaranteed to succeed. `DDI_UMEM_NOSLEEP` allocations do not sleep but may fail (return `NULL`) if memory is currently not available.

After the I/O has completed and the buffer structure is no longer needed, the driver calls [freerbuf\(9F\)](#) to free the buffer structure.

**Return Values** The `ddi_umem_iosetup(9F)` function returns a pointer to the initialized buffer header, or `NULL` if no space is available.

**Context** The `ddi_umem_iosetup(9F)` function can be called from any context only if `flag` is set to `DDI_UMEM_NOSLEEP`. If `DDI_UMEM_SLEEP` is set, `ddi_umem_iosetup(9F)` can be called from user and kernel context only.

**See Also** [ddi\\_umem\\_lock\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [freerbuf\(9F\)](#), [physio\(9F\)](#), [buf\(9S\)](#)

**Name** ddi\_umem\_lock, ddi\_umem\_unlock – lock and unlock memory pages

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int ddi_umem_lock(caddr_t addr, size_t len, int flags,
                 ddi_umem_cookie_t *cookiep);

void ddi_umem_unlock(ddi_umem_cookie_t cookie);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

### Parameters

ddi_umem_lock	<i>addr</i>	Virtual address of memory object
	<i>len</i>	Length of memory object in bytes
	<i>flags</i>	Valid flags include:
	DDI_UMEMLOCK_READ	Memory pages are locked to be read from. (Disk write or a network send.)
	DDI_UMEMLOCK_WRITE	Memory pages are locked to be written to. (Disk read or a network receive.)
	<i>cookiep</i>	Pointer to a kernel memory cookie.
ddi_umem_unlock	<i>cookie</i>	Kernel memory cookie allocated by ddi_umem_lock().

**Description** The ddi\_umem\_lock() function locks down the physical pages (including I/O pages) that correspond to the current process' virtual address range [*addr*, *addr* + *size*) and fills in a cookie representing the locked pages. This cookie can be used to create a [buf\(9S\)](#) structure that can be used to perform I/O (see [ddi\\_umem\\_iosetup\(9F\)](#) and [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#)), or it can be used with [devmap\\_umem\\_setup\(9F\)](#) to export the memory to an application.

The virtual address and length specified must be at a page boundary and the mapping performed in terms of the system page size. See [pagesize\(1\)](#).

The flags argument indicates the intended use of the locked memory. Set flags to DDI\_UMEMLOCK\_READ if the memory pages will be read (for example, in a disk write or a network send.) Set flags to DDI\_UMEMLOCK\_WRITE if the memory pages will be written (for example, in a disk read or a network receive). You must choose one (and only one) of these values.

To unlock the locked pages, the drivers call ddi\_umem\_unlock(9F) with the cookie obtained from ddi\_umem\_lock().

The process is not allowed to [exec\(2\)](#) or [fork\(2\)](#) while its physical pages are locked down by the device driver.

---

The device driver must ensure that the physical pages have been unlocked after the application has called `close(2)`.

**Return Values** On success, a `0` is returned. Otherwise, one of the following `errno` values is returned.

- EFAULT** User process has no mapping at that address range or does not support locking
- EACCES** User process does not have the required permission.
- ENOMEM** The system does not have sufficient resources to lock memory, or locking *len* memory would exceed a limit or resource control on locked memory.
- EAGAIN** Could not allocate system resources required to lock the pages. The `ddi_umem_lock()` could succeed at a later time.
- EINVAL** Requested memory is not aligned on a system page boundary.

**Context** The `ddi_umem_lock()` function can only be called from user context; `ddi_umem_unlock()` from user, kernel, and interrupt contexts.

**See Also** [ddi\\_umem\\_iosetup\(9F\)](#), [ddi\\_dma\\_buf\\_bind\\_handle\(9F\)](#), [devmap\\_umem\\_setup\(9F\)](#), [ddi\\_umem\\_alloc\(9F\)](#)

**Notes** The `ddi_umem_unlock()` function consumes physical memory. The driver is responsible for a speedy unlock to free up the resources.

The `ddi_umem_unlock()` function can defer unlocking of the pages to a later time depending on the implementation.

**Name** delay – delay execution for a specified number of clock ticks

**Synopsis** #include <sys/ddi.h>

```
void delay(clock_t ticks);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *ticks* The number of clock cycles to delay.

**Description** delay() provides a mechanism for a driver to delay its execution for a given period of time. Since the speed of the clock varies among systems, drivers should base their time values on microseconds and use [drv\\_usectohz\(9F\)](#) to convert microseconds into clock ticks.

delay() uses [timeout\(9F\)](#) to schedule an internal function to be called after the specified amount of time has elapsed. delay() then waits until the function is called. Because timeout() is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use [cv\\_timedwait\(9F\)](#) rather than delay().

delay() does not busy-wait. If busy-waiting is required, use [drv\\_usecwait\(9F\)](#).

**Context** delay() can be called from user and kernel contexts.

**Examples** EXAMPLE 1 delay() Example

Before a driver I/O routine allocates buffers and stores any user data in them, it checks the status of the device (line 12). If the device needs manual intervention (such as, needing to be refilled with paper), a message is displayed on the system console (line 14). The driver waits an allotted time (line 17) before repeating the procedure.

```

1 struct device { /* layout of physical device registers */
2     int     control; /* physical device control word */
3     int     status; /* physical device status word */
4     short  xmit_char; /* transmit character to device */
5 };
6
7
8     . . .
9 /* get device registers */
10 register struct device *rp = ...
11
12 while (rp->status & NOPAPER) { /* while printer is out of paper */
13     /* display message and ring bell */
14     /* on system console */
15     cmn_err(CE_WARN, "\007",
16             (getminor(dev) & 0xf));
17     /* wait one minute and try again */
18     delay(60 * drv_usectohz(1000000));
19 }
```

**See Also** `biodone(9F)`, `biowait(9F)`, `cv_timedwait(9F)`, `ddi_in_panic(9F)`, `drv_hztousec(9F)`, `drv_usectohz(9F)`, `drv_usecwait(9F)`, `timeout(9F)`, `untimeout(9F)`

*Writing Device Drivers*

**Name** devmap\_default\_access – default driver memory access function

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int devmap_default_access(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, uint_t type, uint_t rw);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dhp* An opaque mapping handle that the system uses to describe the mapping.  
*pvtp* Driver private mapping data.  
*off* User offset within the logical device memory at which the access begins.  
*len* Length (in bytes) of the memory being accessed.  
*type* Type of access operation.  
*rw* Type of access.

**Description** devmap\_default\_access() is a function providing the semantics of devmap\_access(9E). The drivers call devmap\_default\_access() to handle the mappings that do not support context switching. The drivers should call devmap\_do\_ctxmgt(9F) for the mappings that support context management.

devmap\_default\_access() can either be called from devmap\_access(9E) or be used as the devmap\_access(9E) entry point. The arguments *dhp*, *pvtp*, *off*, *len*, *type*, and *rw* are provided by the devmap\_access(9E) entry point and must not be modified.

**Return Values** 0 Successful completion.  
Non-zero An error occurred.

**Context** devmap\_default\_access() must be called from the driver's devmap\_access(9E) entry point.

**Examples** EXAMPLE 1 Using devmap\_default\_access in devmap\_access.

The following shows an example of using devmap\_default\_access() in the devmap\_access(9E) entry point.

```
. . .
#define OFF_DO_CTXMGT 0x40000000
#define OFF_NORMAL 0x40100000
#define CTXMGT_SIZE 0x100000
#define NORMAL_SIZE 0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
 */
```



EXAMPLE 1 Using devmap\_default\_access in devmap\_access. (Continued)

```

static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset,
              size_t length, uint_t type, uint_t rw)
{
    .....
    /*
     * see devmap_contextmgt(9E) for an example
     */
}

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
               size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;

    /*
     * check if off is within the range that supports
     * context management.
     */
    if ((diff = off - OFF_DO_CTXMG) >= 0 && diff < CTXMG_SIZE) {
        /*
         * calculates the length for context switching
         */
        if ((len + off) > (OFF_DO_CTXMG + CTXMG_SIZE))
            return (-1);
        /*
         * perform context switching
         */
        err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
                              rw, xx_context_mgt);
    }
    /*
     * check if off is within the range that does normal
     * memory mapping.
     */
    } else if ((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) {
        if ((len + off) > (OFF_NORMAL + NORMAL_SIZE))
            return (-1);
        err = devmap_default_access(dhp, pvtp, off, len, type, rw);
    } else
        return (-1);
}

```

**EXAMPLE 1** Using `devmap_default_access` in `devmap_access`. *(Continued)*

```
        return (err);
    }
```

**See Also** [devmap\\_access\(9E\)](#), [devmap\\_do\\_ctxmgt\(9F\)](#), [devmap\\_callback\\_ctl\(9S\)](#)

*Writing Device Drivers*

**Name** devmap\_devmem\_setup, devmap\_umem\_setup – set driver memory mapping parameters

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int devmap_devmem_setup(devmap_cookie_t dhp, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, uint_t rnumber,
    offset_t roff, size_t len, uint_t maxprot,
    uint_t flags, ddi_device_acc_attr_t *accattrp);

int devmap_umem_setup(devmap_cookie_t dhp, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie,
    offset_t koff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** devmap\_devmem\_setup() parameters:

<i>dhp</i>	An opaque mapping handle that the system uses to describe the mapping.										
<i>dip</i>	Pointer to the device's dev_info structure.										
<i>callbackops</i>	Pointer to a <a href="#">devmap_callback_ctl(9S)</a> structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping. The framework will copy the structure to the system private memory.										
<i>rnumber</i>	Index number to the register address space set.										
<i>roff</i>	Offset into the register address space.										
<i>len</i>	Length (in bytes) of the mapping to be mapped.										
<i>maxprot</i>	Maximum protection flag possible for attempted mapping. Some combinations of possible settings are: <table> <tr> <td>PROT_READ</td> <td>Read access is allowed.</td> </tr> <tr> <td>PROT_WRITE</td> <td>Write access is allowed.</td> </tr> <tr> <td>PROT_EXEC</td> <td>Execute access is allowed.</td> </tr> <tr> <td>PROT_USER</td> <td>User-level access is allowed. The mapping is done as a result of a <a href="#">mmap(2)</a> system call.</td> </tr> <tr> <td>PROT_ALL</td> <td>All access is allowed.</td> </tr> </table>	PROT_READ	Read access is allowed.	PROT_WRITE	Write access is allowed.	PROT_EXEC	Execute access is allowed.	PROT_USER	User-level access is allowed. The mapping is done as a result of a <a href="#">mmap(2)</a> system call.	PROT_ALL	All access is allowed.
PROT_READ	Read access is allowed.										
PROT_WRITE	Write access is allowed.										
PROT_EXEC	Execute access is allowed.										
PROT_USER	User-level access is allowed. The mapping is done as a result of a <a href="#">mmap(2)</a> system call.										
PROT_ALL	All access is allowed.										
<i>flags</i>	Used to determine the cache attribute. <p>Possible values of the cache attribute are:</p>										

IOMEM_DATA_CACHED	The CPU can cache the data it fetches and push it to memory at a later time. This is the default attribute that is used if no cache attributes are specified.
IOMEM_DATA_UC_WR_COMBINE	The CPU never caches the data, but writes can occur out of order or can be combined. Reordering is implied.  If IOMEM_DATA_UC_WR_COMBINE is specified but not supported, IOMEM_DATA_UNCACHED is used instead.
IOMEM_DATA_UNCACHED	The CPU never caches data, but has uncacheable access to memory. Strict ordering is implied.

The cache attributes are mutually exclusive. Any combination of the values leads to a failure. On the SPARC architecture, only IOMEM\_DATA\_CACHED is meaningful. Others lead to a failure.

*accattrp* Pointer to a `ddi_device_acc_attr()` structure of the device. See [ddi\\_device\\_acc\\_attr\(9S\)](#). The value in `devacc_attr_dataorder` is ignored in the current release. The value in `devacc_attr_endian_flags` is meaningful on the SPARC architecture only.

`devmap_umem_setup()` parameters:

<i>dhp</i>	An opaque data structure that the system uses to describe the mapping.
<i>dip</i>	Pointer to the device's <code>dev_info</code> structure.
<i>callbacks</i>	Pointer to a <a href="#">devmap_callback_ctl(9S)</a> structure. The structure contains pointers to device driver-supplied functions that manage events on the device mapping.
<i>cookie</i>	A kernel memory <i>cookie</i> (see <a href="#">ddi_umem_alloc(9F)</a> ).
<i>koff</i>	Offset into the kernel memory defined by <i>cookie</i> .
<i>len</i>	Length (in bytes) of the mapping to be mapped.
<i>maxprot</i>	Maximum protection flag possible for attempted mapping. Some combinations of possible settings are:
PROT_READ	Read access is allowed.
PROT_WRITE	Write access is allowed.
PROT_EXEC	Execute access is allowed.

	PROT_USER	User-level access is allowed (the mapping is being done as a result of a <code>mmap(2)</code> system call).
	PROT_ALL	All access is allowed.
<i>flags</i>		Must be set to 0.
<i>accattrp</i>		Pointer to a <code>ddi_device_acc_attr(9S)</code> structure. Ignored in the current release. Reserved for future use.

**Description** The `devmap_devmem_setup()` and `devmap_umem_setup()` functions are used in the `devmap(9E)` entry point to pass mapping parameters from the driver to the system.

The *dhp* argument specifies a device mapping handle that the system uses to store all mapping parameters of a physical contiguous memory. The system copies the data pointed to by *callbackops* to a system private memory. This allows the driver to free the data after returning from either `devmap_devmem_setup()` or `devmap_umem_setup()`. The driver is notified of user events on the mappings via the entry points defined by `devmap_callback_ctl(9S)`. The driver is notified of the following user events:

Mapping Setup	User has called <code>mmap(2)</code> to create a mapping to the device memory.
Access	User has accessed an address in the mapping that has no translations.
Duplication	User has duplicated the mapping. Mappings are duplicated when the process calls <code>fork(2)</code> .
Unmapping	User has called <code>munmap(2)</code> on the mapping or is exiting, <code>exit(2)</code> .

See `devmap_map(9E)`, `devmap_access(9E)`, `devmap_dup(9E)`, and `devmap_unmap(9E)` for details on these entry points.

By specifying a valid *callbackops* to the system, device drivers can manage events on a device mapping. For example, the `devmap_access(9E)` entry point allows the drivers to perform context switching by unloading the mappings of other processes and to load the mapping of the calling process. Device drivers may specify NULL to *callbackops* which means the drivers do not want to be notified by the system.

The maximum protection allowed for the mapping is specified in *maxprot*. *accattrp* defines the device access attributes. See `ddi_device_acc_attr(9S)` for more details.

`devmap_devmem_setup()` is used for device memory to map in the register set given by *rnumber* and the offset into the register address space given by *roff*. The system uses *rnumber* and *roff* to go up the device tree to get the physical address that corresponds to *roff*. The range to be affected is defined by *len* and *roff*. The range from *roff* to *roff + len* must be a physical contiguous memory and page aligned.

Drivers use `devmap_umem_setup()` for kernel memory to map in the kernel memory described by *cookie* and the offset into the kernel memory space given by *koff*. *cookie* is a kernel memory

pointer obtained from [ddi\\_umem\\_alloc\(9F\)](#). If *cookie* is `NULL`, `devmap_umem_setup()` returns `-1`. The range to be affected is defined by *len* and *koff*. The range from *koff* to *koff* + *len* must be within the limits of the kernel memory described by *koff* + *len* and must be page aligned.

Drivers use `devmap_umem_setup()` to export the kernel memory allocated by [ddi\\_umem\\_alloc\(9F\)](#) to user space. The system selects a user virtual address that is aligned with the kernel virtual address being mapped to avoid cache incoherence if the mapping is not `MAP_FIXED`.

**Return Values** `0` Successful completion.  
`-1` An error occurred.

**Context** `devmap_devmem_setup()` and `devmap_umem_setup()` can be called from user, kernel, and interrupt context.

**See Also** [exit\(2\)](#), [fork\(2\)](#), [mmap\(2\)](#), [munmap\(2\)](#), [devmap\(9E\)](#), [ddi\\_umem\\_alloc\(9F\)](#), [ddi\\_device\\_acc\\_attr\(9S\)](#), [devmap\\_callback\\_ctl\(9S\)](#)

*Writing Device Drivers*

**Name** devmap\_do\_ctxmgt – perform device context switching on a mapping

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int devmap_do_ctxmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
                    size_t len, uint_t type,
                    uint_t rw, int (*devmap_contextmgt)devmap_cookie_t,
                    void *, offset_t, size_t, uint_t, uint_t);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

<b>Parameters</b>	<i>dhp</i>	An opaque mapping handle that the system uses to describe the mapping.
	<i>pvtp</i>	Driver private mapping data.
	<i>off</i>	User offset within the logical device memory at which the access begins.
	<i>len</i>	Length (in bytes) of the memory being accessed.
	<i>devmap_contextmgt</i>	The address of driver function that the system will call to perform context switching on a mapping. See <a href="#">devmap_contextmgt(9E)</a> for details.
	<i>type</i>	Type of access operation. Provided by <a href="#">devmap_access(9E)</a> . Should not be modified.
	<i>rw</i>	Direction of access. Provided by <a href="#">devmap_access(9E)</a> . Should not be modified.

**Description** Device drivers call `devmap_do_ctxmgt()` in the [devmap\\_access\(9E\)](#) entry point to perform device context switching on a mapping. `devmap_do_ctxmgt()` passes a pointer to a driver supplied callback function, [devmap\\_contextmgt\(9E\)](#), to the system that will perform the actual device context switching. If [devmap\\_contextmgt\(9E\)](#) is not a valid driver callback function, the system will fail the memory access operation which will result in a SIGSEGV or SIGBUS signal being delivered to the process.

`devmap_do_ctxmgt()` performs context switching on the mapping object identified by *dhp* and *pvtp* in the range specified by *off* and *len*. The arguments *dhp*, *pvtp*, *type*, and *rw* are provided by the [devmap\\_access\(9E\)](#) entry point and must not be modified. The range from *off* to *off+len* must support context switching.

The system will pass through *dhp*, *pvtp*, *off*, *len*, *type*, and *rw* to [devmap\\_contextmgt\(9E\)](#) in order to perform the actual device context switching. The return value from [devmap\\_contextmgt\(9E\)](#) will be returned directly to `devmap_do_ctxmgt()`.

**Return Values** 0            Successful completion.  
 Non-zero        An error occurred.

**Context** devmap\_do\_ctxmgt ( ) must be called from the driver's [devmap\\_access\(9E\)](#) entry point.

**Examples** EXAMPLE 1 Using devmap\_do\_ctxmgt in the devmap\_access entry point.

The following shows an example of using devmap\_do\_ctxmgt ( ) in the [devmap\\_access\(9E\)](#) entry point.

```
. . .
#define OFF_DO_CTXMGT  0x40000000
#define OFF_NORMAL    0x40100000
#define CTXMGT_SIZE   0x100000
#define NORMAL_SIZE   0x100000

/*
 * Driver devmap_contextmgt(9E) callback function.
 */
static int
xx_context_mgt(devmap_cookie_t dhp, void *pvtp, offset_t offset,
              size_t length, uint_t type, uint_t rw)
{
    . . . . .
    /*
     * see devmap_contextmgt(9E) for an example
     */
}

/*
 * Driver devmap_access(9E) entry point
 */
static int
xxdevmap_access(devmap_cookie_t dhp, void *pvtp, offset_t off,
               size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int err;

    /*
     * check if off is within the range that supports
     * context management.
     */
    if ((diff = off - OFF_DO_CTXMGT) >= 0 && diff < CTXMGT_SIZE) {
        /*
         * calculates the length for context switching
         */
        if ((len + off) > (OFF_DO_CTXMGT + CTXMGT_SIZE))
```



**EXAMPLE 1** Using `devmap_do_ctxmgt` in the `devmap_access` entry point. *(Continued)*

```

        return (-1);
    /*
     * perform context switching
     */
    err = devmap_do_ctxmgt(dhp, pvtp, off, len, type,
                          rw, xx_context_mgt);
    /*
     * check if off is within the range that does normal
     * memory mapping.
     */
    } else if ((diff = off - OFF_NORMAL) >= 0 && diff < NORMAL_SIZE) {
        if ((len + off) > (OFF_NORMAL + NORMAL_SIZE))
            return (-1);
        err = devmap_default_access(dhp, pvtp, off, len, type, rw);
    } else
        return (-1);

    return (err);
}

```

**See Also** [devmap\\_access\(9E\)](#), [devmap\\_contextmgt\(9E\)](#), [devmap\\_default\\_access\(9F\)](#)

*Writing Device Drivers*

**Name** devmap\_set\_ctx\_timeout – set the timeout value for the context management callback

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
void devmap_set_ctx_timeout(devmap_cookie_t dhp, clock_t ticks);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dhp* An opaque mapping handle that the system uses to describe the mapping.  
*ticks* Number of clock ticks to wait between successive calls to the context management callback function.

**Description** The devmap\_set\_ctx\_timeout() function specifies the time interval for the system to wait between successive calls to the driver's context management callback function, [devmap\\_contextmgt\(9E\)](#).

Device drivers typically call devmap\_set\_ctx\_timeout() in the [devmap\\_map\(9E\)](#) routine. If the drivers do not call devmap\_set\_ctx\_timeout() to set the timeout value, the default timeout value of 0 will result in no delay between successive calls to the driver's [devmap\\_contextmgt\(9E\)](#) callback function.

**Context** The devmap\_set\_ctx\_timeout() function can be called from user, interrupt, or kernel context.

**See Also** [devmap\\_contextmgt\(9E\)](#), [devmap\\_map\(9E\)](#), [timeout\(9F\)](#)

**Name** devmap\_setup, ddi\_devmap\_segmap – set up a user mapping to device memory using the devmap framework

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int devmap_setup(dev_t dev, offset_t off, ddi_as_handle_t as,
    caddr_t *addrp, size_t len, uint_t prot, uint_t maxprot,
    uint_t flags, cred_t *cred);
```

```
int ddi_devmap_segmap(dev_t dev, off_t off, ddi_as_handle_t as,
    caddr_t *addrp, off_t len, uint_t prot, uint_t maxprot,
    uint_t flags, cred_t *cred);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dev</i>	Device whose memory is to be mapped.										
<i>off</i>	User offset within the logical device memory at which the mapping begins.										
<i>as</i>	An opaque data structure that describes the address space into which the device memory should be mapped.										
<i>addrp</i>	Pointer to the starting address in the address space into which the device memory should be mapped.										
<i>len</i>	Length (in bytes) of the memory to be mapped.										
<i>prot</i>	A bit field that specifies the protections. Some possible settings combinations are: <table> <tr> <td>PROT_READ</td> <td>Read access is desired.</td> </tr> <tr> <td>PROT_WRITE</td> <td>Write access is desired.</td> </tr> <tr> <td>PROT_EXEC</td> <td>Execute access is desired.</td> </tr> <tr> <td>PROT_USER</td> <td>User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).</td> </tr> <tr> <td>PROT_ALL</td> <td>All access is desired.</td> </tr> </table>	PROT_READ	Read access is desired.	PROT_WRITE	Write access is desired.	PROT_EXEC	Execute access is desired.	PROT_USER	User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).	PROT_ALL	All access is desired.
PROT_READ	Read access is desired.										
PROT_WRITE	Write access is desired.										
PROT_EXEC	Execute access is desired.										
PROT_USER	User-level access is desired (the mapping is being done as a result of a <code>mmap(2)</code> system call).										
PROT_ALL	All access is desired.										
<i>maxprot</i>	Maximum protection flag possible for attempted mapping; the PROT_WRITE bit may be masked out if the user opened the special file read-only.										
<i>flags</i>	Flags indicating type of mapping. The following flags can be specified: <table> <tr> <td>MAP_PRIVATE</td> <td>Changes are private.</td> </tr> <tr> <td>MAP_SHARED</td> <td>Changes should be shared.</td> </tr> <tr> <td>MAP_FIXED</td> <td>The user specified an address in <i>*addrp</i> rather than letting the system choose an address.</td> </tr> </table>	MAP_PRIVATE	Changes are private.	MAP_SHARED	Changes should be shared.	MAP_FIXED	The user specified an address in <i>*addrp</i> rather than letting the system choose an address.				
MAP_PRIVATE	Changes are private.										
MAP_SHARED	Changes should be shared.										
MAP_FIXED	The user specified an address in <i>*addrp</i> rather than letting the system choose an address.										

*cred*            Pointer to the user credential structure.

**Description** `devmap_setup()` and `ddi_devmap_segmap()` allow device drivers to use the devmap framework to set up user mappings to device memory. The devmap framework provides several advantages over the default device mapping framework that is used by `ddi_segmap(9F)` or `ddi_segmap_setup(9F)`. Device drivers should use the devmap framework, if the driver wants to:

- use an optimal MMU pagesize to minimize address translations,
- conserve kernel resources,
- receive callbacks to manage events on the mapping,
- export kernel memory to applications,
- set up device contexts for the user mapping if the device requires context switching,
- assign device access attributes to the user mapping, or
- change the maximum protection for the mapping.

`devmap_setup()` must be called in the `segmap(9E)` entry point to establish the mapping for the application. `ddi_devmap_segmap()` can be called in, or be used as, the `segmap(9E)` entry point. The differences between `devmap_setup()` and `ddi_devmap_segmap()` are in the data type used for *off* and *len*.

When setting up the mapping, `devmap_setup()` and `ddi_devmap_segmap()` call the `devmap(9E)` entry point to validate the range to be mapped. The `devmap(9E)` entry point also translates the logical offset (as seen by the application) to the corresponding physical offset within the device address space. If the driver does not provide its own `devmap(9E)` entry point, `EINVAL` will be returned to the `mmap(2)` system call.

**Return Values** `0`            Successful completion.  
 Non-zero            An error occurred. The return value of `devmap_setup()` and `ddi_devmap_segmap()` should be used directly in the `segmap(9E)` entry point.

**Context** `devmap_setup()` and `ddi_devmap_segmap()` can be called from user or kernel context only.

**See Also** `mmap(2)`, `devmap(9E)`, `segmap(9E)`, `ddi_segmap(9F)`, `ddi_segmap_setup(9F)`, `cb_ops(9S)`

*Writing Device Drivers*

**Name** devmap\_unload, devmap\_load – control validation of memory address translations

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int devmap_load(devmap_cookie_t dhp, offset_t off, size_t len,
               uint_t type, uint_t rw);

int devmap_unload(devmap_cookie_t dhp, offset_t off, size_t len);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dhp</i>	An opaque mapping handle that the system uses to describe the mapping.
<i>off</i>	User offset within the logical device memory at which the loading or unloading of the address translations begins.
<i>len</i>	Length (in bytes) of the range being affected.
devmap_load() only <i>type</i>	Type of access operation.
<i>rw</i>	Direction of access.

**Description** devmap\_unload() and devmap\_load() are used to control the validation of the memory mapping described by *dhp* in the specified range. devmap\_unload() invalidates the mapping translations and will generate calls to the [devmap\\_access\(9E\)](#) entry point next time the mapping is accessed. The drivers use devmap\_load() to validate the mapping translations during memory access.

A typical use of devmap\_unload() and devmap\_load() is in the driver's context management callback function, [devmap\\_contextmgt\(9E\)](#). To manage a device context, a device driver calls devmap\_unload() on the context about to be switched out. It switches contexts, and then calls devmap\_load() on the context switched in. devmap\_unload() can be used to unload the mappings of other processes as well as the mappings of the calling process, but devmap\_load() can only be used to load the mappings of the calling process. Attempting to load another process's mappings with devmap\_load() will result in a system panic.

For both routines, the range to be affected is defined by the *off* and *len* arguments. Requests affect the entire page containing the *off* and all pages up to and including the page containing the last byte as indicated by *off* + *len*. The arguments *type* and *rw* are provided by the system to the calling function (for example, [devmap\\_contextmgt\(9E\)](#)) and should not be modified.

Supplying a value of 0 for the *len* argument affects all addresses from the *off* to the end of the mapping. Supplying a value of 0 for the *off* argument and a value of 0 for *len* argument affect all addresses in the mapping.

A non-zero return value from either devmap\_unload() or devmap\_load() will cause the corresponding operation to fail. The failure may result in a SIGSEGV or SIGBUS signal being delivered to the process.

**Return Values** 0            Successful completion.  
Non-zero        An error occurred.

**Context** These routines can be called from user or kernel context only.

**Examples** EXAMPLE 1 Managing a One-Page Device Context

The following shows an example of managing a device context that is one page in length.

```
struct xx_context cur_ctx;

static int
xxdevmap_contextmgt(devmap_cookie_t dhp, void *pvtp, offset_t off,
    size_t len, uint_t type, uint_t rw)
{
    int err;
    devmap_cookie_t cur_dhp;
    struct xx_pvt *p;
    struct xx_pvt *pvp = (struct xx_pvt *)pvtp;
    /* enable access callbacks for the current mapping */
    if (cur_ctx != NULL && cur_ctx != pvp->ctx) {
        p = cur_ctx->pvt;
        /*
         * unload the region from off to the end of the mapping.
         */
        cur_dhp = p->dhp;
        if ((err = devmap_unload(cur_dhp, off, len)) != 0)
            return (err);
    }
    /* Switch device context - device dependent*/
    ...
    /* Make handle the new current mapping */
    cur_ctx = pvp->ctx;
    /*
     * Disable callbacks and complete the access for the
     * mapping that generated this callback.
     */
    return (devmap_load(pvp->dhp, off, len, type, rw));
}
```

**See Also** [devmap\\_access\(9E\)](#), [devmap\\_contextmgt\(9E\)](#)

*Writing Device Drivers*

**Name** disksort – single direction elevator seek sort for buffers

**Synopsis** `#include <sys/conf.h>`  
`#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
void disksort(struct diskhd *dp, struct buf *bp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dp* A pointer to a `diskhd` structure. A `diskhd` structure is essentially identical to head of a buffer structure (see [buf\(9S\)](#)). The only defined items of interest for this structure are the `av_forw` and `av_back` structure elements which are used to maintain the front and tail pointers of the forward linked I/O request queue.

*bp* A pointer to a buffer structure. Typically this is the I/O request that the driver receives in its strategy routine (see [strategy\(9E\)](#)). The driver is responsible for initializing the `b_resid` structure element to a meaningful sort key value prior to calling `disksort()`.

**Description** The function `disksort()` sorts a pointer to a buffer into a single forward linked list headed by the `av_forw` element of the argument *dp*.

It uses a one-way elevator algorithm that sorts buffers into the queue in ascending order based upon a key value held in the argument buffer structure element `b_resid`.

This value can either be the driver calculated cylinder number for the I/O request described by the buffer argument, or simply the absolute logical block for the I/O request, depending on how fine grained the sort is desired to be or how applicable either quantity is to the device in question.

The head of the linked list is found by use of the `av_forw` structure element of the argument *dp*. The tail of the linked list is found by use of the `av_back` structure element of the argument *dp*. The `av_forw` element of the *bp* argument is used by `disksort()` to maintain the forward linkage. The value at the head of the list presumably indicates the currently active disk area.

**Context** This function can be called from user, interrupt, or kernel context.

**See Also** [strategy\(9E\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Warnings** The `disksort()` function does no locking. Therefore, any locking is completely the responsibility of the caller.

**Name** dlbindack, dlphysaddrack, dllockack, dlerrorrack, dluderrorind – DLPI device driver helper functions

**Synopsis** #include <sys/dlpi.h>

```
void dllockack(queue_t *wq, mblk_t *mp, t_uscalar_t correct_primitive);
void dlerrorrack(queue_t *wq, mblk_t *mp, t_uscalar_t error_primitive,
                 t_uscalar_t error, t_uscalar_t unix_errno);
void dlbindack(queue_t *wq, mblk_t *mp, t_scalar_t sap, const void *addrp,
               t_uscalar_t addrlen, t_uscalar_t maxconind, t_uscalar_t xidtest);
void dlphysaddrack(queue_t *wq, mblk_t *mp, const void *addrp,
                   t_uscalar_t addrlen);
void dluderrorind(queue_t *wq, mblk_t *mp, const void *addrp,
                  t_uscalar_t addrlen, t_uscalar_t error, t_uscalar_t unix_errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

<b>Parameters</b>	<i>wq</i>	Streams write queue.
	<i>mp</i>	Pointer to the bind request message.
	<i>sap</i>	Service access point being requested.
	<i>addrp</i>	Pointer to the dlpi layer source address.
	<i>addrlen</i>	Size of the dlpi layer address pointed to by <i>addrp</i> .
	<i>maxconind</i>	Maximum number of DL_CONNECT_IND messages allowed to be outstanding per stream.
	<i>xidtest</i>	The XID and TEST responses supported.
	<i>correct_primitive</i>	Identifies the DL primitive completing successfully.
	<i>error_primitive</i>	Identifies the DL primitive in error.
	<i>error</i>	DLPI error associated with the failure in the DLPI request.
	<i>unix_errno</i>	Corresponding UNIX system error that can be associated with the failure in the DLPI request.

**Description** All functions described in this manpage take a pointer to the message passed to the DLPI provider (*mblk\_t*) and attempt to reuse it in formulating the M\_PROTO reply. If the message block is too small to be reused, it is freed and a new one is allocated.

All functions reply upstream using [qreply\(9F\)](#). The write-side queue pointer must be provided.



The `dlokack()` function provides the successful acknowledgement `DL_OK_ACK` message reply to the DLPI provider and is used to complete many of the DLPI requests in the DLPI consumer.

The `dlerrorack()` function provides the unsuccessful acknowledgement `DL_ERROR_ACK` message reply to the DLPI provider and is used for error completions were required for DLPI requests in the DLPI consumer.

The `dlbindack()` function provides the `DL_BIND_ACK` message reply to the DLPI provider and is used to complete the `DL_BIND_REQ` processing in the DLPI consumer.

The `dlphysaddrack()` function provides the `DL_PHYS_ADDR_ACK` message reply used to complete the `DL_PHYS_ADDR_ACK` processing.

The `dluderrorind()` function provides the `DL_UDERROR_IND` message reply used to complete an unsuccessful `DL_UNITDATA_REQ`.

**Return Values** None.

**Notes** These functions are not required if you are writing a DLPI device driver using [gld\(7D\)](#).

**Context** All DLPI helper functions can be called from user, interrupt, or kernel context.

**See Also** [gld\(7D\)](#), [dlpi\(7P\)](#), [qreply\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** drv\_getparm – retrieve kernel state information

**Synopsis** #include <sys/ddi.h>

```
int drv_getparm(unsigned int parm, void *value_p);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *parm* The kernel parameter to be obtained. Possible values are:

**LBOLT** Read the value of `lbolt`. `lbolt` is a `clock_t` that represents the number of clock ticks since system boot. No special treatment is applied when this value overflows the maximum value of the signed integral type `clock_t`. When this occurs, its value will be negative and its magnitude will be decreasing until it again passes zero. It can therefore not be relied upon to provide an indication of the amount of time that passes since the last system reboot, nor should it be used to mark an absolute time in the system. Only the difference between two measurements of `lbolt` is significant. It is used in this way inside the system kernel for timing purposes.

**PPGRP** Read the process group identification number. This number determines which processes should receive a HANGUP or BREAK signal when detected by a driver.

**UPROCP** Read the process table token value.

**PPID** Read process identification number.

**PSID** Read process session identification number.

**TIME** Read time in seconds.

**UCRED** Return a pointer to the caller's credential structure.

*value\_p* A pointer to the data space in which the value of the parameter is to be copied.

**Description** Since the release of the Solaris 2.6 operating environment, the `drv_getparm()` function has been replaced by `ddi_get_lbolt(9F)`, `ddi_get_time(9F)`, and `ddi_get_pid(9F)`.

The `drv_getparm()` function verifies that *parm* corresponds to a kernel parameter that may be read. If the value of *parm* does not correspond to a parameter or corresponds to a parameter that may not be read, -1 is returned. Otherwise, the value of the parameter is stored in the data space pointed to by *value\_p*.

The `drv_getparm()` function does not explicitly check to see whether the device has the appropriate context when the function is called and the function does not check for correct alignment in the data space pointed to by *value\_p*. It is the responsibility of the driver writer to use this function only when it is appropriate to do so and to correctly declare the data space needed by the driver.

**Return Values** The `drv_getparm()` function returns `0` to indicate success, `-1` to indicate failure. The value stored in the space pointed to by *value\_p* is the value of the parameter if `0` is returned, or undefined if `-1` is returned. `-1` is returned if you specify a value other than `LBOLT`, `PPGRP`, `PPID`, `PSID`, `TIME`, `UCRED`, or `UPROCP`. Always check the return code when using this function.

**Context** The `drv_getparm()` function can be called from user context only when using `PPGRP`, `PPID`, `PSID`, `UCRED`, or `UPROCP`. It can be called from user, interrupt, or kernel context when using the `LBOLT` or `TIME` argument.

**See Also** [ddi\\_get\\_lbolt\(9F\)](#), [ddi\\_get\\_pid\(9F\)](#), [ddi\\_get\\_time\(9F\)](#), [buf\(9S\)](#)

*Writing Device Drivers*

**Name** drv\_hztousec – convert clock ticks to microseconds

**Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>
```

```
clock_t drv_hztousec(clock_t hertz);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *hertz* The number of clock ticks to convert.

**Description** The `drv_hztousec()` function converts into microseconds the time expressed by *hertz*, which is in system clock ticks.

The length of time the system has been up since boot can be retrieved by calling [ddi\\_get\\_lbolt\(9F\)](#), which will return a value of type `clock_t` containing the number of clock ticks since boot. Drivers often use the value of `lbolt` before and after an I/O request to measure the amount of time it took the device to process the request. The `drv_hztousec()` function can be used by the driver to convert the reading from clock ticks to a known unit of time.

**Return Values** The number of microseconds equivalent to the *hertz* parameter. No error value is returned. If the microsecond equivalent to *hertz* is too large to be represented as a `clock_t`, then the maximum `clock_t` value will be returned.

**Context** The `drv_hztousec()` function can be called from user, interrupt, or kernel context.

**See Also** [ddi\\_get\\_lbolt\(9F\)](#), [drv\\_usectohz\(9F\)](#), [drv\\_usecwait\(9F\)](#)

*Writing Device Drivers*

- 
- Name** `drv_priv` – determine driver privilege
- Synopsis**
- ```
#include <sys/types.h>
#include <sys/cred.h>
#include <sys/ddi.h>

int drv_priv(cred_t *cr);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** `cr` Pointer to the user credential structure.
- Description** The `drv_priv()` function provides a general interface to the system privilege policy. It determines whether the credentials supplied by the user credential structure pointed to by `cr` identify a process that has the `{PRIV_SYS_DEVICES}` privilege asserted in its effective set. This function should be used only when file access modes, special minor device numbers, and the device policy (see [privileges\(5\)](#), [add\\_drv\(1M\)](#)) are insufficient to provide protection for the requested driver function. It is intended to replace all calls to `suser()` and any explicit checks for effective user ID = 0 in driver code.
- Return Values** This routine returns 0 if it succeeds, EPERM if it fails.
- Context** The `drv_priv()` function can be called from user, interrupt, or kernel context.
- See Also** [add\\_drv\(1M\)](#), [update\\_drv\(1M\)](#), [privileges\(5\)](#)
- Writing Device Drivers*

**Name** drv\_usectohz – convert microseconds to clock ticks

**Synopsis** #include <sys/types.h>  
#include <sys/ddi.h>

```
clock_t drv_usectohz(clock_t microsecs);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *microsecs* The number of microseconds to convert.

**Description** The `drv_usectohz()` function converts a length of time expressed in microseconds to a number of system clock ticks. The time arguments to [timeout\(9F\)](#) and [delay\(9F\)](#) are expressed in clock ticks.

The `drv_usectohz()` function is a portable interface for drivers to make calls to [timeout\(9F\)](#) and [delay\(9F\)](#) and remain binary compatible should the driver object file be used on a system with a different clock speed (a different number of ticks in a second).

**Return Values** The value returned is the number of system clock ticks equivalent to the *microsecs* argument. No error value is returned. If the clock tick equivalent to *microsecs* is too large to be represented as a `clock_t`, then the maximum `clock_t` value will be returned.

**Context** The `drv_usectohz()` function can be called from user, interrupt, or kernel context.

**See Also** [delay\(9F\)](#), [drv\\_hztousec\(9F\)](#), [timeout\(9F\)](#)

*Writing Device Drivers*

**Notes** If the *microsecs* argument to `drv_usectohz()` is less than [drv\\_hztousec\(9F\)](#), `drv_usectohz()` returns one tick. This, coupled with multiplication, can result in significantly longer durations than expected. For example, on a machine where `hz` is 100, calling `drv_usectohz()` with a *microsecs* value less than 10000 returns a result equivalent to 10000 (1 tick). This type of mistake causes code such as “`5000 * drv_usectohz(1000)`” to compute a duration of 50 seconds instead of the intended 5 seconds.

- 
- Name** drv\_usecwait – busy-wait for specified interval
- Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>

void drv_usecwait(clock_t microsecs);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *microsecs* The number of microseconds to busy-wait.
- Description** The `drv_usecwait()` function gives drivers a means of busy-waiting for a specified microsecond count. The amount of time spent busy-waiting may be greater than the microsecond count but will minimally be the number of microseconds specified.
- [delay\(9F\)](#) can be used by a driver to delay for a specified number of system ticks, but it has two limitations. First, the granularity of the wait time is limited to one clock tick, which may be more time than is needed for the delay. Second, [delay\(9F\)](#) can be invoked from user or kernel context and hence cannot be used at interrupt time or system initialization.
- Often, drivers need to delay for only a few microseconds, waiting for a write to a device register to be picked up by the device. In this case, even in user context, [delay\(9F\)](#) produces too long a wait period.
- Context** The `drv_usecwait()` function can be called from user, interrupt, or kernel context.
- See Also** [delay\(9F\)](#), [timeout\(9F\)](#), [untimeout\(9F\)](#)
- Writing Device Drivers*
- Notes** The driver wastes processor time by making this call since `drv_usecwait()` does not block but simply busy-waits. The driver should only make calls to `drv_usecwait()` as needed, and only for as much time as needed. The `drv_usecwait()` function does not mask out interrupts.

**Name** dupb – duplicate a message block descriptor

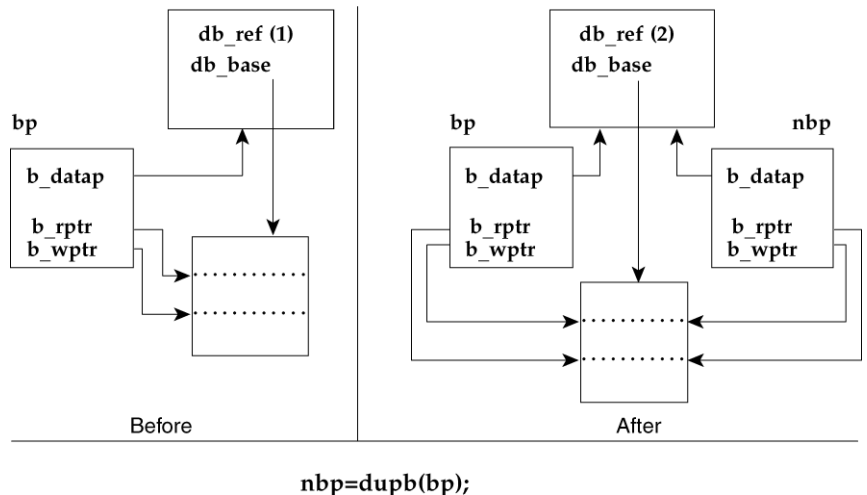
**Synopsis** #include <sys/stream.h>

```
mbblk_t *dupb(mblk_t *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Description** The dupb() function creates a new mblk\_t structure (see [msgb\(9S\)](#)) to reference the message block pointed to by bp.

Unlike [copyb\(9F\)](#), dupb() does not copy the information in the dbblk\_t structure (see [datab\(9S\)](#)), but creates a new mblk\_t structure to point to it. The reference count in the dbblk\_t structure (db\_ref) is incremented. The new mblk\_t structure contains the same information as the original. Note that b\_rptr and b\_wptr are copied from the bp.



**Parameters** bp Pointer to the message block to be duplicated. mblk\_t is an instance of the [msgb\(9S\)](#) structure.

**Return Values** If successful, dupb() returns a pointer to the new message block. A NULL pointer is returned if dupb() cannot allocate a new message block descriptor or if the db\_ref field of the data block structure (see [datab\(9S\)](#)) has reached a maximum value (255).

**Context** dupb() can be called from user, kernel, or interrupt context.



**Examples** EXAMPLE 1 Using dupb()

This `srv(9E)` (service) routine adds a header to all `M_DATA` messages before passing them along. `dupb` is used instead of `copyb(9F)` because the contents of the header block are not changed.

For each message on the queue, if it is a priority message, pass it along immediately (lines 10–11). Otherwise, if it is anything other than an `M_DATA` message (line 12), and if it can be sent along (line 13), then do so (line 14). Otherwise, put the message back on the queue and return (lines 16–17). For all `M_DATA` messages, first check to see if the stream is flow-controlled (line 20). If it is, put the message back on the queue and return (lines 37–38). If it is not, the header block is duplicated (line 21).

`dupb()` can fail either due to lack of resources or because the message block has already been duplicated 255 times. In order to handle the latter case, the example calls `copyb(9F)` (line 22). If `copyb(9F)` fails, it is due to buffer allocation failure. In this case, `qbufcall(9F)` is used to initiate a callback (lines 30–31) if one is not already pending (lines 26–27).

The callback function, `xxxcallback()`, clears the recorded `qbufcall(9F)` callback id and schedules the service procedure (lines 49–50). Note that the close routine, `xxxclose()`, must cancel any outstanding `qbufcall(9F)` callback requests (lines 58–59).

If `dupb()` or `copyb(9F)` succeed, link the `M_DATA` message to the new message block (line 34) and pass it along (line 35).

```

1  xxxsrv(q)
2      queue_t *q;
3  {
4      struct xx *xx = (struct xx *)q->q_ptr;
5      mblk_t *mp;
6      mblk_t *bp;
7      extern mblk_t *hdr;
8
9      while ((mp = getq(q)) != NULL) {
10         if (mp->b_datap->db_type >= QPCTL) {
11             putnext(q, mp);
12         } else if (mp->b_datap->db_type != M_DATA) {
13             if (canputnext(q))
14                 putnext(q, mp);
15             else {
16                 putbq(q, mp);
17                 return;
18             }
19         } else { /* M_DATA */
20             if (canputnext(q)) {
21                 if ((bp = dupb(hdr)) == NULL)
22                     bp = copyb(hdr);
23                 if (bp == NULL) {

```

## EXAMPLE 1 Using dupb() (Continued)

```

24             size_t size = msgdsize(mp);
25             putbq(q, mp);
26             if (xx->xx_qbufcall_id) {
27                 /* qbufcall pending */
28                 return;
29             }
30             xx->xx_qbufcall_id = qbufcall(q, size,
31                 BPRI_MED, xxxcallback, (intptr_t)q);
32             return;
33         }
34         linkb(bp, mp);
35         putnext(q, bp);
36     } else {
37         putbq(q, mp);
38         return;
39     }
40 }
41 }
42 }
43 void
44 xxxcallback(q)
45     queue_t *q;
46 {
47     struct xx *xx = (struct xx *)q->q_ptr;
48
49     xx->xx_qbufcall_id = 0;
50     qenable(q);
51 }
52 xxxclose(q, cflag, crp)
53     queue_t *q;
54     int cflag;
55     cred_t *crp;
56 {
57     struct xx *xx = (struct xx *)q->q_ptr;
58     ...
59     if (xx->xx_qbufcall_id)
60         qunbufcall(q, xx->xx_qbufcall_id);
61     ...
62 }

```

**See Also** [srv\(9E\)](#), [copyb\(9F\)](#), [qbufcall\(9F\)](#), [datab\(9S\)](#), [msgb\(9S\)](#)

*Writing Device Drivers STREAMS Programming Guide*

- 
- Name** dupmsg – duplicate a message
- Synopsis** `#include <sys/stream.h>`
- ```
mbblk_t *dupmsg(mblk_t *mp);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *mp* Pointer to the message.
- Description** dupmsg ( ) forms a new message by copying the message block descriptors pointed to by *mp* and linking them. dupb(9F) is called for each message block. The data blocks themselves are not duplicated.
- Return Values** If successful, dupmsg ( ) returns a pointer to the new message block. Otherwise, it returns a NULL pointer. A return value of NULL indicates either memory depletion or the data block reference count, db\_ref (see datab(9S)), has reached a limit (255). See dupb(9F).
- Context** dupmsg ( ) can be called from user, kernel, or interrupt context.
- Examples** EXAMPLE 1 Using dupmsg ( )  
See copyb(9F) for an example using dupmsg ( ).
- See Also** copyb(9F), copymsg(9F), dupb(9F), datab(9S)  
*Writing Device Drivers*  
*STREAMS Programming Guide*

**Name** enableok – reschedule a queue for service

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

void enableok(queue_t *q);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* A pointer to the queue to be rescheduled.

**Description** The `enableok()` function enables queue *q* to be rescheduled for service. It reverses the effect of a previous call to [noenable\(9F\)](#) on *q* by turning off the QNOENB flag in the queue.

**Context** The `enableok()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `enableok()`

The `qrestart()` routine uses two STREAMS functions to restart a queue that has been disabled. The `enableok()` function turns off the QNOENB flag, allowing the [qenable\(9F\)](#) to schedule the queue for immediate processing.

```
1 void
2 qrestart(rdwr_q)
3     register queue_t *rdwr_q;
4 {
5     enableok(rdwr_q);
6     /* re-enable a queue that has been disabled */
7     (void) qenable(rdwr_q);
8 }
```

**See Also** [noenable\(9F\)](#), [qenable\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** esballoc, desballoc – allocate a message block using a caller-supplied buffer

**Synopsis** #include <sys/stream.h>

```
mblk_t *esballoc(uchar_t *base, size_t size, uint_t pri,
                frtn_t *fr_rtnp);

mblk_t *desballoc(uchar_t *base, size_t size, uint_t pri,
                 frtn_t *fr_rtnp);
```

**Interface Level** esballoc(): Architecture independent level 1 (DDI/DKI)

desballoc(): Solaris DDI specific (Solaris DDI)

**Parameters** *base* Address of caller-supplied data buffer.  
*size* Number of bytes in data buffer.  
*pri* Priority of the request (no longer used).  
*fr\_rtnp* Free routine data structure.

**Description** The esballoc() and desballoc() functions operate identically to [allobcb\(9F\)](#), except that the data buffer to associate with the message is specified by the caller. The allocated message will have both the `b_wptr` and `b_rptr` set to the supplied data buffer starting at *base*. Only the buffer itself can be specified by the caller. The message block and data block header are allocated as if by [allobcb\(9F\)](#).

When [freeb\(9F\)](#) is called to free the message, the driver's message-freeing routine, referenced through the [free\\_rtn\(9S\)](#) structure, is called with appropriate arguments to free the data buffer.

The [free\\_rtn\(9S\)](#) structure includes the following members:

```
void (*free_func)();      /* caller's freeing routine */
caddr_t free_arg;        /* argument to free_func() */
```

Instead of requiring a specific number of arguments, the `free_arg` field is defined of type `caddr_t`. This way, the driver can pass a pointer to a structure if more than one argument is needed.

If esballoc() was used, then `free_func` will be called asynchronously at some point after the message is no longer referenced. If desballoc() was used, then `free_func` will be called synchronously by the thread releasing the final reference. See [freeb\(9F\)](#).

The `free_func` routine must not sleep, and must not access any dynamically allocated data structures that could be freed before or during its execution. In addition, because messages allocated with desballoc() are freed in the context of the caller, `free_func` must not call another module's put procedure, or attempt to acquire a private module lock which might be held by another thread across a call to a STREAMS utility routine that could free a message

block. Finally, `free_func` routines specified using `desballoc` may run in interrupt context and thus must only use synchronization primitives that include an interrupt priority returned from `ddi_intr_get_pri(9F)` or `ddi_intr_get_softint_pri(9F)`. If any of these restrictions are not followed, the possibility of lock recursion or deadlock exists.

**Return Values** On success, a pointer to the newly allocated message block is returned. On failure, `NULL` is returned.

**Context** The `esballoc()` and `desballoc()` functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [allocb\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_intr\\_get\\_softint\\_pri\(9F\)](#), [freeb\(9F\)](#), [datab\(9S\)](#), [free\\_rtn\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** esbbscall – call function when buffer is available
- Synopsis** `#include <sys/stream.h>`
- ```
bufcall_id_t esbbscall(uint_t pri, void (*func)(void *arg),
                      void(arg));
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters**
- pri* Priority of allocation request (to be used by [allocb\(9F\)](#) function, called by `esbbscall()`).
  - func* Function to be called when buffer becomes available.
  - arg* Argument to *func*.
- Description** The `esbbscall()` function, like [bufcall\(9F\)](#), serves as a [timeout\(9F\)](#) call of indeterminate length. If [esbbsalloc\(9F\)](#) is unable to allocate a message and data block header to go with its externally supplied data buffer, `esbbscall()` can be used to schedule the routine *func*, to be called with the argument *arg* when a buffer becomes available. The *func* argument can be a routine that calls [esbbsalloc\(9F\)](#) or it may be another kernel function.
- Return Values** On success, a `bufcall` ID is returned. On failure, `0` is returned. The value returned from a successful call should be saved for possible future use with `unbufcall()` should it become necessary to cancel the `esbbscall()` request (as at driver close time).
- Context** The `esbbscall()` function can be called from user, interrupt, or kernel context.
- See Also** [allocb\(9F\)](#), [bufcall\(9F\)](#), [esbbsalloc\(9F\)](#), [timeout\(9F\)](#), [datab\(9S\)](#), [unbufcall\(9F\)](#)
- Writing Device Drivers STREAMS Programming Guide*

**Name** flushband – flush messages for a specified priority band

**Synopsis** #include <sys/stream.h>

```
void flushband(queue_t *q, unsigned char pri, int flag);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue.

*pri* Priority of messages to be flushed.

*flag* Valid *flag* values are:

FLUSHDATA Flush only data messages (types M\_DATA, M\_DELAY, M\_PROTO, and M\_PCPROTO).

FLUSHALL Flush all messages.

**Description** The flushband() function flushes messages associated with the priority band specified by *pri*. If *pri* is 0, only normal and high priority messages are flushed. Otherwise, messages are flushed from the band *pri* according to the value of *flag*.

**Context** The flushband() function can be called from user, interrupt, or kernel context.

**See Also** flushq(9F)

*Writing Device Drivers STREAMS Programming Guide*



**Name** flushq – remove messages from a queue

**Synopsis** #include <sys/stream.h>

```
void flushq(queue_t *q, int flag);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue to be flushed.

*flag* Valid *flag* values are:

FLUSHDATA Flush only data messages (types M\_DATA M\_DELAY M\_PROTO and M\_PCPROTO).

FLUSHALL Flush all messages.

**Description** The flushq() function frees messages and their associated data structures by calling freemsg(9F). If the queue's count falls below the low water mark and the queue was blocking an upstream service procedure, the nearest upstream service procedure is enabled.

**Context** The flushq() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using flushq()

This example depicts the canonical flushing code for STREAMS modules. The module has a write service procedure and potentially has messages on the queue. If it receives an M\_FLUSH message, and if the FLUSHR bit is on in the first byte of the message (line 10), then the read queue is flushed (line 11). If the FLUSHW bit is on (line 12), then the write queue is flushed (line 13). Then the message is passed along to the next entity in the stream (line 14). See the example for qreply(9F) for the canonical flushing code for drivers.

```

1  /*
2  * Module write-side put procedure.
3  */
4  xxxwput(q, mp)
5      queue_t *q;
6      mblk_t *mp;
7  {
8      switch(mp->b_datap->db_type) {
9          case M_FLUSH:
10             if (*mp->b_rptr & FLUSHR)
11                 flushq(RD(q), FLUSHALL);
12             if (*mp->b_rptr & FLUSHW)
13                 flushq(q, FLUSHALL);
14             putnext(q, mp);
15             break;
16             . . .
16     }
17 }
```

**See Also** [flushband\(9F\)](#), [freemsg\(9F\)](#), [putq\(9F\)](#), [qreply\(9F\)](#)

*Writing Device Drivers STREAMS Programming Guide*

- 
- Name** freeb – free a message block
- Synopsis** `#include <sys/stream.h>`
- ```
void freeb(mblk_t *bp);
```
- Parameters** *bp* Pointer to the message block to be deallocated. `mblk_t` is an instance of the [msgb\(9S\)](#) structure.
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Description** The `freeb()` function deallocates a message block. If the reference count of the `db_ref` member of the [datab\(9S\)](#) structure is greater than 1, `freeb()` decrements the count. If `db_ref` equals 1, it deallocates the message block and the corresponding data block and buffer.
- If the data buffer to be freed was allocated with the [esballoc\(9F\)](#), the buffer may be a non-STREAMS resource. In that case, the driver must be notified that the attached data buffer needs to be freed, and run its own freeing routine. To make this process independent of the driver used in the stream, `freeb()` finds the [free\\_rtn\(9S\)](#) structure associated with the buffer. The `free_rtn` structure contains a pointer to the driver-dependent routine, which releases the buffer. Once this is accomplished, `freeb()` releases the STREAMS resources associated with the buffer.
- Context** The `freeb()` function can be called from user, interrupt, or kernel context.
- Examples** **EXAMPLE 1** Using `freeb()`
- See [copyb\(9F\)](#) for an example of using `freeb()`.
- See Also** [allocb\(9F\)](#), [copyb\(9F\)](#), [dupb\(9F\)](#), [esballoc\(9F\)](#), [free\\_rtn\(9S\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*

**Name** freemsg – free all message blocks in a message

**Synopsis** #include <sys/stream.h>

```
void freemsg(mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the message blocks to be deallocated. *mblk\_t* is an instance of the [msgb\(9S\)](#) structure. If *mp* is NULL, `freemsg()` immediately returns.

**Description** The `freemsg()` function calls [freeb\(9F\)](#) to free all message and data blocks associated with the message pointed to by *mp*.

**Context** The `freemsg()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `freemsg()`

See [copymsg\(9F\)](#).

**See Also** [copymsg\(9F\)](#), [freeb\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The behavior of `freemsg()` when passed a NULL pointer is Solaris-specific.

**Name** freerbuf – free a raw buffer header

**Synopsis**

```
#include <sys/buf.h>
#include <sys/ddi.h>
```

```
void freerbuf(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to a previously allocated buffer header structure.

**Description** The `freerbuf()` function frees a raw buffer header previously allocated by [getrbuf\(9F\)](#). This function does not sleep and so may be called from an interrupt routine.

**Context** The `freerbuf()` function can be called from user, interrupt, or kernel context.

**See Also** [getrbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [kmem\\_free\(9F\)](#), [kmem\\_zalloc\(9F\)](#)

**Name** freezestr, unfreezestr – freeze, thaw the state of a stream

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

void freezestr(queue_t *q);
void unfreezestr(queue_t *q);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the message queue to freeze/unfreeze.

**Description** `freezestr()` freezes the state of the entire stream containing the queue pair *q*. A frozen stream blocks any thread attempting to enter any open, close, put or service routine belonging to any queue instance in the stream, and blocks any thread currently within the stream if it attempts to put messages onto or take messages off of any queue within the stream (with the sole exception of the caller). Threads blocked by this mechanism remain so until the stream is thawed by a call to `unfreezestr()`.

Drivers and modules must freeze the stream before manipulating the queues directly (as opposed to manipulating them through programmatic interfaces such as [getq\(9F\)](#), [putq\(9F\)](#), [putbq\(9F\)](#), etc.)

**Context** These routines may be called from any stream open, close, put or service routine as well as interrupt handlers, callouts and call-backs.

**See Also** *Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The `freezestr()` and `unfreezestr()` functions can have a serious impact on system performance. Their use should be very limited. In most cases, there is no need to use `freezestr()` and there are usually better ways to accomplish what you need to do than by freezing the stream.

Calling `freezestr()` to freeze a stream that is already frozen by the caller will result in a single-party deadlock.

The caller of `unfreezestr()` must be the thread who called `freezestr()`.

STREAMS utility functions such as [getq\(9F\)](#), [putq\(9F\)](#), [putbq\(9F\)](#), and so forth, should not be called by the caller of `freezestr()` while the stream is still frozen, as they indirectly freeze the stream to ensure atomicity of queue manipulation.

**Name** geterror – return I/O error

**Synopsis**

```
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/ddi.h>
```

```
int geterror(struct buf *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *bp* Pointer to a [buf\(9S\)](#) structure.

**Description** The `geterror()` function returns the error number from the error field of the buffer header structure.

**Return Values** An error number indicating the error condition of the I/O request is returned. If the I/O request completes successfully, 0 is returned.

**Context** The `geterror()` function can be called from user, interrupt, or kernel context.

**See Also** [buf\(9S\)](#)

*Writing Device Drivers*

**Name** gethrtime – get high resolution time

**Synopsis** `#include <sys/time.h>`

```
hrtime_t gethrtime(void);
```

**Description** The `gethrtime()` function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of [adjtime\(2\)](#) or [settimeofday\(3C\)](#). The hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.

**Return Values** `gethrtime()` always returns the current high-resolution real time. There are no error conditions.

**Context** There are no restrictions on the context from which `gethrtime()` can be called.

**See Also** [proc\(1\)](#), [gettimeofday\(3C\)](#), [settimeofday\(3C\)](#), [attributes\(5\)](#)

**Notes** Although the units of hi-res time are always the same (nanoseconds), the actual resolution is hardware dependent. Hi-res time is guaranteed to be monotonic (it does not go backward, it does not periodically wrap) and linear (it does not occasionally speed up or slow down for adjustment, as the time of day can), but not necessarily unique: two sufficiently proximate calls might return the same value.

The time base used for this function is the same as that for [gethrtime\(3C\)](#). Values returned by both of these functions can be interleaved for comparison purposes.



**Name** getmajor – get major device number

**Synopsis** `#include <sys/types.h>`  
`#include <sys/mkdev.h>`  
`#include <sys/ddi.h>`

```
major_t getmajor(dev_t dev);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *dev* Device number.

**Description** The `getmajor()` function extracts the major number from a device number.

**Return Values** The major number.

**Context** The `getmajor()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `getmajor()`

The following example shows both the `getmajor()` and [getminor\(9F\)](#) functions used in a `debug cmn_err(9F)` statement to return the major and minor numbers for the device supported by the driver.

```
dev_t dev;

#ifdef DEBUG
cmn_err(CE_NOTE, "Driver Started. Major# = %d,
           Minor# = %d", getmajor(dev), getminor(dev));
#endif
```

**See Also** [cmn\\_err\(9F\)](#), [getminor\(9F\)](#), [makedevice\(9F\)](#)

*Writing Device Drivers*

**Warnings** No validity checking is performed. If *dev* is invalid, an invalid number is returned.

**Name** getminor – get minor device number

**Synopsis**

```
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>
```

```
minor_t getminor(dev_t dev);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *dev* Device number.

**Description** The `getminor()` function extracts the minor number from a device number.

**Return Values** The minor number.

**Context** The `getminor()` function can be called from user, interrupt, or kernel context.

**Examples** See the [getmajor\(9F\)](#) manual page for an example of how to use `getminor()`.

**See Also** [getmajor\(9F\)](#), [makedevice\(9F\)](#)

*Writing Device Drivers*

**Warnings** No validity checking is performed. If *dev* is invalid, an invalid number is returned.

---

<b>Name</b>	get_pktiopb, free_pktiopb – allocate/free a SCSI packet in the iopb map																
<b>Synopsis</b>	<pre>#include &lt;sys/scsi/scsi.h&gt;  struct scsi_pkt *get_pktiopb(struct scsi_address *ap,     caddr_t *datap, int cdblen, int statuslen, int datalen,     int readflag, int (*callback));  void free_pktiopb(struct scsi_pkt *pkt, caddr_t datap, int datalen);</pre>																
<b>Interface Level</b>	These interfaces are obsolete. Use <a href="#">scsi_alloc_consistent_buf(9F)</a> instead of get_pktiopb(). Use <a href="#">scsi_free_consistent_buf(9F)</a> instead of free_pktiopb().																
<b>Parameters</b>	<table> <tr> <td><i>ap</i></td> <td>Pointer to the target's scsi_address structure.</td> </tr> <tr> <td><i>datap</i></td> <td>Pointer to the address of the packet, set by this function.</td> </tr> <tr> <td><i>cdblen</i></td> <td>Number of bytes required for the SCSI command descriptor block (CDB).</td> </tr> <tr> <td><i>statuslen</i></td> <td>Number of bytes required for the SCSI status area.</td> </tr> <tr> <td><i>datalen</i></td> <td>Number of bytes required for the data area of the SCSI command.</td> </tr> <tr> <td><i>readflag</i></td> <td>If non-zero, data will be transferred from the SCSI target.</td> </tr> <tr> <td><i>callback</i></td> <td>Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC</td> </tr> <tr> <td><i>pkt</i></td> <td>Pointer to a <a href="#">scsi_pkt(9S)</a> structure.</td> </tr> </table>	<i>ap</i>	Pointer to the target's scsi_address structure.	<i>datap</i>	Pointer to the address of the packet, set by this function.	<i>cdblen</i>	Number of bytes required for the SCSI command descriptor block (CDB).	<i>statuslen</i>	Number of bytes required for the SCSI status area.	<i>datalen</i>	Number of bytes required for the data area of the SCSI command.	<i>readflag</i>	If non-zero, data will be transferred from the SCSI target.	<i>callback</i>	Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC	<i>pkt</i>	Pointer to a <a href="#">scsi_pkt(9S)</a> structure.
<i>ap</i>	Pointer to the target's scsi_address structure.																
<i>datap</i>	Pointer to the address of the packet, set by this function.																
<i>cdblen</i>	Number of bytes required for the SCSI command descriptor block (CDB).																
<i>statuslen</i>	Number of bytes required for the SCSI status area.																
<i>datalen</i>	Number of bytes required for the data area of the SCSI command.																
<i>readflag</i>	If non-zero, data will be transferred from the SCSI target.																
<i>callback</i>	Pointer to a callback function, or NULL_FUNC or SLEEP_FUNC																
<i>pkt</i>	Pointer to a <a href="#">scsi_pkt(9S)</a> structure.																
<b>Description</b>	<p>The get_pktiopb() function allocates a scsi_pkt structure that has a small data area allocated. It is used by some SCSI commands such as REQUEST_SENSE, which involve a small amount of data and require cache-consistent memory for proper operation. It uses <a href="#">ddi_iopb_alloc(9F)</a> for allocating the data area and <a href="#">scsi_realloc(9F)</a> to allocate the packet and DMA resources.</p> <p><i>callback</i> indicates what get_pktiopb() should do when resources are not available:</p> <table> <tr> <td>NULL_FUNC</td> <td>Do not wait for resources. Return a NULL pointer.</td> </tr> <tr> <td>SLEEP_FUNC</td> <td>Wait indefinitely for resources.</td> </tr> </table> <p>Other Values <i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> must return either 0 (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.</p> <p>The free_pktiopb() function is used for freeing the packet and its associated resources.</p>	NULL_FUNC	Do not wait for resources. Return a NULL pointer.	SLEEP_FUNC	Wait indefinitely for resources.												
NULL_FUNC	Do not wait for resources. Return a NULL pointer.																
SLEEP_FUNC	Wait indefinitely for resources.																
<b>Return Values</b>	The get_pktiopb() function returns a pointer to the newly allocated scsi_pkt or a NULL pointer.																

**Context** If *callback* is SLEEP\_FUNC, then this routine can be called only from user or kernel context. Otherwise, it can be called from user, interrupt, or kernel context. The *callback* function should not block or call routines that block.

The free\_pktiopb() function can be called from user, interrupt, or kernel context.

**Attributes** See attributes(5) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**See Also** attributes(5), ddi\_iopb\_alloc(9F), scsi\_alloc\_consistent\_buf(9F), scsi\_free\_consistent\_buf(9F), scsi\_pktalloc(9F), scsi\_realloc(9F), scsi\_pkt(9S)

*Writing Device Drivers*

**Notes** The get\_pktiopb() and free\_pktiopb() functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, scsi\_alloc\_consistent\_buf(9F) and scsi\_free\_consistent\_buf(9F).

The get\_pktiopb() function uses scarce resources. For this reason and its obsolescence (see above), its use is discouraged.

- 
- Name** getq – get the next message from a queue
- Synopsis** `#include <sys/stream.h>`
- ```
mblk_t *getq(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the queue from which the message is to be retrieved.
- Description** The `getq()` function is used by a service ([srv\(9E\)](#)) routine to retrieve its enqueued messages.
- A module or driver may include a service routine to process enqueued messages. Once the STREAMS scheduler calls `srv()` it must process all enqueued messages, unless prevented by flow control. `getq()` obtains the next available message from the top of the queue pointed to by *q*. It should be called in a `while` loop that is exited only when there are no more messages or flow control prevents further processing.
- If an attempt was made to write to the queue while it was blocked by flow control, `getq()` back-enables (restarts) the service routine once it falls below the low water mark.
- Return Values** If there is a message to retrieve, `getq()` returns a pointer to it. If no message is queued, `getq()` returns a NULL pointer.
- Context** The `getq()` function can be called from user, interrupt, or kernel context.
- Examples** See [dupb\(9F\)](#).
- See Also** [srv\(9E\)](#), [bcanput\(9F\)](#), [canput\(9F\)](#), [dupb\(9F\)](#), [putbq\(9F\)](#), [putq\(9F\)](#), [qenable\(9F\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*

**Name** getrbuf – get a raw buffer header

**Synopsis**

```
#include <sys/buf.h>
#include <sys/kmem.h>
#include <sys/ddi.h>

struct buf *getrbuf(int sleepflag);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *sleepflag* Indicates whether driver should sleep for free space.

**Description** The `getrbuf()` function allocates the space for a buffer header to the caller. It is used in cases where a block driver is performing raw (character interface) I/O and needs to set up a buffer header that is not associated with the buffer cache.

The `getrbuf()` function calls [kmem\\_alloc\(9F\)](#) to perform the memory allocation. `kmem_alloc()` requires the information included in the *sleepflag* argument. If *sleepflag* is set to `KM_SLEEP`, the driver may sleep until the space is freed up. If *sleepflag* is set to `KM_NOSLEEP`, the driver will not sleep. In either case, a pointer to the allocated space is returned or `NULL` to indicate that no space was available.

**Return Values** The `getrbuf()` function returns a pointer to the allocated buffer header, or `NULL` if no space is available.

**Context** The `getrbuf()` function can be called from user, interrupt, or kernel context. (Drivers must not allow `getrbuf()` to sleep if called from an interrupt routine.)

**See Also** [bioinit\(9F\)](#), [freerbuf\(9F\)](#), [kmem\\_alloc\(9F\)](#), [kmem\\_free\(9F\)](#)

*Writing Device Drivers*

**Name** gld, gld\_mac\_alloc, gld\_mac\_free, gld\_register, gld\_unregister, gld\_recv, gld\_sched, gld\_intr  
– Generic LAN Driver service routines

**Synopsis** #include <sys/gld.h>

```
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
void gld_mac_free(gld_mac_info_t *macinfo);
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
int gld_unregister(gld_mac_info_t *macinfo);
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
void gld_sched(gld_mac_info_t *macinfo);
uint_t gld_intr(caddr_t);
void gld_linkstate(gld_mac_info_t *macinfo, int32_t newstate);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters**

|                 |                                                          |
|-----------------|----------------------------------------------------------|
| <i>macinfo</i>  | Pointer to a <a href="#">gld_mac_info(9S)</a> structure. |
| <i>dip</i>      | Pointer to dev_info structure.                           |
| <i>name</i>     | Device interface name.                                   |
| <i>mp</i>       | Pointer to a message block containing a received packet. |
| <i>newstate</i> | Media link state.                                        |

**Description** `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 may 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()`.

`gld_mac_free()` frees a [gld\\_mac\\_info\(9S\)](#) structure previously allocated by `gld_mac_alloc()`.

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

- links the device-specific driver with the GLD system;
- sets the device-specific driver's private data pointer (using [ddi\\_set\\_driver\\_private\(9F\)](#)) to point to the `macinfo` structure;
- creates the minor device node.

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

The driver's `attach(9E)` routine should return `DDI_SUCCESS` if `gld_register()` succeeds. If `gld_register()` returns `DDI_FAILURE`, the `attach(9E)` routine should deallocate any resources it allocated before calling `gld_register()` and then also return `DDI_FAILURE`.

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

- ensures 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.

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()` returns `DDI_FAILURE`, the driver's `detach(9E)` routine must leave the device operational and return `DDI_FAILURE`.

`gld_rcv()` is called by the driver's interrupt handler to pass a received packet upstream. The driver must construct and pass a `STREAMSM_DATA` message containing the raw packet. `gld_rcv()` 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_rcv()`. In particular, locks that could be taken by a transmit thread may not be held during a call to `gld_rcv()`: the interrupt thread that calls `gld_rcv()` may 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_rcv()`, this could result in a panic due to recursive mutex entry.

`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.)

`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()` may be specified as the soft interrupt handler in the call to `ddi_add_softintr()`.

`gld_linkstate()` is called by the device driver to notify GLD of changes in the media link state. The *newstate* argument should be set to one of the following:

|                                    |                                          |
|------------------------------------|------------------------------------------|
| <code>GLD_LINKSTATE_DOWN</code>    | The media link is unavailable.           |
| <code>GLD_LINKSTATE_UP</code>      | The media link is unavailable.           |
| <code>GLD_LINKSTATE_UNKNOWN</code> | The status of the media link is unknown. |

If a driver calls `gld_linkstate()`, it must also set the `GLD_CAP_LINKSTATE` bit in the `gldm_capabilities` field of the `gld_mac_info(9S)` structure.

**Return Values** `gld_mac_alloc()` returns a pointer to a new `gld_mac_info(9S)` structure.

`gld_register()` and `gld_unregister()` return:

|                          |             |
|--------------------------|-------------|
| <code>DDI_SUCCESS</code> | on success. |
| <code>DDI_FAILURE</code> | on failure. |

`gld_intr()` returns a value appropriate for an interrupt handler.

**See Also** `gld(7D)`, `gld(9E)`, `gld_mac_info(9S)`, `gld_stats(9S)`, `dlpi(7P)`, `attach(9E)`, `ddi_add_intr(9F)`.

*Writing Device Drivers*

**Name** hat\_getkpfnum – get page frame number for kernel address

**Synopsis**

```
#include <sys/types.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
pfn_t hat_getkpfnum(caddr_t addr);
```

**Interface Level** This interface is obsolete. A driver [devmap\(9E\)](#) entry point should be provided instead.

**Parameters** *addr* The kernel virtual address for which the page frame number is to be returned.

**Description** `hat_getkpfnum()` returns the page frame number corresponding to the kernel virtual address, *addr*.

*addr* must be a kernel virtual address which maps to device memory. [ddi\\_map\\_regs\(9F\)](#) can be used to obtain this address. For example, [ddi\\_map\\_regs\(9F\)](#) can be called in the driver's [attach\(9E\)](#) routine. The resulting kernel virtual address can be saved by the driver (see [ddi\\_soft\\_state\(9F\)](#)) and used in [mmap\(9E\)](#). The corresponding [ddi\\_unmap\\_regs\(9F\)](#) call can be made in the driver's [detach\(9E\)](#) routine. Refer to [mmap\(9E\)](#) for more information.

**Return Values** The page frame number corresponding to the valid, device-mapped virtual address *addr*. Otherwise the return value is undefined.

**Context** `hat_getkpfnum()` can be called only from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Obsolete        |

**See Also** [attach\(9E\)](#), [detach\(9E\)](#), [devmap\(9E\)](#), [mmap\(9E\)](#), [ddi\\_map\\_regs\(9F\)](#), [ddi\\_soft\\_state\(9F\)](#), [ddi\\_unmap\\_regs\(9F\)](#)

*Writing Device Drivers*

**Notes** For some devices, mapping device memory in the driver's [attach\(9E\)](#) routine and unmapping device memory in the driver's [detach\(9E\)](#) routine is a sizeable drain on system resources. This is especially true for devices with a large amount of physical address space. Refer to [mmap\(9E\)](#) for alternative methods.

**Name** hook\_alloc – allocate a hook\_t data structure

**Synopsis** #include <sys/hook.h>

```
hook_t *hook_alloc(const int version);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *version* must always be the symbol HOOK\_VERSION.

**Description** The hook\_alloc() function allocates a hook\_t structure, returning a pointer for the caller to use.

**Return Values** Upon success, hook\_alloc() returns a pointer to the allocated *hook\_t* structure. On failure, hook\_alloc() returns a NULL pointer.

**Context** The hook\_alloc() function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/core-os  |
| Interface Stability | Committed       |

**See Also** [hook\\_free\(9F\)](#), [hook\\_t\(9S\)](#)

**Name** hook\_free – free a hook\_t data structure

**Synopsis** #include <sys/hook.h>

```
void hook_free(hook_t * hook);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *hook* pointer returned by [hook\\_alloc\(9F\)](#).

**Description** The `hook_free()` function frees a `hook_t` structure that was originally allocated by [hook\\_alloc\(9F\)](#).

**Context** The `hook_free()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/core-os  |
| Interface Stability | Committed       |

**See Also** [hook\\_alloc\(9F\)](#), [hook\\_t\(9S\)](#)

**Name** id32\_alloc, id32\_free, id32\_lookup – 32-bit driver ID management routines

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/id32.h>

uint32_t id32_alloc(void *ptr, int flag);
void id32_free(uint32_t token);
void *id32_lookup(uint32_t token);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters**

*ptr* any valid 32- or 64-bit pointer

*flag* determines whether caller can sleep for memory (see [kmem\\_alloc\(9F\)](#) for a description)

**Description** These routines were originally developed so that device drivers could manage 64-bit pointers on devices that save space only for 32-bit pointers.

Many device drivers need to pass a 32-bit value to the hardware when attempting I/O. Later, when that I/O completes, the only way the driver has to identify the request that generated that I/O is via a "token". When the I/O is initiated, the driver passes this token to the hardware. When the I/O completes the hardware passes back this 32-bit token.

Before Solaris supported 64-bit pointers, device drivers just passed a raw 32-bit pointer to the hardware. When pointers grew to be 64 bits this was no longer possible. The `id32_*` routines were created to help drivers translate between 64-bit pointers and a 32-bit token.

Given a 32- or 64-bit pointer, the routine `id32_alloc()` allocates a 32-bit token, returning 0 if `KM_NOSLEEP` was specified and memory could not be allocated. The allocated token is passed back to `id32_lookup()` to obtain the original 32- or 64-bit pointer.

The routine `id32_free()` is used to free an allocated token. Once `id32_free()` is called, the supplied token is no longer valid.

Note that these routines have some degree of error checking. This is done so that an invalid token passed to `id32_lookup()` will not be accepted as valid. When `id32_lookup()` detects an invalid token it returns NULL. Calling routines should check for this return value so that they do not try to dereference a NULL pointer.

**Context** These functions can be called from user or interrupt context. The routine `id32_alloc()` should not be called from interrupt context when the `KM_SLEEP` flag is passed in. All other routines can be called from interrupt or kernel context.

**See Also** [kmem\\_alloc\(9F\)](#)

*Writing Device Drivers*

**Name** inb, inw, inl, repinsb, repinsw, repinsd – read from an I/O port

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

unsigned char inb(int port);

unsigned short inw(int
port);

unsigned long inl(int port);

void repinsb(int port, unsigned char *addr, int
count);

void repinsw(int port, unsigned short *addr, int
count);

void repinsd(int port, unsigned long *addr, int
count);
```

**Interface Level** The functions described here are obsolete. For the `inb()`, `inw()`, and `inl()` functions, use, respectively, [ddi\\_get8\(9F\)](#), [ddi\\_get16\(9F\)](#), and [ddi\\_get32\(9F\)](#) instead. For `repinsb()`, `repinsw()`, and `repinsl()`, use, respectively, [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_get16\(9F\)](#), and [ddi\\_rep\\_get32\(9F\)](#) instead.

**Parameters**

*port* A valid I/O port address.

*addr* The address of a buffer where the values will be stored.

*count* The number of values to be read from the I/O port.

**Description** These routines read data of various sizes from the I/O port with the address specified by *port*.

The `inb()`, `inw()`, and `inl()` functions read 8 bits, 16 bits, and 32 bits of data respectively, returning the resulting values.

The `repinsb()`, `repinsw()`, and `repinsd()` functions read multiple 8-bit, 16-bit, and 32-bit values, respectively. *count* specifies the number of values to be read. A pointer to a buffer will receive the input data; the buffer must be long enough to hold *count* values of the requested size.

**Return Values** The `inb()`, `inw()`, and `inl()` functions return the value that was read from the I/O port.

**Context** These functions may be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

---

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Architecture    | x86             |
| Stability Level | Obsolete        |

**See Also** [isa\(4\)](#), [attributes\(5\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_get8\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [ddi\\_rep\\_get8\(9F\)](#), [outb\(9F\)](#)

*Writing Device Drivers*

**Name** insq – insert a message into a queue

**Synopsis** #include <sys/stream.h>

```
int insq(queue_t *q, mblk_t *emp, mblk_t *nmp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters**

- q* Pointer to the queue containing message *emp*.
- emp* Enqueued message before which the new message is to be inserted. *mblk\_t* is an instance of the [msgb\(9S\)](#) structure.
- nmp* Message to be inserted.

**Description** The `insq()` function inserts a message into a queue. The message to be inserted, *nmp*, is placed in *q* immediately before the message *emp*. If *emp* is NULL, the new message is placed at the end of the queue. The queue class of the new message is ignored. All flow control parameters are updated. The service procedure is enabled unless QNOENB is set.

**Return Values** The `insq()` function returns 1 on success, and 0 on failure.

**Context** The `insq()` function can be called from user, interrupt, or kernel context.

**Examples** This routine illustrates the steps a transport provider may take to place expedited data ahead of normal data on a queue (assume all M\_DATA messages are converted into M\_PROTO T\_DATA\_REQ messages). Normal T\_DATA\_REQ messages are just placed on the end of the queue (line 16). However, expedited T\_EXDATA\_REQ messages are inserted before any normal messages already on the queue (line 25). If there are no normal messages on the queue, *bp* will be NULL and we fall out of the for loop (line 21). `insq` acts like [putq\(9F\)](#) in this case.

```
1 #include <sys/stream.h>
2 #include <sys/tihdr.h>
3
4 static int
5 xxxwput(queue_t *q, mblk_t *mp)
6 {
7     union T_primitives *tp;
8     mblk_t *bp;
9     union T_primitives *ntp;
10
11     switch (mp->b_datap->db_type) {
12     case M_PROTO:
13         tp = (union T_primitives *)mp->b_rptr;
14         switch (tp->type) {
15         case T_DATA_REQ:
16             putq(q, mp);
17             break;
18
```



```

19     case T_EXDATA_REQ:
20         /* Insert code here to protect queue and message block */
21         for (bp = q->q_first; bp; bp = bp->b_next) {
22             if (bp->b_datap->db_type == M_PROTO) {
23                 ntp = (union T_primitives *)bp->b_rptr;
24                 if (ntp->type != T_EXDATA_REQ)
25                     break;
26             }
27         }
28         (void)insq(q, bp, mp);
29         /* End of region that must be protected */
30         break;
31     . . .
32 }
33 }

```

When using `insq()`, you must ensure that the queue and the message block is not modified by another thread at the same time. You can achieve this either by using STREAMS functions or by implementing your own locking.

**See Also** [putq\(9F\)](#), [rmvq\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** If `emp` is non-NULL, it must point to a message on `q` or a system panic could result.

**Name** IOC\_CONVERT\_FROM – determine if there is a need to translate M\_IOCTL contents.

**Synopsis** #include <sys/stream.h>

```
uint_t IOC_CONVERT_FROM(struct iocblk *iocp);
```

**Interface Level** Solaris DDI Specific (Solaris DDI)

**Parameters** *iocp* A pointer to the M\_IOCTL control structure.

**Description** The IOC\_CONVERT\_FROM macro is used to see if the contents of the current M\_IOCTL message had its origin in a different C Language Type Model.

**Return Values** The IOC\_CONVERT\_FROM() function returns the following values:

IOC\_ILP32 This is an LP64 kernel and the M\_IOCTL originated in an ILP32 user process.

IOC\_NONE The M\_IOCTL message uses the same C Language Type Model as this calling module or driver.

**Context** The IOC\_CONVERT\_FROM() macro can be called from user, interrupt, or kernel context.

**See Also** [ddi\\_model\\_convert\\_from\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** kiconv – buffer-based code conversion function

**Synopsis** #include <sys/types.h>  
#include <sys/errno.h>  
#include <sys/sunddi.h>

```
size_t kiconv(kiconv_t cd, char **inbuf, size_t *inbytesleft,
              char **outbuf, size_t *outbytesleft, int *errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** The parameters for the `kiconv` function are as follows:

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|--------------------------------------------------------------------------------------------------|-------|-----------------------------------------------------------------------------|--------|-------------------------------------------------------------------------------------------------------------------|-------|-------------------------------------------------------------------------------|
| <i>cd</i>           | Code conversion descriptor indicating the code conversion and conversion state.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| <i>inbuf</i>        | Points to an address of a buffer containing a sequence of character bytes in fromcode codeset to be converted. After the conversion, the variable is updated to point to the byte following the last byte that was successfully used in the conversion.                                                                                                                                                                                                                                                                                                                                                                                        |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| <i>inbytesleft</i>  | As an input parameter, the number of bytes to be converted in <i>inbuf</i> . As an output parameter, the number of bytes in <i>inbuf</i> still not converted after the conversion.                                                                                                                                                                                                                                                                                                                                                                                                                                                             |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| <i>outbuf</i>       | Points to an address of a buffer where converted character bytes in tocode codeset can be saved. After the conversion, the variable is updated to point to the byte following the last byte of converted output data.                                                                                                                                                                                                                                                                                                                                                                                                                          |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| <i>outbytesleft</i> | As an input parameter, the number of available bytes at <i>outbuf</i> where converted character bytes can be saved. As an output parameter, the number of bytes still available at <i>outbuf</i> after the conversion.                                                                                                                                                                                                                                                                                                                                                                                                                         |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| <i>errno</i>        | Indicates the error when conversion is not completed or failed. The following are possible values: <table> <tr> <td>EILSEQ</td> <td>The input conversion was stopped due to an input byte that does not belong to the input codeset.</td> </tr> <tr> <td>E2BIG</td> <td>The input conversion was stopped due to lack of space in the output buffer.</td> </tr> <tr> <td>EINVAL</td> <td>The input conversion was stopped due to an incomplete character or shift sequence at the end of the input buffer.</td> </tr> <tr> <td>EBADF</td> <td>The <i>cd</i> input parameter is not a valid open code conversion descriptor.</td> </tr> </table> | EILSEQ | The input conversion was stopped due to an input byte that does not belong to the input codeset. | E2BIG | The input conversion was stopped due to lack of space in the output buffer. | EINVAL | The input conversion was stopped due to an incomplete character or shift sequence at the end of the input buffer. | EBADF | The <i>cd</i> input parameter is not a valid open code conversion descriptor. |
| EILSEQ              | The input conversion was stopped due to an input byte that does not belong to the input codeset.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| E2BIG               | The input conversion was stopped due to lack of space in the output buffer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| EINVAL              | The input conversion was stopped due to an incomplete character or shift sequence at the end of the input buffer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |
| EBADF               | The <i>cd</i> input parameter is not a valid open code conversion descriptor.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |        |                                                                                                  |       |                                                                             |        |                                                                                                                   |       |                                                                               |

**Description** The `kiconv()` function converts the sequence of characters from one codeset, in the array specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the

array specified by *outbuf*. The codesets are those specified in the `kiconv_open()` call that returned the code conversion descriptor, *cd*. The *inbuf* parameter points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* parameter points to a variable that points to the first available byte in the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When `kiconv()` is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, `kiconv()` places, if any, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, `kiconv()` fails and sets *errno* to E2BIG. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the byte following the last byte that was successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `kiconv()` encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target codeset, `kiconv()` performs an implementation-defined conversion (that is, a non-identical conversion) on this character.

**Return Values** The `kiconv()` function updates the variables pointed to by the parameters to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* is 0. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft* is non-zero and *errno* is set to indicate the condition. If such and other error occurs, `kiconv()` returns  $(\text{size\_t})-1$  and sets *errno* to indicate the error.

**Context** `kiconv()` can be called from user or interrupt context.

**Examples** EXAMPLE 1 Performing a Simple Conversion

The following example shows how to perform a simple conversion using `kiconv()` with a limited size of output buffer:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

int
doconversion(char *fromcode, char *tocode, char *inbuf, char *outbuf,
             size_t inlen, size_t *outlen)
{
    kiconv_t cd;
    size_t ileft, ret;
    int err;

    cd = kiconv_open((const char *)tocode, (const char *)fromcode);
    if (cd == (kiconv_t)-1) {
        /* Cannot open conversion. */
        return (-1);
    }

    ret = kiconv(cd, &inbuf, &inlen, &outbuf, outlen, &err);
    if (ret == (size_t)-1)
        goto doconv_error_return;

    /*
     * Reset the conversion descriptor. This will also
     * make sure to write to output buffer any saved bytes
     * in the conversion descriptor state.
     */
    ileft = 0;
    ret = kiconv(cd, (char *)NULL, &ileft, &outbuf, outlen, &err);
    if (ret == (size_t)-1)
        goto doconv_error_return;

    (void) kiconv_close(cd);

    return (0);

doconv_error_return:
    (void) kiconv_close(cd);

    /* Need more output buffer. */
    if (err == E2BIG)
        return (-2);
}
```

**EXAMPLE 1** Performing a Simple Conversion *(Continued)*

```

/* Illegal sequence? */
if (err == EILSEQ)
    return (-3);

/* Incomplete character? */
if (err == EINVAL)
    return (-4);

/*
 * Bad code conversion descriptor or any other unknown error.
 */
return (-5);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [iconv\(3C\)](#), [iconv\\_close\(3C\)](#), [iconv\\_open\(3C\)](#), [u8\\_strcmp\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [uconv\\_u16tou32\(3C\)](#), [uconv\\_u16tou8\(3C\)](#), [uconv\\_u32tou16\(3C\)](#), [uconv\\_u32tou8\(3C\)](#), [uconv\\_u8tou16\(3C\)](#), [uconv\\_u8tou32\(3C\)](#), [attributes\(5\)](#), [kiconvstr\(9F\)](#), [kiconv\\_close\(9F\)](#), [kiconv\\_open\(9F\)](#), [u8\\_strcmp\(9F\)](#), [u8\\_textprep\\_str\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#), [uconv\\_u16tou8\(9F\)](#), [uconv\\_u32tou16\(9F\)](#), [uconv\\_u32tou8\(9F\)](#), [uconv\\_u8tou16\(9F\)](#), [uconv\\_u8tou32\(9F\)](#)

The Unicode Standard:

<http://www.unicode.org/standard/standard.html>

**Notes** The [iconv\(3C\)](#) man page also has a good example code that can be referenced.

**Name** kiconv\_close – code conversion descriptor deallocation function

**Synopsis** `#include <sys/errno.h>`  
`#include <sys/sunddi.h>`

```
int kiconv_close(kiconv_t cd);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *cd* Code conversion descriptor to be deallocated.

**Description** The `kiconv_close()` function deallocates the conversion descriptor *cd* and all other associated resources allocated by the `kiconv_open()` function.

**Return Values** Upon successful completion, `kiconv_close()` returns 0; otherwise, it returns the following *errno* value to indicate the error:

EBADF The code conversion descriptor is invalid.

**Context** `kiconv_close()` can be called from user or interrupt context.

**Examples** EXAMPLE 1 Closing the Code Conversion

The following example shows how to close the code conversion descriptor with error checking:

```
if (kiconv_close(cd) == EBADF) {
    /* Code conversion descriptor is invalid. */
    return (-1);
}
```

The following example shows how to close the code conversion descriptor without error checking:

```
(void) kiconv_close(cd);
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [iconv\(3C\)](#), [iconv\\_close\(3C\)](#), [iconv\\_open\(3C\)](#), [u8\\_strerror\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [uconv\\_u16tou32\(3C\)](#), [uconv\\_u16tou8\(3C\)](#), [uconv\\_u32tou16\(3C\)](#), [uconv\\_u32tou8\(3C\)](#), [uconv\\_u8tou16\(3C\)](#), [uconv\\_u8tou32\(3C\)](#), [attributes\(5\)](#), [kiconv\(9F\)](#), [kiconvstr\(9F\)](#), [kiconv\\_open\(9F\)](#), [u8\\_strerror\(9F\)](#), [u8\\_textprep\\_str\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#), [uconv\\_u16tou8\(9F\)](#), [uconv\\_u32tou16\(9F\)](#), [uconv\\_u32tou8\(9F\)](#), [uconv\\_u8tou16\(9F\)](#), [uconv\\_u8tou32\(9F\)](#)

The Unicode Standard

<http://www.unicode.org/standard/standard.html>



**Name** kiconv\_open – code conversion descriptor allocation function

**Synopsis** #include <sys/sunddi.h>

```
kiconv_t kiconv_open(const char *tocode, const char *fromcode);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *to*code Points to a target codeset name string.  
*from*code Points to a source codeset name string.

**Description** The `kiconv_open()` function returns a code conversion descriptor that describes a conversion from the codeset specified by *from*code to the codeset specified by *to*code. For state-dependent encodings, the conversion descriptor is in a codeset-dependent initial state (ready for immediate use with the `kiconv()` function).

Supported code conversions are between UTF-8 and the following:

| Name        | Description                                                       |
|-------------|-------------------------------------------------------------------|
| Big5        | Traditional Chinese Big5                                          |
| Big5-HKSCS  | Traditional Chinese Big5-Hong Kong<br>Supplementary Character Set |
| CP720       | DOS Arabic                                                        |
| CP737       | DOS Greek                                                         |
| CP850       | DOS Latin-1 (Western European)                                    |
| CP852       | DOS Latin-2 (Eastern European)                                    |
| CP857       | DOS Latin-5 (Turkish)                                             |
| CP862       | DOS Hebrew                                                        |
| CP866       | DOS Cyrillic Russian                                              |
| CP932       | Japanese Shift JIS (Windows)                                      |
| CP950-HKSCS | Traditional Chinese HKSCS-2001 (Windows)                          |
| CP1250      | Central Europe                                                    |
| CP1251      | Cyrillic                                                          |
| CP1252      | Western Europe                                                    |
| CP1253      | Greek                                                             |
| CP1254      | Turkish                                                           |
| CP1255      | Hebrew                                                            |
| CP1256      | Arabic                                                            |
| CP1257      | Baltic                                                            |
| EUC-CN      | Simplified Chinese EUC                                            |
| EUC-JP      | Japanese EUC                                                      |
| EUC-JP-MS   | Japanese EUC MS                                                   |
| EUC-KR      | Korean EUC                                                        |
| EUC-TW      | Traditional Chinese EUC                                           |
| GB18030     | Simplified Chinese GB18030                                        |
| GBK         | Simplified Chinese GBK                                            |
| ISO-8859-1  | Latin-1 (Western European)                                        |

---

|                |                                           |
|----------------|-------------------------------------------|
| ISO-8859-2     | Latin-2 (Eastern European)                |
| ISO-8859-3     | Latin-3 (Southern European)               |
| ISO-8859-4     | Latin-4 (Northern European)               |
| ISO-8859-5     | Cyrillic                                  |
| ISO-8859-6     | Arabic                                    |
| ISO-8859-7     | Greek                                     |
| ISO-8859-8     | Hebrew                                    |
| ISO-8859-9     | Latin-5 (Turkish)                         |
| ISO-8859-10    | Latin-6 (Nordic)                          |
| ISO-8859-13    | Latin-7 (Baltic)                          |
| ISO-8859-15    | Latin-9 (Western European with euro sign) |
| KOI8-R         | Cyrillic                                  |
| Shift_JIS      | Japanese Shift JIS (JIS)                  |
| TIS_620        | Thai (a.k.a. ISO 8859-11)                 |
| Unified-Hangul | Korean Unified Hangul                     |

UTF-8 and the above names can be used at *to*code and *from*code to specify the desired code conversion. The following aliases are also supported as alternative names to be used:

| Aliases     | Original Name  |
|-------------|----------------|
| 720         | CP720          |
| 737         | CP737          |
| 850         | CP850          |
| 852         | CP852          |
| 857         | CP857          |
| 862         | CP862          |
| 866         | CP866          |
| 932         | CP932          |
| 936, CP936  | GBK            |
| 949, CP949  | Unified-Hangul |
| 950, CP950  | Big5           |
| 1250        | CP1250         |
| 1251        | CP1251         |
| 1252        | CP1252         |
| 1253        | CP1253         |
| 1254        | CP1254         |
| 1255        | CP1255         |
| 1256        | CP1256         |
| 1257        | CP1257         |
| ISO-8859-11 | TIS_620        |
| PCK, SJIS   | Shift_JIS      |

A conversion descriptor remains valid until it is closed by using `kiconv_close()`.

**Return Values** Upon successful completion, `kiconv_open()` returns a code conversion descriptor for use on subsequent calls to `kiconv()`. Otherwise, if the conversion specified by *fromcode* and *to*code is

not supported or for any other reasons the code conversion descriptor cannot be allocated, `kiconv_open()` returns `(kiconv_t)-1` to indicate the error.

**Context** `kiconv_close()` can be called from user context only.

**Examples** EXAMPLE 1 Opening a Code Conversion

The following example shows how to open a code conversion from ISO 8859-15 to UTF-8

```
#include <sys/sunddi.h>

kiconv_t cd;

cd = kiconv_open("UTF-8", "ISO-8859-15");
if (cd == (kiconv_t)-1) {
    /* Cannot open up the code conversion. */
    return (-1);
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [iconv\(3C\)](#), [iconv\\_close\(3C\)](#), [iconv\\_open\(3C\)](#), [u8\\_strerror\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [uconv\\_u16tou32\(3C\)](#), [uconv\\_u16tou8\(3C\)](#), [uconv\\_u32tou16\(3C\)](#), [uconv\\_u32tou8\(3C\)](#), [uconv\\_u8tou16\(3C\)](#), [uconv\\_u8tou32\(3C\)](#), [attributes\(5\)](#), [kiconv\(9F\)](#), [kiconvstr\(9F\)](#), [kiconv\\_close\(9F\)](#), [u8\\_strerror\(9F\)](#), [u8\\_textprep\\_str\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#), [uconv\\_u16tou8\(9F\)](#), [uconv\\_u32tou16\(9F\)](#), [uconv\\_u32tou8\(9F\)](#), [uconv\\_u8tou16\(9F\)](#), [uconv\\_u8tou32\(9F\)](#)

The Unicode Standard

<http://www.unicode.org/standard/standard.html>

**Notes** The code conversions are available between UTF-8 and the above noted *codesets*. For example, to convert from EUC-JP to Shift\_JIS, first convert EUC-JP to UTF-8 and then convert UTF-8 to Shift\_JIS.

The code conversions supported are based on simple one-to-one mappings. There is no special treatment or processing done during code conversions such as case conversion, Unicode Normalization, or mapping between combining or conjoining sequences of UTF-8 and pre-composed characters in non-UTF-8 codesets.

All supported non-UTF-8 codesets use pre-composed characters only. However, UTF-8 allows combining or conjoining characters too. For this reason, using a form of Unicode Normalizations on UTF-8 text with `u8_textprep_str()` before or after doing code conversions might be necessary.

**Name** kiconvstr – string-based code conversion function

**Synopsis** #include <sys/types.h>  
#include <sys/errno.h>  
#include <sys/sunddi.h>

```
size_t kiconvstr(const char *tocode, const char *fromcode, char *inarray,
                size_t *inlen, char *outarray, size_t *outlen, int flag, int *errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** The parameters for the `kiconvstr` function are as follows:

- tocode* Points to a target codeset name string. Supported names are specified at `kiconv_open()`.
- fromcode* Points to a source codeset name string. Supported names are specified at `kiconv_open()`.
- inarray* Points to a byte array containing a sequence of character bytes in *fromcode* codeset to be converted.
- inlen* As an input parameter, the number of bytes to be converted in *inarray*. As an output parameter, the number of bytes in *inarray* still not converted after the conversion.
- outarray* Points to a byte array where converted character bytes in *tocode* codeset can be saved.
- outlen* As an input parameter, the number of available bytes at *outarray* where converted character bytes can be saved. As an output parameter, the number of bytes still available at *outarray* after the conversion.
- flag* Indicates possible conversion options constructed by a bitwise-inclusive-OR of the following values:
- KICONV\_IGNORE\_NULL**  
Normally, `kiconvstr()` stops the conversion if it encounters a string terminating null character of *fromcode* even if the current *inlen* is pointing to a value larger than zero; it does not copy over the null character to *outarray*.
- If this option is set, a null character does not stop the conversion and the conversion continues until either the number of *inarray* bytes pointed by *inlen* are all consumed for conversion, including the null character byte(s), or an error occurred.
- KICONV\_REPLACE\_INVALID**  
Normally, `kiconvstr()` stops the conversion if it encounters illegal or incomplete characters with corresponding *errno* values.

If this option is set, `kiconvstr()` does not stop the conversion and instead treats such characters as non-identical conversion cases.

|              |                                                                                                                  |
|--------------|------------------------------------------------------------------------------------------------------------------|
| <i>errno</i> | Indicates the error when conversion is not completed or failed. The following are possible values:               |
| EILSEQ       | The input conversion was stopped due to an input byte that does not belong to the input codeset.                 |
| E2BIG        | The input conversion was stopped due to lack of space in the output array.                                       |
| EINVAL       | The input conversion was stopped due to an incomplete character or shift sequence at the end of the input array. |
| EBADF        | The requested conversion is not supported.                                                                       |

**Description** The `kiconvstr()` function converts the sequence of characters from one codeset, in the array specified by *inarray*, into a sequence of corresponding characters in another codeset, in the array specified by *outarray*. The codesets are those specified in *fromcode* and *toctype* parameters. The *inarray* parameter points to a character byte array to the first character in the input array and *inlen* indicates the number of bytes to the end of the array to be converted. The *outarray* parameter points to a character byte array to the first available byte in the output array and *outlen* indicates the number of the available bytes to the end of the array. Unless `KICONV_IGNORE_NULL` is specified at *flag*, `kiconvstr()` function normally stops when it encounters NULL byte from the input array regardless of the current *inlen* value.

If `KICONV_REPLACE_INVALID` is not specified at *flag* and if a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If `KICONV_REPLACE_INVALID` is not specified in *flag* and if the input array ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output array is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output array to overflow. The value pointed to by *inlen* is decremented to reflect the number of bytes still not converted in the input array. The value pointed to by *outlen* is decremented to reflect the number of bytes still available in the output array.

If `kiconvstr()` encounters a character in the input array that is legal, but for which an identical character does not exist in the target codeset, `kiconvstr()` performs an implementation-defined conversion (that is, a non-identical conversion) on this character.

If `KICONV_REPLACE_INVALID` is specified in *flag* and if `kiconvstr()` encounters illegal or incomplete characters in the input array, instead of stopping the conversion, `kiconvstr()` treats such characters as if they are non-identical characters and does non-identical conversions on such character bytes.

**Return Values** The `kiconvstr()` function updates the values pointed to by the *inlen* and *outlen* parameters to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input array is converted, the value pointed to by *inlen* is 0. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inlen* is non-zero and *errno* is set to indicate the condition. If an error occurs, `kiconvstr()` returns `(size_t)-1` and sets *errno* to indicate the error.

**Context** `kiconvstr()` can be called from user or interrupt context.

**Examples** **EXAMPLE 1** Performing a Code Conversion

The following example converts a NULL-terminated ISO8859-2 pathname string to a UTF-8 string and treats illegal and incomplete characters as non-identical conversion cases. The conversion does not stop even if it encounters a NULL byte from the input array.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

:

size_t ret;
char ib[MAXPATHLEN];
char ob[MAXPATHLEN];
size_t il, ol;
int err;

:

/*
 * We got the pathname from somewhere.
 *
 * Calculate the length of input string including the terminating
 * NULL byte and prepare other parameters for the conversion.
 */
(void) strncpy(ib, pathname, MAXPATHLEN);
il = strlen(ib) + 1;
ol = MAXPATHLEN;

/*
 * Do the conversion. If the ret > 0, that's the number of
 * non-identical character conversions happened during the conversion.
 * Regardless of whether we have conversion failure or not,
 * outarray could contain some converted characters.
 */
ret = kiconvstr("UTF-8", "ISO-8859-2", ib, &il, ob, &ol,
               (KICONV_IGNORE_NULL|KICONV_REPLACE_INVALID), &err);
if (ret == (size_t)-1) {
```

**EXAMPLE 1** Performing a Code Conversion *(Continued)*

```

/* Code conversion not supported? */
if (err == EBADF)
    return (-1);

/* Output array too small? */
if (err == E2BIG)
    return (-2);

/* Unknown error which isn't possible BTW. */
return (-3);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [iconv\(3C\)](#), [iconv\\_close\(3C\)](#), [iconv\\_open\(3C\)](#), [u8\\_strcmp\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [uconv\\_u16tou32\(3C\)](#), [uconv\\_u16tou8\(3C\)](#), [uconv\\_u32tou16\(3C\)](#), [uconv\\_u32tou8\(3C\)](#), [uconv\\_u8tou16\(3C\)](#), [uconv\\_u8tou32\(3C\)](#), [attributes\(5\)](#), [kiconv\(9F\)](#), [kiconv\\_close\(9F\)](#), [kiconv\\_open\(9F\)](#), [u8\\_strcmp\(9F\)](#), [u8\\_textprep\\_str\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#), [uconv\\_u16tou8\(9F\)](#), [uconv\\_u32tou16\(9F\)](#), [uconv\\_u32tou8\(9F\)](#), [uconv\\_u8tou16\(9F\)](#), [uconv\\_u8tou32\(9F\)](#)

The Unicode Standard:

<http://www.unicode.org/standard/standard.html>



**Name** kmem\_alloc, kmem\_zalloc, kmem\_free – allocate kernel memory

**Synopsis** #include <sys/types.h>  
#include <sys/kmem.h>

```
void *kmem_alloc(size_t size, int flag);
void *kmem_zalloc(size_t size, int flag);
void kmem_free(void*buf, size_t size);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *size* Number of bytes to allocate.

*flag* Determines whether caller can sleep for memory. Possible flags are `KM_SLEEP` to allow sleeping until memory is available, or `KM_NOSLEEP` to return `NULL` immediately if memory is not available.

*buf* Pointer to allocated memory.

**Description** The `kmem_alloc()` function allocates *size* bytes of kernel memory and returns a pointer to the allocated memory. The allocated memory is at least double-word aligned, so it can hold any C data structure. No greater alignment can be assumed. *flag* determines whether the caller can sleep for memory. `KM_SLEEP` allocations may sleep but are guaranteed to succeed. `KM_NOSLEEP` allocations are guaranteed not to sleep but may fail (return `NULL`) if no memory is currently available. The initial contents of memory allocated using `kmem_alloc()` are random garbage.

The `kmem_zalloc()` function is like `kmem_alloc()` but returns zero-filled memory.

The `kmem_free()` function frees previously allocated kernel memory. The buffer address and size must exactly match the original allocation. Memory cannot be returned piecemeal.

**Return Values** If successful, `kmem_alloc()` and `kmem_zalloc()` return a pointer to the allocated memory. If `KM_NOSLEEP` is set and memory cannot be allocated without sleeping, `kmem_alloc()` and `kmem_zalloc()` return `NULL`.

**Context** The `kmem_alloc()` and `kmem_zalloc()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid *flag*. The `kmem_free()` function can be called from from user, interrupt, or kernel context.

**See Also** [copyout\(9F\)](#), [freerbuf\(9F\)](#), [getrbuf\(9F\)](#)

*Writing Device Drivers*

**Warnings** Memory allocated using `kmem_alloc()` is not paged. Available memory is therefore limited by the total physical memory on the system. It is also limited by the available kernel virtual address space, which is often the more restrictive constraint on large-memory configurations.

Excessive use of kernel memory is likely to affect overall system performance. Overcommitment of kernel memory will cause the system to hang or panic.

Misuse of the kernel memory allocator, such as writing past the end of a buffer, using a buffer after freeing it, freeing a buffer twice, or freeing a null or invalid pointer, will corrupt the kernel heap and may cause the system to corrupt data or panic.

The initial contents of memory allocated using `kmem_alloc()` are random garbage. This random garbage may include secure kernel data. Therefore, uninitialized kernel memory should be handled carefully. For example, never `copyout(9F)` a potentially uninitialized buffer.

**Notes** `kmem_alloc(0, flag)` always returns NULL. `kmem_free(NULL, 0)` is legal.

**Name** kmem\_cache\_create, kmem\_cache\_alloc, kmem\_cache\_free, kmem\_cache\_destroy, kmem\_cache\_set\_move – kernel memory cache allocator operations

**Synopsis** #include <sys/types.h>  
#include <sys/kmem.h>

```
kmem_cache_t *kmem_cache_create(char *name, size_t bufsize,
                                size_t align, int (*constructor)(void *, void *, int),
                                void (*destructor)(void *, void *), void (*reclaim)(void *),
                                void *private, void *vmp, int cflags);

void kmem_cache_destroy(kmem_cache_t *cp);

void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);

void kmem_cache_free(kmem_cache_t *cp, void *obj);

void kmem_cache_set_move(kmem_cache_t *cp, kmem_cbrt_t (*move)(void *,
  void *, size_t *, void *));

[Synopsis for callback functions:]

int (*constructor)(void *buf, void *user_arg, int kmflags);

void (*destructor)(void *buf, void *user_arg);

kmem_cbrt_t (*move)(void *old, void *new, size_t bufsize,
                   void *user_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** The parameters for the kmem\_cache\_\* functions are as follows:

|                    |                                                                                                                               |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <i>name</i>        | Descriptive name of a <a href="#">kstat(9S)</a> structure of class kmem_cache. Names longer than 31 characters are truncated. |
| <i>bufsize</i>     | Size of the objects it manages.                                                                                               |
| <i>align</i>       | Required object alignment.                                                                                                    |
| <i>constructor</i> | Pointer to an object constructor function. Parameters are defined below.                                                      |
| <i>destructor</i>  | Pointer to an object destructor function. Parameters are defined below.                                                       |
| <i>reclaim</i>     | Drivers should pass NULL.                                                                                                     |
| <i>private</i>     | Pass-through argument for constructor/destructor.                                                                             |
| <i>vmp</i>         | Drivers should pass NULL.                                                                                                     |
| <i>cflags</i>      | Drivers must pass 0.                                                                                                          |
| <i>kmflag</i>      | Possible flags are:                                                                                                           |
|                    | <b>KM_SLEEP</b> Allow sleeping (blocking) until memory is available.                                                          |

`KM_NOSLEEP` Return NULL immediately if memory is not available.

`KM_PUSHPAGE` Allow the allocation to use reserved memory.

*obj* Pointer to the object allocated by `kmem_cache_alloc()`.

*move* Pointer to an object relocation function. Parameters are defined below.

The parameters for the callback constructor function are as follows:

`void *buf` Pointer to the object to be constructed.

`void *user_arg` The *private* parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

`int kmflags` Propagated *kmflag* values.

The parameters for the callback destructor function are as follows:

`void *buf` Pointer to the object to be deconstructed.

`void *user_arg` The *private* parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

The parameters for the callback `move()` function are as follows:

`void *old` Pointer to the object to be moved.

`void *new` Pointer to the object that serves as the copy destination for the contents of the old parameter.

`size_t bufsize` Size of the object to be moved.

`void *user_arg` The private parameter from the call to `kmem_cache_create()`; it is typically a pointer to the soft-state structure.

**Description** In many cases, the cost of initializing and destroying an object exceeds the cost of allocating and freeing memory for it. The functions described here address this condition.

Object caching is a technique for dealing with objects that are:

- frequently allocated and freed, and
- have setup and initialization costs.

The idea is to allow the allocator and its clients to cooperate to preserve the invariant portion of an object's initial state, or constructed state, between uses, so it does not have to be destroyed and re-created every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once, the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of `mutex_destroy()` and `mutex_init()` each time. An object's embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as

constructed state. The essential requirement is that the client must free the object (using `kmem_cache_free()`) in its constructed state. The allocator cannot enforce this, so programming errors will lead to hard-to-find bugs.

A driver should call `kmem_cache_create()` at the time of `_init(9E)` or `attach(9E)`, and call the corresponding `kmem_cache_destroy()` at the time of `_fini(9E)` or `detach(9E)`.

`kmem_cache_create()` creates a cache of objects, each of size *bufsize* bytes, aligned on an *align* boundary. Drivers not requiring a specific alignment can pass 0. *name* identifies the cache for statistics and debugging. *constructor* and *destructor* convert plain memory into objects and back again; *constructor* can fail if it needs to allocate memory but cannot. *private* is a parameter passed to the constructor and destructor callbacks to support parameterized caches (for example, a pointer to an instance of the driver's soft-state structure). To facilitate debugging, `kmem_cache_create()` creates a `kstat(9S)` structure of class `kmem_cache` and name *name*. It returns an opaque pointer to the object cache.

`kmem_cache_alloc()` gets an object from the cache. The object will be in its constructed state. *kmflag* has either `KM_SLEEP` or `KM_NOSLEEP` set, indicating whether it is acceptable to wait for memory if none is currently available.

A small pool of reserved memory is available to allow the system to progress toward the goal of freeing additional memory while in a low memory situation. The `KM_PUSHPAGE` flag enables use of this reserved memory pool on an allocation. This flag can be used by drivers that implement `strategy(9E)` on memory allocations associated with a single I/O operation. The driver guarantees that the I/O operation will complete (or timeout) and, on completion, that the memory will be returned. The `KM_PUSHPAGE` flag should be used only in `kmem_cache_alloc()` calls. All allocations from a given cache should be consistent in their use of the flag. A driver that adheres to these restrictions can guarantee progress in a low memory situation without resorting to complex private allocation and queuing schemes. If `KM_PUSHPAGE` is specified, `KM_SLEEP` can also be used without causing deadlock.

`kmem_cache_free()` returns an object to the cache. The object must be in its constructed state.

`kmem_cache_destroy()` destroys the cache and releases all associated resources. All allocated objects must have been previously freed.

`kmem_cache_set_move()` registers a function that the allocator may call to move objects from sparsely allocated pages of memory so that the system can reclaim pages that are tied up by the client. Since caching objects of the same size and type already makes severe memory fragmentation unlikely, there is generally no need to register such a function. The idea is to make it possible to limit worst-case fragmentation in caches that exhibit a tendency to become highly fragmented. Only clients that allocate a mix of long- and short-lived objects from the same cache are prone to exhibit this tendency, making them candidates for a `move()` callback.

The `move()` callback supplies the client with two addresses: the allocated object that the allocator wants to move and a buffer selected by the allocator for the client to use as the copy destination. The new parameter is an allocated, constructed object ready to receive the contents of the old parameter. The *bufsize* parameter supplies the size of the object, in case a single move function handles multiple caches whose objects differ only in size. Finally, the private parameter passed to the constructor and destructor is also passed to the `move()` callback.

Only the client knows about its own data and when it is a good time to move it. The client cooperates with the allocator to return unused memory to the system, and the allocator accepts this help at the client's convenience. When asked to move an object, the client can respond with any of the following:

```
typedef enum kmem_cbrc {
    KMEM_CBRC_YES,
    KMEM_CBRC_NO,
    KMEM_CBRC_LATER,
    KMEM_CBRC_DONT_NEED,
    KMEM_CBRC_DONT_KNOW
} kmem_cbrc_t;
```

The client must not explicitly free either of the objects passed to the `move()` callback, since the allocator wants to free them directly to the slab layer (bypassing the per-CPU magazine layer). The response tells the allocator which of the two object parameters to free:

|                                  |                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>KMEM_CBRC_YES</code>       | The client moved the object; the allocator frees the old parameter.                                                                                                                                                                                                                                                                                                           |
| <code>KMEM_CBRC_NO</code>        | The client refused to move the object; the allocator frees the new parameter (the unused copy destination).                                                                                                                                                                                                                                                                   |
| <code>KMEM_CBRC_LATER</code>     | The client is using the object and cannot move it now; the allocator frees the new parameter (the unused copy destination). The client should use <code>KMEM_CBRC_LATER</code> instead of <code>KMEM_CBRC_NO</code> if the object is likely to become movable soon.                                                                                                           |
| <code>KMEM_CBRC_DONT_NEED</code> | The client no longer needs the object; the allocator frees both the old and new parameters. This response is the client's opportunity to be a model citizen and give back as much as it can.                                                                                                                                                                                  |
| <code>KMEM_CBRC_DONT_KNOW</code> | The client does not know about the object because: <ol style="list-style-type: none"><li>the client has just allocated the object and has not yet put it wherever it expects to find known objects</li><li>the client has removed the object from wherever it expects to find known objects and is about to free the object</li><li>the client has freed the object</li></ol> |

In all of these cases above, the allocator frees the new parameter (the unused copy destination) and searches for the old parameter in the magazine layer. If the object is found, it is removed from the magazine layer and freed to the slab layer so that it will no longer tie up an entire page of memory.

Any object passed to the `move()` callback is guaranteed to have been touched only by the allocator or by the client. Because memory patterns applied by the allocator always set at least one of the two lowest order bits, the bottom two bits of any pointer member (other than `char *` or `short *`, which may not be 8-byte aligned on all platforms) are available to the client for marking cached objects that the client is about to free. This way, the client can recognize known objects in the `move()` callback by the unmarked (valid) pointer value.

If the client refuses to move an object with either `KMEM_CBRC_NO` or `KMEM_CBRC_LATER`, and that object later becomes movable, the client can notify the allocator by calling `kmem_cache_move_notify()`. Alternatively, the client can simply wait for the allocator to call back again with the same object address. Responding `KMEM_CBRC_NO` even once or responding `KMEM_CBRC_LATER` too many times for the same object makes the allocator less likely to call back again for that object.

[Synopsis for notification function:]

```
void kmem_cache_move_notify(kmem_cache_t *cp, void *obj);
```

The parameters for the notification function are as follows:

*cp*     Pointer to the object cache.

*obj*     Pointer to the object that has become movable since an earlier refusal to move it.

**Context** Constructors can be invoked during any call to `kmem_cache_alloc()`, and will run in that context. Similarly, destructors can be invoked during any call to `kmem_cache_free()`, and can also be invoked during `kmem_cache_destroy()`. Therefore, the functions that a constructor or destructor invokes must be appropriate in that context. Furthermore, the allocator may also call the constructor and destructor on objects still under its control without client involvement.

`kmem_cache_create()` and `kmem_cache_destroy()` must not be called from interrupt context. `kmem_cache_create()` can also block for available memory.

`kmem_cache_alloc()` can be called from interrupt context only if the `KM_NOSLEEP` flag is set. It can be called from user or kernel context with any valid flag.

`kmem_cache_free()` can be called from user, kernel, or interrupt context.

`kmem_cache_set_move()` is called from the same context as `kmem_cache_create()`, immediately after `kmem_cache_create()` and before allocating any objects from the cache.

The registered `move()` callback is always invoked in the same global callback thread dedicated for move requests, guaranteeing that no matter how many clients register a `move()` function, the allocator never tries to move more than one object at a time. Neither the allocator nor the client can be assumed to know the object's whereabouts at the time of the callback.

### Examples EXAMPLE 1 Object Caching

Consider the following data structure:

```
struct foo {
    kmutex_t foo_lock;
    kondvar_t foo_cv;
    struct bar *foo_barlist;
    int foo_refcnt;
};
```

Assume that a `foo` structure cannot be freed until there are no outstanding references to it (`foo_refcnt == 0`) and all of its pending bar events (whatever they are) have completed (`foo_barlist == NULL`). The life cycle of a dynamically allocated `foo` would be something like this:

```
foo = kmem_alloc(sizeof (struct foo), KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;
    use foo;
ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a `foo` object we perform a sequence of operations that constitutes nothing but expensive overhead. All of this overhead (that is, everything other than `use foo` above) can be eliminated by object caching.

```
int
foo_constructor(void *buf, void *arg, int tags)
{
    struct foo *foo = buf;
    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
    return (0);
}

void
```



**EXAMPLE 1** Object Caching *(Continued)*

```

foo_destructor(void *buf, void *arg)
{
    struct foo *foo = buf;
    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}

user_arg = ddi_get_soft_state(foo_softc, instance);
(void) snprintf(buf, KSTAT_STRLEN, "foo%d_cache",
               ddi_get_instance(dip));
foo_cache = kmem_cache_create(buf,
                              sizeof (struct foo), 0,
                              foo_constructor, foo_destructor,
                              NULL, user_arg, 0);

```

To allocate, use, and free a foo object:

```

foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
    use foo;
kmem_cache_free(foo_cache, foo);

```

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. foo\_constructor and foo\_destructor will be invoked only to populate and drain the cache, respectively.

**EXAMPLE 2** Registering a Move Callback

To register a move() callback:

```

object_cache = kmem_cache_create(...);
kmem_cache_set_move(object_cache, object_move);

```

**Return Values** If successful, the constructor function must return 0. If KM\_NOSLEEP is set and memory cannot be allocated without sleeping, the constructor must return -1.

kmem\_cache\_create() returns a pointer to the allocated cache.

If successful, kmem\_cache\_alloc() returns a pointer to the allocated object. If KM\_NOSLEEP is set and memory cannot be allocated without sleeping, kmem\_cache\_alloc() returns NULL.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [condvar\(9F\)](#), [kmem\\_alloc\(9F\)](#), [mutex\(9F\)](#), [kstat\(9S\)](#)

*Writing Device Drivers*

*The Slab Allocator: An Object-Caching Kernel Memory Allocator*, Bonwick, J.; USENIX Summer 1994 Technical Conference (1994).

*Magazines and vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources*, Bonwick, J. and Adams, J.; USENIX 2001 Technical Conference (2001).

**Notes** The constructor must be immediately reversible by the destructor, since the allocator may call the constructor and destructor on objects still under its control at any time without client involvement.

The constructor must respect the *kmflags* argument by forwarding it to allocations made inside the *constructor*, and must not ASSERT anything about the given flags.

The user argument forwarded to the constructor must be fully operational before it is passed to `kmem_cache_create()`.

**Name** kstat\_create – create and initialize a new kstat

**Synopsis** #include <sys/types.h>  
#include <sys/kstat.h>

```
kstat_t *kstat_create(const char *ks_module, int ks_instance,
                    const char *ks_name, const char *ks_class, uchar_t ks_type,
                    ulong_t ks_ndata, uchar_t ks_flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|------------------------------------------------|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>ks_module</i>      | The name of the provider's module (such as “sd”, “esp”, ...). The “core” kernel uses the name “unix”.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_instance</i>    | The provider's instance number, as from <a href="#">ddi_get_instance(9F)</a> . Modules which do not have a meaningful instance number should use 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_name</i>        | A pointer to a string that uniquely identifies this structure. Only KSTAT_STRLEN – 1 characters are significant.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_class</i>       | The general class that this kstat belongs to. The following classes are currently in use: disk, tape, net, controller, vm, kvm, hat, streams, kstat, and misc.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_type</i>        | The type of kstat to allocate. Valid types are: <table border="0" style="margin-left: 2em;"> <tr> <td>KSTAT_TYPE_NAMED</td> <td>Allows more than one data record per kstat.</td> </tr> <tr> <td>KSTAT_TYPE_INTR</td> <td>Interrupt; only one data record per kstat.</td> </tr> <tr> <td>KSTAT_TYPE_IO</td> <td>I/O; only one data record per kstat</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | KSTAT_TYPE_NAMED   | Allows more than one data record per kstat.                                                                                                                                                                                         | KSTAT_TYPE_INTR     | Interrupt; only one data record per kstat.     | KSTAT_TYPE_IO         | I/O; only one data record per kstat                                                                                                                                                                                                                                                                                                                                                     |
| KSTAT_TYPE_NAMED      | Allows more than one data record per kstat.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| KSTAT_TYPE_INTR       | Interrupt; only one data record per kstat.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| KSTAT_TYPE_IO         | I/O; only one data record per kstat                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_ndata</i>       | The number of type-specific data records to allocate.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>ks_flag</i>        | A bit-field of various flags for this kstat. <i>ks_flag</i> is some combination of: <table border="0" style="margin-left: 2em;"> <tr> <td>KSTAT_FLAG_VIRTUAL</td> <td>Tells kstat_create() not to allocate memory for the kstat data section; instead, the driver will set the ks_data field to point to the data it wishes to export. This provides a convenient way to export existing data structures.</td> </tr> <tr> <td>KSTAT_FLAG_WRITABLE</td> <td>Makes the kstat data section writable by root.</td> </tr> <tr> <td>KSTAT_FLAG_PERSISTENT</td> <td>Indicates that this kstat is to be persistent over time. For persistent kstats, <a href="#">kstat_delete(9F)</a> simply marks the kstat as dormant; a subsequent kstat_create() reactivates the kstat. This feature is provided so that statistics are not lost across driver close/open (such as raw disk I/O on a disk with no mounted partitions.) Note: Persistent kstats cannot be</td> </tr> </table> | KSTAT_FLAG_VIRTUAL | Tells kstat_create() not to allocate memory for the kstat data section; instead, the driver will set the ks_data field to point to the data it wishes to export. This provides a convenient way to export existing data structures. | KSTAT_FLAG_WRITABLE | Makes the kstat data section writable by root. | KSTAT_FLAG_PERSISTENT | Indicates that this kstat is to be persistent over time. For persistent kstats, <a href="#">kstat_delete(9F)</a> simply marks the kstat as dormant; a subsequent kstat_create() reactivates the kstat. This feature is provided so that statistics are not lost across driver close/open (such as raw disk I/O on a disk with no mounted partitions.) Note: Persistent kstats cannot be |
| KSTAT_FLAG_VIRTUAL    | Tells kstat_create() not to allocate memory for the kstat data section; instead, the driver will set the ks_data field to point to the data it wishes to export. This provides a convenient way to export existing data structures.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| KSTAT_FLAG_WRITABLE   | Makes the kstat data section writable by root.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |
| KSTAT_FLAG_PERSISTENT | Indicates that this kstat is to be persistent over time. For persistent kstats, <a href="#">kstat_delete(9F)</a> simply marks the kstat as dormant; a subsequent kstat_create() reactivates the kstat. This feature is provided so that statistics are not lost across driver close/open (such as raw disk I/O on a disk with no mounted partitions.) Note: Persistent kstats cannot be                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                    |                                                                                                                                                                                                                                     |                     |                                                |                       |                                                                                                                                                                                                                                                                                                                                                                                         |

virtual, since `ks_data` points to garbage as soon as the driver goes away.

**Description** `kstat_create()` is used in conjunction with [kstat\\_install\(9F\)](#) to allocate and initialize a [kstat\(9S\)](#) structure. The method is generally as follows:

`kstat_create()` allocates and performs necessary system initialization of a [kstat\(9S\)](#) structure. `kstat_create()` allocates memory for the entire `kstat` (header plus data), initializes all header fields, initializes the data section to all zeroes, assigns a unique `kstat` ID (KID), and puts the `kstat` onto the system's `kstat` chain. The returned `kstat` is marked invalid because the provider (caller) has not yet had a chance to initialize the data section.

After a successful call to `kstat_create()` the driver must perform any necessary initialization of the data section (such as setting the name fields in a `kstat` of type `KSTAT_TYPE_NAMED`). Virtual `kstats` must have the `ks_data` field set at this time. The provider may also set the `ks_update`, `ks_private`, and `ks_lock` fields if necessary.

Once the `kstat` is completely initialized, [kstat\\_install\(9F\)](#) is used to make the `kstat` accessible to the outside world.

**Return Values** If successful, `kstat_create()` returns a pointer to the allocated `kstat`. `NULL` is returned upon failure.

**Context** `kstat_create()` can be called from user or kernel context.

**Examples** **EXAMPLE 1** Allocating and Initializing a `kstat` Structure

```
pkstat_t *ksp;
ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}
```

**See Also** [kstat\(3KSTAT\)](#), [ddi\\_get\\_instance\(9F\)](#), [kstat\\_delete\(9F\)](#), [kstat\\_install\(9F\)](#), [kstat\\_named\\_init\(9F\)](#), [kstat\(9S\)](#), [kstat\\_named\(9S\)](#)

*Writing Device Drivers*

**Name** kstat\_delete – remove a kstat from the system

**Synopsis**

```
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_delete(kstat_t *ksp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *ksp* Pointer to a currently installed [kstat\(9S\)](#) structure.

**Description** `kstat_delete()` removes *ksp* from the kstat chain and frees all associated system resources.

**Return Values** None.

**Context** `kstat_delete()` can be called from any context.

**See Also** [kstat\\_create\(9F\)](#), [kstat\\_install\(9F\)](#), [kstat\\_named\\_init\(9F\)](#), [kstat\(9S\)](#)

*Writing Device Drivers*

**Notes** When calling `kstat_delete()`, the driver must not be holding that kstat's `ks_lock`. Otherwise, it may deadlock with a kstat reader.

**Name** kstat\_install – add a fully initialized kstat to the system

**Synopsis**

```
#include <sys/types.h>
#include <sys/kstat.h>

void kstat_install(kstat_t *ksp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *ksp* Pointer to a fully initialized [kstat\(9S\)](#) structure.

**Description** `kstat_install()` is used in conjunction with [kstat\\_create\(9F\)](#) to allocate and initialize a [kstat\(9S\)](#) structure.

After a successful call to `kstat_create()` the driver must perform any necessary initialization of the data section (such as setting the name fields in a kstat of type `KSTAT_TYPE_NAMED`). Virtual kstats must have the `ks_data` field set at this time. The provider may also set the `ks_update`, `ks_private`, and `ks_lock` fields if necessary.

Once the kstat is completely initialized, `kstat_install` is used to make the kstat accessible to the outside world.

**Return Values** None.

**Context** `kstat_install()` can be called from user or kernel context.

**Examples** **EXAMPLE 1** Allocating and Initializing a kstat Structure

The method for allocating and initializing a kstat structure is generally as follows:

```
kstat_t *ksp;
ksp = kstat_create(module, instance, name, class, type, ndata, flags);
if (ksp) {
    /* ... provider initialization, if necessary */
    kstat_install(ksp);
}
```

**See Also** [kstat\\_create\(9F\)](#), [kstat\\_delete\(9F\)](#), [kstat\\_named\\_init\(9F\)](#), [kstat\(9S\)](#)

*Writing Device Drivers*

**Name** kstat\_named\_init, kstat\_named\_setstr – initialize a named kstat

**Synopsis** #include <sys/types.h>  
#include <sys/kstat.h>

```
void kstat_named_init(kstat_named_t *knp, const char *name,
    uchar_t data_type);
```

```
void kstat_named_setstr(kstat_named_t *knp, const char *str);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

|                  |                                                                                                                                  |
|------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <i>knp</i>       | Pointer to a <a href="#">kstat_named(9S)</a> structure.                                                                          |
| <i>name</i>      | The name of the statistic.                                                                                                       |
| <i>data_type</i> | The type of value. This indicates which field of the <a href="#">kstat_named(9S)</a> structure should be used. Valid values are: |
|                  | KSTAT_DATA_CHAR           The “char” field.                                                                                      |
|                  | KSTAT_DATA_LONG           The “long” field.                                                                                      |
|                  | KSTAT_DATA_ULONG          The “unsigned long” field.                                                                             |
|                  | KSTAT_DATA_LONGLONG        Obsolete. Use KSTAT_DATA_INT64.                                                                       |
|                  | KSTAT_DATA_ULONGLONG       Obsolete. Use KSTAT_DATA_UINT64.                                                                      |
|                  | KSTAT_DATA_STRING          Arbitrary length “long string” field.                                                                 |
| <i>str</i>       | Pointer to a NULL-terminated string.                                                                                             |

**Description** kstat\_named\_init() associates a name and a type with a [kstat\\_named\(9S\)](#) structure.

kstat\_named\_setstr() associates *str* with the named kstat *knp*. It is an error for *knp* to be of type other than KSTAT\_DATA\_STRING. The string argument must remain valid even after the function that is calling kstat\_named\_setstr() is returned. This is the only supported method of changing the value of long strings.

**Return Values** None.

**Context** kstat\_named\_init() and kstat\_named\_setstr() can be called from user or kernel context.

**See Also** [kstat\\_create\(9F\)](#), [kstat\\_install\(9F\)](#), [kstat\(9S\)](#), [kstat\\_named\(9S\)](#)

*Writing Device Drivers*

**Name** kstat\_queue, kstat\_waitq\_enter, kstat\_waitq\_exit, kstat\_runq\_enter, kstat\_runq\_exit, kstat\_waitq\_to\_runq, kstat\_runq\_back\_to\_waitq – update I/O kstat statistics

**Synopsis** #include <sys/types.h>  
#include <sys/kstat.h>

```
void kstat_waitq_enter(kstat_io_t *kiop);
void kstat_waitq_exit(kstat_io_t *kiop);
void kstat_runq_enter(kstat_io_t *kiop);
void kstat_runq_exit(kstat_io_t *kiop);
void kstat_waitq_to_runq(kstat_io_t *kiop);
void kstat_runq_back_to_waitq(kstat_io_t *kiop);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *kiop* Pointer to a [kstat\\_io\(9S\)](#) structure.

**Description** A large number of I/O subsystems have at least two basic "lists" (or queues) of transactions they manage: one for transactions that have been accepted for processing but for which processing has yet to begin, and one for transactions which are actively being processed (but not done). For this reason, two cumulative time statistics are kept: wait (pre-service) time, and run (service) time.

The `kstat_queue()` family of functions manage these times based on the transitions between the driver wait queue and run queue.

|                                    |                                                                                                                                                                                                                                                      |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>kstat_waitq_enter()</code>   | <code>kstat_waitq_enter()</code> should be called when a request arrives and is placed into a pre-service state (such as just prior to calling <a href="#">disksort(9F)</a> ).                                                                       |
| <code>kstat_waitq_exit()</code>    | <code>kstat_waitq_exit()</code> should be used when a request is removed from its pre-service state. (such as just prior to calling the driver's start routine).                                                                                     |
| <code>kstat_runq_enter()</code>    | <code>kstat_runq_enter()</code> is also called when a request is placed in its service state (just prior to calling the driver's start routine, but after <code>kstat_waitq_exit()</code> ).                                                         |
| <code>kstat_runq_exit()</code>     | <code>kstat_runq_exit()</code> is used when a request is removed from its service state (just prior to calling <a href="#">biodone(9F)</a> ).                                                                                                        |
| <code>kstat_waitq_to_runq()</code> | <code>kstat_waitq_to_runq()</code> transitions a request from the wait queue to the run queue. This is useful wherever the driver would have normally done a <code>kstat_waitq_exit()</code> followed by a call to <code>kstat_runq_enter()</code> . |



`kstat_runq_back_to_waitq()`      `kstat_runq_back_to_waitq()` transitions a request from the run queue back to the wait queue. This may be necessary in some cases (write throttling is an example).

**Return Values** None.

**Context** `kstat_create()` can be called from user or kernel context.

**Warnings** These transitions must be protected by holding the `kstat`'s `ks_lock`, and must be completely accurate (all transitions are recorded). Forgetting a transition may, for example, make an idle disk appear 100% busy.

**See Also** [biodone\(9F\)](#), [disksort\(9F\)](#), [kstat\\_create\(9F\)](#), [kstat\\_delete\(9F\)](#), [kstat\\_named\\_init\(9F\)](#), [kstat\(9S\)](#), [kstat\\_io\(9S\)](#)

*Writing Device Drivers*

**Name** `ldi_add_event_handler` – add NDI event service callback handler

**Synopsis** `#include <sys/ddi_obsolete.h>`

```
int ldi_add_event_handler(ldi_handle_t lh, ddi_eventcookie_t ec,  
    void (*handler)(ldi_handle_t, ddi_eventcookie_t,  
    void *, void *) void *arg, ldi_callback_id_t *id);
```

**Interface Level** Obsolete

**Parameters** `ldi_handle_t lh`

Layered handle representing event notification device.

`ddi_eventcookie_t ec`

Cookie returned from call to [ldi\\_get\\_eventcookie\(9F\)](#).

`void (*handler)(ldi_handle_t, ddi_eventcookie_t, void *, void *)`

Callback handler for NDI event service notification.

`void *arg`

Pointer to opaque data supplied by caller. Typically, this is a pointer to the layered driver's softstate structure.

`ldi_callback_id_t *id`

Pointer to registration id, where a unique registration id is returned. Registration id must be saved and used when calling [ldi\\_remove\\_event\\_handler\(9F\)](#) to unregister a callback handler.

**Description** This function is obsolete and is only maintained for compatibility. Use of this function is strongly discouraged. For equivalent functionality provided by new interfaces, see [ldi\\_ev\\_get\\_cookie\(9F\)](#) and [ldi\\_ev\\_register\\_callbacks\(9F\)](#).

The `ldi_add_event_handler()` function adds a callback handler to be invoked at the occurrence of the event specified by the cookie. Adding a callback handler is also known as subscribing to an event. Upon successful subscription, the handler is invoked when the event occurs. You can unregister the handler by using [ldi\\_remove\\_event\\_handler\(9F\)](#).

An instance of a layered driver can register multiple handlers for an event or a single handler for multiple events. Callback order is not defined and should be assumed to be random.

The routine handler is invoked with the following arguments:

`ldi_handle_t lh` Layered handle representing the device for which the event notification is requested.

`ddi_eventcookie_t ec` Structure describing event that occurred.

`void *arg` Opaque data pointer provided by the driver during callback registration.

`void *impl_data` Pointer to event specific data defined by the framework that invokes the callback function.

**Return Values** `DDI_SUCCESS` Callback handler registered successfully.  
`DDI_FAILURE` Failed to register callback handler. Possible reasons include lack of resources or a bad cookie.

**Context** The `ldi_add_event_handler()` function can be called from user and kernel contexts only.

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_get\\_eventcookie\(9F\)](#), [ldi\\_remove\\_event\\_handler\(9F\)](#)

#### *Writing Device Drivers*

**Notes** Layered drivers must remove all registered callback handlers for a device instance, represented by the layered handle, by calling [ldi\\_remove\\_event\\_handler\(9F\)](#) before the layered driver's `detach(9E)` routine completes.

**Name** ldi\_ared, ldi\_awrite – Issue an asynchronous read or write request to a device

**Synopsis** #include <sys/sunldi.h>

```
int ldi_ared(ldi_handle_t lh, struct aio_req *aio_req, cred_t *cr);  
int ldi_awrite(ldi_handle_t lh, struct aio_req *aio_req, cred_t *cr);
```

**Parameters**

|                 |                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------|
| <i>lh</i>       | Layered handle.                                                                                      |
| <i>cr</i>       | Pointer to a credential structure.                                                                   |
| <i>aio_reqp</i> | Pointer to the aio_req(9S) structure that describes where the data is to be stored or obtained from. |

**Description** The ldi\_awrite() function passes an asynchronous write request to a device entry point specified by the layered handle. This operation is supported for block and character devices.

The ldi\_ared() function passes an asynchronous read request to a device entry point specified by the layered handle. This operation is supported for block and character devices.

**Return Values** The ldi\_awrite() and ldi\_ared() functions return 0 upon success. If a failure occurs before the request is passed on to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

|         |                                             |
|---------|---------------------------------------------|
| EINVAL  | Invalid input parameters.                   |
| ENOTSUP | Operation is not supported for this device. |

**Context** These functions may be called from user context.

**Name** ldi\_devmap – Issue a devmap request to a device

**Synopsis** #include <sys/sunldi.h>

```
int ldi_devmap(ldi_handle_t lh, devmap_cookie_t dhp, offset_t off,
              size_t len, size_t *maplen, uint_t model);
```

**Parameters**

- lh* Layered handle.
- dhp* Opaque mapping handle used by the system to describe mapping.
- off* User offset within the logical device memory at which mapping begins.
- len* Mapping length (in bytes).
- maplen* Pointer to length (in bytes) of validated mapping. (Less than or equal to *len*).
- model* Data model type of current thread.

**Description** The `ldi_devmap()` function passes an devmap request to the device entry point for the device specified by the layered handle. This operation is supported for character devices.

**Return Values** The `ldi_devmap()` function returns 0 upon success. If a failure occurs before the request is passed to the device, possible return values are shown below. Otherwise any other error number may be returned by the device.

- EINVAL Invalid input parameters.
- ENOTSUP Operation is not supported for this device.

**Context** This function may be called from user or kernel context.

**Name** ldi\_dump – Issue a dump request to a device

**Synopsis** #include <sys/sunlidi.h>

```
int ldi_dump(ldi_handle_t lh, caddr_t addr, daddr_t blkno, int nblk);
```

**Parameters** *lh* Layered handle.

*addr* Area dump address.

*blkno* Block offset to dump memory.

*nblk* Number of blocks to dump.

**Description** The `ldi_dump()` function passes a dump request to the device entry point specified by the layered handle. This operation is supported for block devices.

**Return Values** The `ldi_dump()` function returns 0 upon success. If a failure occurs before the request is passed on to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL Invalid input parameters.

ENOTSUP Operation is not supported for this device.

**Context** These functions may be called from user or kernel context.

- 
- Name** `ldi_ev_finalize` – propagate disposition of a state change event
- Synopsis** `#include <sys/sunlndi.h>`
- ```
void ldi_ev_finalize(dev_info_t *dip, minor_t minor, int spec_type,
                   int ldi_result, ldi_ev_cookie_t cookie, void *ev_data);
```
- Interface Level** Solaris DDI specific (Solaris DDI)
- Parameters**
- dev\_info\_t* \*dip  
The devinfo node of the layered consumer exporting the minor device.
- minor\_t* minor  
The minor number of the exported minor device.
- int* spec\_type  
The type of minor device (S\_IFCHR or S\_IFBLK).
- int* ldi\_result  
The final disposition of the state change.
- ldi\_ev\_cookie\_t* cookie  
An opaque event cookie for the event type returned by a previous call to [ldi\\_ev\\_get\\_cookie\(9F\)](#).
- void* \*ev\_data  
Event specific data.
- Description** The `ldi_ev_finalize()` function propagates the final disposition of an event up the software stack. It may result in two actions:
- Invocation of “finalize” LDI callback handlers registered by layered drivers up the software stack.
  - Device contract “negotiation end” (CT\_EV\_NEGEND) events generated on minors exported to userland.
- The event propagated up the software stack may be different than the event received by the layered driver invoking `ldi_ev_finalize()`. For example, a volume manager may receive an “offline” event on one of its LDI opened disks, but may choose to propagate a “degraded” event on minors it exports to userland (since it may have more than one copy of the data). The event cookie argument to [ldi\\_ev\\_notify\(9F\)](#) may be different from the event cookie currently possessed by the layered driver. If that is the case, the layered driver must generate another event cookie via a new [ldi\\_ev\\_get\\_cookie\(9F\)](#) call.
- Return Values** None.
- Context** This function can be called from user and kernel contexts only.

**Examples** EXAMPLE 1 Invoking `ldi_ev_finalize(9F)` from widget's finalize callback

The following example shows how the `ldi_ev_finalize()` function is invoked from a widget's finalize callback:

```
static void
widget_finalize(ldi_handle_t lh, ldi_ev_cookie_t foo_cookie,
               int ldi_result, void *arg, void *ev_data)
{
    ASSERT(strcmp(ldi_ev_get_type(foo_cookie), LDI_EV_FOO) == 0);

    /* Map imported minor to exported minors */
    widget_map(lh, &minor, &spec_type);

    if (ldi_result == LDI_EV_SUCCESS) {
        ldi_ev_finalize(dip, minor, spec_type,
                       LDI_EV_SUCCESS, foo_cookie, ev_data);
    }

    /*
     * The event foo failed. Reconfigure yourself
     * *before* propagating
     */
    widget_reconfigure(lh, LDI_EV_FOO, REACQUIRE);

    ldi_ev_finalize(dip, minor, spec_type, LDI_EV_FAILURE,
                   foo_cookie, ev_data);
}
```

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_ev\\_remove\\_callbacks\(9F\)](#)



- 
- Name** `ldi_ev_get_cookie` – get an LDI event cookie for a specified event
- Synopsis** `#include <sys/sunlndi.h>`
- ```
int ldi_ev_get_cookie(ldi_handle_t lh, char *evname,
                    ldi_ev_cookie_t *cookiep);
```
- Interface Level** Solaris DDI specific (Solaris DDI)
- Parameters** *ldi\_handle\_t lh*  
A layered handle representing the device for which the event notification was requested.
- char \*evname*  
The string name of the event.
- ldi\_ev\_cookie\_t \*cookiep*  
A pointer to type `ldi_ev_cookie_t`. Contains a pointer to the event cookie on return.
- Description** The `ldi_ev_get_cookie()` function accepts the string name of a state change event affecting the device represented by the layered driver handle “`lh`” and returns an opaque cookie on success. The call is successful if the framework supports event notification for the event named by “`evname`”. If successful, the function returns an opaque cookie through the “`cookiep`” parameter. The cookie is required in subsequent calls for registering callbacks on events.
- The following two LDI events are currently defined:
- LDI\_EV\_OFFLINE**  
The device is moving to the offline state.
- LDI\_EV\_DEGRADE**  
The device is moving to the degraded state.
- `ldi_ev_get_cookie()` also accepts the string name of any events defined for NDI event services. For such events, `ldi_ev_get_cookie()` replaces `ldi_get_eventcookie(9F)` (which is now obsolete).
- Return Values** The return values for this function are:
- LDI\_EV\_SUCCESS**  
The event cookie was created successfully.
- LDI\_EV\_FAILURE**  
An error occurred and the cookie was not created.
- Context** This function can be called from user and kernel contexts only.
- See Also** `ldi_ev_register_callbacks(9F)`, `ldi_ev_remove_callbacks(9F)`

**Name** ldi\_ev\_get\_type – get event name string from event cookie

**Synopsis** #include <sys/sunldi.h>

```
int ldi_ev_get_type(ldi_ev_cookie_t cookie);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *ldi\_ev\_cookie\_t* cookie

An opaque event cookie for the event type returned by a previous call to [ldi\\_ev\\_get\\_cookie\(9F\)](#).

**Description** The `ldi_ev_get_type()` function returns the event string represented by the LDI event cookie “cookie”.

**Return Values** On success, this function returns the event string represented by “cookie”. On failure, this function returns NULL.

**Context** This function can be called from user and kernel contexts only.

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_ev\\_remove\\_callbacks\(9F\)](#)

---

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Name</b>            | ldi_ev_notify – propagate notification of a state change event                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Synopsis</b>        | <pre>#include &lt;sys/sunlndi.h&gt;  int ldi_ev_notify(dev_info_t *dip, minor_t minor, int spec_type,                  ldi_ev_cookie_t cookie, void *ev_data);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Interface Level</b> | Solaris DDI specific (Solaris DDI)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Parameters</b>      | <p><i>dev_info_t *dip</i><br/>The devinfo node of the layered consumer exporting the minor device.</p> <p><i>minor_t minor</i><br/>The minor number of the exported minor device.</p> <p><i>int spec_type</i><br/>The type of minor device (S_IFCHR or S_IFBLK).</p> <p><i>ldi_ev_cookie_t cookie</i><br/>An opaque event cookie for the event type returned by a previous call to <a href="#">ldi_ev_get_cookie(9F)</a>.</p> <p><i>void *ev_data</i><br/>Event specific data.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Description</b>     | <p>The <code>ldi_ev_notify()</code> function propagates an event up the software stack. It may result in two actions:</p> <ul style="list-style-type: none"> <li>▪ Invocation of LDI callback handlers registered by layered drivers up the software stack.</li> <li>▪ Device contract events generated on minors exported to userland.</li> </ul> <p>The event propagated up the software stack may be different than the event received by the layered driver invoking <code>ldi_ev_notify()</code>. For example, a volume manager may receive an “offline” event on one of its LDI opened disks, but may choose to propagate a “degraded” event on minors it exports to userland (since it may have more than one copy of the data). The event cookie argument to <code>ldi_ev_notify()</code> may be different from the event cookie currently possessed by the layered driver. If that is the case, the layered driver must generate another event cookie via a new <a href="#">ldi_ev_get_cookie(9F)</a> call.</p> <p>The <code>ldi_ev_*</code> interfaces are designed to ensure that a “finalize” call is generated for layered driver consumers at the earliest point where an <code>LDI_EV_FAILURE</code> is detected. If this happens inside the LDI event framework, then the framework will invoke <code>finalize</code>. In the event a layered driver detects/generates an <code>LDI_EV_FAILURE</code>, then the layered driver must invoke <code>ldi_ev_finalize()</code>. The following is an example of a layered driver invoking <code>ldi_ev_finalize()</code> for the “foo” event:</p> <pre>static int widget_notify(ldi_handle_t lh, ldi_ev_cookie_t foo_cookie, void *arg,               void *ev_data) {</pre> |

```
    ASSERT(strcmp(ldi_ev_get_type(foo_cookie), LDI_EV_FOO) == 0);

    /* Map imported minors to exported minor */
    widget_map(lh, &minor, &spec_type);

    /*
     * Call ldi_ev_notify() to propagate events to our consumers.
     * This *must* happen before we check if widget should block
     * foo
     */
    if (ldi_ev_notify(dip, minor, spec_type, foo_cookie, ev_data)
        != LDI_EV_SUCCESS)
        return (LDI_EV_FAILURE);

    /*
     * Next, check if we can allow the foo event
     */
    if (widget_release(lh, LDI_EV_FOO) == WIDGET_SUCCESS) {
        return (LDI_EV_SUCCESS)
    }

    /*
     * We cannot permit the foo event. The first layer that detects
     * failure i.e. us, must generate finalize events for *our*
     * consumers
     */
    ldi_ev_finalize(dip, minor, spec_type, LDI_EV_FAILURE,
        foo_cookie, ev_data);

    return (LDI_EV_FAILURE);
}
```

**Return Values** The return values for this function are:

LDI\_EV\_SUCCESS

Consumers up the software stack permit state change.

LDI\_EV\_FAILURE

Consumers are blocking the state change.

**Context** This function can be called from user and kernel context only.

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_ev\\_remove\\_callbacks\(9F\)](#)

**Name** `ldi_ev_register_callbacks` – add a notify and/or finalize callback

**Synopsis** `#include <sys/sunldi.h>`

```
int ldi_ev_get_cookie(ldi_handle_t lh, ldi_ev_cookie_t *cookie,
                    ldi_ev_callback_t *callb, void *arg, ldi_ev_callback_id_t *id);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** `ldi_handle_t lh`

A layered handle representing the device for which the event notification was requested.

`ldi_ev_cookie_t *cookie`

An opaque event cookie for the event type returned by a previous call to [ldi\\_ev\\_get\\_cookie\(9F\)](#).

`ldi_ev_callback_t *callb`

A data structure which currently has the following members:

```
struct ldi_ev_callback {
    uint_t    cb_vers;
    int      (*cb_notify)(ldi_handle_t,
                        ldi_ev_cookie_t cookie,
                        void *arg, void *ev_data);
    void     (*cb_finalize)(ldi_handle_t,
                        ldi_ev_cookie_t cookie,
                        int ldi_result,
                        void *arg,
                        void *ev_data);
} ldi_ev_callback_t;
```

where

`cb_vers`      Version of callback vector. Must be set to `LDI_EV_CB_VERS` by the caller.

The arguments passed into the callbacks when they are invoked, include:

`int ldi_result`

The actual result of the state change operation/event passed to finalize callback: `LDI_EV_SUCCESS`: The state change succeeded `LDI_EV_FAILURE`: The state change failed.

`void *ev_data`

Event specific data.

`void *arg`

A pointer to opaque caller private data.

`ldi_ev_callback_id_t *id`

Unique system wide registration id returned by `ldi_ev_register_callbacks(9F)` upon successful registration.

**Description** The `ldi_ev_register_callbacks()` interface allows layered drivers to register notify and finalize callbacks for certain events. These events are listed in the `ldi_ev_get_cookie(9F)` man page. The notify callback is invoked only for events that can be blocked, just before the event occurs. The notify event is not called for events serviced by the NDI event service framework since such events are by definition asynchronous. Only the finalize callback is invoked for such events. Layered drivers that have registered notify callbacks for that event have the opportunity of blocking such events. The finalize callback is invoked once the final disposition of the state of a device (specifically a device minor node) is known. The callback is invoked with this result, either `LDI_EV_SUCCESS` (state change succeeded) or `LDI_EV_FAILURE` (state change failed). This allows layered driver consumers to finalize any changes they made in response to a previous “notify” callback.

For example, a layered driver's notify callback may be invoked in response to a `LDI_EV_OFFLINE` event. The layered driver may reconfigure itself to stop using the device and permit the change to go forward. Once that happens, the I/O framework attempts to actually take the device offline. This offline attempt can have two possible outcomes: success or failure. In the former case, the finalize callback is invoked with the `ldi_result` argument set to `LDI_EV_SUCCESS` and the layered driver knows that the device has been taken offline. In the latter case, finalize is invoked with the `ldi_result` set to `LDI_EV_FAILURE` and the layered driver knows that the state change failed. In this case, it may choose to reconfigure itself to start using the device again.

Finalize callbacks can be registered for all events including events that cannot be blocked.

A layered driver can also propagate these events up the software stack by using interfaces offered by the LDI event framework. The layered driver may use `ldi_ev_notify()` to propagate notify events occurring on minors it imports onto minors it exports. Similarly, it may use `ldi_ev_finalize()` to propagate finalize events. Both `ldi_ev_notify()` and `ldi_ev_finalize()` propagate events to device contracts as well as LDI callbacks registered against the exported minor nodes.

The LDI event framework has the following guarantees and requirements with respect to these callbacks:

1. The `notify()` callback is invoked before an event (represented by the event cookie) occurs on a device (represented by the layered driver handle) and is invoked only for events that can be blocked. If the callback returns `LDI_EV_FAILURE`, the event is blocked. If the callback returns `LDI_EV_SUCCESS`, the event is allowed to proceed. If any other value is returned, it is an error. An error message is logged and the event is blocked. An example of an event that can be blocked and for which notify callbacks may be invoked is the offline event `LDI_EV_OFFLINE`.
2. The finalize callback is invoked for all events (including events that cannot be blocked) after the event has occurred. It is invoked with either `LDI_EV_SUCCESS` indicating that the event successfully happened or `LDI_EV_FAILURE` indicating that the event did not occur.

The finalize callback returns no values. Good examples of events that cannot be blocked are the degrade event (`LDI_EV_DEGRADE`) and events serviced by the NDI event service framework.

3. Layered drivers may register one or both of these callbacks (that is, only for a notify event or only for a finalize event or for both) against any LDI handle that they may possess. If a finalize or notify event is not being registered, the corresponding pointer in the `ldi_ev_callback_t` structure must be set to NULL. It is an error to attempt a registration with both callbacks set to NULL.
4. A notify and/or finalize callback is invoked only if the corresponding LDI handle is open. If an LDI handle against which the callbacks are registered is closed, the corresponding finalize and notify callbacks is not invoked as it is assumed that the layered driver is no longer interested in the device. See number 5 below for the exception to this rule.
5. A layered driver that closes its LDI handle in its notify routine receives the corresponding finalize callback after the event has occurred. Because the LDI handle has been closed, the finalize callback is invoked with a NULL LDI handle. It is the responsibility of the layered driver to maintain state in its private “*arg*” parameter so that it can reopen the device (if desired) in its finalize callback.

One example where this may happen is with the `LDI_EV_OFFLINE` event. A layered driver's notify callback may be invoked for an offline event. The layered driver may choose to allow this event to proceed. In that case, since it has a layered open of the device, it must close the LDI handle so that the offline event can succeed (an offline of a device does not succeed if there is any open of the device, layered or otherwise). Since the layered driver has closed the LDI handle in the notify routine, its finalize callback (if any) is invoked with a NULL LDI handle. It is the responsibility of the layered driver to maintain state (such as the device path or `dev_id`) in its private “*arg*” parameter, so that in the finalize routine, it can do a layered open of the device if the device offline failed.

This is the only exception where the finalize callback is invoked if the LDI handle has been closed. In all other cases if the LDI handle has been closed, no corresponding callbacks is invoked.

6. In order for the offline event to succeed (`LDI_EV_OFFLINE`), it is imperative that there be no opens (including LDI handles) to the device. If a layered driver's notify callback is invoked for an offline event and the driver intends to allow the offline to proceed, the driver must close the corresponding LDI handle.
7. The notify and finalize callbacks are not automatically unregistered even if the corresponding LDI handle has been closed. It is the responsibility of the layered driver to unregister these callbacks when they are not required. It may do so using the [ldi\\_ev\\_remove\\_callbacks\(9F\)](#) interface. The LDI framework may panic if the entity registering the callback (such as a `dip`, `dev_t` or `module`) no longer exists on the system and the corresponding callbacks have not been unregistered.

8. The LDI event framework guarantees that if a layered driver receives a notify event, it also receives a finalize event except if the layered consumer itself blocked the event (that is, it returned `LDI_EV_FAILURE` from its notify callback. In this case, the layered driver knows that the event has been blocked and therefore does not need the finalize callback.
9. If a layered driver propagates notify events on minors it imports to minors it exports, it must first propagate these events up the software stack via `ldi_ev_notify()` in its notify callback. It must do so before attempting to check if it blocks the event. This is required, because a layered driver cannot release the device if consumers up the stack are still using the device. If `ldi_ev_notify()` returns `LDI_EV_FAILURE`, the callback must immediately return `LDI_EV_FAILURE` from its notify callback. If `ldi_ev_notify()` returns `LDI_EV_SUCCESS`, then the state change is permissible as far as consumers higher up in the software stack are concerned. The layered driver must then determine if it can permit the state change. If the state change is to be allowed, the layered driver must return `LDI_EV_SUCCESS`. If the layered driver determines that the state change should not be permitted, it must invoke `ldi_ev_finalize()` on minors it exports with a result of `LDI_EV_FAILURE` (to inform consumers up the stack) and then return `LDI_EV_FAILURE` from its notify callback.
10. The LDI event framework generates finalize events at the earliest point where a failure is detected. If the failure is detected in the framework (such as in `ldi_ev_notify()`) the framework generates the finalize events. In the event that a failure is first detected in a layered driver (that is, in the notify callback of a layered driver) the layered driver must use `ldi_ev_finalize()` to send finalize events up the software stack. See the examples for code snippets describing this scenario.
11. The finalize callback must first reconfigure itself before attempting to propagate the event up the software stack via `ldi_ev_finalize(9F)`. This is so that the minors it exports are available and ready for use before the finalize event is propagated up the software stack.
12. It may so happen that the event propagated up the software stack is not the same as the event for which a layered driver's notify/finalize callback is invoked. For example, a layered driver's callback(s) may be invoked for an offline event, but the driver may choose to only propagate the degraded event to its consumers (since it may have a mirror/copy of the data on the device.) In that case, the layered driver must generate a different event cookie (that is, one corresponding to the degraded event via `ldi_ev_get_cookie(9F)`) and use that cookie in its propagation calls (that is, `ldi_ev_notify(9F)` and `ldi_ev_finalize(9F)`).

Once the registration of the callback(s) is successful, an opaque `ldi_ev_callback_id_t` structure is returned which may be used to unregister the callback(s) later.

**Return Values** The return values for this function are:

`LDI_EV_SUCCESS`  
Callback(s) added successfully.

`LDI_EV_FAILURE`  
Failed to add callback(s).



**Context** This function can be called from user and kernel contexts only.

**Examples** EXAMPLE 1 Registration and Callbacks for the OFFLINE Event

The following example shows how the `ldi_ev_register_callbacks()` function performs a registration and callback for the offline event:

```
static int
event_register(void)
{
    ldi_handle_t lh;
    ldi_ev_callback_t callb;
    ldi_ev_cookie_t off_cookie;

    if (ldi_ev_get_cookie(lh, LDI_EV_OFFLINE, &off_cookie)
        == LDI_EV_FAILURE)
        goto fail;

    callb.cb_vers = LDI_EV_CB_VERS;
    callb.cb_notify = off_notify;
    callb.cb_finalize = off_finalize;

    if (ldi_ev_register_callbacks(lh, off_cookie, &callb, arg, &id)
        != LDI_EV_SUCCESS)
        goto fail;
}

static void
event_unregister(ldi_ev_callback_id_t id)
{
    ldi_ev_remove_callbacks(id);
}

static int
off_notify(ldi_handle_t lh, ldi_ev_cookie_t off_cookie, void *arg,
           void *ev_data)
{
    ASSERT(strcmp(ldi_ev_get_type(off_cookie), LDI_EV_OFFLINE) == 0);

    /* Map imported minors to exported minor */
    widget_map(lh, &minor, &spec_type);

    /*
     * Call ldi_ev_notify() to propagate events to our consumers.
     * This *must* happen before we check if offline should be blocked
     */
    if (ldi_ev_notify(dip, minor, spec_type, off_cookie, ev_data)
```

## EXAMPLE 1 Registration and Callbacks for the OFFLINE Event (Continued)

```

        != LDI_EV_SUCCESS)
            return (LDI_EV_FAILURE);

    /*
     * Next, check if we can allow the offline
     */
    if (widget_check(lh) == WIDGET_SUCCESS) {
        widget_save_path(arg, lh);
        widget_reconfigure(lh, RELEASE);
        ldi_close(lh);
        return (LDI_EV_SUCCESS)
    }

    /*
     * We cannot permit the offline. The first layer that detects
     * failure i.e. us, must generate finalize events for our
     * consumers
     */
    ldi_ev_finalize(dip, minor, spec_type, LDI_EV_FAILURE, off_cookie,
        ev_data);

    return (LDI_EV_FAILURE);
}

/*
/*
 * The finalize callback will only be called if we returned LDI_EV_SUCCESS
 * in our notify callback. ldi_result passed in may be SUCCESS or FAILURE
 */
static void
off_finalize(ldi_handle_t NULL_lh, ldi_ev_cookie_t off_cookie,
    int ldi_result, void *arg, void *ev_data)
{
    ldi_handle_t lh;

    ASSERT(strcmp(ldi_ev_get_type(off_cookie), LDI_EV_OFFLINE) == 0);

    path = widget_get_path(arg);

    widget_map_by_path(path, &minor, &spec_type);

    if (ldi_result == LDI_EV_SUCCESS) {
        ldi_ev_finalize(dip, minor, spec_type, LDI_EV_SUCCESS,
            off_cookie, ev_data);
        return;
    }
}

```

**EXAMPLE 1** Registration and Callbacks for the OFFLINE Event *(Continued)*

```

/* The offline failed. Reopen the device */
ldi_open_by_name(path, &lh);
widget_reconfigure(lh, REACQUIRE);

ldi_ev_finalize(dip, minor, spec_type, LDI_EV_FAILURE, off_cookie,
ev_data);
}

```

**EXAMPLE 2** Registration and Callbacks for the DEGRADE Event

The following example shows how the `ldi_ev_register_callbacks()` function performs a registration and callback for the degrade event:

```

static int
event_register(void)
{
    ldi_handle_t lh;
    ldi_ev_callback_t callb;
    ldi_ev_cookie_t dgrd_cookie;

    if (ldi_ev_get_cookie(lh, LDI_EV_DEGRADE, &dgrd_cookie)
        == LDI_EV_FAILURE)
        goto fail;

    /* no notify callbacks allowed for degrade events */
    callb.cb_vers = LDI_EV_CB_VERS;
    callb.cb_notify = NULL; /* NULL, notify cannot be used for
                             DEGRADE */
    callb.cb_finalize = dgrd_finalize;

    if (ldi_ev_register_callbacks(lh, dgrd_cookie, &callb, arg, &id)
        != LDI_EV_SUCCESS)
        goto fail;
}

static void
event_unregister(ldi_ev_callback_id_t id)
{
    ldi_ev_remove_callbacks(id);
}

/*
 * For degrade events. ldi_result will always be LDI_EV_SUCCESS
 */
static void
dgrd_finalize(ldi_handle_t lh, ldi_ev_cookie_t off_cookie,

```

**EXAMPLE 2** Registration and Callbacks for the DEGRADE Event *(Continued)*

```
int ldi_result, void *arg, void *ev_data)
{
    ASSERT(ldi_result == LDI_EV_SUCCESS);
    ASSERT(strcmp(ldi_ev_get_type(off_cookie), LDI_EV_DEGRADE) == 0);

    widget_map(lh, &minor, &spec_type);

    widget_reconfigure(lh, RELEASE);

    ldi_ev_finalize(dip, minor, spec_type, LDI_EV_SUCCESS, d
        grd_cookie, ev_data);
}
```

**See Also** [ldi\\_ev\\_finalize\(9F\)](#), [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_notify\(9F\)](#),  
[ldi\\_ev\\_remove\\_callbacks\(9F\)](#)

**Name** `ldi_ev_remove_callbacks` – remove all callbacks for a given callback ID

**Synopsis** `#include <sys/sunldi.h>`

```
void ldi_ev_remove_callbacks(ldi_ev_callback_id_t id);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *ldi\_ev\_callback\_id\_t* *id*

An opaque data structure returned on successful calls to [ldi\\_ev\\_register\\_callbacks\(9F\)](#).

**Description** The `ldi_ev_remove_callback()` function unregisters any callbacks that were registered via [ldi\\_ev\\_register\\_callbacks\(9F\)](#). Once this function returns, the callback ID is no longer valid.

The finalize and notify callbacks exist independently of the LDI handle and are not automatically removed when the LDI handle is closed. The layered driver removes these callbacks via `ldi_ev_remove_callbacks()` when the callbacks are no longer needed. The LDI framework may panic the system if the entity registering the callback (a `dev_t`, `dip` or `module`) no longer exists on the system and the callbacks have not been unregistered.

**Return Values** None.

**Context** This function can be called from user and kernel contexts only.

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#)

**Name** `ldi_get_dev`, `ldi_get_otyp`, `ldi_get_devid`, `ldi_get_minor_name` – Extract information from a layered handle

**Synopsis** `#include <sys/sunldi.h>`

```
int ldi_get_dev(ldi_handle_t lh, dev_t *devp);
int ldi_get_otyp(ldi_handle_t lh, int *otyp);
int ldi_get_devid(ldi_handle_t lh, ddi_devid_t *devid);
int ldi_get_minor_name(ldi_handle_t lh, char **minor_name);
```

**Parameters**

|                   |                                                                                                                                                                     |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>lh</i>         | Layered handle                                                                                                                                                      |
| <i>otyp</i>       | Indicates on which interface the driver was opened. Valid settings are:<br>OTYP_BLK    Open device block interface.<br>OTYP_CHR    Open device character interface. |
| <i>devp</i>       | Pointer to a device number.                                                                                                                                         |
| <i>devid</i>      | Device ID.                                                                                                                                                          |
| <i>minor_name</i> | Minor device node name.                                                                                                                                             |

**Description** The `ldi_get_dev()` function retrieves the `dev_t` associated with a layered handle.

The `ldi_get_otyp()` retrieves the open flag that was used to open the device associated with the layered handle.

The `ldi_get_devid()` function retrieves a *devid* for the device associated with the layered handle. The caller should use `ddi_devid_free()` to free the *devid* when done with it.

The `ldi_get_minor_name()` function retrieves the name of the minor node opened for the device associated with the layered handle. `ldi_get_minor_name()` allocates a buffer containing the minor node name and returns it via the *minor\_name* parameter. The caller should use `kmem_free()` to release the buffer when done with it.

**Return Values** The `ldi_get_dev()`, `ldi_get_otyp()`, `ldi_get_devid()`, and `ldi_get_devid()` functions return 0 upon success.

In case of an error, the following values may be returned:

|         |                                                 |
|---------|-------------------------------------------------|
| EINVAL  | Invalid input parameters.                       |
| ENOTSUP | The operation is not supported for this device. |

**Context** These functions may be called from user or kernel context.

**Name** ldi\_get\_eventcookie – retrieve NDI event service cookie

**Synopsis** #include <sys/ddi\_obsolete.h>

```
int ldi_get_eventcookie(ldi_handle_t lh, char *name
    ddi_eventcookie_t *ecp);
```

**Interface Level** Obsolete

**Parameters**

|                               |                                                   |
|-------------------------------|---------------------------------------------------|
| <i>ldi_handle_t lh</i>        | Layered handle.                                   |
| <i>char *name</i>             | null-terminated string containing the event name. |
| <i>ddi_eventcookie_t *ecp</i> | Pointer to the kernel event cookie.               |

**Description** This function is obsolete and is only maintained for compatibility. Use of this function is strongly discouraged. For equivalent functionality provided by new interfaces, see [ldi\\_ev\\_get\\_cookie\(9F\)](#) and [ldi\\_ev\\_register\\_callbacks\(9F\)](#).

The `ldi_get_eventcookie()` function queries the device tree for a cookie matching the given event name and returns a reference to that cookie. The search is performed by calling up the device tree hierarchy of the device represented by the layered driver handle until the request is satisfied by a bus nexus driver, or the top of the `dev_info` tree is reached.

The cookie returned by this function can be used to register a callback handler with [ldi\\_add\\_event\\_handler\(9F\)](#).

**Return Values**

|             |                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------|
| DDI_SUCCESS | Cookie handle is returned.                                                                              |
| DDI_FAILURE | Request was not serviceable by any nexus driver in the target device's ancestral device tree hierarchy. |

**Context** This function may be called from user or kernel contexts.

**See Also** [ldi\\_ev\\_get\\_cookie\(9F\)](#), [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_add\\_event\\_handler\(9F\)](#), [ldi\\_remove\\_event\\_handler\(9F\)](#)

*Writing Device Drivers*

**Name** `ldi_get_size`, `ldi_get_blksize`, `ldi_get_pblksize` – Retrieve device size properties

**Synopsis** `#include <sys/sunlndi.h>`

```
int ldi_get_size(ldi_handle_t lh, uint64_t *sizep);
int ldi_get_blksize(ldi_handle_t lh, uint34_t *blksizep);
int ldi_get_pblksize(ldi_handle_t lh, uint32_t *pblksizep);
```

**Parameters** *lh* Layered handle.  
*sizep* Pointer to the caller's unsigned 64-bit integer buffer.  
*blksizep* Pointer to the caller's unsigned 32-bit integer buffer  
*pblksizep* Pointer to the caller's unsigned 32-bit integer buffer

**Description** The `ldi_get_size()` function uses the layered driver handle to calculate and return a device's size. The device size is returned within the caller supplied buffer (*\*sizep*).

The `ldi_get_blksize()` function uses the layered driver handle to return a device's logical block size. Its value is returned within the caller supplied buffer (*\*blksizep*).

The `ldi_get_pblksize()` function uses the layered driver handle to return a device's physical block size. Its value is returned within the caller supplied buffer (*\*pblksizep*).

A valid layered driver handle must be obtained via the [ldi\\_open\\_by\\_name\(9F\)](#) interface prior to calling `ldi_get_size()`, `ldi_get_blksize()`, and `ldi_get_pblksize()`.

**Return Values** The `ldi_get_size()` function returns the following values:

`DDI_SUCCESS` The device size property has been returned within the caller supplied buffer.  
`DDI_FAILURE` The device size property could not be found or determined.

**Context** These functions may be called from user or kernel context.

**See Also** [ldi\\_open\\_by\\_name\(9F\)](#)

*Writing Device Drivers*



**Name** ldi\_ident\_from\_dev, ldi\_ident\_from\_stream, ldi\_ident\_from\_dip, ldi\_ident\_release – ldi cookie management

**Synopsis** #include <sys/sunldi.h>

```
int ldi_ident_from_dip(dev_info_t *dip, ldi_ident_t *lip);
int ldi_ident_from_dev(dev_t dev, ldi_ident_t *lip);
int ldi_ident_from_stream(struct queue *sq, ldi_ident_t *lip);
void ldi_ident_release(ldi_ident_t li);
```

**Parameters**

- li* ldi identifier
- lip* ldi identifier pointer
- dip* pointer to device info node
- dev* device number
- sq* pointer to a stream queue

**Description** The `ldi_ident_from_dev()` function allocates and returns an ldi identifier that is associated with the device number specified by *dev*. The new ldi identifier is returned in the ldi identifier pointer parameter *lip*.

The `ldi_ident_from_dip()` function allocates and returns an ldi identifier that is associated with the device info node pointed to by *dip*. The new ldi identifier is returned in the ldi identifier pointer parameter *lip*.

The `ldi_ident_from_stream()` function allocates and returns an ldi identifier that is associated with the stream pointed to by *queue*. The new ldi identifier is returned in the ldi identifier pointer parameter *lip*.

The `ldi_ident_release()` function releases an identifier that was allocated by one of the `ldi_ident_from_*` functions.

**Return Values** The `ldi_ident_from_dev()`, `ldi_ident_from_dip()`, and `ldi_ident_from_stream()` functions return 0 upon success.

All of these functions return EINVAL for invalid input parameters.

**Context** These functions can be called from user or kernel context, and `ldi_ident_release()` can also be called from interrupt context.

**Name** ldi\_ioctl – send an ioctl to a device

**Synopsis** #include <sys/sunlndi.h>

```
int ldi_ioctl(ldi_handle_t lh, int cmd, intptr_t arg, int mode,
             cred_t *cr, int *rvalp);
```

**Parameters**

- lh* Layered handle.
- cr* Pointer to a credential structure used to open a device.
- rvalp* Caller return value. (May be set by driver and is valid only if the `ioctl()` succeeds).
- cmd* Command argument. Interpreted by driver `ioctl()` as the operation to be performed.
- arg* Driver parameter. Argument interpretation is driver dependent and usually depends on the command type.
- mode* Bit field that contains:
  - FKIOCTL Inform the target device that the ioctl originated from within the kernel.

**Description** The `ldi_ioctl()` function passes an ioctl request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.

If *arg* is interpreted as a pointer (that is, as not an immediate value) and the data pointed to by *arg* is in the kernel's address space, the FKIOCTL flag should be set. This indicates to the target driver that no data model conversion is necessary.

If the caller of `ldi_ioctl()` is not the originator of the ioctl data pointed to by *arg*, (for example, when passing on an ioctl request from a user process), the caller must pass on the mode parameter from the original ioctl. This is because the mode parameter contains the FMODELS bits that enable the target driver to determine the data model of the process which originated the ioctl and perform any necessary conversions. See [ddi\\_model\\_convert\\_from\(9F\)](#) for more information.

**Stream ioctls** For a general description of streams ioctls see [streamio\(7I\)](#). `ldi_ioctl()` supports a number of streams ioctls, using layered handles in the place of file descriptors. When issuing streams ioctls the FKIOCTL parameter should be specified. The possible return values for supported ioctl commands are also documented in [streamio\(7I\)](#).

The following streams ioctls are supported:

- I\_PLINK Behaves as documented in [streamio\(7I\)](#). The layered handle *lh* should point to the streams multiplexer. The *arg* parameter should point to a layered handle for another streams driver.

**I\_UNPLINK** Behaves as documented in [streamio\(7I\)](#). The layered handle *lh* should point to the streams multiplexer. The *arg* parameter is the multiplexor ID number returned by **I\_PLINK** when the streams were linked.

**Return Values** The `ldi_ioctl()` function returns 0 upon success. If a failure occurs before the request is passed on to the device, possible return values are shown below. Otherwise any other error number may be returned by the device.

**EINVAL** Invalid input parameters.

**ENOTSUP** Operation is not supported for this device.

**Context** These functions can be called from user or kernel context.

**See Also** [streamio\(7I\)](#), [ddi\\_model\\_convert\\_from\(9F\)](#)

**Name** ldi\_open\_by\_dev, ldi\_open\_by\_name, ldi\_open\_by\_devid, ldi\_close – open and close devices

**Synopsis** #include <sys/sunldi.h>

```
int ldi_open_by_dev(dev_t *devp, int otyp, int flag, cred_t *cr,
    ldi_handle_t *lhp, ldi_ident_t li);

int ldi_open_by_name(char *pathname, int flag, cred_t *cr,
    ldi_handle_t *lhp, ldi_ident_t li);

int ldi_open_by_devid(ddi_devid_t devid, char *minor_name, int flag,
    cred_t *cr, ldi_handle_t *lhp, ldi_ident_t li);

int ldi_close(ldi_handle_t lh, int flag, cred_t *cr);
```

**Parameters**

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>lh</i>         | Layered handle                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>lhp</i>        | Pointer to a layered handle that is returned upon a successful open.                                                                                                                                                                                                                                                                                                                                                                                                           |
| <i>li</i>         | LDI identifier.                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <i>cr</i>         | Pointer to the credential structure used to open a device.                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <i>devp</i>       | Pointer to a device number.                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <i>pathname</i>   | Pathname to a device.                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <i>devid</i>      | Device ID.                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <i>minor_name</i> | Minor device node name.                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <i>otyp</i>       | Flag passed to the driver indicating which interface is open. Valid settings are: <ul style="list-style-type: none"> <li>OTYP_BLK    Open the device block interface.</li> <li>OTYP_CHR    Open the device character interface.</li> </ul> Only one OTYP flag can be specified. To open streams devices, specify OTYP_CHR.                                                                                                                                                     |
| <i>flag</i>       | Bit field that instructs the driver on how to open the device. Valid settings are: <ul style="list-style-type: none"> <li>FEXCL        Open the device with exclusive access; fail all other attempts to open the device.</li> <li>FNDELAY      Open the device and return immediately. Do not block the open even if something is wrong.</li> <li>FREAD        Open the device with read-only permission. (If ORed with FWRITE, allow both read and write access).</li> </ul> |

- `FWRITE` Open a device with write-only permission (if ORed with `FREAD`, then allow both read and write access).
- `FNOCTTY` Open the device. If the device is a tty, do not attempt to open it as a session-controlling tty.

**Description** The `ldi_open_by_dev()`, `ldi_open_by_name()` and `ldi_open_by_devid()` functions allow a caller to open a block, character, or streams device. Upon a successful open, a layered handle to the device is returned via the layered handle pointed to by `lhp`. The ldi identifier passed to these functions is previously allocated with `ldi_ident_from_stream(9F)`, `ldi_ident_from_dev(9F)`, and `ldi_ident_from_dip(9F)`.

The `ldi_open_by_dev()` function opens a device specified by the `dev_t` pointed to by `devp`. Upon successful open, the caller should check the value of the `dev_t` to see if it has changed. (Cloning devices will change this value during opens.) When opening a streams device, `otyp` must be `OTYP_CHR`.

The `ldi_open_by_devid()` function opens a device by `devid`. The caller must specify the minor node name to open.

The `ldi_open_by_name()` function opens a device by pathname. Pathname is a null terminated string in the kernel address space. Pathname must be an absolute path, meaning that it must begin with `'/'`. The format of the pathname supplied to this function is either a `/devices` path or any other filesystem path to a device node. Opens utilizing `/devices` paths are supported before root is mounted. Opens utilizing other filesystem paths to device nodes are supported only if root is already mounted.

The `ldi_close()` function closes a layered handle that was obtained with either `ldi_open_by_dev()`, `ldi_open_by_name()`, or `ldi_open_by_devid()`. After `ldi_close()` returns the layered handle, the `lh` that was previously passed in is no longer valid.

**Return Values** The `ldi_close()` function returns `0` for success. `EINVAL` is returned for invalid input parameters. Otherwise, any other error number may be returned by the device.

The `ldi_open_by_dev()` and `ldi_open_by_devid()` functions return `0` upon success. If a failure occurs before the device is open, possible return values are shown below. Otherwise any other error number may be returned by the device.

- `EINVAL` Invalid input parameters.
- `ENODEV` Requested device does not exist.
- `ENXIO` Unsupported device operation or access mode.

The `ldi_open_by_name()` function returns `0` upon success. If a failure occurs before the device is open, possible return values are shown below. Otherwise any other error number may be returned by the device.

|        |                                                                                                                                          |
|--------|------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | Invalid input parameters.                                                                                                                |
| ENODEV | Requested device path does not exist.                                                                                                    |
| EACCES | Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by <i>cr</i> are denied. |
| ENXIO  | Unsupported device operation or access mode.                                                                                             |

**Context** These functions may be called from user or kernel context.

These functions should not be called from a device's attach, detach, or power entry point. This could result in a system crash or deadlock.

**See Also** [scsi\\_vhci\(7D\)](#), [ldi\\_ident\\_from\\_dev\(9F\)](#), [ldi\\_ident\\_from\\_dip\(9F\)](#),  
[ldi\\_ident\\_from\\_stream\(9F\)](#)

**Notes** Use only OTYP\_CHR or OTYP\_BLK options when you use the `ldi_open_by_dev()` and `ldi_open_by_devid()` functions to open a device. Other flags, including OTYP\_LYR, have been deprecated and should not be used with these interfaces.

The caller should be aware of cases when multiple paths to a single device may exist. (This can occur for scsi disk devices if [scsi\\_vhci\(7D\)](#) is disabled or a disk is connected to multiple controllers not supported by [scsi\\_vhci\(7D\)](#)).

In these cases, `ldi_open_by_devid()` returns a device handle that corresponds to a particular path to a target device. This path may not be the same across multiple calls to `ldi_open_by_devid()`. Device handles associated with the same device but different access paths should have different filesystem device paths and `dev_t` values.

In the cases where multiple paths to a device exist and access to the device has not been virtualized via MPXIO (as with scsi disk devices not accessed via [scsi\\_vhci\(7D\)](#)), the LDI does not provide any path fail-over capabilities. If the caller wishes to do their own path management and failover they should open all available paths to a device via `ldi_open_by_name()`.

**Name** `ldi_poll` – Poll a device

**Synopsis** `#include <sys/sunlndi.h>`

```
int ldi_poll(ldi_handle_t lh, short events, int anyyet, short *reventsp,
            struct pollhead **phpp);
```

**Parameters**

*lh* Layered handle.

*events* Potential events. Valid events are:

|            |                                                                  |
|------------|------------------------------------------------------------------|
| POLLIN     | Data other than high priority data may be read without blocking. |
| POLLOUT    | Normal data may be written without blocking.                     |
| POLLPRI    | High priority data may be received without blocking.             |
| POLLHUP    | Device hangup has occurred.                                      |
| POLLERR    | An error has occurred on the device.                             |
| POLLRDNORM | Normal data (priority band = 0) may be read without blocking.    |
| POLLRDBAND | Data from a non-zero priority band may be read without blocking. |
| POLLWRNORM | Data other than high priority data may be read without blocking. |
| POLLWRBAND | Priority data (priority band > 0) may be written.                |

*anyyet* A flag that is non-zero if any other file descriptors in the pollfd array have events pending. The `poll(2)` system call takes a pointer to an array of pollfd structures as one of its arguments. See `poll(2)` for more details.

*reventsp* Pointer to a bitmask of the returned events satisfied.

*phpp* Pointer to a pointer to a pollhead structure.

**Description** The `ldi_poll()` function passes a poll request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.

**Return Values** The `ldi_poll()` function returns 0 upon success. If a failure occurs before the request is passed on to the device, possible return values are:

|         |                                             |
|---------|---------------------------------------------|
| EINVAL  | Invalid input parameters.                   |
| ENOTSUP | Operation is not supported for this device. |

**Context** These functions may be called from user or kernel context.



**Name** ldi\_prop\_exists – Check for the existence of a property

**Synopsis** #include <sys/sunlidi.h>

```
int ldi_prop_exists(ldi_handle_t lh, uint_t flags, char *name);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *lh* Layered handle.

*flags* Possible flag values are some combination of:

**LDI\_DEV\_T\_ANY** Match the lookup request independent of the actual dev\_t value that was used when the property was created. The flag indicates any dev\_t value (including DDI\_DEV\_T\_NONE) associated with a possible property match satisfies the matching criteria.

**DDI\_PROP\_DONTPASS** Do not pass request to parent device information node if the property is not found.

**DDI\_PROP\_NOTPROM** Do not look at PROM properties (ignored on platforms that do not support PROM properties).

*name* String containing the name of the property.

**Description** ldi\_prop\_exists() checks for the existence of a property associated with a device represented by the layered driver handle, regardless of the property value data type.

Properties are searched for based on the dip and dev\_t values associated with the layered handle, and the property name. This association is handled by the layered driver infrastructure on behalf of the consumers of ldi\_prop\_exists().

The property search order is as follows:

1. Search software-properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI\_PROP\_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI\_PROP\_DONTPASS is not set, pass this request to the parent device information node of the device represented by the layered handle.
6. Return 0 if not found and 1 if found.

Typically, the specific dev\_t value associated with the device represented by the layered handle (ldi\_handle\_t) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the ldi property look up functions.

However, if the `LDI_DEV_T_ANY` flag is used, the `ldi` property lookup functions will match the request regardless of the `dev_t` value associated with the property at the time of its creation. If a property was created with a `dev_t` set to `DDI_DEV_T_NONE`, the only way to look up this property is with the `LDI_DEV_T_ANY` flag. PROM properties are always created with a `dev_t` set to `DDI_DEV_T_NONE`.

name must always be set to the name of the property being looked up.

**Return Values** `ldi_prop_exists()` returns 1 if the property exists and 0 otherwise.

**Context** This function may be called from user or kernel context.

**Example** The following example demonstrates the use of `ldi_prop_exists()`.

```
/* Determine the existence of the "interrupts" property */
ldi_prop_exists(lh, LDI_DEV_T_ANY|DDI_PROP_NOTPROM, "interrupts");
```

**See Also** `ddi_prop_exists(9F)`

*Writing Device Drivers*

**Name** ldi\_prop\_get\_int, ldi\_prop\_get\_int64 – Lookup integer property

**Synopsis** #include <sys/sunldi.h>

```
int ldi_prop_get_int(ldi_handle_t lh, uint_t flags, char *name,
    int defvalue);

int64_t ldi_prop_get_int64(ldi_handle_t lh, uint_t flags, char *name,
    int64_t defvalue);
```

**Parameters** *lh* Layered handle.

*flags* Possible flag values are some combination of:

|                   |                                                                                                                                                                                                                                             |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDI_DEV_T_ANY     | Match the lookup request independent of the actual dev_t value that was used when the property was created. Indicates any dev_t value (including DDI_DEV_T_NONE) associated with a possible property match satisfies the matching criteria. |
| DDI_PROP_DONTPASS | Do not pass request to parent device information node if property not found.                                                                                                                                                                |
| DDI_PROP_NOTPROM  | Do not look at PROM properties (ignored on platforms that do not support PROM properties).                                                                                                                                                  |

*name* String containing the property name.

*defvalue* Integer value that is returned if the property is not found.

**Interface Level** Solaris DDI specific (Solaris DDI)

**Description** The ldi\_prop\_get\_int() and ldi\_prop\_get\_int64() functions search for an integer property associated with a device represented by the layered driver handle. If the integer property is found, the functions return the property value.

Properties are searched for based on the dip and dev\_t values associated with the layered handle, the property name, and type of the data (integer).

The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If DDI\_PROP\_NOTPROM is not set, search the PROM properties (if they exist).
5. If DDI\_PROP\_DONTPASS is not set, pass this request to the parent device information node of the device represented by the layered handle.

## 6. Return defvalue.

Typically, the specific `dev_t` value associated with the device represented by the layered handle (`ldi_handle_t`) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the `ldi` property look up functions.

However, if the `LDI_DEV_T_ANY` flag is used, the `ldi` property lookup functions match the request regardless of the `dev_t` value associated with the property at the time of its creation. If a property was created with a `dev_t` set to `DDI_DEV_T_NONE`, the only way to look up this property is with the `LDI_DEV_T_ANY` flag. PROM properties are always created with a `dev_t` set to `DDI_DEV_T_NONE`.

name must always be set to the name of the property being looked up.

The return value of the routine is the value of property. If the property is not found, the argument `defvalue` is returned as the property value.

`ldi_prop_get_int64()` does not search the PROM for 64-bit property values.

**Return Values** `ldi_prop_get_int()` and `ldi_prop_get_int64()` return the property value. If the property is not found, the argument `defvalue` is returned. If the property is found, but cannot be decoded into an `int` or an `int64_t`, `DDI_PROP_NOT_FOUND` is returned.

**Context** `ldi_prop_get_int()` and `ldi_prop_get_int64()` can be called from user or kernel context.

**Examples** Using `ldi_prop_get_int64()`.

The following example demonstrates the use of `ldi_prop_get_int64()`.

```
/*
 * Get the value of the integer "timeout" property, using
 * our own default if no such property exists
 */

int64_t timeout, defval;

timeout = ldi_prop_get_int64(lh, LDI_DEV_T_ANY|DDI_PROP_DONTPASS,
    propname, defval);
```

**See Also** [ddi\\_prop\\_get\\_int\(9F\)](#), [ddi\\_prop\\_get\\_int64\(9F\)](#), [ldi\\_prop\\_exists\(9F\)](#).

*Writing Device Drivers*

**Name** ldi\_prop\_lookup\_int\_array, ldi\_prop\_lookup\_int64\_array, ldi\_prop\_lookup\_string\_array, ldi\_prop\_lookup\_string, ldi\_prop\_lookup\_byte\_array – Lookup property information

**Synopsis** #include <sys/sunldi.h>

```
int ldi_prop_lookup_int_array(ldi_handle_t lh, uint_t flags, char *name,
    int **datap, uint_t *nelements);

int ldi_prop_lookup_int64_array(ldi_handle_t lh, uint_t flags, char *name,
    int64_t **datap, uint_t *nelements);

int ldi_prop_lookup_string_array(ldi_handle_t lh, uint_t flags,
    char *name, char ***datap, uint_t *nelements);

int ldi_prop_lookup_string(ldi_handle_t lh, uint_t flags, char *name,
    char **datap);

int ldi_prop_lookup_byte_array(ldi_handle_t lh, uint_t flags, char *name,
    uchar_t **datap, uint_t *nelements);
```

**Parameters** *lh* Layered handle.

*flags* Possible flag values are some combination of:

|                   |                                                                                                                                                                                                                                                         |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDI_DEV_T_ANY     | Match the lookup request independent of the actual dev_t value that was used when the property was created. The flag indicates any dev_t value (including DDI_DEV_T_NONE) associated with a possible property match will satisfy the matching criteria. |
| DDI_PROP_DONTPASS | Do not pass request to parent device information node if the property is not found.                                                                                                                                                                     |
| DDI_PROP_NOTPROM  | Do not look at PROM properties (ignored on platforms that do not support PROM properties).                                                                                                                                                              |

*name* String containing the property name.

*nelements* The address of an unsigned integer which, upon successful return, contains the number of elements accounted for in the memory pointed at by datap. Depending on the interface you use, the elements are either integers, strings or bytes.

*datap*

ldi\_prop\_lookup\_int\_array() Pointer address to an array of integers which, upon successful return, point to memory containing the integer array property value.

ldi\_prop\_lookup\_int64\_array() Pointer address to an array of 64-bit integers which, upon successful return, point to memory containing the integer array property value.

|                                             |                                                                                                                                                                                                                                                                               |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ldi_prop_lookup_string_array()</code> | Pointer address to an array of strings which, upon successful return, point to memory containing the array of strings. The string array is formatted as an array of pointers to NULL terminated strings, much like the <code>argv</code> argument to <code>execve(2)</code> . |
| <code>ldi_prop_lookup_string()</code>       | Pointer address to a string which, upon successful return, points to memory containing the NULL terminated string value of the property.                                                                                                                                      |
| <code>ldi_prop_lookup_byte_array()</code>   | Pointer address to an array of bytes which, upon successful return, point to memory containing the property byte array value.                                                                                                                                                 |

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The property look up functions search for and, if found, return the value of a given property. Properties are searched for based on the `dip` and `dev_t` values associated with the layered handle, the property name, and type of the data (integer, string, or byte).

The property search order is as follows:

1. Search software properties created by the driver.
2. Search the software properties created by the system (or nexus nodes in the device info tree).
3. Search the driver global properties list.
4. If `DDI_PROP_NOTPROM` is not set, search the PROM properties (if they exist).
5. If `DDI_PROP_DONTPASS` is not set, pass this request to the parent device information node of the device represented by the layered handle.
6. Return `DDI_PROP_NOT_FOUND`.

Typically, the specific `dev_t` value associated with the device represented by the layered handle (`ldi_handle_t`) is used as a part of the property match criteria. This association is handled by the layered driver infrastructure on behalf of the consumers of the `ldi` property look up functions.

However, if the `LDI_DEV_T_ANY` flag is used, the `ldi` property lookup functions match the request regardless of the `dev_t` value associated with the property at the time of its creation. If a property was created with a `dev_t` set to `DDI_DEV_T_NONE`, then the only way to look up this property is with the `LDI_DEV_T_ANY` flag. PROM properties are always created with a `dev_t` set to `DDI_DEV_T_NONE`.

name must always be set to the name of the property being looked up.

For the `ldi_prop_lookup_int_array()`, `ldi_prop_lookup_int64_array()`, `ldi_prop_lookup_string_array()`, `ldi_prop_lookup_string()`, and `ldi_prop_lookup_byte_array()` functions, `datap` is the address of a pointer which, upon successful return, points to memory containing the value of the property. In each case `*datap` points to a different type of property value. See the individual descriptions of the functions below for details on the different return values. `nelementsp` is the address of an unsigned integer which, upon successful return, contains the number of integer, string or byte elements accounted for in the memory pointed at by `*datap`.

All of the property look up functions may block to allocate memory needed to hold the value of the property.

When a driver has obtained a property with any look up function and is finished with that property, it must be freed by call `ddi_prop_free()`. `ddi_prop_free()` must be called with the address of the allocated property. For instance, if you call `ldi_prop_lookup_int_array()` with `datap` set to the address of a pointer to an integer, `&my-int-ptr`, the companion free call is `ddi_prop_free(my-int-ptr)`.

Property look up functions are described below:

|                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ldi_prop_lookup_int_array()</code>    | This function searches for and returns an array of integer property values. An array of integers is defined to <code>*nelementsp</code> number of 4 byte long integer elements. <code>datap</code> should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property.                                                                   |
| <code>ldi_prop_lookup_int64_array()</code>  | This function searches for and returns an array of integer property values. An array of integers is defined to <code>*nelementsp</code> number of 8 byte long integer elements. <code>datap</code> should be set to the address of a pointer to an array of integers which, upon successful return, will point to memory containing the integer array value of the property This function does not search the PROM for 64-bit property values. |
| <code>ldi_prop_lookup_string_array()</code> | This function searches for and returns a property that is an array of strings. <code>datap</code> should be set to an address of a pointer to an array of strings which, upon successful return, will point to memory containing the array of strings. The array of strings is formatted as an array of pointers to null-terminated strings, much like the <code>argv</code> argument to <code>execve(2)</code> .                              |

|                                           |                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ldi_prop_lookup_string()</code>     | This function searches for and returns a property that is a null-terminated string. <code>datap</code> should be set to the address of a pointer to a string which, upon successful return, points to memory containing the string value of the property.                                               |
| <code>ldi_prop_lookup_byte_array()</code> | This function searches for and returns a property that is an array of bytes. <code>datap</code> should be set to the address of a pointer to an array of bytes which, upon successful return, points to memory containing the byte array value of the property.                                         |
| <code>ddi_prop_free()</code>              | Frees the resources associated with a property previously allocated using <code>ldi_prop_lookup_int_array()</code> , <code>ldi_prop_lookup_int64_array()</code> , <code>ldi_prop_lookup_string_array()</code> , <code>ldi_prop_lookup_string()</code> , and <code>ldi_prop_lookup_byte_array()</code> . |

**Return Values** The functions `ldi_prop_lookup_int_array()`, `ldi_prop_lookup_int64_array()`, `ldi_prop_lookup_string_array()`, `ldi_prop_lookup_string()`, and `ldi_prop_lookup_byte_array()` return the following values:

|                                     |                                                                                                                                                       |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DDI_PROP_SUCCESS</code>       | Property found and returned.                                                                                                                          |
| <code>DDI_PROP_INVALID_ARG</code>   | If an attempt is made to look up a property with a NULL <code>ldi</code> handle, <code>name</code> is NULL or <code>name</code> is a the null string. |
| <code>DDI_PROP_NOT_FOUND</code>     | Property not found.                                                                                                                                   |
| <code>DDI_PROP_UNDEFINED</code>     | Prop explicitly undefined (see <code>ddi_prop_undefine(9F)</code> ).                                                                                  |
| <code>DDI_PROP_CANNOT_DECODE</code> | Property value cannot be decoded.                                                                                                                     |

**Context** These functions may be called from user or kernel context.

**Example** Using `ldi_prop_lookup_int64_array()`.

```
The following example demonstrates the use of
ldi_prop_lookup_int64_array().
```

```
int64_t *options;
uint_t  noptions;

/*
```



```
* Get the data associated with the integer "options" property
* array, along with the number of option integers
*/

if (ldi_prop_lookup_int64_array(lh,
    LDI_DEV_T_ANY|DDI_PROP_NOTPROM, "options",
    &options, &noptions) == DDI_PROP_SUCCESS) {
    /*
     * Process the options data from the property
     * we just received. Let's do "our thing" with data.
     */
    xx_process_options(options, noptions);

    /*
     * Free the memory allocated for the property data
     */
    ddi_prop_free(options);
}
```

**See Also** [execve\(2\)](#), [ddi\\_prop\\_free\(9F\)](#), [ddi\\_prop\\_lookup\(9F\)](#), [ldi\\_prop\\_exists\(9F\)](#).

*Writing Device Drivers*

**Name** ldi\_putmsg, ldi\_getmsg – Read/write message blocks from/to a stream

**Synopsis** #include <sys/sunldi.h>

```
int ldi_putmsg(ldi_handle_t lh, mblk_t *smp);  
int ldi_getmsg(ldi_handle_t lh, mblk_t **rmp, timestruc_t *timeo);
```

**Parameters**

- lh* Layered handle.
- smp* Message block to send.
- rmp* Message block to receive.
- timeo* Optional timeout for data reception.

**Description** The `ldi_putmsg` function allows a caller to send a message block to a streams device specified by the layered handle *lh*. Once the message (*smp*) has been passed to `ldi_putmsg()`, the caller must not free the message even if an error occurs.

The `ldi_getmsg()` function allows a caller to receive a message block from a streams device specified by the layered handle *lh*. Callers must free the message received with [freemsg\(9F\)](#).

If a NULL timeout value is specified when the caller receives a message, the caller sleeps until a message is received.

**Return Values** The `ldi_putmsg()` and `ldi_getmsg()` functions return 0 upon success. If a failure occurs before the request is passed to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.

EINVAL Invalid input parameters.

ENOTSUP Operation is not supported for this device.

The `ldi_getmsg()` function may also return:

ETIME Returned if the timeout *timeo* expires with no messages received.

**Context** These functions may be called from user or kernel context.

- 
- Name** `ldi_read`, `ldi_write` – Read and write from a device
- Synopsis** `#include <sys/sunlidi.h>`
- ```
int ldi_read(ldi_handle_t lh, struct uio *uiop, cred_t *cr);
int ldi_write(ldi_handle_t lh, struct uio *uiop, cred_t *cr);
```
- Parameters**
- lh* Layered handle.
- cr* Pointer to a credential structure used to open a device.
- uiop* Pointer to the `uio(9S)` structure. `uio(9S)` specifies the location of the read or write data. (Either userland or kernel.)
- Description**
- The `ldi_read()` function passes a read request to the device entry point for the device specified by the layered handle. This operation is supported for block, character, and streams devices.
- The `ldi_write()` function passes a write request to the device entry point for a device specified by the layered handle. This operation is supported for block, character, and streams devices.
- Return Values**
- The `ldi_read()` and `ldi_write()` functions return `0` upon success. If a failure occurs before the request is passed to the device, the possible return values are shown below. Otherwise any other error number may be returned by the device.
- `EINVAL` Invalid input parameters.
- `ENOTSUP` Operation is not supported for this device.
- Context** These functions may be called from user or kernel context.

**Name** `ldi_remove_event_handler` – remove an NDI event service callback

**Synopsis** `#include <sys/ddi_obsolete.h>`

```
int ldi_remove_event_handler(ldi_handle_t lh,  
                             ldi_callback_id_t id);
```

**Interface Level** Obsolete

**Parameters** `ldi_handle_t lh` Layered handle representing the device for which the event notification is requested.

`ldi_callback_id_t id` Unique system-wide registration ID returned by [ldi\\_add\\_event\\_handler\(9F\)](#) upon successful registration.

**Description** This function is obsolete and is only maintained for compatibility. Use of this function is strongly discouraged. For equivalent functionality provided by new interfaces, see [ldi\\_ev\\_register\\_callbacks\(9F\)](#) and [ldi\\_ev\\_remove\\_callbacks\(9F\)](#).

The `ldi_remove_event_handler()` function removes the callback handler specified by the registration ID (`ldi_callback_id_t`). Upon successful removal, the callback handler is removed from the system and is not invoked at the event occurrence.

**Return Values** `DDI_SUCCESS` Callback handler removed successfully.

`DDI_FAILURE` Failed to remove callback handler.

**Context** This function can be called from user and kernel contexts only.

**See Also** [ldi\\_ev\\_register\\_callbacks\(9F\)](#), [ldi\\_ev\\_remove\\_callbacks\(9F\)](#),  
[ldi\\_add\\_event\\_handler\(9F\)](#), [ldi\\_get\\_eventcookie\(9F\)](#)

*Writing Device Drivers*

- 
- Name** ldi\_strategy – Device strategy request
- Synopsis** #include <sys/sunlidi.h>
- ```
int ldi_strategy(ldi_handle_t lh, struct buf *bp);
```
- Parameters** *lh* Layered handle.  
*bp* Pointer to the buf(9S) structure.
- Description** The `ldi_strategy()` function passes a strategy request to the device entry point for the device specified by the layered handle. This operation is supported for block devices.
- Return Values** The `ldi_strategy()` function returns 0 if the strategy request has been passed on to the target device. Other possible return values are:
- EINVAL Invalid input parameters.
  - ENOTSUP Operation is not supported for this device.
- Once the request has been passed on to the target devices strategy entry point, any further errors will be reported by [bioerror\(9F\)](#) and [biodone\(9F\)](#). See the [strategy\(9E\)](#) entry point for more information.
- Context** This function may be called from user or kernel context.

**Name** linkb – concatenate two message blocks

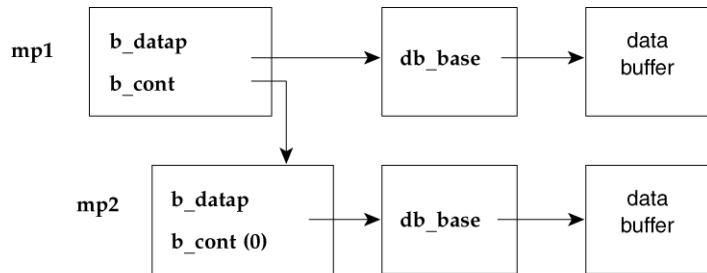
**Synopsis** #include <sys/stream.h>

```
void linkb(mblk_t *mp1, mblk_t *mp2);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Description** The linkb() function creates a new message by adding mp2 to the tail of mp1. The continuation pointer, b\_cont, of mp1 is set to point to mp2.

The following figure describes how the linkb(m1, m2) function concatenates two message blocks, mp1 and mp2:



**linkb(mp1, mp2);**

**Parameters** *mp1* The message to which *mp2* is to be added. mblk\_t is an instance of the msgb(9S) structure.

*mp2* The message to be added.

**Context** The linkb() function can be called from user, interrupt, or kernel context.

**Examples** See dupb(9F) for an example that uses linkb().

**See Also** dupb(9F), unlinkb(9F), msgb(9S)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** list\_create, list\_destroy, list\_insert\_after, list\_insert\_before, list\_insert\_head, list\_insert\_tail, list\_remove, list\_remove\_head, list\_remove\_tail, list\_head, list\_tail, list\_next, list\_prev, list\_is\_empty, list\_link\_init, list\_link\_active, list\_move\_tail, list\_link\_replace – list functions

**Synopsis** #include <sys/list.h>

```
void list_create(list_t * list, size_t size, size_t offset);
void list_destroy(list_t * list);
void list_insert_after(list_t * list, void *reference_item,
    void *new_item);
void list_insert_before(list_t * list, void *reference_item,
    void *new_item);
void list_insert_head(list_t * list, void *new_item);
void list_insert_tail(list_t * list, void *new_item);
void list_remove(list_t * list, void *item);
void *list_remove_head(list_t * list);
void *list_remove_tail(list_t * list);
void *list_head(list_t * list);
void *list_tail(list_t * list);
void *list_next(list_t * list, void *reference_item);
void *list_prev(list_t * list, void *reference_item);
int list_is_empty(list_t * list);
void list_link_init(list_node_t *node);
int list_link_active(list_node_t *node);
void list_move_tail(list_t *dst, list_t *src);
void list_link_replace(list_node_t *node1, list_node_t *node2);
```

**Description** The list\_create() function initializes a new list. The driver supplies the storage for the list handle, the size of an individual element, and the offset of a list\_node\_t within the element to use for the links of the list.

The list\_destroy() function destroys the list handle, including freeing any resources that may have been internally allocated for the list. The list must be empty when this function is called.

The list\_insert\_after() and list\_insert\_before() functions insert new\_item into the linked list at a location after or before the reference item, which must already be on the list.

The `list_insert_head()` and `list_insert_tail()` functions insert the *new\_item* on the list at either the head or tail of the list. (The head is the first item, the tail is the last item).

The `list_remove()` function removes the item from the list.

The `list_remove_head()` and `list_remove_tail()` functions remove the head (first) or tail (last) item from the list. The item removed is returned to the caller. If the list is empty when these functions are called, then no change is made and `NULL` is returned to the caller.

The `list_head()` and `list_tail()` functions simply return the head (first) or tail (last) item on the list. `NULL` is returned if the list is empty.

The `list_next()` and `list_prev()` functions return the next or previous item in the list, relative to the named reference item which must be linked on the list.

The `list_is_empty()` function returns 0 if the list has items in it, or non-zero otherwise.

The `list_link_init()` function initializes the `list_node_t`. It is functionally equivalent to `bzero(node, sizeof(*node));`

The `list_link_active()` function returns non-zero if the node is on an active list.

The `list_move_tail()` function is used to append the items on the *src* list to the end of the *dst* list. It is mandatory that the two lists were initialized using identical size and offset parameters. Upon completion, the *src* list will be empty.

The `list_link_replace()` function swaps two items on a list. Note that the items need not be on the same list, but extreme care must be used to ensure that both lists are protected from concurrent accesses and that the lists were initialized with identical size and offset parameters.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#)



**Name** mac, mac\_alloc, mac\_free, mac\_register, mac\_unregister, mac\_tx\_update, mac\_link\_update, mac\_rx, mac\_init\_ops, mac\_fini\_ops – MAC driver service routines

**Synopsis** #include <sys/mac\_provider.h>

```
mac_register_t *mac_alloc(uint_t version);
void mac_free(mac_register_t *mregp);
int mac_register(mac_register_t *mregp, mac_handle_t *mhp);
int mac_unregister(mac_handle_t mh);
void mac_tx_update(mac_handle_t mh);
void mac_link_update(mac_handle_t mh, link_state_t new_state);
void mac_rx(mac_handle_t mh, void *reserved, mblk_t *mp_chain);
void mac_init_ops(struct dev_ops *ops, const char *name);
void mac_fini_ops(struct dev_ops *ops);
```

|                   |                  |                                                      |
|-------------------|------------------|------------------------------------------------------|
| <b>Parameters</b> | <i>version</i>   | MAC version                                          |
|                   | <i>mh</i>        | MAC handle                                           |
|                   | <i>mhp</i>       | pointer to a MAC handle                              |
|                   | <i>mregp</i>     | pointer to a mac_register_t(9S) structure            |
|                   | <i>reserved</i>  | reserved argument                                    |
|                   | <i>mp_chain</i>  | chain of message blocks containing a received packet |
|                   | <i>new_state</i> | media link state                                     |
|                   | <i>ops</i>       | device operations structure                          |
|                   | <i>name</i>      | device driver name                                   |

**Interface Level** Solaris architecture specific (Solaris DDI)

**Description** The `mac_alloc()` function allocates a new `mac_register(9S)` structure and returns a pointer to it. The allocated structure may contain some MAC-private elements. These private elements are initialized by the MAC layer before `mac_alloc()` returns, and the other elements of the structure are initialized to 0. The device driver must initialize the structure members as described by `mac_register` before passing a pointer to the structure to `mac_register`. The version argument should be set to `MAC_VERSION_V1`.

The `mac_free()` function frees a `mac_register` structure previously allocated by `mac_alloc()`.

The `mac_register()` function is called from the device driver's [attach\(9E\)](#) entry point, and is used to register the MAC-based device driver with the MAC layer. The `mac_register()` entry point is passed an instance of the `mac_register` structure previously allocated by `mac_alloc()`.

On success, `mac_register()` returns 0 and sets `mhp` to point to a new MAC handle corresponding to the new MAC instance. This MAC handle is subsequently passed by the driver to the framework as an argument to other MAC routines such as the ones described here. The `attach()` entry point of the driver should return `DDI_SUCCESS` in this case. On failure, `mac_register()` returns a non-zero error as described by [Intro\(2\)](#). The `attach()` entry point of the driver should return `DDI_FAILURE` in this case.

The `mac_unregister()` function is called by the driver from its [detach\(9E\)](#) entry point to unregister the instance from the MAC layer. It should pass the MAC handle which was previously obtained from `mac_register()`. `mac_unregister()` returns 0 on success, in which case the driver's `detach()` entry point should return `DDI_SUCCESS`. `mac_unregister()` returns a non-zero error as described by [Intro\(2\)](#) on failure. In this case the driver's `detach()` entry point should return `DDI_FAILURE`.

The `mac_tx_update()` function should be called by the driver to reschedule stalled outbound packets. Whenever the driver's [mc\\_tx\(9E\)](#) has returned a non-empty chain of packets, it must later call `mac_tx_update()` to inform the MAC layer that it should retry the packets that previously could not be sent. `mac_tx_update()` should be called as soon as possible after resources are again available, to ensure that MAC resumes passing outbound packets to the driver's `mc_tx()` entry point.

The `mac_link_update()` function is called by the device driver to notify the MAC layer of changes in the media link state. The `new_state` argument must be set to one of the following:

|                                 |                            |
|---------------------------------|----------------------------|
| <code>LINK_STATE_UP</code>      | The media link is up.      |
| <code>LINK_STATE_DOWN</code>    | The media link is down.    |
| <code>LINK_STATE_UNKNOWN</code> | The media link is unknown. |

The `mac_rx()` function is called by the driver's interrupt handler to pass a chain of one or more packets to the MAC layer. Packets of a chain are linked with the `b_next` pointer. The driver should avoid holding mutex or other locks during the call to `mac_rx()`. In particular, locks that could be taken by a transmit thread may not be held during a call to `mac_rx()`.

The `mac_init_ops()` function must be invoked from the [\\_init\(9E\)](#) entry point of the device driver before a call to [mod\\_install\(9F\)](#). It is passed a pointer to the device driver's operations structure, and the name of the device driver.

The `mac_fini_ops()` function must be called from [\\_fini\(9E\)](#) before the driver is unloaded after invoking [mod\\_remove\(9F\)](#), or before returning from `_init()` in the case of an error returned by `mod_install()`.

**Return Values** The `mac_alloc()` function returns a pointer to a new `mac_register(9S)` structure.

The `mac_register()` and `mac_unregister()` functions return a non-zero error, as defined by [Intro\(2\)](#).

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/header   |
| Interface Stability | Committed       |

**See Also** [Intro\(2\)](#), [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [\\_fini\(9E\)](#), [\\_init\(9E\)](#), [mac\(9E\)](#), [mc\\_tx\(9E\)](#), [mod\\_install\(9F\)](#), [mod\\_remove\(9F\)](#), [dev\\_ops\(9S\)](#), [mac\\_register\(9S\)](#)

**Name** mac\_hcksum\_get, mac\_hcksum\_set – hardware checksumming offload routines

**Synopsis** #include <sys/mac\_provider.h>

```
void mac_hcksum_get(mblk_t *mp, uint32_t *start, uint32_t *stuff,
    uint32_t *end, uint32_t *value, uint32_t *flags);
```

```
void mac_hcksum_set(mblk_t *mp, uint32_t start, uint32_t stuff,
    uint32_t end, uint32_t value, uint32_t flags);
```

**Parameters**

|              |                                                                                                                                                     |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>mp</i>    | pointer to a message block                                                                                                                          |
| <i>start</i> | offset, in bytes, from the start of the IP header to the start of the checksum span                                                                 |
| <i>end</i>   | offset, in bytes, from the start of the IP header to the end of the checksum span                                                                   |
| <i>stuff</i> | offset, in bytes, from the start of the IP header to the checksum field in the protocol header                                                      |
| <i>value</i> | hardware computed checksum value                                                                                                                    |
| <i>flags</i> | per-packet flags indicating the hardware checksumming to be performed on outbound packets, or the hardware checksumming performed on inbound packet |

**Interface Level** Solaris architecture specific (Solaris DDI)

**Description** Hardware checksumming allows the checksum computation to be offloaded to the network device hardware for lower CPU utilization. Hardware checksumming capabilities are advertised from the driver's [mc\\_getcapab\(9E\)](#) entry point. The description of [mc\\_getcapab\(\)](#) also includes more information about the expected behavior of drivers for full and partial checksumming offload.

For received traffic, the hardware can enable hardware checksumming, and the network stack will know how to handle packets for which checksum computation or verification has been performed. The `mac_hcksum_set()` function can be used by a device driver to associate information related to the hardware checksumming performed on the packet.

The *flags* argument can be a combination of the following:

|                      |                                                                                                                                              |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| HCK_FULLCKSUM        | The full checksum was computed, and is passed through the <i>value</i> argument.                                                             |
| HCK_FULLCKSUM_OK     | The full checksum was verified in hardware and is correct.                                                                                   |
| HCK_PARTIALCKSUM     | Partial checksum computed and passed through the <i>value</i> argument. The <i>start</i> and <i>end</i> arguments specify the checksum span. |
| HCK_IPV4_HDRCKSUM_OK | IP header checksum was verified in hardware and is correct.                                                                                  |

HCK\_PARTIALCKSUM is mutually exclusive with the HCK\_FULLCKSUM and HCK\_FULLCKSUM\_OK flags.

For outbound packets, hardware checksumming capabilities are queried via the `mc_getcapab()` entry point. Hardware checksumming is enabled by the network stack based on the `MAC_CAPAB_HCKSUM` capability. A device driver that advertised support for this capability can subsequently receive outbound packets that may not have a fully computed checksum. It is the responsibility of the driver to invoke `mac_hcksum_get()` to retrieve the per-packet hardware checksumming metadata.

HCK\_FULLCKSUM            Compute full checksum for this packet.

HCK\_PARTIALCKSUM        Compute partial 1's complement checksum based on the *start*, *stuff*, and offset.

HCK\_IPV4\_HDRCKSUM       Compute the IP header checksum.

HCK\_PARTIALCKSUM is mutually exclusive with HCK\_FULLCKSUM.

The flags HCK\_FULLCKSUM, HCK\_FULLCKSUM\_OK, and HCK\_PARTIALCKSUM are used for both IPv4 and IPv6 packets. The driver advertises support for IPv4 and/or IPv6 full checksumming during capability negotiation. See [mc\\_getcapab\(9E\)](#).

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/header   |
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [mac\(9E\)](#)

**Name** mac\_ Iso\_get – LSO routine

**Synopsis** #include <sys/mac\_provider.h>

```
void mac_ Iso_get(mblk_t *mp, uint32_t *mss, uint32_t *flags);
```

**Description** LSO (Large Segment Offload, or Large Send Offload) allows the network stack to send larger buffers to a device driver. These large buffers can then be segmented in hardware, allowing for reduced CPU utilization, PCI overhead, and reduced buffer management costs.

LSO is enabled only for device driver instances that advertise support for the MAC\_CAPAB\_LSO capability through the [mc\\_getcapab\(9E\)](#) entry point.

Once a device driver advertises the LSO capability, it must use the `mac_ Iso_get()` entry point to query whether LSO must be performed on the packet. The following values for the *flags* argument are supported:

**HW\_LSO** When set, this flag indicates that LSO is enabled for that packet. The maximum segment size (MSS) to be used during segmentation of the large segment is returned through the location pointed to by *mss*.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTETYPE       | ATTRIBUTEVALUE |
|---------------------|----------------|
| Availability        | system/header  |
| Interface Stability | Committed      |

**See Also** [attributes\(5\)](#), [mac\(9E\)](#)

**Name** `mac_prop_info_set_perm`, `mac_prop_info_set_default_uint8`,  
`mac_prop_info_set_default_str`, `mac_prop_info_set_default_link_flowctrl`,  
`mac_prop_info_set_range_uint32` – set property information

**Synopsis** `#include <sys/mac_provider.h>`

```
void mac_prop_info_set_perm(mac_prop_info_handle_t ph,
    uint8_t perm);

void mac_prop_info_set_default_uint8(mac_prop_info_handle_t ph,
    uint8_t val);

void mac_prop_info_set_default_str(mac_prop_info_handle_t ph,
    const char *str);

void mac_prop_info_set_default_link_flowctrl(
    mac_prop_info_handle_t ph, link_flowctrl_t val);

void mac_prop_info_set_range_uint32(mac_prop_info_handle_t ph,
    uint32_t min, uint32_t max);
```

**Description** The entry points described here are invoked from a device driver's `mc_propinfo(9E)` entry point to associate information such as default values, permissions, or allowed value ranges.

Each one of these functions takes as first argument the property information handle which is passed to `mc_propinfo()` as argument.

The `mac_prop_info_set_perm()` function specifies the property of the property. The permission is passed through the `perm` argument and can be set to one of the following values.

|                                  |                                       |
|----------------------------------|---------------------------------------|
| <code>MAC_PROP_PERM_READ</code>  | The property is read-only.            |
| <code>MAC_PROP_PERM_WRITE</code> | The property is write-only.           |
| <code>MAC_PROP_PERM_RW</code>    | The property can be read and written. |

The driver is not required to call `mac_prop_info_set_perm()` for every property. If the driver does not call that function for a specific property, the framework will assume that the property has read and write permissions, corresponding to `MAC_PROP_PERM_RW`.

The `mac_prop_info_set_default_uint8()`, `mac_prop_info_set_default_str()`, and `mac_prop_info_set_default_link_flowctrl()` functions are used to associate a default value with a specific property.

The `mac_prop_info_set_range_uint32()` function is used by a driver to associate an allowed range of values for a specific property. The range is defined by the `min` and `max` arguments passed by the device driver. This function can be called successively to specify multiple non-overlapping ranges.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/header   |
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [mac\(9E\)](#)



**Name** makecom, makecom\_g0, makecom\_g0\_s, makecom\_g1, makecom\_g5 – make a packet for SCSI commands

**Synopsis** #include <sys/scsi/scsi.h>

```
void makecom_g0(struct scsi_pkt *pkt, struct scsi_device *devp,
               int flag, int cmd, int addr, int cnt);

void makecom_g0_s(struct scsi_pkt *pkt, struct scsi_device *devp,
                 int flag, int cmd, int cnt, int fixbit);

void makecom_g1(struct scsi_pkt *pkt, struct scsi_device *devp,
               int flag, int cmd, int addr, int cnt);

void makecom_g5(struct scsi_pkt *pkt, struct scsi_device *devp,
               int flag, int cmd, int addr, int cnt);
```

**Interface Level** These interfaces are obsolete. [scsi\\_setup\\_cdb\(9F\)](#) should be used instead.

**Parameters**

- pkt* Pointer to an allocated [scsi\\_pkt\(9S\)](#) structure.
- devp* Pointer to the target's [scsi\\_device\(9S\)](#) structure.
- flag* Flags for the `pkt_flags` member.
- cmd* First byte of a group 0 or 1 or 5 SCSI CDB.
- addr* Pointer to the location of the data.
- cnt* Data transfer length in units defined by the SCSI device type. For sequential devices *cnt* is the number of bytes. For block devices, *cnt* is the number of blocks.
- fixbit* Fixed bit in sequential access device commands.

**Description** The `makecom` functions initialize a packet with the specified command descriptor block, *devp* and transport flags. The `pkt_address`, `pkt_flags`, and the command descriptor block pointed to by `pkt_cdbp` are initialized using the remaining arguments. Target drivers may use `makecom_g0()` for Group 0 commands (except for sequential access devices), or `makecom_g0_s()` for Group 0 commands for sequential access devices, or `makecom_g1()` for Group 1 commands, or `makecom_g5()` for Group 5 commands. *fixbit* is used by sequential access devices for accessing fixed block sizes and sets the tag portion of the SCSI CDB.

**Context** These functions can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `makecom` Functions

```
if (blkno >= (1<<20)) {
    makecom_g1(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE_G1,
              (int) blkno, nblk);
} else {
    makecom_g0(pkt, SD_SCSI_DEVP, pflag, SCMD_WRITE,
              (int) blkno, nblk);
```

EXAMPLE 1 Using makecom Functions (Continued)

```
}
```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [scsi\\_setup\\_cdb\(9F\)](#), [scsi\\_device\(9S\)](#), [scsi\\_pkt\(9S\)](#)

*ANSI Small Computer System Interface-2 (SCSI-2)*

*Writing Device Drivers*

**Notes** The `makecom_g0()`, `makecom_g0_s()`, `makecom_g1()`, and `makecom_g5()` functions are obsolete and will be discontinued in a future release. These functions have been replaced by the `scsi_setup_cdb()` function. See [scsi\\_setup\\_cdb\(9F\)](#).

**Name** makedevice – make device number from major and minor numbers

**Synopsis**

```
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/ddi.h>
```

```
dev_t makedevice(major_t majnum, minor_t minnum);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *majnum* Major device number.

*minnum* Minor device number.

**Description** The `makedevice()` function creates a device number from a major and minor device number. `makedevice()` should be used to create device numbers so the driver will port easily to releases that treat device numbers differently.

**Return Values** The device number, containing both the major number and the minor number, is returned. No validation of the major or minor numbers is performed.

**Context** The `makedevice()` function can be called from user, interrupt, or kernel context.

**See Also** [getmajor\(9F\)](#), [getminor\(9F\)](#)

**Name** max – return the larger of two integers

**Synopsis** `#include <sys/ddi.h>`

```
int max(int int1, int int2);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *int1* The first integer.

*int2* The second integer.

**Description** The `max()` function compares two signed integers and returns the larger of the two.

**Return Values** The larger of the two numbers.

**Context** The `max()` function can be called from user, interrupt, or kernel context.

**See Also** [min\(9F\)](#)

*Writing Device Drivers*

**Name** MBLKHEAD, MBLKIN, MBLKL, MBLKSIZE, MBLKTAIL – Message block utility macros

**Synopsis** #include <sys/stream.h>  
#include <sys/strsun.h>

```
int MBLKHEAD(mblk_t *mp);
int MBLKTAIL(mblk_t *mp);
int MBLKSIZE(mblk_t *mp);
int MBLKL(mblk_t *mp);
int MBLKIN(mblk_t *mp, int offset, int len);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* Message to be examined.  
*offset* Offset from *mp->b\_rptr* from which to start examining.  
*len* Number of bytes to examine.

**Description** The MBLKHEAD ( ) macro calculates the number of bytes between the first byte and the first unread byte of the message block, that is: *mp->b\_rptr - mp->b\_datap->db\_base*.

The MBLKTAIL ( ) macro calculates the number of bytes between the first unwritten byte and the last byte of the message block, that is: *mp->b\_datap->db\_lim - mp->b\_wptr*.

The MBLKSIZE ( ) macros calculates the total size of the message block, that is: *mp->b\_datap->db\_lim - mp->b\_datap->db\_base*.

The MBLKL ( ) macro calculates the length of the message block, that is: *mp->b\_wptr - mp->b\_rptr*.

The MBLKIN ( ) macro checks whether the byte range specified by *offset* and *len* resides entirely within the message block.

**Return Values** The MBLKHEAD ( ), MBLKTAIL ( ), MBLKL ( ) and MBLKSIZE ( ) functions all return the appropriate byte count, as specified above. MBLKIN ( ) returns non-zero if the check succeeds, or zero if it fails.

**Context** These functions can be called from user, kernel or interrupt context.

**Notes** These macros may evaluate any of their arguments more than once. This precludes passing arguments with side effects.

These macros assume the message itself is well formed, that is: *mp->b\_datap->db\_base <= mp->b\_rptr <= mp->b\_wptr <= mp->b\_datap->db\_lim*.

**See Also** [msgb\(9S\)](#)

*STREAMS Programming Guide*

**Name** mcopyin – Convert an M\_IOCTL or M\_IOCTLDATA message to an M\_COPYIN

**Synopsis** #include <sys/stream.h>  
#include <sys/strsun.h>

```
void mcopyin(mblk_t *mp, void *private, size_t size,
             void * useraddr);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* M\_IOCTL or M\_IOCTLDATA message.  
*private* Value to which the *cq\_private* field of [copyreq\(9S\)](#) is set.  
*size* Value to which the *cq\_size* field of [copyreq\(9S\)](#) is set.  
*useraddr* Optionally, the value to which the *cq\_addr* field of [copyreq\(9S\)](#) is set.

**Description** The `mcopyin()` function converts an M\_IOCTL or M\_IOCTLDATA message into an M\_COPYIN message using the supplied arguments.

To convert the message, `mcopyin()` changes the message type to M\_COPYIN, and its payload from a [iocblk\(9S\)](#) to a [copyreq\(9S\)](#). Since the [iocblk\(9S\)](#) and [copyreq\(9S\)](#) are designed to overlay one another, the only fields which must be updated are *cq\_private*, *cq\_size*, and *cq\_addr*, which are set to the supplied values. If *useraddr* is passed as NULL, *mp* must be a transparent M\_IOCTL, and *cq\_addr* is assigned the pointer-sized quantity found at `mp->b_cont->b_rptr`.

Any trailing message blocks associated with *mp* are freed.

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**See Also** [mcopyout\(9F\)](#), [copyreq\(9S\)](#)

*STREAMS Programming Guide*

**Name** mcopymsg – Copy message contents into a buffer and free message

**Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>
```

```
void mcopymsg(mblk_t *mp, void *buf);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* Message to be copied.

*buf* Buffer in which to copy.

**Description** The `mcopymsg()` function copies the contents of the specified message into the specified buffer. If the message consists of more than a single message block, the contents of each message block are placed consecutively into the buffer. After copying the message contents to *buf*, `mcopymsg()` frees the message *mp*.

The provided buffer must be large enough to accommodate the message. If the buffer is not large enough, the results are unspecified. The [msgsize\(9F\)](#) function can be used to calculate the total size of the message beforehand.

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**See Also** [freemsg\(9F\)](#), [msgsize\(9F\)](#)

*STREAMS Programming Guide*



- 
- Name** mcopyout – Convert an M\_IOCTL or M\_IOCTLDATA message to an M\_COPYOUT
- Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>

void mcopyout(mblk_t *mp, void *private, size_t size, void *useraddr,
              mblk_t *dp);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters**
- mp* M\_IOCTL or M\_IOCTLDATA message.
  - private* Value to set the cq\_private field of the [copyreq\(9S\)](#) to.
  - size* Value to set the cq\_size field of the [copyreq\(9S\)](#) to.
  - useraddr* Optionally, the value to set the cq\_addr field of the [copyreq\(9S\)](#) to.
  - dp* Optionally, the payload to copy out.
- Description** The mcopyout () function converts an M\_IOCTL or M\_IOCTLDATA message into an M\_COPYOUT message using the supplied arguments.
- To convert the message, mcopyout () changes the message type to M\_COPYOUT, and its payload from a [iocblk\(9S\)](#) to a [copyreq\(9S\)](#). Since the [iocblk\(9S\)](#) and [copyreq\(9S\)](#) are designed to overlay one another, the only fields which must be updated are *cq\_private*, *cq\_size*, and *cq\_addr*, which are set to the supplied values. If *useraddr* is passed as NULL, the M\_IOCTL must be transparent and *cq\_addr* is assigned the pointer-sized quantity found at *mp->b\_cont->b\_rptr*.
- If *dp* is not NULL, any trailing message blocks associated with *mp* are freed, *mp->b\_cont* is reset to *dp* and *dp->b\_wptr* is set to *dp->b\_rptr + size*. Otherwise, any trailing message blocks are unaffected.
- Return Values** None.
- Context** This function can be called from user, kernel or interrupt context.
- See Also** [mcopyin\(9F\)](#), [copyreq\(9S\)](#), [iocblk\(9S\)](#)
- STREAMS Programming Guide*

**Name** membar\_ops, membar\_enter, membar\_exit, membar\_producer, membar\_consumer – memory access synchronization barrier operations

**Synopsis** #include <sys/atomic.h>

```
void membar_enter(void);
void membar_exit(void);
void membar_producer(void);
void membar_consumer(void);
```

**Description** The `membar_enter()` function is a generic memory barrier used during lock entry. It is placed after the memory operation that acquires the lock to guarantee that the lock protects its data. No stores from after the memory barrier will reach visibility and no loads from after the barrier will be resolved before the lock acquisition reaches global visibility.

The `membar_exit()` function is a generic memory barrier used during lock exit. It is placed before the memory operation that releases the lock to guarantee that the lock protects its data. All loads and stores issued before the barrier will be resolved before the subsequent lock update reaches visibility.

The `membar_enter()` and `membar_exit()` functions are used together to allow regions of code to be in relaxed store order and then ensure that the load or store order is maintained at a higher level. They are useful in the implementation of mutex exclusion locks.

The `membar_producer()` function arranges for all stores issued before this point in the code to reach global visibility before any stores that follow. This is useful in producer modules that update a data item, then set a flag that it is available. The memory barrier guarantees that the available flag is not visible earlier than the updated data, thereby imposing store ordering.

The `membar_consumer()` function arranges for all loads issued before this point in the code to be completed before any subsequent loads. This is useful in consumer modules that check if data is available and read the data. The memory barrier guarantees that the data is not sampled until after the available flag has been seen, thereby imposing load ordering.

**Return Values** No values are returned.

**Errors** No errors are defined.

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [atomic\\_add\(9F\)](#), [atomic\\_and\(9F\)](#), [atomic\\_bits\(9F\)](#), [atomic\\_cas\(9F\)](#), [atomic\\_dec\(9F\)](#), [atomic\\_inc\(9F\)](#), [atomic\\_ops\(9F\)](#), [atomic\\_or\(9F\)](#), [atomic\\_swap\(9F\)](#), [attributes\(5\)](#), [atomic\\_ops\(3C\)](#)

**Notes** Atomic instructions (see [atomic\\_ops\(9F\)](#)) ensure global visibility of atomically-modified variables on completion. In a relaxed store order system, this does not guarantee that the visibility of other variables will be synchronized with the completion of the atomic instruction. If such synchronization is required, memory barrier instructions must be used.

**Name** memchr, memcmp, memcpy, memmove, memset – Memory operations

**Synopsis** #include <sys/ddi.h>

```
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dst* Pointers to character strings.  
*n* Count of characters to be copied.  
*s1, s2* Pointers to character strings.

**Description** These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The `memchr()` function returns a pointer to the first occurrence of *c* (converted to an unsigned char) in the first *n* bytes (each interpreted as an unsigned char) of memory area *s*, or a null pointer if *c* does not occur.

The `memcmp()` function compares its arguments, looking at the first *n* bytes (each interpreted as an unsigned char), and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2* when taken to be unsigned characters.

The `memcpy()` function copies *n* bytes from memory area *s2* to *s1*. It returns *s1*. If copying takes place between objects that overlap, the behavior is undefined.

The `memmove()` function copies *n* bytes from memory area *s2* to memory area *s1*. Copying between objects that overlap will take place correctly. It returns *s1*.

The `memset()` function sets the first *n* bytes in memory area *s* to the value of *c* (converted to an unsigned char). It returns *s*.

**Usage** Using `memcpy()` might be faster than using `memmove()` if the application knows that the objects being copied do not overlap.

**Context** These functions can be called from user, interrupt, or kernel context.

**See Also** [bcopy\(9F\)](#), [ddi\\_copyin\(9F\)](#), [strcpy\(9F\)](#)

*Writing Device Drivers*

**Name** merror – Send an M\_ERROR message upstream

**Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>
```

```
void merror(queue_t *wq, mblk_t *mp, int error);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *wq* Write queue associated with the read queue to send the M\_ERROR on.

*mp* Optionally, a STREAMS message to convert to an M\_ERROR.

*error* Error code to include in the M\_ERROR message.

**Description** The `merror()` function constructs an M\_ERROR message, and sends the resulting message upstream.

If *mp* is NULL, `merror()` allocates a one-byte M\_ERROR message. If *mp* is non-NULL, `merror()` attempts to convert the passed-in message to an M\_ERROR. However, if the passed-in message has more than one reference (see [dupmsg\(9F\)](#)), or if it is of zero length, it is freed and a new message is allocated.

If the allocation or conversion fails, `merror()` silently fails. Otherwise, the resulting one-byte data block is assigned the specified error code and sent upstream.

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**Notes** Callers must not hold any locks across an `merror()` that can be acquired as part of [put\(9E\)](#) processing.

**See Also** [put\(9E\)](#), [dupmsg\(9F\)](#)

*STREAMS Programming Guide*

- 
- Name** mexchange – Exchange one message for another
- Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>

mblk_t *mexchange(queue_t *wq, mblk_t *mp, size_t size,
    uchar_t type, int32_t primtype);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters**
- wq*            Optionally, write queue associated with the read queue to be used on failure (see below).
  - mp*            Optionally, the message to exchange.
  - size*           Size of the returned message.
  - type*           Type of the returned message.
  - primtype*      Optionally, a 4 byte value to store at the beginning of the returned message.
- Description** The `mexchange()` function exchanges the passed in message for another message of the specified *size* and *type*.
- If *mp* is not NULL, is of at least *size* bytes, and has only one reference (see [dupmsg\(9F\)](#)), *mp* is converted to be of the specified *size* and *type*. Otherwise, a new message of the specified *size* and *type* is allocated. If allocation fails, and *wq* is not NULL, [merror\(9F\)](#) attempts to send an error to the stream head.
- Finally, if *primtype* is not -1 and *size* is at least 4 bytes, the first 4 bytes are assigned to be *primtype*. This is chiefly useful for STREAMS-based protocols such as DLPI and TPI which store the protocol message type in the first 4 bytes of each message.
- Return Values** A pointer to the requested message is returned on success. NULL is returned on failure.
- Context** This function can be called from user, kernel or interrupt context.
- See Also** [dupmsg\(9F\)](#), [merror\(9F\)](#)
- STREAMS Programming Guide*

**Name** min – return the lesser of two integers

**Synopsis** `#include <sys/ddi.h>`

```
int min(int int1, int int2);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *int1* The first integer.

*int2* The second integer.

**Description** The `min()` function compares two signed integers and returns the lesser of the two.

**Return Values** The lesser of the two integers.

**Context** The `min()` function can be called from user, interrupt, or kernel context.

**See Also** [max\(9F\)](#)

*Writing Device Drivers*



**Name** mioc2ack – Convert an M\_IOCTL message to an M\_IOCACK message

**Synopsis** #include <sys/stream.h>  
#include <sys/strsun.h>

```
void mioc2ack(mblk_t *mp, mblk_t *dp, size_t count, int rval);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* M\_IOCTL message.  
*dp* Payload to associate with M\_IOCACK message.  
*count* Value to set the *ioc\_count* of the [iocblk\(9S\)](#) to.  
*rval* Value to set the *ioc\_rval* of the [iocblk\(9S\)](#) to.

**Description** The `mioc2ack()` function converts an M\_IOCTL message into an M\_IOCACK message using the supplied arguments.

To convert the message, `mioc2ack()` changes the message type to M\_IOCACK, sets the *ioc\_count* and *ioc\_rval* members of the [iocblk\(9S\)](#) associated with *mp* to the passed-in values, and clears the *ioc\_error* field. Further, it frees any message blocks chained off of *mp->b\_cont* and resets *mp->b\_cont* to *dp*. Finally, if *dp* is not NULL, `mioc2ack()` resets *dp->b\_wptr* to be *dp->b\_rptr + count* (that is, it sets *dp* to be exactly *count* bytes in length).

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**See Also** [miocack\(9F\)](#), [miocnak\(9F\)](#), [iocblk\(9S\)](#)

*STREAMS Programming Guide*

**Name** miocack – Positively acknowledge an M\_IOCTL message

**Synopsis** #include <sys/stream.h>  
#include <sys/strsun.h>

```
void miocack(queue_t *wq, mblk_t *mp, intcount, int rval);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *wq* Write queue associated with the read queue to send the M\_IOCACK on.  
*mp* M\_IOCTL message.  
*count* Value to set the `ioc_count` of the [iocblk\(9S\)](#) to.  
*rval* Value to set the `ioc_rval` of the [iocblk\(9S\)](#) to.

**Description** The `miocack()` function converts an M\_IOCTL message into a M\_IOCACK message and sends the resulting message upstream.

To convert the message, `miocack()` changes the message type to M\_IOCACK, sets the 'ioc\_count' and 'ioc\_rval' members of the [iocblk\(9S\)](#) associated with *mp* to the passed-in values, and clears the 'ioc\_error' field. If the caller specifies a non-zero value for count, it is expected that the caller has already set 'mp->b\_cont' field to point to a message block with a length of at least *count* bytes.

Callers that only need to perform the message conversion, or need to perform additional steps between the conversion and the sending of the M\_IOCACK should use [mioc2ack\(9F\)](#).

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**Notes** Callers must not hold any locks across a `miocack()` that can be acquired as part of [put\(9E\)](#) processing.

**See Also** [mioc2ack\(9F\)](#), [put\(9E\)](#), [iocblk\(9S\)](#)

*STREAMS Programming Guide*

**Name** miocnak – Negatively acknowledge an M\_IOCTL message

**Synopsis** #include <sys/stream.h>  
#include <sys/strsun.h>

```
void miocnak(queue_t *wq, mblk_t *mp, int count, int error);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *wq* Write queue associated with the read queue to send the M\_IOCNAK on.  
*mp* M\_IOCTL message.  
*count* Value to set the *ioc\_count* of the [iocblk\(9S\)](#) to.  
*error* Value to set the *ioc\_error* of the [iocblk\(9S\)](#) to.

**Description** The `miocnak()` function converts an M\_IOCTL message into an M\_IOCNAK message and sends the resulting message upstream.

To convert the message, `miocnak()` changes the message type to M\_IOCNAK, sets the *ioc\_count* and *ioc\_error* members of the [iocblk\(9S\)](#) associated with *mp* to the passed-in values, and clears the *ioc\_rval* field. Since payloads cannot currently be associated with M\_IOCNAK messages, *count* must always be zero. If *error* is passed as zero, EINVAL is assumed.

**Return Values** None.

**Context** This function can be called from user, kernel or interrupt context.

**Notes** Callers must not hold any locks across a `miocnak()` that can be acquired as part of [put\(9E\)](#) processing.

**See Also** [mioc2ack\(9F\)](#), [miocack\(9F\)](#), [put\(9E\)](#), [iocblk\(9S\)](#)

*STREAMS Programming Guide*

**Name** miocpullup – Prepare the payload of an M\_IOCTL message for access

**Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>
```

```
int miocpullup(mblk_t *mp, size_t size);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *mp* M\_IOCTL message.  
*size* Number of bytes to prepare.

**Description** The `miocpullup()` function prepares the payload of the specified M\_IOCTL message for access by ensuring that it consists of at least *size* bytes of data.

If the M\_IOCTL message is transparent, or its total payload is less than *size* bytes, an error is returned. Otherwise, the payload is concatenated as necessary to provide contiguous access to at least *size* bytes of data. As a special case, if *size* is zero, `miocpullup()` returns successfully, even if no payload exists.

**Return Values** Zero is returned on success. Otherwise an `errno` value is returned indicating the problem.

**Context** This function can be called from user, kernel or interrupt context.

**See Also** *STREAMS Programming Guide*

**Name** mkiocb – allocates a STREAMS ioctl block for M\_IOCTL messages in the kernel.

**Synopsis** #include <sys/stream.h>

```
mblk_t *mkiocb(uint_t command);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *command* ioctl command for the *ioc\_cmd* field.

**Description** STREAMS modules or drivers might need to issue an ioctl to a lower module or driver. The `mkiocb()` function tries to allocate (using `alloca(9F)`) a STREAMS M\_IOCTL message block (`iocblk(9S)`). Buffer allocation fails only when the system is out of memory. If no buffer is available, the `qbufcall(9F)` function can help a module recover from an allocation failure.

The `mkiocb` function returns a `mblk_t` structure which is large enough to hold any of the ioctl messages (`iocblk(9S)`, `copyreq(9S)` or `copyresp(9S)`), and has the following special properties:

`b_wptr` Set to `b_rptr + sizeof(struct iocblk)`.

`b_cont` Set to NULL..

`b_datap->db_type` Set to M\_IOCTL.

The fields in the `iocblk` structure are initialized as follows:

`ioc_cmd` Set to the command value passed in.

`ioc_id` Set to a unique identifier.

`ioc_cr` Set to point to a credential structure encoding the maximum system privilege and which does not need to be freed in any fashion.

`ioc_count` Set to 0.

`ioc_rval` Set to 0.

`ioc_error` Set to 0.

`ioc_flags` Set to IOC\_NATIVE to reflect that this is native to the running kernel.

**Return Values** Upon success, the `mkiocb()` function returns a pointer to the allocated `mblk_t` of type M\_IOCTL.

On failure, it returns a null pointer.

**Context** The `mkiocb()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 M\_IOCTL Allocation

The first example shows an M\_IOCTL allocation with the ioctl command TEST\_CMD. If the [iocblk\(9S\)](#) cannot be allocated, NULL is returned, indicating an allocation failure (line 5). In line 11, the [putnext\(9F\)](#) function is used to send the message downstream.

```

1 test_function(queue_t *q, test_info_t *testinfo)
2 {
3     mblk_t *mp;
4
5     if ((mp = mkiocb(TEST_CMD)) == NULL)
6         return (0);
7
8     /* save off ioctl ID value */
9     testinfo->xx_iocid = ((struct iocblk *)mp->b_rptr)->ioc_id;
10
11    putnext(q, mp);      /* send message downstream */
12    return (1);
13 }
```

## EXAMPLE 2 The ioctl ID Value

During the read service routine, the ioctl ID value for M\_IOCACK or M\_IOCNAK should equal the ioctl that was previously sent by this module before processing.

```

1 test_lrsrv(queue_t *q)
2 {
3     ...
4
5     switch (DB_TYPE(mp)) {
6     case M_IOCACK:
7     case M_IOCNAK:
8         /* Does this match the ioctl that this module sent */
9         ioc = (struct iocblk*)mp->b_rptr;
10        if (ioc->ioc_id == testinfo->xx_iocid) {
11            /* matches, so process the message */
12            ...
13            freemsg(mp);
14        }
15        break;
16    }
17    ...
18 }
```

## EXAMPLE 3 An iocblk Allocation Which Fails

The next example shows an iocblk allocation which fails. Since the open routine is in user context, the caller may block using [qbufcall\(9F\)](#) until memory is available.

**EXAMPLE 3** An iocblk Allocation Which Fails *(Continued)*

```

1 test_open(queue_t *q, dev_t devp, int oflag, int sflag,
           cred_t *credp)
2 {
3     while ((mp = mkiocb(TEST_IOCTL)) == NULL) {
4         int id;
5
6         id = qbufcall(q, sizeof (union ioctypes), BPRI_HI,
7                     dummy_callback, 0);
8         /* Handle interrupts */
9         if (!qwait_sig(q)) {
10            qunbufcall(q, id);
11            return (EINTR);
12        }
13    }
14    putnext(q, mp);
15 }

```

**See Also** [allocb\(9F\)](#), [putnext\(9F\)](#), [qbufcall\(9F\)](#), [qwait\\_sig\(9F\)](#), [copyreq\(9S\)](#), [copyresp\(9S\)](#), [iocblk\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** It is the module's responsibility to remember the ID value of the `M_IOCTL` that was allocated. This will ensure proper cleanup and ID matching when the `M_IOCACK` or `M_IOCNAK` is received.

**Name** mod\_install, mod\_remove, mod\_info, mod\_modname – add, remove or query a loadable module

**Synopsis** #include <sys/modctl.h>

```
int mod_install(struct modlinkage *modlinkage);
int mod_remove(struct modlinkage *modlinkage);
int mod_info(struct modlinkage *modlinkage,
             struct modinfo *modinfo);
const char *mod_modname(struct modlinkage *modlinkage);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *modlinkage* Pointer to the loadable module's modlinkage structure which describes what type(s) of module elements are included in this loadable module.

*modinfo* Pointer to the modinfo structure passed to [\\_info\(9E\)](#).

**Description** mod\_install() must be called from a module's [\\_init\(9E\)](#) routine.

mod\_remove() must be called from a module's [\\_fini\(9E\)](#) routine.

mod\_info() must be called from a module's [\\_info\(9E\)](#) routine.

mod\_modname() can be called from any module routine.

When [\\_init\(9E\)](#) is executing, its call to mod\_install() enables other threads to call [attach\(9E\)](#) even prior to mod\_install() returning and [\\_init\(9E\)](#) completion. From a programming standpoint this means that all [\\_init\(9E\)](#) initialization must occur prior to [\\_init\(9E\)](#) calling mod\_install(). If mod\_install() fails (non-zero return value), any initialization must be undone.

When [\\_fini\(9E\)](#) is executing, another thread may call [attach\(9E\)](#) prior to [\\_fini\(9E\)](#) calling mod\_remove(). If this occurs, the mod\_remove() fails (non-zero return). From a programming standpoint, this means that [\\_init\(9E\)](#) initializations should only be undone after a successful return from mod\_remove().

To query the name of a module, use mod\_modname(). If the module is a driver, the module name is the same as the driver name. This function can be used to avoid compiled-in knowledge of the module name, and is of particular use when [\\_init\(9E\)](#) needs to call a function like [kstat\\_create\(9F\)](#).

**Return Values** mod\_install() and mod\_remove() return 0 upon success and non-zero on failure.

mod\_info() returns a non-zero value on success and 0 upon failure.



`mod_modname()` returns the name of the module. The returned name remains valid as long as the module remains loaded.

**Examples** See [\\_init\(9E\)](#) for an example that uses these functions.

**See Also** [\\_fini\(9E\)](#), [\\_info\(9E\)](#), [\\_init\(9E\)](#), [kstat\\_create\(9F\)](#), [modldrv\(9S\)](#), [modlinkage\(9S\)](#), [modlstrmod\(9S\)](#)

*Writing Device Drivers*

**Name** msgdsize – return the number of bytes in a message

**Synopsis** #include <sys/stream.h>

```
size_t msgdsize(mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Message to be evaluated.

**Description** The `msgdsize()` function counts the number of bytes in a data message. Only bytes included in the data blocks of type `M_DATA` are included in the count.

**Return Values** The number of data bytes in a message, expressed as an integer.

**Context** The `msgdsize()` function can be called from user, interrupt, or kernel context.

**Examples** See [bufcall\(9F\)](#) for an example that uses `msgdsize()`.

**See Also** [bufcall\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** msgpullup – concatenate bytes in a message
- Synopsis** #include <sys/stream.h>
- ```
mbblk_t *msgpullup(mblk_t *mp, ssize_t len);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *mp* Pointer to the message whose blocks are to be concatenated.  
*len* Number of bytes to concatenate.
- Description** The msgpullup() function concatenates and aligns the first *len* data bytes of the message pointed to by *mp*, copying the data into a new message. Any remaining bytes in the remaining message blocks will be copied and linked onto the new message. The original message is unaltered. If *len* equals -1, all data are concatenated. If *len* bytes of the same message type cannot be found, msgpullup() fails and returns NULL.
- Return Values** The msgpullup function returns the following values:
- |          |  |
|----------|--|
| Non-null | Successful completion. A pointer to the new message is returned. |
| NULL     | An error occurred.   |
- Context** The msgpullup() function can be called from user, interrupt, or kernel context.
- See Also** [srv\(9E\)](#), [allocb\(9F\)](#), [pullupmsg\(9F\)](#), [msgb\(9S\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*
- Notes** The msgpullup() function is a DKI-compliant replacement for the older [pullupmsg\(9F\)](#) routine. Users are strongly encouraged to use msgpullup() instead of [pullupmsg\(9F\)](#).

**Name** msgsize – Return the total number of bytes in a message

**Synopsis**

```
#include <sys/stream.h>
#include <sys/strsun.h>
```

```
size_t msgsize(mblk_t *mp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *mp* Message to be evaluated.

**Description** The `msgsize()` function counts the number of bytes in a message, regardless of the data type of the underlying data blocks.

**Return Values** Number of bytes in the message.

**Context** This function can be called from user, kernel or interrupt context.

**See Also** [msgdsize\(9F\)](#)

*STREAMS Programming Guide*

**Name** mt-streams – STREAMS multithreading

**Synopsis** #include <sys/conf.h>

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** STREAMS drivers configure the degree of concurrency using the `cb_flag` field in the `cb_ops` structure (see [cb\\_ops\(9S\)](#)). The corresponding field for STREAMS modules is the `f_flag` in the `fmodsw` structure.

For the purpose of restricting and controlling the concurrency in drivers/modules, we define the concepts of *inner* and *outer perimeters*. A driver/module can be configured either to have no perimeters, to have only an inner or an outer perimeter, or to have both an inner and an outer perimeter. Each perimeter acts as a readers-writers lock, that is, there can be multiple concurrent readers or a single writer. Thus, each perimeter can be entered in two modes: shared (reader) or exclusive (writer). The mode depends on the perimeter configuration and can be different for the different STREAMS entry points ( [open\(9E\)](#), [close\(9E\)](#), [put\(9E\)](#), or [srv\(9E\)](#)).

The concurrency for the different entry points is (unless specified otherwise) to enter with exclusive access at the inner perimeter (if present) and shared access at the outer perimeter (if present).

The perimeter configuration consists of flags that define the presence and scope of the inner perimeter, the presence of the outer perimeter (which can only have one scope), and flags that modify the default concurrency for the different entry points.

All MT safe modules/drivers specify the `D_MP` flag.

**Inner Perimeter Flags** The inner perimeter presence and scope are controlled by the mutually exclusive flags:

<code>D_MTPERQ</code>	The module/driver has an inner perimeter around each queue.
<code>D_MTQPAIR</code>	The module/driver has an inner perimeter around each read/write pair of queues.
<code>D_MTPERMOD</code>	The module/driver has an inner perimeter that encloses all the module's/driver's queues.
None of the above	The module/driver has no inner perimeter.

**Outer Perimeter Flags** The outer perimeter presence is configured using:

<code>D_MTOUTPERIM</code>	In addition to any inner perimeter, the module/driver has an outer perimeter that encloses all the module's/driver's queues. This can be combined with all the inner perimeter options except <code>D_MTPERMOD</code> .
---------------------------	---

Note that acquiring exclusive access at the outer perimeter (that is, using [qwriter\(9F\)](#) with the `PERIM_OUTER` flag) can incur significant performance penalties, which grow linearly with the number of open instances of the module or driver in the system.

The default concurrency can be modified using:

`D_MTPUTSHARED` This flag modifies the default behavior when [put\(9E\)](#) procedure are invoked so that the inner perimeter is entered shared instead of exclusively.

`D_MTOCEXCL` This flag modifies the default behavior when [open\(9E\)](#) and [close\(9E\)](#) procedures are invoked so the outer perimeter is entered exclusively instead of shared.

Note that drivers and modules using this flag can cause significant system performance degradation during stream open or close when many instances of the driver or module are in use simultaneously. For this reason, use of this flag is discouraged. Instead, since [open\(9E\)](#) and [close\(9E\)](#) both execute with user context, developers are encouraged to use traditional synchronization routines such as [cv\\_wait\\_sig\(9F\)](#) to coordinate with other open instances of the module or driver.

The module/driver can use [qwait\(9F\)](#) or `qwait_sig()` in the [open\(9E\)](#) and [close\(9E\)](#) procedures if it needs to wait “outside” the perimeters.

The module/driver can use [qwriter\(9F\)](#) to upgrade the access at the inner or outer perimeter from shared to exclusive.

The use and semantics of `qprocson()` and [qprocsoff\(9F\)](#) is independent of the inner and outer perimeters.

**See Also** [close\(9E\)](#), [open\(9E\)](#), [put\(9E\)](#), [srv\(9E\)](#), [qprocsoff\(9F\)](#), [qprocson\(9F\)](#), [qwait\(9F\)](#), [qwriter\(9F\)](#), [cb\\_ops\(9S\)](#)

*[STREAMS Programming Guide](#)*

*[Writing Device Drivers](#)*

**Name** mutex, mutex\_enter, mutex\_exit, mutex\_init, mutex\_destroy, mutex\_owned, mutex\_tryenter  
– mutual exclusion lock routines

**Synopsis** #include <sys/ksynch.h>

```
void mutex_init(kmutex_t *mp, char *name, kmutex_type_t type,
               void *arg);

void mutex_destroy(kmutex_t *mp);

void mutex_enter(kmutex_t *mp);

void mutex_exit(kmutex_t *mp);

int mutex_owned(kmutex_t *mp);

int mutex_tryenter(kmutex_t *mp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>mp</i>	Pointer to a kernel mutex lock (kmutex_t).
<i>name</i>	Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.)
<i>type</i>	Type of mutex lock.
<i>arg</i>	Type-specific argument for initialization routine.

**Description** A mutex enforces a policy of mutual exclusion. Only one thread at a time may hold a particular mutex. Threads trying to lock a held mutex will block until the mutex is unlocked.

Mutexes are strictly bracketing and may not be recursively locked, meaning that mutexes should be exited in the opposite order they were entered, and cannot be reentered before exiting.

mutex\_init() initializes a mutex. It is an error to initialize a mutex more than once.

The *type* argument specifies the type of mutex lock. The basic mutex type is MUTEX\_ADAPTIVE, which is expected to be used in almost all of the kernel. MUTEX\_SPIN provides interrupt blocking and must be used in interrupt handlers above LOCK\_LEVEL. The *iblock* cookie argument to mutex\_init() encodes the interrupt level to block. The *iblock* cookie must be NULL for adaptive locks.

MUTEX\_DEFAULT is the type usually specified (except in drivers) to mutex\_init(). It is identical to MUTEX\_ADAPTIVE.

MUTEX\_DRIVER is always used by drivers. mutex\_init() converts this to either MUTEX\_ADAPTIVE or MUTEX\_SPIN, depending on the *iblock* cookie.

The *arg* argument provides type-specific information for a given variant type of mutex. When `mutex_init()` is called for driver mutexes, if the mutex is used by the interrupt handler, the *arg* should be the interrupt priority returned from `ddi_intr_get_pri(9F)` or `ddi_intr_get_softint_pri(9F)`. Note that *arg* should be the value of the interrupt priority cast by calling the `DDI_INTR_PRI` macro. If the mutex is never used inside an interrupt handler, the argument should be `NULL`.

`mutex_enter()` is used to acquire a mutex. If the mutex is already held, then the caller blocks. After returning, the calling thread is the owner of the mutex. If the mutex is already held by the calling thread, a panic ensues.

`mutex_owned()` should only be used in `ASSERT()` and may be enforced by not being defined unless the preprocessor symbol `DEBUG` is defined. Its return value is non-zero if the current thread (or, if that cannot be determined, at least some thread) holds the mutex pointed to by *mp*.

`mutex_tryenter()` is very similar to `mutex_enter()` except that it doesn't block when the mutex is already held. `mutex_tryenter()` returns non-zero when it acquired the mutex and 0 when the mutex is already held.

`mutex_exit()` releases a mutex and will unblock another thread if any are blocked on the mutex.

`mutex_destroy()` releases any resources that might have been allocated by `mutex_init()`. `mutex_destroy()` must be called before freeing the memory containing the mutex, and should be called with the mutex unheld (not owned by any thread). The caller must be sure that no other thread attempts to use the mutex.

**Return Values** `mutex_tryenter()` returns a non-zero value on success and zero on failure.

`mutex_owned()` returns a non-zero value if the calling thread currently holds the mutex pointed to by *mp*, or when that cannot be determined, if any thread holds the mutex. Otherwise `mutex_owned()` returns zero.

**Context** These functions can be called from user, kernel, or high-level interrupt context, except for `mutex_init()` and `mutex_destroy()`, which can be called from user or kernel context only.

**Examples** **EXAMPLE 1** Initializing a Mutex

A driver might do this to initialize a mutex that is part of its unit structure and used in its interrupt routine:

```
ddi_intr_get_pri(hdhp, &pri);
mutex_init(&un->un_lock, NULL, MUTEX_DRIVER, DDI_INTR_PRI(pri));
ddi_intr_add_handler(hdhp, xxintr, (caddr_t)un, NULL);
```



**EXAMPLE 2** Calling a Routine with a Lock

A routine that expects to be called with a certain lock held might have the following ASSERT:

```
xxstart(struct xxunit *un)
{
    ASSERT(mutex_owned(&un->un_lock));
    ...
}
```

**See Also** [lockstat\(1M\)](#), [lockstat\(7D\)](#), [Intro\(9F\)](#), [condvar\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_intr\\_get\\_softint\\_pri\(9F\)](#), [rwlock\(9F\)](#), [semaphore\(9F\)](#)

*Writing Device Drivers*

**Notes** Compiling with `_LOCKTEST` or `_MPSTATS` defined has no effect. To gather lock statistics, see [lockstat\(1M\)](#). Mutex statistics can be gathered on the fly, without rebooting or recompiling the kernel, via the [lockstat\(7D\)](#) driver.

The address of a `kmutex_t` lock must be aligned on an 8-byte boundary for 64-bit kernels, or a 4-byte boundary for 32-bit kernels. Violation of this requirement will result in undefined behavior, including, but not limited to, failure of mutual exclusion or a system panic.

To write scalable, responsive drivers that do not hang, panic or deadlock the system, follow these guidelines:

Never return from a driver entry point with a mutex held.

Never hold a mutex when calling a service that may block, for example [kmem\\_alloc\(9F\)](#) with `KM_SLEEP` or [delay\(9F\)](#).

Always acquire mutexes in a consistent order. If a critical section acquires mutex A followed by B, and elsewhere in the driver mutex B is acquired before A, the driver can deadlock with one thread holding A and waiting for B and another thread holding B while waiting for A.

Always use a mutex to enforce exclusive access to data, not instruction paths.

Acquiring a lock in user context that is also acquired in interrupt context means that, as long as that lock is held, the driver instance holding the lock is subject to all the rules and limitations of interrupt context.

In most cases, a mutex can and should be acquired and released within the same function.

Liberal use of debugging aids like `ASSERT(mutex_owned(&mutex))` can help find callers of a function which should be holding a mutex but are not. This means you need to test your driver compiled with `DEBUG`.

Do not use a mutex to set driver state. However, you should use a mutex to protect driver state data.

Use per-instance and automatic data where possible to reduce the amount of shared data.

Per-instance data can be protected by a per-instance lock to improve scalability and reduce contention with multiple hardware instances.

Avoid global data and global mutexes whenever possible.

**Name** net\_event\_notify\_register, net\_event\_notify\_unregister – add/delete a function to be called for changes to a event

**Synopsis** #include <sys/hook.h>  
#include <sys/neti.h>

```
int net_event_notify_register(net_handle_t family, char
    *event, hook_notify_fn_t *callback, void *arg);

int net_event_notify_unregister(net_handle_t family, char
    *event, hook_notify_fn_t *callback);

typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,
    void *arg, const char *name1, const char *name2, const char
    *name3);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *family* value returned from a successful call to net\_protocol\_lookup().  
*callback* function to call when a change occurs.  
*event* name of the event for which notification of change is desired.  
*arg* pointer to pass into the callback() function when a change occurs.

**Description** The net\_event\_notify\_register() function registers a function represented by the pointer *callback* to be called when there is a change to the event represented by *family*. The types of changes for which notifications are available for is currently limited to the addition and removal of hooks.

The net\_event\_notify\_unregister() function indicates that there is no longer any desire to receive notification of changes to the event through function calls to the specified *callback*.

The name of a hook should be considered a private interface unless otherwise specified. The hook names used by IPFilter in Solaris are a public, but uncommitted, interface.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called when the *event* occurs must not block any other events.

The arguments passed through to the callback are as follows (the command is either HN\_REGISTER or HN\_UNREGISTER):

*name1* is the name of the protocol.  
*name2* is the name of the event

*name3* is the name of the hook being added/removed

**Return Values** If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST the given callback function is already registered.

**Context** These functions may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [net\\_hook\\_register\(9F\)](#), [net\\_hook\\_unregister\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_getifname – determine the name given to a network interface

**Synopsis** #include <sys/neti.h>

```
int net_getifname(const net_data_t net, const phy_if_t ifp,
                 char *buffer, size_t buflen);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

*ifp* value returned from a successful call to [net\\_phylookup\(9F\)](#) or [net\\_phygetnext\(9F\)](#).

*buffer* pointer to the buffer in which to write the interface name.

*buflen* size of the buffer in which to store the interface name.

**Description** The `net_getifname()` function copies the name of the network interface into the buffer provided. The name will always be null-terminated. If the buffer is too small to fit both the interface name and the null-terminated name, the name in the buffer is truncated to fit. See [net\\_phygetnext\(9F\)](#) for an example on how to use this function.

**Return Values** The `net_getifname()` function returns:

-1 The network protocol does not support this function.

0 Successful completion.

1 Unsuccessful.

**Context** The `net_getifname()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_phygetnext\(9F\)](#), [net\\_phylookup\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_getlifaddr – determine a network address for a given interface

**Synopsis** #include <sys/neti.h>

```
int net_getlifaddr(const net_data_t net, const phy_if_t ifp,
                  const net_if_t lif, int const type,
                  struct sockaddr* storage);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>net</i>	value returned from a successful call to <a href="#">net_protocol_lookup(9F)</a> .
<i>ifp</i>	value returned from a successful call to <a href="#">net_phylookup(9F)</a> or <a href="#">net_phygetnext(9F)</a> , indicating which network interface the information should be returned from.
<i>lif</i>	indicates the logical interface from which to fetch the address.
<i>type</i>	indicates what type of address should be returned. See below for more details on this field.
<i>storage</i>	pointer to an area of memory to store the address data.

**Description** The `net_getlifaddr()` function retrieves the address information for each logical interface. Each call to `net_getlifaddr()` requires that the caller pass a pointer to an array of address information types to retrieve, and an accompanying pointer to an array of pointers to `struct sockaddr` structures to which to copy the address information. See [net\\_lifgetnext\(9F\)](#) for an example on how to use this function.

Each member of the address type array should be one of the values listed here.

NA_ADDRESS	Return the network address associated with the logical interface ( <i>lif</i> ) that belongs to the network interface ( <i>ifp</i> ).
NA_PEER	Return the address assigned to the remote host for point to point network interfaces for the given network/logical interface.
NA_BROADCAST	Return the broadcast address assigned to the given network/logical interface for network interfaces that support broadcast packets.
NA_NETMASK	Return the netmask associated with the given network/logical interface for network interfaces that support broadcast packets.

**Return Values** The `net_getlifaddr()` function returns:

-1	The network protocol does not support this function.
0	Successful completion.
1	Unsuccessful.

**Context** The `net_getlifaddr()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_lifgetnext\(9F\)](#), [net\\_phylookup\(9F\)](#), [net\\_phygetnext\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_getmtu – determine the MTU of a given network interface

**Synopsis** #include <sys/neti.h>

```
int net_getmtu(const net_data_t net, const phy_if_t ifp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

*ifp* value returned from a successful call to [net\\_phylookup\(9F\)](#) or [net\\_phygetnext\(9F\)](#).

**Description** The `net_getmtu()` function receives information about the current MTU of a network interface. The value returned from this function call should not be cached as the MTU of a network interface since it is not guaranteed to be constant.

**Return Values** The `net_getmtu()` function returns -1 if the network protocol does not support this feature and otherwise returns the current MTU of the network interface.

**Context** The `net_getmtu()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_phygetnext\(9F\)](#), [net\\_phylookup\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_getnetid – returns the instance identifier

**Synopsis** #include <sys/neti.h>

```
netid_t net_getnetid(const net_data_t net);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

**Description** The `net_getnetid()` function returns the instance identifier for the protocol instance returned via a call to [net\\_protocol\\_lookup\(9F\)](#).

**Return Values** The `net_getnetid()` function returns the value of the instance identifier.

**Context** The `net_getnetid()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_protocol\\_lookup\(9F\)](#)



**Name** net\_getpmtuenabled – determine if path MTU discovery is enabled for a network protocol

**Synopsis** #include <sys/neti.h>

```
int net_getpmtuenabled(const net_data_t net);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

**Description** The `net_getpmtuenabled()` function returns a value to indicate whether or not path MTU (PMTU) discovery is enabled for this network protocol.

**Return Values** The `net_getpmtuenabled()` function returns:

- 1 The network protocol does not support this function.
- 0 PATH MTU discovery is disabled.
- 1 PATH MTU discovery is enabled.

**Context** The `net_getpmtuenabled()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_getmtu\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_hook\_register – add a hook to be called in event processing

**Synopsis** #include <sys/neti.h>

```
int net_hook_unregister(net_handle_t net, char *hook_name, hook_t hook);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>net</i>	value returned from a successful call to net_protocol_register().
<i>hook_name</i>	hook name to be registered
<i>hook</i>	pointer to a hook_t structure

**Description** The net\_hook\_register() function uses hooks that allow callbacks to be registered with events that belong to a network protocol. A successful call to net\_hook\_register() requires that a valid handle for a network protocol be provided (the *net* parameter), along with a hook description that includes a reference to an available event.

While it is possible to use the same hook\_t structure with multiple calls to net\_hook\_register(), it is not encouraged.

The hook\_t structure passed in with this function is described by hook\_t(9S). The following describes how this structure is used.

<i>h_func</i>	Must be non-NULL and represent a function that fits the specified interface.
<i>h_name</i>	Gives the hook a name that represents its owner. No duplication of <i>h_name</i> among the hooks present for an event is allowed.
<i>h_flags</i>	Currently unused and must be set to 0.
<i>h_hint, h_hintvalue</i>	Specify a hint to net_hook_register() on how to insert this hook. If the hint cannot be specified, then an error is returned.
<i>h_arg;</i>	May take any value that the consumer wishes to have passed back when the hook is activated.

**Return Values** If the net\_hook\_register() function succeeds, 0 is returned. Otherwise, one of the following errors is returned:

ENOMEM	The system cannot allocate any more memory to support registering this hook.
ENXIO	A hook cannot be found among the given family of events.
EEXIST	A hook with the given <i>h_name</i> already exists on that event.
ESRCH	A before or after dependency cannot be satisfied due to the hook with
EBUSY	The <i>h_hint</i> field specifies a hint that cannot currently be satisfied because it conflicts with another hook. An example of this might be specifying HH_FIRST or

HH\_LAST when another hook has already been registered with this value.

**Context** The `net_hook_register()` function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_hook\\_unregister\(9F\)](#), [hook\\_t\(9S\)](#)

**Name** net\_hook\_unregister – disable a hook that was called in event processing

**Synopsis** #include <sys/neti.h>

```
int net_hook_unregister(net_handle_t net, char *hook_name, hook_t hook);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to net\_protocol\_register().

*hook\_name* hook name to be unregistered

*hook* value returned from a successful call to net\_hook\_register(9F).

**Description** The net\_hook\_unregister() function disables the callback hooks that were registered with the net\_hook\_register() function.

**Return Values** If the net\_hook\_unregister() function succeeds, 0 is returned. Otherwise, an error indicating the problem encountered.

**Context** The net\_hook\_unregister() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_hook\\_register\(9F\)](#)

**Name** netinfo – interface to network data/functionality

**Description** The `net_getnetid()` interface is designed to provide the framework for accessing functionality and data within an implementation of a network layer protocol (OSI layer 3.) A protocol may or may not provide full coverage for each of the functions that is described within this interface. Where it does not, it must return an appropriate error condition for that call. Documentation pertaining to the network protocol, as found in man page section 7pP, must list which functions provided by this interface are and are not supported.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [hook\\_alloc\(9F\)](#), [hook\\_free\(9F\)](#), [hook\\_nic\\_event\(9S\)](#), [hook\\_pkt\\_event\(9S\)](#), [hook\\_t\(9S\)](#), [net\\_getifname\(9F\)](#), [net\\_getlifaddr\(9F\)](#), [net\\_getmtu\(9F\)](#), [net\\_getnetid\(9F\)](#), [net\\_getpmtuenabled\(9F\)](#), [net\\_hook\\_register\(9F\)](#), [net\\_hook\\_unregister\(9F\)](#), [net\\_inject\(9F\)](#), [net\\_inject\\_alloc\(9F\)](#), [net\\_inject\\_free\(9F\)](#), [net\\_inject\\_t\(9S\)](#), [net\\_instance\\_alloc\(9F\)](#), [net\\_instance\\_free\(9F\)](#), [net\\_instance\\_register\(9F\)](#), [net\\_instance\\_unregister\(9F\)](#), [net\\_inject\\_t\(9S\)](#), [net\\_ispartialchecksum\(9F\)](#), [net\\_isvalidchecksum\(9F\)](#), [net\\_kstat\\_create\(9F\)](#), [net\\_kstat\\_delete\(9F\)](#), [net\\_lifgetnext\(9F\)](#), [net\\_netidtozonid\(9F\)](#), [net\\_phygetnext\(9F\)](#), [net\\_phylookup\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [net\\_protocol\\_release\(9F\)](#), [net\\_protocol\\_walk\(9F\)](#), [net\\_routeto\(9F\)](#), [net\\_zoneidtonetid\(9F\)](#)

**Name** net\_inject – determine if a network interface name exists for a network protocol

**Synopsis** #include <sys/neti.h>

```
int net_inject(const net_data_t net, inject_t style,
               net_inject_t *packet);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>net</i>	value returned from a successful call to <a href="#">net_protocol_lookup(9F)</a> .
<i>style</i>	method that determines how this packet is to be injected into the network or kernel.
<i>packet</i>	details about the packet to be injected.

**Description** The `net_inject()` function provides an interface to allow delivery of network layer (layer 3) packets either into the kernel or onto the network. The method of delivery is determined by *style*.

If `NI_QUEUE_IN` is specified, the packet is scheduled for delivery up into the kernel, imitating its reception by a network interface. In this mode, `packet->ni_addr` is ignored and `packet->ni_physical` specifies the interface for which the packet is made to appear as if it arrived on.

If `NI_QUEUE_OUT` is specified, the packet is scheduled for delivery out of the kernel, as if it were being sent by a raw socket. In this mode, `packet->ni_addr` and `packet->ni_physical` are both ignored.

Neither `NI_QUEUE_IN` or `NI_QUEUE_OUT` cause the packet to be immediately processed by the kernel. Instead, the packet is added to a list and a timeout is scheduled (if there are none already pending) to deliver the packet. The call to `net_inject()` returns once the setup has been completed, and not after the packet has been processed. The packet processing is completed on a different thread and in a different context to that of the original packet. Thus, a packet queued up using `net_inject()` for either `NI_QUEUE_IN` or `NI_QUEUE_OUT` is presented to the packet event again. A packet received by a hook from `NH_PHYSICAL_IN` and then queued up with `NI_QUEUE_IN` is seen by the hook as another `NH_PHYSICAL_IN` packet. This also applies to both `NH_PHYSICAL_OUT` and `NI_QUEUE_OUT` packets.

If `NI_DIRECT_OUT` is specified, an attempt is made to send the packet out to a network interface immediately. No processing on the packet, aside from prepending any required layer 2 information, is made. In this instance, `packet->ni_addr` may be used to specify the next hop (for the purpose of link layer address resolution) and `packet->ni_physical` determines which interface the packet should be sent out.

For all three packets, `packet->ni_packet` must point to an `mb1k` structure with the packet to be delivered.

See [net\\_inject\\_t\(9S\)](#) for more details on the structure `net_inject_t`.

**Return Values** The `net_inject()` function returns:

- 1 The network protocol does not support this function.
- 0 The packet is successfully queued or sent.
- 1 The packet could not be queued up or sent out immediately.

**Context** The `net_inject()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_protocol\\_lookup\(9F\)](#), [netinfo\(9F\)](#), [net\\_inject\\_t\(9S\)](#)

**Name** net\_inject\_alloc – allocate a net\_inject\_t structure

**Synopsis** #include <sys/neti.h>

```
net_inject_t *net_inject_alloc(const int version);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *version* must always be the symbol NETI\_VERSION.

**Description** The net\_inject\_alloc() function allocates a net\_inject\_t structure, returning a pointer for the caller to use.

**Return Values** Upon success, net\_inject\_alloc() returns a pointer to the allocated net\_inject\_t structure. On failure, hook\_alloc() returns a NULL pointer.

**Context** The net\_inject\_alloc() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_inject\\_free\(9F\)](#), [net\\_inject\\_t\(9S\)](#)



**Name** net\_inject\_free – free a net\_inject\_t structure

**Synopsis** #include <sys/neti.h>

```
void net_inject_free(net_inject_t *inject);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *inject* pointer returned by [net\\_inject\\_alloc\(9F\)](#).

**Description** The `net_inject_free()` function frees a `net_inject_t` structure that was originally allocated by [net\\_inject\\_alloc\(9F\)](#).

**Context** The `net_inject_free()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_inject\\_alloc\(9F\)](#), [attributes\(5\)](#), [net\\_inject\\_t\(9S\)](#)

**Name** net\_instance\_alloc – allocate a net\_instance\_t structure

**Synopsis** #include <sys/neti.h>

```
net_instance_t *net_instance_alloc(const int version);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *version* must always be the symbol NETI\_VERSION.

**Description** The net\_instance\_alloc() function allocates a net\_instance\_t structure, returning a pointer for the caller to use.

**Return Values** Upon success, net\_instance\_alloc() returns a pointer to the allocated net\_instance\_t structure. On failure, it returns a NULL pointer.

**Context** The net\_instance\_alloc() function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_instance\\_free\(9F\)](#), [attributes\(5\)](#), [net\\_inject\\_t\(9S\)](#)

**Name** net\_instance\_free – free a net\_instance\_t structure

**Synopsis** #include <sys/neti.h>

```
void net_instance_free(net_instance_t *net_instance);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net\_instance* pointer returned by [net\\_instance\\_alloc\(9F\)](#).

**Description** The `net_instance_free()` function frees a `net_instance_t` structure that was originally allocated by [net\\_instance\\_alloc\(9F\)](#).

**Context** The `net_instance_free()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_instance\\_alloc\(9F\)](#), [attributes\(5\)](#), [net\\_instance\\_t\(9S\)](#)

**Name** net\_instance\_notify\_register, net\_instance\_notify\_unregister – add/delete a function to be called for changes to an instance

**Synopsis** #include <sys/hook.h>  
#include <sys/neti.h>

```
int net_instance_notify_register(net_id_t net_id,  
    hook_notify_fn_t *callback, void *arg);  
  
int net_instance_notify_unregister(net_id_t net_id,  
    hook_notify_fn_t *callback);  
  
typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,  
    void *arg, const char *name1, const char *name2, const char  
    *name3);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *netid* value from either *callback* registered with net\_instance\_register() or net\_zoneidtonetid().

*callback* function to call when a change occurs.

*arg* pointer to pass into the callback() function when a change occurs.

**Description** The net\_instance\_notify\_register() function registers a function represented by the pointer *callback* to be called when there is a new instance added or removed from the given network instance (represented by *netid*.)

The net\_instance\_notify\_unregister() function indicates that there is no longer any desire to receive notification of changes to the instance through function calls to the specified *callback*.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called must not block any other events.

The arguments passed through to the callback are as follows (the command is either HN\_REGISTER or HN\_UNREGISTER):

*name1* is the *netid* represented as a string.

*name2* is NULL.

*name3* is the name of the instance being added/removed

**Return Values** If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST the given callback function is already registered.

**Context** These functions may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [net\\_instance\\_register\(9F\)](#), [net\\_instance\\_unregister\(9F\)](#), [net\\_zoneidtonetid\(9F\)](#)

**Name** net\_instance\_register – register a set of instances to occur with IP instance events

**Synopsis** #include <sys/neti.h>

```
int net_instance_register(net_instance_t *instances);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *instances* must be a pointer returned by [net\\_instance\\_alloc\(9F\)](#).

**Description** The `net_instance_register()` function attempts to record the set of functions passed by instances that are to be called when an event related to IP instance maintenance occurs.

**Return Values** If the `net_instance_register()` function succeeds, `DDI_SUCCESS` is returned. Otherwise, `DDI_FAILURE` is returned to indicate failure due to the name in the instance already being present.

**Context** The `net_instance_register()` function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_instance\\_alloc\(9F\)](#), [net\\_instance\\_unregister\(9F\)](#), [attributes\(5\)](#), [net\\_instance\\_t\(9S\)](#)

**Name** net\_instance\_unregister – disable a set of instances

**Synopsis** #include <sys/neti.h>

```
void net_instance_unregister(net_instance_t *instances);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *instances* must be a pointer returned by [net\\_instance\\_alloc\(9F\)](#).

**Description** The `net_instance_unregister()` function removes the set of instances that were previously registered with the `net_instance_register()` function.

**Return Values** If the `net_instance_unregister()` function succeeds, 0 is returned. Otherwise, an error indicating the problem encountered.

**Context** The `net_instance_unregister()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_instance\\_alloc\(9F\)](#), [net\\_instance\\_register\(9F\)](#), [attributes\(5\)](#), [net\\_instance\\_t\(9S\)](#)

**Name** net\_ispartialchecksum – indicate if a packet is being scheduled for hardware checksum calculation

**Synopsis** #include <sys/neti.h>

```
int net_ispartialchecksum(const net_data_t net, mblk_t *mb);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

*mb* the mblk structure holding a packet that is the subject of this query.

**Description** The net\_ispartialchecksum() function looks at the fields within the mblk structure to determine if the packet contained inside contains headers with only partial checksum values. Partial checksum values are stored inside headers when the calculation of the complete checksum is being handled by the hardware.

**Return Values** The net\_ispartialchecksum() function returns:

–1 The network protocol does not support this function.

0 The packet does not contain partial checksums.

If a packet is marked for hardware checksum'ing, the following values are returned:

NET\_HCK\_L3\_FULL Complete layer 3 checksum calculated

NET\_HCK\_L3\_PART Partial layer 3 checksum calculated

NET\_HCK\_L4\_FULL Complete layer 4 checksum calculated

NET\_HCK\_L4\_PART Partial layer 4 checksum calculated

**Context** The net\_ispartialchecksum() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_isvalidchecksum\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [attributes\(5\)](#)



**Name** net\_isvalidchecksum – verify layer 3 and layer 4 checksums

**Synopsis** #include <sys/neti.h>

```
int net_isvalidchecksum(const net_data_t net, mblk_t *mb);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

*mb* the mblk structure holding a packet that is the subject of this query.

**Description** The `net_isvalidchecksum()` function verifies the layer 3 checksum (and, in some case, the layer 4 checksum) in the packet. If possible, fields that are used by hardware checksum'ing are examined rather than manually verifying that the checksums are present for packets received from a network interface.

For both IPv4 and IPv6, TCP, UDP and ICMP (including ICMPV6 for IPv6) layer 4 checksums are currently validated.

**Return Values** The `net_isvalidchecksum()` function returns:

-1 The network protocol does not support this function.

0 The packet does not contain partial checksums.

1 The packet does contain partial checksums.

**Context** The `net_isvalidchecksum()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_ispartialchecksum\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [attributes\(5\)](#)

**Name** net\_kstat\_create – create and initialize a new kstat for a specific instance of IP

**Synopsis**

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/neti.h>
```

```
kstat_t *net_kstat_create(netid_t netid, char *module,
    int instance, char *name, char *class, uchar_type type,
    ulong_t ndata, uchar_t ks_flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>netid</i>	IP instance identifier.
<i>module</i>	The name of the provider's module (such as “sd”, “esp”, ...). The “core” kernel uses the name “unix”.
<i>instance</i>	The provider's instance number, as from <a href="#">ddi_get_instance(9F)</a> . Modules which do not have a meaningful instance number should use 0.
<i>name</i>	A pointer to a string that uniquely identifies this structure. Only KSTAT_STRLLEN – 1 characters are significant.
<i>class</i>	The general class that this kstat belongs to. The following classes are currently in use: disk, tape, net, controller, vm, kvm, hat, streams, kstat, and misc.
<i>type</i>	The type of kstat to allocate. Valid types are: <ul style="list-style-type: none"> <li>KSTAT_TYPE_NAMED Allows more than one data record per kstat.</li> <li>KSTAT_TYPE_INTR Interrupt; only one data record per kstat.</li> <li>KSTAT_TYPE_IO I/O; only one data record per kstat</li> </ul>
<i>ndata</i>	The number of type-specific data records to allocate.
<i>ks_flag</i>	A bit-field of various flags for this kstat. <i>ks_flag</i> is some combination of: <ul style="list-style-type: none"> <li>KSTAT_FLAG_VIRTUAL Tells <code>net_kstat_create()</code> not to allocate memory for the kstat data section; instead, the driver will set the <code>ks_data</code> field to point to the data it wishes to export. This provides a convenient way to export existing data structures.</li> <li>KSTAT_FLAG_WRITABLE Makes the kstat data section writable by root.</li> <li>KSTAT_FLAG_PERSISTENT Indicates that this kstat is to be persistent over time. For persistent kstats, <a href="#">kstat_delete(9F)</a> simply marks the kstat as dormant; a subsequent</li> </ul>

`kstat_create()` reactivates the `kstat`. This feature is provided so that statistics are not lost across driver close/open (such as raw disk I/O on a disk with no mounted partitions.) Note: Persistent `kstats` cannot be virtual, since `ks_data` points to garbage as soon as the driver goes away.

**Description** The `net_kstat_create()` function allocates and initializes a [kstat\(9S\)](#) structure. See [kstat\\_create\(9F\)](#) for a complete discussion of this function.

**Return Values** If successful, `net_kstat_create()` returns a pointer to the allocated `kstat`. NULL is returned upon failure.

**Context** The `net_kstat_create()` function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [ddi\\_get\\_instance\(9F\)](#), [kstat\\_create\(9F\)](#), [kstat\\_delete\(9F\)](#), [net\\_kstat\\_delete\(9F\)](#), [hook\\_t\(9S\)](#), [kstat\\_named\(9S\)](#)

**Name** net\_kstat\_delete – remove a kstat for a specific instance of IP from the system

**Synopsis**

```
#include <sys/types.h>
#include <sys/kstat.h>
#include <sys/neti.h>
```

```
void net_kstat_delete(netid_t netid, kstat_t *ksp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *netid* IP instance identifier.  
*ksp* Pointer to a currently installed [kstat\(9S\)](#) structure.

**Description** The net\_kstat\_delete() function removes *ksp* from the kstat chain and frees all associated system resources. See [kstat\\_delete\(9F\)](#) for a complete discussion of this function.

**Return Values** If successful, net\_kstat\_delete() returns a pointer to the allocated kstat. NULL is returned upon failure.

**Context** The net\_kstat\_delete() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [kstat\\_create\(9F\)](#), [kstat\\_delete\(9F\)](#), [net\\_kstat\\_create\(9F\)](#), [hook\\_t\(9S\)](#), [kstat\\_named\(9S\)](#)

**Name** net\_lifgetnext – search through a list of logical network interfaces

**Synopsis** #include <sys/neti.h>

```
net_if_t net_lifgetnext(const net_data_t net, const phy_if_t ifp,
    net_if_t lif);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).  
*ifp* value returned from a successful call to [net\\_phylookup\(9F\)](#) or [net\\_phygetnext\(9F\)](#).  
*lif* value returned from a successful call to this function.

**Description** The `net_lifgetnext()` function is used to search through all of the logical interfaces that are associated with a physical network interface. To start searching through this list, a value of 0 should be passed through as the value of `lif`. When 0 is returned by this function, the last of the interfaces owned by this protocol has been reached.

When called successfully, the value returned represents a logical interface that exists, at the time of the call, within the scope of the network interface and its assigned network protocol. This value is only guaranteed to be unique for a name within the scope of the network interface and its assigned protocol.

**Examples**

```
net_data_t net;
phy_if_t ifp;
net_if_t lif;
char buffer[32];
net_ifaddr_t atype[1];
struct sockaddr_in sin[1];

net = net_protocol_lookup("inet");

if (net != NULL) {
    atype[0] = NA_ADDRESS;
    ifp = net_phylookup(net, "hme0");
    for (lif = net_lifgetnext(net, 0); lif != 0;
        lif = net_lifgetnext(net, lif)) {
        /* Do something with lif */
        if (net_getlifaddr(net, ifp, lif, 1, atype, sin) == 0)
            printf("hme0:%d %x0, lif,
                ntohl(sin[0].sin_addr.s_addr));
    }
}
```

**Return Values** The `net_lifgetnext()` function returns a value of -1 if it is not supported by the network protocol and a value of 0 if an attempt to go beyond the last network interface is made. Otherwise, it returns a value representing a network interface.

**Context** The `net_lifgetnext()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_phygetnext\(9F\)](#), [net\\_phylookup\(9F\)](#)

**Name** net\_netidtozonid – map a netid\_t structure identifier to a zoneid\_t structure

**Synopsis** #include <sys/neti.h>

```
zoneid_t net_netidtozonid(const netid_t net);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* valid netid\_t structure that refers to an existing IP instance.

**Description** The net\_netidtozonid() function maps the given netid\_t (used to represent an instance of IP) into the matching zoneid\_t structure that is associated with it.

**Return Values** The net\_netidtozonid() function returns -1 if no mapping took place. Otherwise, it returns the zoneid\_t structure currently associated with a netid\_t structure. For shared stack zones, the zoneid\_t structure for the global zone is returned.

**Context** The net\_netidtozonid() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_zoneidtonetid\(9F\)](#), [attributes\(5\)](#)

**Name** net\_phygetnext – search through the current list of network interfaces

**Synopsis** #include <sys/neti.h>

```
phy_if_t net_phygetnext(const net_data_t net, const phy_if_t ifp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

*ifp* value returned from a successful call to this function or [net\\_phylookup\(9F\)](#).

**Description** The `net_phygetnext()` function searches through all of the network interfaces that a network protocol “owns”. To start searching through all of the interfaces owned by a protocol, a value of 0 should be passed through as the value of *ifp*. When 0 is returned by this function, the last of the interfaces owned by this protocol has been reached.

When called successfully, the value returned represents a network interface that exists, at the time of the call, within the scope of the network interface. This value is only guaranteed to be unique for a name within the scope of the network protocol.

**Examples**

```
net_data_t net;
phy_if_t ifp;
char buffer[32];

net = net_protocol_lookup("inet");

if (net != NULL) {
    for (ifp = net_phygetnext(net, 0); ifp != 0;
         ifp = net_phygetnext(net, ifp)) {
        /* Do something with ifp */
        if (net_getifname(net, ifp, buffer,
                        sizeof(buffer) >= 0)
            printf("Interface %s0, buffer);
    }
}
```

**Return Values** The `net_phygetnext()` function returns -1 if it is not supported by the network protocol or 0 if an attempt to go beyond the last network interface is made. Otherwise, it returns a value representing a network interface.

**Context** The `net_phygetnext()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os



ATTRIBUTETYPE	ATTRIBUTEVALUE
Interface Stability	Committed

**See Also** [net\\_phylookup\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [attributes\(5\)](#)

**Name** net\_phylookup – determine if a network interface name exists for a network protocol

**Synopsis** #include <sys/neti.h>

```
phy_if_t net_phylookup(const net_data_t net, const char *name);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).  
*name* name of the network interface to find.

**Description** The net\_phylookup() function attempts to resolve the interface name passed in with the network protocol.

When called successfully, the value returned represents a network interface that exists, at the time of the call, within the scope of the network interface. This value is only guaranteed to be unique for a name within the scope of the network protocol.

**Return Values** The net\_phylookup() function returns -1 if it is not supported by the network protocol, and 0 if the named network interface does not exist (or is otherwise unknown). Otherwise, it returns a value greater than 0 representing a network interface that currently exists within the scope of this network protocol.

**Context** The net\_phylookup() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_getifname\(9F\)](#), [net\\_phygetnext\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [attributes\(5\)](#)

**Name** net\_protocol\_lookup – locate an implementation of a network layer protocol

**Synopsis** #include <sys/neti.h>

```
net_data_t net_protocol_lookup(netid_t id, const char *protocol);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *id* network instance identifier.  
*family* name of the network protocol to find.

**Description** The net\_protocol\_lookup() function attempts to locate a data structure that defines what capabilities it is exporting through this interface. The value returned by this call is guaranteed to be valid until it is passed into a call to [net\\_protocol\\_release\(9F\)](#), after which it should no longer be treated as valid.

The protocol must be a registered name of a network protocol that has been registered. The symbols NHF\_INET and NHF\_INET6 should be passed to net\_protocol\_lookup() as the protocol name to gain access to either IPv4 or IPv6 respectively.

**Return Values** The net\_protocol\_lookup() function returns NULL if it does not find any knowledge about the network protocol referenced. Otherwise, it returns a value that can be used with other calls in this framework.

**Context** The net\_protocol\_lookup() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_protocol\\_release\(9F\)](#), [attributes\(5\)](#)

**Name** net\_protocol\_notify\_register, net\_instance\_protocol\_unregister – add/delete a function to be called for changes to a protocol

**Synopsis** #include <sys/hook.h>  
#include <sys/neti.h>

```
int net_protocol_notify_register(net_handle_t family,
    hook_notify_fn_t *callback, void *arg);

int net_protocol_notify_unregister(net_handle_t family,
    hook_notify_fn_t *callback);

typedef int (* hook_notify_fn_t)(hook_notify_cmd_t command,
    void *arg, const char *name1, const char *name2, const char
    *name3);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *family* value returned from net\_protocol\_lookup().  
*callback* function to call when a change occurs.  
*arg* pointer to pass into the callback() function when a change occurs.

**Description** The net\_protocol\_notify\_register() function registers a function represented by the pointer *callback* to be called when there is a change to the protocol represented by *family*. The types of changes for which notifications are available for is currently limited to the addition and removal of protocols.

The net\_protocol\_notify\_unregister() function removes the function specified by the pointer *callback* from the list of functions to call. This call may fail if the specified function cannot be found.

Multiple *callback* functions may be registered through this interface. The same set of parameters is passed to each *callback* function. The memory referenced through the pointers passed to the *callback* should be treated as pointing to read-only memory. Changing this data is strictly prohibited.

The function that is called must not block any other protocols.

The arguments passed through to the callback are as follows (the command is either HN\_REGISTER or HN\_UNREGISTER):

*name1* is the name of the protocol  
*name2* is NULL.  
*name3* is the name of the protocol being added/removed

**Return Values** If these functions succeed, 0 is returned. Otherwise, the following error is returned:

EEXIST the given callback function is already registered.

**Context** These functions may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [net\\_protocol\\_lookup\(9F\)](#)

**Name** net\_protocol\_release – indicate that a reference to a network protocol is no longer required

**Synopsis** #include <sys/neti.h>

```
int net_protocol_release(net_data_t *net);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

**Description** The `net_protocol_release()` function indicates to the network information framework that the caller is no longer interested in any knowledge about the network protocol to which the parameter being passed through applies.

**Return Values** The `net_protocol_release()` function returns:

-1 The value passed in is unknown to this framework.

0 Successful completion.

1 Unsuccessful because this function has been called too many times.

**Context** The `net_protocol_release()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_protocol\\_lookup\(9F\)](#), [net\\_protocol\\_walk\(9F\)](#), [attributes\(5\)](#)

**Name** net\_protocol\_walk – step through the list of registered network protocols

**Synopsis** #include <sys/neti.h>

```
net_handle_t net_protocol_walk(netid_t stack_id, net_handle_t net);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *stack\_id* the IP stack instance ID obtained by a calling [getzoneid\(3C\)](#) and typecasting its return value to `netid_t`

*net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).

**Description** The `net_protocol_walk()` function walks through all of the network protocols that have been registered with this interface. The initial call to `net_protocol_walk()` should be made by passing in NULL as the value for *net*. When this function returns NULL, the end of the list has been reached.

A caller of `net_protocol_walk()` is required to walk through the entire list of network protocols, until NULL is returned or, when finished with using the value returned, pass it into a call to [net\\_protocol\\_release\(9F\)](#).

**Return Values** The `net_protocol_walk()` function returns NULL when the end of the list is returned. Otherwise, it returns a non-NULL value as a token for being passed into other function calls within this interface.

**Context** The `net_protocol_walk()` function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_protocol\\_lookup\(9F\)](#), [net\\_protocol\\_release\(9F\)](#), [attributes\(5\)](#)

**Name** net\_routeto – indicate which network interface packets are sent

**Synopsis** #include <sys/neti.h>

```
phy_if_t net_routeto(const net_data_t *net, struct sockaddr *address,
                    struct sockaddr *nexthop);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- net* value returned from a successful call to [net\\_protocol\\_lookup\(9F\)](#).
- address* network address to find a path out of the machine for.
- nexthop* pointer to the sockaddr structure in which to store the address of the next hop. If this information is not required, the value NULL may be passed instead.

**Description** The net\_routeto() function indicates which network interface packets destined for a particular address would be sent out of, according to the systems network routing tables. If next is supplied as a non-NULL pointer, the IP address of the next hop router to be used is returned in it.

**Return Values** The net\_routeto() function returns:

- 1 The network protocol does not support this function.
- 0 This function cannot find a route for the address given.
- >0 Indicates which network interface can be used to reach the given address.

**Context** The net\_routeto() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_phygetnext\(9F\)](#), [net\\_phylookup\(9F\)](#), [net\\_protocol\\_lookup\(9F\)](#), [attributes\(5\)](#)



**Name** net\_zoneidtonetid – map a zoneid\_t structure identifier to a netid\_t structure

**Synopsis** #include <sys/neti.h>

```
netid_t net_zoneidtonetid(const zoneid_t zone);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *zone* valid zoneid\_t structure that refers to a running zone.

**Description** The net\_zoneidtonetid() function maps the given zoneid\_t structure (used to represent a zone that is currently running) into a netid\_t structure that is associated with the IP instance supporting network functions for that zone.

**Return Values** The net\_zoneidtonetid() function returns -1 if no mapping took place. Otherwise, it returns the netid\_t structure currently used by the zoneid\_t structure. For zones that are using a shared IP instance, the netid\_t structure for the instance owned by the global zone is returned.

**Context** The net\_zoneidtonetid() function may be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/core-os
Interface Stability	Committed

**See Also** [net\\_netidtozonid\(9F\)](#), [attributes\(5\)](#)

**Name** nochpoll – error return function for non-pollable devices

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int nochpoll(dev_t dev, short events, int anyyet, short *reventsp,
             struct pollhead **pollhdrp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>dev</i>	Device number.
<i>events</i>	Event flags.
<i>anyyet</i>	Check current events only.
<i>reventsp</i>	Event flag pointer.
<i>pollhdrp</i>	Poll head pointer.

**Description** The `nochpoll()` function is a routine that simply returns the value `ENXIO`. It is intended to be used in the `cb_ops(9S)` structure of a device driver for devices that do not support the `poll(2)` system call.

**Return Values** The `nochpoll()` function returns `ENXIO`.

**Context** The `nochpoll()` function can be called from user, interrupt, or kernel context.

**See Also** `poll(2)`, `chpoll(9E)`, `cb_ops(9S)`

*Writing Device Drivers*

**Name** nodev – error return function

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>

```
int nodev( );
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Description** nodev( ) returns ENXIO. It is intended to be used in the [cb\\_ops\(9S\)](#) data structure of a device driver for device entry points which are not supported by the driver. That is, it is an error to attempt to call such an entry point.

**Return Values** nodev( ) returns ENXIO.

**Context** nodev( ) can be only called from user context.

**See Also** [nulldev\(9F\)](#), [cb\\_ops\(9S\)](#)

*Writing Device Drivers*

**Name** noenable – prevent a queue from being scheduled

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

void noenable(queue_t *q);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue.

**Description** The `noenable()` function prevents the *q* from being scheduled for service by [insq\(9F\)](#), [putq\(9F\)](#) or [putbq\(9F\)](#) when enqueueing an ordinary priority message. The queue can be re-enabled with the [enableok\(9F\)](#) function.

**Context** The `noenable()` function can be called from user, interrupt, or kernel context.

**See Also** [enableok\(9F\)](#), [insq\(9F\)](#), [putbq\(9F\)](#), [putq\(9F\)](#), [qenable\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** nulldev – zero return function

**Synopsis** #include <sys/conf.h>  
#include <sys/ddi.h>

```
int nulldev();
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Description** nulldev() returns 0. It is intended to be used in the [cb\\_ops\(9S\)](#) data structure of a device driver for device entry points that do nothing.

**Return Values** nulldev() returns a 0.

**Context** nulldev() can be called from any context.

**See Also** [nodev\(9F\)](#), [cb\\_ops\(9S\)](#)

*Writing Device Drivers*

**Name** nvlst\_add\_boolean, nvlst\_add\_boolean\_value, nvlst\_add\_byte, nvlst\_add\_int8, nvlst\_add\_uint8, nvlst\_add\_int16, nvlst\_add\_uint16, nvlst\_add\_int32, nvlst\_add\_uint32, nvlst\_add\_int64, nvlst\_add\_uint64, nvlst\_add\_string, nvlst\_add\_nvlst, nvlst\_add\_nvpair, nvlst\_add\_boolean\_array, nvlst\_add\_int8\_array, nvlst\_add\_uint8\_array, nvlst\_add\_nvlst\_array, nvlst\_add\_byte\_array, nvlst\_add\_int16\_array, nvlst\_add\_uint16\_array, nvlst\_add\_int32\_array, nvlst\_add\_uint32\_array, nvlst\_add\_int64\_array, nvlst\_add\_uint64\_array, nvlst\_add\_string\_array, nvlst\_t – value pair functions

**Synopsis** #include <sys/nvpair.h>

```
int nvlst_add_boolean(nvlst_t *nvl, const char *name);
int nvlst_add_boolean_value(nvlst_t *nvl, const char *name,
    boolean_t val);
int nvlst_add_byte(nvlst_t *nvl, const char *name, uchar_t val);
int nvlst_add_int8(nvlst_t *nvl, const char *name, int8_t val);
int nvlst_add_uint8(nvlst_t *nvl, const char *name, uint8_t val);
int nvlst_add_int16(nvlst_t *nvl, const char *name, int16_t val);
int nvlst_add_uint16(nvlst_t *nvl, const char *name, uint16_t val);
int nvlst_add_int32(nvlst_t *nvl, const char *name, int32_t val);
int nvlst_add_uint32(nvlst_t *nvl, const char *name, uint32_t val);
int nvlst_add_int64(nvlst_t *nvl, const char *name, int64_t val);
int nvlst_add_uint64(nvlst_t *nvl, const char *name, uint64_t val);
int nvlst_add_string(nvlst_t *nvl, const char *name, char *val);
int nvlst_add_nvlst(nvlst_t *nvl, const char *name, nvlst_t *val);
int nvlst_add_nvpair(nvlst_t *nvl, nvpair_t *nvp);
int nvlst_add_boolean_array(nvlst_t *nvl, const char *name,
    boolean_t *val, uint_t nelem);
int nvlst_add_byte_array(nvlst_t *nvl, const char *name, uchar_t *val,
    uint_t nelem);
int nvlst_add_int8_array(nvlst_t *nvl, const char *name, int8_t *val,
    uint_t nelem);
int nvlst_add_uint8_array(nvlst_t *nvl, const char *name,
    uint8_t *val, uint_t nelem);
int nvlst_add_int16_array(nvlst_t *nvl, const char *name, int16_t *val,
    uint_t nelem);
```

```

int nvlst_add_uint16_array(nvlst_t *nvl, const char *name, uint16_t *val,
                          uint_t nelem);

int nvlst_add_int32_array(nvlst_t *nvl, const char *name, int32_t *val,
                          uint_t nelem);

int nvlst_add_uint32_array(nvlst_t *nvl, const char *name, uint32_t *val,
                          uint_t nelem);

int nvlst_add_int64_array(nvlst_t *nvl, const char *name, int64_t *val,
                          uint_t nelem);

int nvlst_add_uint64_array(nvlst_t *nvl, const char *name, uint64_t *val,
                          uint_t nelem);

int nvlst_add_string_array(nvlst_t *nvl, const char *name,
                          const *char *val, uint_t nelem);

int nvlst_add_nvlst_array(nvlst_t *nvl, const char *name,
                          nvlst_t **val, uint_t nelem);

```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>nvl</i>	The <code>nvlst_t</code> to be processed.
<i>nvp</i>	The <code>nvpair_t</code> (name-value pair) to be processed.
<i>name</i>	Name of the name-value pair ( <code>nvpair</code> ).
<i>nelem</i>	Number of elements in value (that is, array size).
<i>val</i>	Value or starting address of the array value.

**Description** These functions add a new name-value pair to an `nvlst_t`. The memory allocation policy follows that specified in `nvlst_alloc()`, `nvlst_unpack()`, or `nvlst_dup()`. See [nvlst\\_alloc\(9F\)](#). The uniqueness of `nvpair` name and data types follows the *nvflag* argument specified in `nvlst_alloc()`.

If `NV_UNIQUE_NAME` was specified for *nvflag*, existing `nvpairs` with matching names are removed before the new `nvpair` is added.

If `NV_UNIQUE_NAME_TYPE` was specified for *nvflag*, existing `nvpairs` with matching names and data types are removed before the new `nvpair` is added.

If neither was specified for *nvflag*, the new `nvpair` is unconditionally added at the end of the list. The library preserves the order of the name-value pairs across packing, unpacking, and duplication.

Multiple threads can simultaneously read the same `nvlst_t`, but only one thread may actively change a given `nvlst_t` at a time. The caller is responsible for the synchronization.

The `nvlst_add_boolean()` function is deprecated and the `nvlst_add_boolean_value()` function is used instead.

**Return Values**

0	success
EINVAL	invalid argument
ENOMEM	insufficient memory

**Context** These functions can be called from interrupt context only if (1) the default allocator is used and the `KM_NOSLEEP` flag is set, or (2) the specified allocator did not sleep for free memory (for example, if it uses a pre-allocated buffer for memory allocations).

See [nvlst\\_alloc\(9F\)](#) for a description of pluggable allocators and `KM_NOSLEEP`. These functions can be called from user or kernel context in all cases.



**Name** nvlst\_alloc, nvlst\_free, nvlst\_size, nvlst\_pack, nvlst\_unpack, nvlst\_dup, nv\_alloc\_init, nv\_alloc\_fini, nvlst\_xalloc, nvlst\_xpack, nvlst\_xunpack, nvlst\_xdup, nvlst\_merge – Manage a name-value pair list

**Synopsis** #include <sys/nvpair.h>

List Manipulation:

```
int nvlst_alloc(nvlst_t **nvlp, uint_t nvflag,
               int kmflag);

int nvlst_xalloc(nvlst_t **nvlp, uint_t nvflag, nv_alloc_t *nva);

void nvlst_free(nvlst_t *nvl);

int nvlst_size(nvlst_t *nvl, size_t *size, int encoding);

int nvlst_pack(nvlst_t *nvl, char **bufp, size_t *buflen, int encoding,
               int flag);

int nvlst_xpack(nvlst_t *nvl, char **bufp, size_t *buflen, int encoding,
                nv_alloc_t *nva);

int nvlst_unpack(char *buf, size_t buflen, nvlst_t **nvlp, int flag);

int nvlst_xunpack(char *buf, size_t buflen, nvlst_t **nvlp,
                  nv_alloc_t *nva);

int nvlst_dup(nvlst_t *nvl, nvlst_t **nvlp, int flag);

int nvlst_xdup(nvlst_t *nvl, nvlst_t **nvlp, nv_alloc_t *nva);

int nvlst_merge(nvlst_t *dst, nvlst_t *nvl, int flag);
```

Pluggable Allocator Configuration:

```
nv_alloc_t *nvlst_lookup_nv_alloc(nvlst_t *);

int nv_alloc_init(nv_alloc_t *nva,
                  const nv_alloc_ops_t * nvo, /* args */ ...);

void nv_alloc_reset(nv_alloc_t *nva);

void nv_alloc_fini(nv_alloc_t *nva);
```

Pluggable Allocation Initialization with Fixed Allocator:

```
int nv_alloc_init(nv_alloc_t *nva,
                  nv_fixed_ops, void * bufptr, size_t sz);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *nvlp* Address of a pointer to list of name-value pairs (nvlst\_t).  
*nvflag* Specify bit fields defining nvlst\_t properties:

	<code>NV_UNIQUE_NAME</code>	nvpair names are unique.
	<code>NV_UNIQUE_NAME_TYPE</code>	Name-data type combination is unique
<i>kmflag</i>		Kernel memory allocation policy, either <code>KM_SLEEP</code> or <code>KM_NOSLEEP</code> .
<i>nvl</i>		<code>nvlist_t</code> to be processed.
<i>dst</i>		Destination <code>nvlist_t</code> .
<i>size</i>		Pointer to buffer to contain the encoded size.
<i>bufp</i>		Address of buffer to pack <code>nvlist</code> into. Must be 8-byte aligned. If <code>NULL</code> , library will allocate memory.
<i>buf</i>		Buffer containing packed <code>nvlist_t</code> .
<i>buflen</i>		Size of buffer <i>bufp</i> or <i>buf</i> points to.
<i>encoding</i>		Encoding method for packing.
<i>nvo</i>		Pluggable allocator operations pointer ( <code>nv_alloc_ops_t</code> ).
<i>nva</i>		Points to a <code>nv_alloc_t</code> structure to be used for the specified <code>nvlist_t</code> .

### Description List Manipulation:

The `nvlist_alloc()` function allocates a new name-value pair list and updates *nvlp* to point to the handle. The argument *nvflag* specifies `nvlist_t` properties to remain persistent across packing, unpacking, and duplication.

If `NV_UNIQUE_NAME` is specified for *nvflag*, existing nvpairs with matching names are removed before the new nvpair is added. If `NV_UNIQUE_NAME_TYPE` is specified for *nvflag*, existing nvpairs with matching names and data types are removed before the new nvpair is added. See [nvlist\\_add\\_byte\(9F\)](#) for more details.

The `nvlist_xalloc()` function differs from `nvlist_alloc()` in that `nvlist_xalloc()` can use a different allocator, as described in the Pluggable Allocators section.

The `nvlist_free()` function frees a name-value pair list.

The `nvlist_size()` function returns the minimum size of a contiguous buffer large enough to pack *nvl*. The *encoding* parameter specifies the method of encoding when packing *nvl*. Supported encoding methods are:

`NV_ENCODE_NATIVE`      Straight `bcopy()` as described in [bcopy\(9F\)](#).

`NV_ENCODE_XDR`          Use XDR encoding, suitable for sending to another host.

The `nvlist_pack()` function packs *nvl* into contiguous memory starting at *\*bufp*. The *encoding* parameter specifies the method of encoding (see above).

- If *\*bufp* is not NULL, *\*bufp* is expected to be a caller-allocated buffer of size *\*buflen*. The *kmflag* argument is ignored.
- If *\*bufp* is NULL, the library allocates memory and updates *\*bufp* to point to the memory and updates *\*buflen* to contain the size of the allocated memory. The value of *kmflag* indicates the memory allocation policy

The `nvlst_xpack()` function differs from `nvlst_pack()` in that `nvlst_xpack()` can use a different allocator.

The `nvlst_unpack()` function takes a buffer with a packed `nvlst_t` and unpacks it into a searchable `nvlst_t`. The library allocates memory for `nvlst_t`. The caller is responsible for freeing the memory by calling `nvlst_free()`.

The `nvlst_xunpack()` function differs from `nvlst_unpack()` in that `nvlst_xunpack()` can use a different allocator.

The `nvlst_dup()` function makes a copy of *nvl* and updates *nvlp* to point to the copy.

The `nvlst_xdup()` function differs from `nvlst_dup()` in that `nvlst_xdup()` can use a different allocator.

The `nvlst_merge()` function adds copies of all name-value pairs from `nvlst_t nvl` to `nvlst_t dst`. Name-value pairs in *dst* are replaced with name-value pairs from *nvl* which have identical names (if *dst* has the type `NV_UNIQUE_NAME`), or identical names and types (if *dst* has the type `NV_UNIQUE_NAME_TYPE`).

The `nvlst_lookup_nv_alloc()` function retrieves the pointer to the allocator used when manipulating a name-value pair list.

#### PLUGGABLE ALLOCATORS Using Pluggable Allocators:

The `nv_alloc_init()`, `nv_alloc_reset()` and `nv_alloc_fini()` functions provide an interface that specifies the allocator to be used when manipulating a name-value pair list.

The `nv_alloc_init()` determines allocator properties and puts them into the *nva* argument. You need to specify the *nv\_arg* argument, the *nvo* argument and an optional variable argument list. The optional arguments are passed to the (`*nv_ao_init()`) function.

The *nva* argument must be passed to `nvlst_xalloc()`, `nvlst_xpack()`, `nvlst_xunpack()` and `nvlst_xdup()`.

The `nv_alloc_reset()` function resets the allocator properties to the data specified by `nv_alloc_init()`. When no (`*nv_ao_reset()`) function is specified, `nv_alloc_reset()` is without effect.

The `nv_alloc_fini()` destroys the allocator properties determined by `nv_alloc_init()`. When a `(*nv_ao_fini())` routine is specified, it is called from `nv_alloc_fini()`.

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

The `'nv_alloc_sleep'` and `'nv_alloc_nosleep'` `nv_alloc_t` pointers may be used with `nvlist_xalloc` to mimic the behavior of `nvlist_alloc` with `KM_SLEEP` and `KM_NOSLEEP`, respectively.

- o `nv_alloc_nosleep`
- o `nv_alloc_sleep`

The `nvpair` framework provides a fixed-buffer allocator, accessible via `nv_fixed_ops`.

- o `nv_fixed_ops`

Given a buffer size and address, the fixed-buffer allocator allows for the creation of `nvlists` in contexts where `malloc` or `kmem_alloc` services may not be available. The fixed-buffer allocator is designed primarily to support the creation of `nvlists`.

Memory freed using `nvlist_free()`, `pair-removal`, or similar routines is not reclaimed.

When used to initialize the fixed-buffer allocator, `nv_alloc_init` should be called as follows:

```
int nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo,
                 void *bufptr, size_t sz);
```

When invoked on a fixed-buffer, the `nv_alloc_reset()` function resets the fixed buffer and prepares it for re-use. The framework consumer is responsible for freeing the buffer passed to `nv_alloc_init()`.

#### CREATING PLUGGABLE ALLOCATORS

Any producer of name-value pairs may possibly specify his own allocator routines. You must provide the following pluggable allocator operations in the allocator implementation.

```
int (*nv_ao_init)(nv_alloc_t *nva, va_list nv_valist);
void (*nv_ao_fini)(nv_alloc_t *nva);
void *(*nv_ao_alloc)(nv_alloc_t *nva, size_t sz);
void (*nv_ao_reset)(nv_alloc_t *nva);
void (*nv_ao_free)(nv_alloc_t *nva, void *buf, size_t sz);
```

The `nva` argument of the allocator implementation is always the first argument.

The optional `(*nv_ao_init())` function is responsible for filling the data specified by `nv_alloc_init()` into the `nva_arg()` argument. The `(*nv_ao_init())` function is called only when `nv_alloc_init()` is executed.

The optional `(*nv_ao_fini())` function is responsible for the cleanup of the allocator implementation. It is called by `nv_alloc_fini()`.

The required (`*nv_ao_alloc()`) function is used in the nvpair allocation framework for memory allocation. The `sz` argument specifies the size of the requested buffer.

The optional (`*nv_ao_reset()`) function is responsible for resetting the `nva_arg` argument to the data specified by `nv_alloc_init()`.

The required (`*nv_ao_free()`) function is used in the nvpair allocator framework for memory de-allocation. The argument `buf` is a pointer to a block previously allocated by (`*nv_ao_alloc()`) function. The size argument `sz` must exactly match the original allocation.

The disposition of the allocated objects and the memory used to store them is left to the allocator implementation.

**Return Values** For `nvlst_alloc()`, `nvlst_dup()`, `nvlst_xalloc()`, and `nvlst_xdup()`:

0            success  
EINVAL      invalid argument  
ENOMEM     insufficient memory

For `nvlst_pack()`, `nvlst_unpack()`, `nvlst_xpack()`, and `nvlst_xunpack()`:

0            success  
EINVAL      invalid argument  
ENOMEM     insufficient memory  
EFAULT     encode/decode error  
ENOTSUP    encode/decode method not supported

For `nvlst_size()`:

0            success  
EINVAL      invalid argument

For `nvlst_lookup_nv_alloc()`:

pointer to the allocator

**Usage** The fixed-buffer allocator is very simple allocator. It uses a pre-allocated buffer for memory allocations and it can be used in interrupt context. You are responsible for allocation and de-allocation for the pre-allocated buffer.

**Examples**

```
/*
 * using the fixed-buffer allocator.
 */
#include <sys/nvpair.h>
```

```

/* initialize the nvpair allocator framework */
static nv_alloc_t *
init(char *buf, size_t size)
{
    nv_alloc_t *nvap;

    if ((nvap = kmem_alloc(sizeof(nv_alloc_t), KM_SLEEP)) == NULL)
        return (NULL);

    if (nv_alloc_init(nvap, nv_fixed_ops, buf, size) == 0)
        return (nvap);

    return (NULL);
}

static void
fini(nv_alloc_t *nvap)
{
    nv_alloc_fini(nvap);
    kmem_free(nvap, sizeof(nv_alloc_t));
}

static int
interrupt_context(nv_alloc_t *nva)
{
    nvlst_t *nvl;
    int error;

    if ((error = nvlst_xalloc(&nvl, NV_UNIQUE_NAME, nva)) != 0)
        return (-1);

    if ((error = nvlst_add_int32(nvl, "name", 1234)) == 0)
        error = send_nvl(nvl);

    nvlst_free(nvl);
    return (error);
}

```

**Context** The `nvlst_alloc()`, `nvlst_pack()`, `nvlst_unpack()`, and `nvlst_dup()` functions can be called from interrupt context only if the `KM_NOSLEEP` flag is set. They can be called from user context with any valid flag.

The `nvlst_xalloc()`, `nvlst_xpack()`, `nvlst_xunpack()`, and `nvlst_xdup()` functions can be called from interrupt context only if (1) the default allocator is used and the `KM_NOSLEEP` flag is set or (2) the specified allocator did not sleep for free memory (for example, it uses a pre-allocated buffer for memory allocations).

These functions can be called from user or kernel context with any valid flag.

**Name** nvlst\_lookup\_boolean, nvlst\_lookup\_boolean\_value, nvlst\_lookup\_byte, nvlst\_lookup\_int8, nvlst\_lookup\_int16, nvlst\_lookup\_int32, nvlst\_lookup\_int64, nvlst\_lookup\_uint8, nvlst\_lookup\_uint16, nvlst\_lookup\_uint32, nvlst\_lookup\_uint64, nvlst\_lookup\_string, nvlst\_lookup\_nvlist, nvlst\_lookup\_boolean\_array, nvlst\_lookup\_byte\_array, nvlst\_lookup\_int8\_array, nvlst\_lookup\_int16\_array, nvlst\_lookup\_int32\_array, nvlst\_lookup\_int64\_array, nvlst\_lookup\_uint8\_array, nvlst\_lookup\_uint16\_array, nvlst\_lookup\_uint32\_array, nvlst\_lookup\_uint64\_array, nvlst\_lookup\_string\_array, nvlst\_lookup\_nvlist\_array, nvlst\_lookup\_pairs – match name and type indicated by the interface name and retrieve data value

**Synopsis** #include <sys/nvpair.h>

```
int nvlst_lookup_boolean(nvlist_t *nvl, const char *name);

int nvlst_lookup_boolean_value(nvlist_t *nvl, const char *name,
                               boolean_t *val);

int nvlst_lookup_byte(nvlist_t *nvl, const char *name,
                      uchar_t *val);

int nvlst_lookup_int8(nvlist_t *nvl, const char *name,
                      int8_t *val);

int nvlst_lookup_uint8(nvlist_t *nvl, const char *name,
                       uint8_t *val);

int nvlst_lookup_int16(nvlist_t *nvl, const char *name,
                       int16_t *val);

int nvlst_lookup_uint16(nvlist_t *nvl, const char *name,
                        uint16_t *val);

int nvlst_lookup_int32(nvlist_t *nvl, const char *name,
                       int32_t *val);

int nvlst_lookup_uint32(nvlist_t *nvl, const char *name,
                        uint32_t *val);

int nvlst_lookup_int64(nvlist_t *nvl, const char *name,
                       int64_t *val);

int nvlst_lookup_uint64(nvlist_t *nvl, const char *name,
                        uint64_t *val);

int nvlst_lookup_string(nvlist_t *nvl, const char *name,
                        char **val);

int nvlst_lookup_nvlist(nvlist_t *nvl, const char *name,
                        nvlist_t **val);

int nvlst_lookup_boolean_array(nvlist_t *nvl, const char *name,
                               boolean_t **val, uint_t *nelem);
```

```

int nvlst_lookup_byte_array(nvlst_t *nvl, const char *name,
    uchar_t **val, uint_t *nelem);

int nvlst_lookup_int8_array(nvlst_t *nvl, const char *name,
    int8_t **val, uint_t *nelem);

int nvlst_lookup_uint8_array(nvlst_t *nvl, const char *name,
    uint8_t **val, uint_t *nelem);

int nvlst_lookup_int16_array(nvlst_t *nvl, const char *name,
    int16_t **val, uint_t *nelem);

int nvlst_lookup_uint16_array(nvlst_t *nvl, const char *name,
    uint16_t **val, uint_t *nelem);

int nvlst_lookup_int32_array(nvlst_t *nvl, const char *name,
    int32_t **val, uint_t *nelem);

int nvlst_lookup_uint32_array(nvlst_t *nvl, const char *name,
    uint32_t **val, uint_t *nelem);

int nvlst_lookup_int64_array(nvlst_t *nvl, const char *name,
    int64_t **val, uint_t *nelem);

int nvlst_lookup_uint64_array(nvlst_t *nvl, const char *name,
    uint64_t **val, uint_t *nelem);

int nvlst_lookup_string_array(nvlst_t *nvl, const char *name,
    char ***val, uint_t *nelem);

int nvlst_lookup_nvlst_array(nvlst_t *nvl, const char *name,
    nvlst_t ***val, uint_t *nelem);

int nvlst_lookup_pairs(nvlst_t *nvl, int flag...);

```

**Interface Level** Solaris DDI specific (Solaris DDI)

<b>Parameters</b>	<i>nvl</i>	The list of name-value pairs ( <i>nvlst_t</i> ) to be processed.
	<i>name</i>	Name of the name-value pair ( <i>nvpair</i> ) to search.
	<i>nelem</i>	Address to store the number of elements in value.
	<i>val</i>	Address to store the value or starting address of the array value.
	<i>flag</i>	Specify bit fields defining lookup behavior:
	NV_FLAG_NOENTOK	The retrieval function will not fail if no matching name-value pair is found.

**Description** These functions find the *nvpair* that matches the name and type as indicated by the interface name. If one is found, *nelem* and *val* are modified to contain the number of elements in value and the starting address of data, respectively.



These interfaces work for `nvlst_t` allocated with `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` specified in `nvlst_alloc()`. See [nvlst\\_alloc\(9F\)](#). If this is not the case, the interface will return `ENOTSUP` because the list potentially contains multiple `nvpairs` with the same name and type.

Multiple threads can simultaneously read the same `nvlst_t` but only one thread should actively change a given `nvlst_t` at a time. The caller is responsible for the synchronization.

All memory required for storing the array elements, including string values, are managed by the library. References to such data remain valid until `nvlst_free()` is called on `nvl`.

The `nvlst_lookup_pairs()` function retrieves a set of `nvpairs`. The arguments are a null-terminated list of pairs (data type `DATA_TYPE_BOOLEAN`), triples (non-array data types) or quads (array data types). As shown below, the interpretation of the arguments depends on the value of `type`. See [nvpair\\_type\(9F\)](#).

*name*      Name of the name-value pair to search.

*type*      Data type.

*val*        Address to store the starting address of the value. When using data type `DATA_TYPE_BOOLEAN`, the *val* argument is ignored.

*nelem*     Address to store the number of elements in value. Non-array data types have only one argument and *nelem* is ignored.

The argument order is *name*, *type*, [*val*], [*nelem*].

When using `NV_FLAG_NOENTOK` and no matching name-value pair is found, the memory pointed to by *val* and *nelem* is not touched.

These functions return `0` on success and an error value on failure.

**Errors** These functions fail under the following conditions.

<code>0</code>	Success
<code>EINVAL</code>	Invalid argument
<code>ENOENT</code>	No matching name-value pair found
<code>ENOTSUP</code>	Encode/decode method not supported

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [nvlst\\_alloc\(9F\)](#), [nvpair\\_type\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** nvlst\_lookup\_nvpair, nvlst\_exists – lookup named pairs

**Synopsis** #include <libnvpair.h>

```
int nvlst_lookup_nvpair(nvlst_t *nvl, const char *name,
                       nvpair_t **nvp);

boolean_t nvlst_exists(nvlst_t *nvl, const char *name);
```

**Description** The `nvlst_lookup_nvpair()` function returns the `nvpair` with the matching name, regardless of type. It is valid only for lists allocated with `NV_UNIQUE_NAME`. See [nvlst\\_alloc\(9F\)](#).

The `nvlst_exists()` function returns success if any `nvpair` exists with the given name. It is valid for all types of lists.

**Return Values** The `nvlst_lookup_nvpair()` function returns 0 on success and an error value on failure.

The `nvlst_exists()` function returns `B_TRUE` if an `nvpair` with the given name exists and `B_FALSE` otherwise.

**Errors** The `nvlst_lookup_nvpair()` function will fail if:

`EINVAL`     There is an invalid argument.  
`ENOENT`     No matching name-value pair is found.  
`ENOTSUP`    The list was not allocated with `NV_UNIQUE_NAME`.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [nvlst\\_lookup\\_nvpair\(3NVP AIR\)](#), [nvlst\\_alloc\(9F\)](#), [attributes\(5\)](#)

**Name** nvlst\_next\_nvpair, nvpair\_name, nvpair\_type – return data regarding name-value pairs

**Synopsis** #include <sys/nvpair.h>

```
nvpair_t *nvlst_next_nvpair(nvlst_t *nvl, nvpair_t *nvpair);
char *nvpair_name(nvpair_t *nvpair);
data_type_t nvpair_type(nvpair_t *nvpair);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *nvl* The list of name-value pairs (nvlst\_t) to be processed.

*nvpair* Handle to a name-value pair.

**Description** The nvlst\_next\_nvpair() function returns a handle to the next name-value pair (nvpair) in the list following *nvpair*. If *nvpair* is NULL, the first pair is returned. If *nvpair* is the last pair in the nvlst\_t, NULL is returned.

The nvpair\_name() function returns a string containing the name of *nvpair*.

The nvpair\_type() function retrieves the value of the *nvpair* in the form of enumerated type data\_type\_t. This is used to determine the appropriate nvpair\_\*() function to call for retrieving the value.

**Return Values** For nvpair\_name(): a string containing the name.

For nvpair\_type(): an enumerated data type data\_type\_t. Possible values for data\_type\_t are:

```
DATA_TYPE_BOOLEAN
DATA_TYPE_BOOLEAN_VALUE
DATA_TYPE_BYTE
DATA_TYPE_INT8
DATA_TYPE_UINT8
DATA_TYPE_INT16
DATA_TYPE_UINT16
DATA_TYPE_INT32
DATA_TYPE_UINT32
DATA_TYPE_INT64
DATA_TYPE_UINT64
DATA_TYPE_STRING
DATA_TYPE_NVLST
DATA_TYPE_BOOLEAN_ARRAY
DATA_TYPE_BYTE_ARRAY
DATA_TYPE_INT8_ARRAY
DATA_TYPE_UINT8_ARRAY
DATA_TYPE_INT16_ARRAY
DATA_TYPE_UINT16_ARRAY
```

```
DATA_TYPE_INT32_ARRAY  
DATA_TYPE_UINT32_ARRAY  
DATA_TYPE_INT64_ARRAY  
DATA_TYPE_UINT64_ARRAY  
DATA_TYPE_STRING_ARRAY  
DATA_TYPE_NVLST_ARRAY
```

After `nvpairs` is removed from or replaced in an `nvlst`, it cannot be manipulated. This includes `nvlst_next_nvpair()`, `nvpair_name()` and `nvpair_type()`. Replacement can happen during pair addition on `nvlsts` created with `NV_UNIQUE_NAME_TYPE` and `NV_UNIQUE_NAME`. See [nvlst\\_alloc\(9F\)](#) for more details.

**Context** These functions can be called from user, interrupt, or kernel context.

**Name** nvlst\_remove, nvlst\_remove\_all – remove name-value pairs

**Synopsis** #include <sys/nvpair.h>

```
int nvlst_remove(nvlst_t *nvl, const char *name, data_type_t type);  
int nvlst_remove_all(nvlst_t *nvl, const char *name);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *nvl* The list of name-value pairs (nvlst\_t) to be processed.  
*name* Name of the name-value pair (nvpair) to be removed.  
*type* Data type of the nvpair to be removed.

**Description** The nvlst\_remove() function removes the first occurrence of *nvpair* that matches the name and the type.

The nvlst\_remove\_all() function removes all occurrences of *nvpair* that match the name, regardless of type.

Multiple threads can simultaneously read the same nvlst\_t but only one thread may actively change a given nvlst\_t at a time. The caller is responsible for the synchronization.

**Return Values** These functions return 0 on success and an error value on failure.

**Context** The nvlst\_remove() and nvlst\_remove\_all() functions can be called from user, interrupt, or kernel context.

**Errors** EINVAL There is an invalid argument.  
ENOENT No name-value pairs were found to match the criteria specified by name and type.

**Name** nvpair\_value\_byte, nvpair\_value\_nvlist, nvpair\_value\_int8, nvpair\_value\_int16, nvpair\_value\_int32, nvpair\_value\_int64, nvpair\_value\_uint8, nvpair\_value\_uint16, nvpair\_value\_uint32, nvpair\_value\_uint64, nvpair\_value\_string, nvpair\_value\_boolean\_array, nvpair\_value\_byte\_array, nvpair\_value\_nvlist\_array, nvpair\_value\_int8\_array, nvpair\_value\_int16\_array, nvpair\_value\_int32\_array, nvpair\_value\_int64\_array, nvpair\_value\_uint8\_array, nvpair\_value\_uint16\_array, nvpair\_value\_uint32\_array, nvpair\_value\_uint64\_array, nvpair\_value\_string\_array – retrieve value from a name-value pair

**Synopsis** #include <sys/nvpair.h>

```
int nvpair_value_boolean_value(nvpair_t *nvpair, boolean_t *val);
int nvpair_value_byte(nvpair_t *nvpair, uchar_t *val);
int nvpair_value_int8(nvpair_t *nvpair, int8_t *val);
int nvpair_value_uint8(nvpair_t *nvpair, uint8_t *val);
int nvpair_value_int16(nvpair_t *nvpair, int16_t *val);
int nvpair_value_uint16(nvpair_t *nvpair, uint16_t *val);
int nvpair_value_int32(nvpair_t *nvpair, int32_t *val);
int nvpair_value_uint32(nvpair_t *nvpair, uint32_t *val);
int nvpair_value_int64(nvpair_t *nvpair, int64_t *val);
int nvpair_value_uint64(nvpair_t *nvpair, uint64_t *val);
int nvpair_value_string(nvpair_t *nvpair, char **val);
int nvpair_value_nvlist(nvpair_t *nvpair, nvlist_t **val);
int nvpair_value_boolean_array(nvpair_t *nvpair, boolean_t **val,
    uint_t *nelem);
int nvpair_value_byte_array(nvpair_t *nvpair, uchar_t **val,
    uint_t *nelem);
int nvpair_value_int8_array(nvpair_t *nvpair, int8_t **val,
    uint_t *nelem);
int nvpair_value_uint8_array(nvpair_t *nvpair, uint8_t **val,
    uint_t *nelem);
int nvpair_value_int16_array(nvpair_t *nvpair, int16_t **val,
    uint_t *nelem);
int nvpair_value_uint16_array(nvpair_t *nvpair, uint16_t **val,
    uint_t *nelem);
int nvpair_value_int32_array(nvpair_t *nvpair, int32_t **val,
    uint_t *nelem);
```

```
int nvpair_value_uint32_array(nvpair_t *nvpair, uint32_t **val,
                             uint_t *nelem);

int nvpair_value_int64_array(nvpair_t *nvpair, int64_t **val,
                             uint_t *nelem);

int nvpair_value_uint64_array(nvpair_t *nvpair, uint64_t **val,
                             uint_t *nelem);

int nvpair_value_string_array(nvpair_t *nvpair, char ***val,
                              uint_t *nelem);

int nvpair_value_nvlist_array(nvpair_t *nvpair, nvlist_t ***val,
                              uint_t *nelem);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *nvpair* Name-value pair (nvpair) to be processed.  
*nelem* Address to store the number of elements in value.  
*val* Address to store the value or starting address of array value.

**Description** These functions retrieve the value of *nvpair*. The data type of *nvpair* must match the function name for the call to be successful.

There is no `nvpair_value_boolean()`; the existence of the name implies the value is true.

For array data types, including string, the memory containing the data is managed by the library and references to the value remains valid until `nvlist_free()` is called on the `nvlist_t` from which *nvpair* is obtained. See [nvlist\\_free\(9F\)](#)

The value of an *nvpair* may not be retrieved after the *nvpair* having been removed from or replaced in an *nvlist*. Replacement can happen during pair addition on *nvlists* created with `NV_UNIQUE_NAME_TYPE` and `NV_UNIQUE_NAME`. See [nvlist\\_alloc\(9F\)](#) for more details.

**Return Values** `0` Success  
`EINVAL` Either one of the arguments is `NULL` or type of *nvpair* does not match the interface name.

**Context** These functions can be called from user, interrupt, or kernel context.



- Name** OTHERQ, otherq – get pointer to queue's partner queue
- Synopsis** `#include <sys/stream.h>`  
`#include <sys/ddi.h>`
- ```
queue_t *OTHERQ(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the queue.
- Description** The OTHERQ() function returns a pointer to the other of the two queue structures that make up a STREAMS module or driver. If *q* points to the read queue the write queue will be returned, and vice versa.
- Return Values** The OTHERQ() function returns a pointer to a queue's partner.
- Context** The OTHERQ() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Setting Queues

This routine sets the minimum packet size, the maximum packet size, the high water mark, and the low water mark for the read and write queues of a given module or driver. It is passed either one of the queues. This could be used if a module or driver wished to update its queue parameters dynamically.

```
1 void
2 set_q_params(q, min, max, hi, lo)
3     queue_t *q;
4     short min;
5     short max;
6     ushort_t hi;
7     ushort_t lo;
8 {
9     q->q_minpsz = min;
10    q->q_maxpsz = max;
11    q->q_hiwat = hi;
12    q->q_lowat = lo;
13    OTHERQ(q)->q_minpsz = min;
14    OTHERQ(q)->q_maxpsz = max;
15    OTHERQ(q)->q_hiwat = hi;
16    OTHERQ(q)->q_lowat = lo;
17 }
```

**See Also** [Writing Device Drivers](#)

[STREAMS Programming Guide](#)

**Name** outb, outw, outl, repoutsb, repoutsw, repoutsd – write to an I/O port

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
void outb(int port, unsigned char value);
void outw(int port, unsigned short value);
void outl(int port, unsigned long value);
void repoutsb(int port, unsigned char *addr, int count);
void repoutsw(int port, unsigned short *addr, int count);
void repoutsd(int port, unsigned long *addr, int count);
```

**Interface Level** The functions described here are obsolete. For the `outb()`, `outw()`, and `outl()` functions use, respectively, `ddi_put8(9F)`, `ddi_put16(9F)`, and `ddi_put32(9F)` instead. For `repoutsb()`, `repoutsw()`, and `repoutsl()`, use, respectively, `ddi_rep_put8(9F)`, `ddi_rep_put16(9F)`, and `ddi_rep_put32(9F)` instead.

**Parameters** *port* A valid I/O port address.  
*value* The data to be written to the I/O port.  
*addr* The address of a buffer from which the values will be fetched.  
*count* The number of values to be written to the I/O port.

**Description** These routines write data of various sizes to the I/O port with the address specified by *port*.

The `outb()`, `outw()`, and `outl()` functions write 8 bits, 16 bits, and 32 bits of data respectively, writing the data specified by *value*.

The `repoutsb()`, `repoutsw()`, and `repoutsd()` functions write multiple 8-bit, 16-bit, and 32-bit values, respectively. *count* specifies the number of values to be written. *addr* is a pointer to a buffer from which the output values are fetched.

**Context** These functions may be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Architecture        | x86             |
| Interface Stability | Obsolete        |

**See Also** [isa\(4\)](#), [attributes\(5\)](#), [ddi\\_put8\(9F\)](#), [ddi\\_put16\(9F\)](#), [ddi\\_put32\(9F\)](#), [ddi\\_rep\\_put8\(9F\)](#), [ddi\\_rep\\_put16\(9F\)](#), [ddi\\_rep\\_put32\(9F\)](#), [inb\(9F\)](#)

*Writing Device Drivers*

**Name** packet\_events – packet events overview

**Description** The implementation of the [ip\(7P\)](#) protocol in Solaris supports a number of packet events that are made available through [netinfo\(9F\)](#). The packet events that are currently available to programmers and applications are listed below. Each packet event below includes a description of the event.

The hook function (`h_func`) supplied in the [hook\\_t\(9S\)](#) structure passed to [net\\_hook\\_register\(9F\)](#) is called once for each packet. Returning more than one packet by the hook function by changing `b_next` is not supported. Only one packet at a time is passed in. On return from the hook function, one packet can be returned.

When the hook function is called, it is passed a pointer to a `hook_pkt_event_t` structure via the `info (hook_data_t)` argument. See [hook\\_pkt\\_event\(9S\)](#) for more details about the fields in this structure and how they are used with respect to the events described herein.

**IP Events** IP events are available individually for each of the IPv4 and IPv6 protocols. Thus a party that is interested in both protocols must register two hooks: one for each protocol family. Similarly, if only one of the two protocols is of interest then the event for that protocol family can be subscribed to with no need to worry about packets from the other being presented.

**Forwarding Events** Packet Event: Forwarding  
Macro: `NH_FORWARDING`  
Name: `forwarding`

IP packets that are received by the host but are destined for another host where the host is doing routing across multiple network interfaces will be delivered to the forwarding packet event when the host has determined the next outgoing network interface. Packets are received by this event after they have been processed by the Internet Load Balancer (see [ilbadm\(1M\)](#)).

Packets that are routed using IPFilter's policy based routing features will not be seen by the forwarding hook as they do not pass through the kernel's routing and instead use [net\\_inject\(9F\)](#) to deliver packets directly to a network interface.

**Loopback Events** Loopback packet events are generated to simulate the transmission and reception of IP packets inside the kernel when it is sending a packet to itself. Packets addressed to the loopback address (127.0.0.1) will can be seen at these events. Additionally, packets that originate from an IP address that is assigned to a network interface and that carry the IP address of a network interface on the same host in the destination field also fall into this category.

The loopback events are the only packet events for which a packet can be received on the loopback interface (for example, `lo0`) or any other network interface that has the loopback flag set.

Network traffic that “transits” between zones that are sharing an instance of IP networking will also trigger loopback packet events. A local zone that is sharing its network instance with the global zone will typically have network interfaces named such as `e1000g0:1`, and there will be no network interfaces that are not logical network interfaces.

If a firewall or other software package is only interested in packets that are being sent and received by the physical hardware and all networking activity local to the host is considered to be safe then it is not necessary to subscribe to the loopback events.

Any software that receives packets from the loopback events must be aware that the packets are optimized for handling within the kernel. Thus it is possible that checksums may be missing or just incorrect because the system has no need to guard against the packet being corrupted in transmission.

```
Packet Event: Loopback In
Macro: NH_LOOPBACK_IN
Name: LOOPBACK_IN
```

The “loopback in” event simulates the kernel receiving a packet from itself. For example, if a user where to run the command “ssh localhost”, a “loopback in” event would be generated for the initial connection establishment where the host “receives” a packet to port 22 from itself. In most cases, the “loopback in” event will be the second event for any packet that is local to the host; the first will be the “loopback out” event that corresponds to the application or kernel sending the packet to itself.

```
Packet Event: Loopback Out
Macro: NH_LOOPBACK_OUT
Name: LOOPBACK_OUT
```

The “loopback out” event simulates the kernel sending a packet to itself.

Physical Events The “Physical” packet events occur in IP to indicate the reception and transmission of packets by IP from an external source to IP. By way of example, all packets exchanged between network interfaces present in the [mac\(9F\)](#) layer and IP will pass through these events.

Each packet received will have the IP header aligned to a 32bit boundary. There will be no other references held to the packet, thus *db\_ref* from [dat\(9S\)](#) will be 1.

```
Packet Event: Physical In
Macro: NH_PHYSICAL_IN
Name: PHYSICAL_IN
```

The “physical in” event occurs for packets that have been accepted by IP and for which a very basic set of tests is completed. This currently includes ensuring that a packet is not from a loopback address nor destined to a loopback address.

All packets presented by the “physical in” will have an IP version number that matches the family of the event. Thus a packet received from the callback attached to the “physical in” event for IPv6 will always have an IP version number of 6.

Packets with impossible addresses, such as those in the loopback network address space for IPv4, will not be seen at the “physical in” event as they are filtered out by IP itself. However, other checks (such as ensuring the checksum is correct) are not done until after the packet passes through this event.

Packet Event: Physical Out  
Macro: NH\_PHYSICAL\_OUT  
Name: PHYSICAL\_OUT

The “physical out” event occurs for packets that are being transmitted by IP. There is one event per packet that is transmitted, thus packets that are internally generated and too big will be seen as fragments when this event occurs.

For network interfaces that support offloading of the checksum calculation from software into hardware for the IP suite of protocols, packets presented by this event may either have incorrect, missing (zero) or partial checksums. The function [net\\_ispartialchecksum\(9F\)](#) can be used to verify if the packet is expecting hardware to complete the checksum calculations.

**kstats** Each of the packet events presented here has its own kstat entry when it is available. To access a particular packet event's kstats, the use of the packet event's name as the kstat name and the protocol as the kstat module is supported.

The following example lists all kstats that have the name “PHYSICAL\_IN”. When present, there will be one for each protocol that is registered with `netinfo: arp, inet` and `inet6`.

```
# kstat -n PHYSICAL_IN
```

The following example displays the kstat information for the “PHYSICAL\_OUT” event that is associated with the “inet” module (protocol).

```
# kstat -m inet -n PHYSICAL_OUT
```

**See Also** [ilbadm\(1M\)](#), [ip\(7P\)](#), [datab\(9S\)](#), [hook\\_pkt\\_event\(9S\)](#), [hook\\_t\(9S\)](#), [mac\(9F\)](#), [net\\_ispartialchecksum\(9F\)](#), [net\\_hook\\_register\(9F\)](#), [net\\_inject\(9F\)](#), [netinfo\(9F\)](#)

**Name** `pci_config_get8`, `pci_config_get16`, `pci_config_get32`, `pci_config_get64`, `pci_config_put8`, `pci_config_put16`, `pci_config_put32`, `pci_config_put64`, `pci_config_getb`, `pci_config_getl`, `pci_config_getll`, `pci_config_getw`, `pci_config_putb`, `pci_config_putl`, `pci_config_putll`, `pci_config_putw` – read or write single datum of various sizes to the PCI Local Bus Configuration space

**Synopsis** `#include <sys/ddi.h>`  
`#include <sys/sunddi.h>`

```
uint8_t pci_config_get8(ddi_acc_handle_t handle, off_t offset);
uint16_t pci_config_get16(ddi_acc_handle_t handle, off_t offset);
uint32_t pci_config_get32(ddi_acc_handle_t handle, off_t offset);
uint64_t pci_config_get64(ddi_acc_handle_t handle, off_t offset);
void pci_config_put8(ddi_acc_handle_t handle, off_t offset,
    uint8_t value);
void pci_config_put16(ddi_acc_handle_t handle, off_t offset,
    uint16_t value);
void pci_config_put32(ddi_acc_handle_t handle, off_t offset,
    uint32_t value);
void pci_config_put64(ddi_acc_handle_t handle, off_t offset,
    uint64_t value);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The `pci_config_getb()`, `pci_config_getl()`, `pci_config_getll()`, `pci_config_getw()`, `pci_config_putb()`, `pci_config_putl()`, `pci_config_putll()`, and `pci_config_putw()` functions are obsolete. The `pci_config_get8()` function replaces `pci_config_getb()`. The `pci_config_get32()` function replaces `pci_config_getl()`. The `pci_config_get64()` function replaces `pci_config_getll()`. The `pci_config_get16()` function replaces `pci_config_getw()`. The `pci_config_put8()` function replaces `pci_config_putb()`. The `pci_config_put32()` function replaces `pci_config_putl()`. The `pci_config_put64()` function replaces `pci_config_putll()`. The `pci_config_put16()` function replaces `pci_config_putw()`.

**Parameters** *handle* The data access handle returned from `pci_config_setup(9F)`.  
*offset* Byte offset from the beginning of the PCI Configuration space.  
*value* Output data.

**Description** These routines read or write a single datum of various sizes from or to the PCI Local Bus Configuration space. The `pci_config_get8()`, `pci_config_get16()`, `pci_config_get32()`, and `pci_config_get64()` functions read 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The `pci_config_put8()`, `pci_config_put16()`, `pci_config_put32()`, and `pci_config_put64()` functions write 8 bits, 16 bits, 32 bits, and 64 bits of data, respectively. The *offset* argument must be a multiple of the datum size.

Since the PCI Local Bus Configuration space is represented in little endian data format, these functions translate the data from or to native host format to or from little endian format.

[pci\\_config\\_setup\(9F\)](#) must be called before invoking these functions.

**Return Values** [pci\\_config\\_get8\(\)](#), [pci\\_config\\_get16\(\)](#), [pci\\_config\\_get32\(\)](#), and [pci\\_config\\_get64\(\)](#) return the value read from the PCI Local Bus Configuration space.

**Context** These routines can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Interface Stability | <p><a href="#">pci_config_getb()</a>, <a href="#">pci_config_getl()</a>, <a href="#">pci_config_getll()</a>, <a href="#">pci_config_getw()</a>, <a href="#">pci_config_putb()</a>, <a href="#">pci_config_putl()</a>, <a href="#">pci_config_putll()</a>, <a href="#">pci_config_putw()</a> are Obsolete</p> <p><a href="#">pci_config_get8()</a>, <a href="#">pci_config_get16()</a>, <a href="#">pci_config_get32()</a>, <a href="#">pci_config_get64()</a>, <a href="#">pci_config_put8()</a>, <a href="#">pci_config_put16()</a>, <a href="#">pci_config_put32()</a>, <a href="#">pci_config_put64()</a> are Committed</p> |

**See Also** [attributes\(5\)](#), [pci\\_config\\_setup\(9F\)](#), [pci\\_config\\_teardown\(9F\)](#)



**Name** pci\_config\_setup, pci\_config\_tearardown – setup or tear down the resources for enabling accesses to the PCI Local Bus Configuration space

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_config_setup(dev_info_t *dip, ddi_acc_handle_t *handle);
void pci_config_tearardown(ddi_acc_handle_t *handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

*dip* Pointer to the device's dev\_info structure.

*handle* Pointer to a data access handle.

**Description** pci\_config\_setup() sets up the necessary resources for enabling subsequent data accesses to the PCI Local Bus Configuration space. pci\_config\_tearardown() reclaims and removes those resources represented by the data access handle returned from pci\_config\_setup().

**Return Values** pci\_config\_setup() returns:

DDI\_SUCCESS Successfully setup the resources.

DDI\_FAILURE Unable to allocate resources for setup.

**Context** pci\_config\_setup() must be called from user or kernel context. pci\_config\_tearardown() can be called from any context.

**Notes** These functions are specific to PCI bus device drivers. For drivers using these functions, a single source to support devices with multiple bus versions may not be easy to maintain.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|----------------|-----------------|
| Architecture   | PCI Local Bus   |

**See Also** [attributes\(5\)](#)

*IEEE 1275 PCI Bus Binding*

**Name** pci\_ereport\_setup, pci\_ereport\_teardown, pci\_ereport\_post – post error reports for the generic PCI errors logged in the PCI Configuration Status register.

**Synopsis** #include <sys/sunddi.h>

```
void pci_ereport_setup(dev_info_t *dip);
void pci_ereport_teardown(dev_info_t *dip);
void pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *dep,
    uin16_t *status);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

|               |                                                  |
|---------------|--------------------------------------------------|
| <i>dip</i>    | Pointer to the dev_info structure of the devices |
| <i>dep</i>    | Pointer to DDI error status                      |
| <i>status</i> | Pointer to status bit storage location           |

**Description** The pci\_ereport\_setup() function initializes support for error report generation and sets up the resources for subsequent access to PCI, PCI/X or PCI Express Configuration space. The caller must have established a fault management capability level of at least DDI\_FM\_EREPOR\_T\_CAPABLE with a previous call to ddi\_fm\_init() for *dip*.

The pci\_ereport\_teardown() function releases any resources allocated and set up by pci\_ereport\_setup() and associated with *dip*.

The pci\_ereport\_post() function is called to scan for and post any PCI, PCI/X or PCI Express Bus errors. On a PCI bus, for example, the errors detected include:

- Detected Parity Error
- Master Data Parity Error
- Target Abort
- Master Abort
- System Error
- Discard Timeout

The pci\_ereport\_post() function must be called only from a driver's error handler callback function. See ddi\_fm\_handler\_register(9F). The error\_status argument to the error handler callback function should be passed through as the dep argument to pci\_ereport\_post() as it may contain bus specific information that might be useful for handling any errors that are discovered.

The fme\_flag in the error\_status argument to the error handler callback function will contain one of the following:

DDI\_FM\_ERR\_UNEXPECTED() Any errors discovered are unexpected.

|                       |                                                                               |
|-----------------------|-------------------------------------------------------------------------------|
| DDI_FM_ERR_EXPECTED() | Errors discovered were the result of a DDI_ACC_CAUTIOUS operation.            |
| DDI_FM_ERR_POKE()     | Errors discovered are the result of a <a href="#">ddi_poke(9F)</a> operation. |
| DDI_FM_ERR_PEEK()     | Errors discovered are the result of a <a href="#">ddi_peek(9F)</a> operation. |

Error report events are generated automatically if `fme_flag` is set to `DDI_FM_ERR_UNEXPECTED` and the corresponding error bits are set in the various PCI, PCI/X or PCI Express Bus error registers of the device associated with *dip*. The generated error report events are posted to the Solaris Fault Manager, [fmd\(1M\)](#), for diagnosis.

If the status argument is non-null, `pci_ereport_post()` saves the contents of the PCI Configuration Status Register to `*status`. If it is not possible to read the PCI Configuration Status Register, -1 is returned in `*status` instead.

On return from the call to `pci_ereport_post()`, the `ddi_fm_error_t` structure pointed at by *dep* will have been updated, and the `fme_status` field contains one of the following values:

|                 |                                                                                                                                                                                                                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DDI_FM_OK       | No errors were detected which might affect this device instance.                                                                                                                                                                                                                                                      |
| DDI_FM_FATAL    | An error which is considered fatal to the operational state of the system was detected.                                                                                                                                                                                                                               |
| DDI_FM_NONFATAL | An error which is not considered fatal to the operational state of the system was detected. The <code>fme_acc_handle</code> or <code>fme_dma_handle</code> fields in the returned <code>ddi_fm_error_t</code> structure will typically reference a handle that belongs to the device instance that has been affected. |
| DDI_FM_UNKNOWN  | An error was detected, but the call was unable to determine the impact of the error on the operational state of the system. This is treated the same way as <code>DDI_FM_FATAL</code> unless some other device is able to evaluate the fault to be <code>DDI_FM_NONFATAL</code> .                                     |

**Context** The `pci_ereport_setup()` and `pci_ereport_tearardown()` functions must be called from user or kernel context.

The `pci_ereport_post()` function can be called in any context.

**Examples**

```
int xxx_fmcap = DDI_FM_EREPORT_CAPABLE | DDI_FM_ERRCB_CAPABLE;
```

```
xxx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd) {
    ddi_fm_init(dip, &xxx_fmcap, &xxx_ibc);
    if (xxx_fmcap & DDI_FM_ERRCB_CAPABLE)
        ddi_fm_handler_register(dip, xxx_err_cb);
    if (xxx_fmcap & DDI_FM_EREPORT_CAPABLE)
        pci_ereport_setup(dip);
}
```

```
    }

    xxx_err_cb(dev_info_t *dip, ddi_fm_error_t *errp) {
        uint16_t status;

        pci_ereport_post(dip, errp, &status);
        return (errp->fme_status);
    }

    xxx_detach(dev_info_t *dip, ddi_attach_cmd_t cmd) {

        if (xxx_fmcap & DDI_FM_EREPOR_T_CAPABLE)
            pci_ereport_teardown(dip);
        if (xxx_fmcap & DDI_FM_ERRRCB_CAPABLE)
            ddi_fm_handler_unregister(dip);
        ddi_fm_fini(dip);
    }
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [fmd\(1M\)](#), [attributes\(5\)](#), [ddi\\_fm\\_handler\\_register\(9F\)](#), [ddi\\_fm\\_init\(9F\)](#), [ddi\\_peek\(9F\)](#), [ddi\\_poke\(9F\)](#), [ddi\\_fm\\_error\(9S\)](#)

- Name** pci\_param\_free – free resources allocated by pci\_param\_get() and pci\_param\_get\_ioctl()
- Synopsis** #include <sys/sunddi.h>
- ```
int pci_param_free(pci_param_t param);
```
- Parameters** *param* A handle obtained from pci\_param\_list() or pci\_param\_get\_ioctl().
- Description** The pci\_param\_free() function frees the resources allocated by [pci\\_param\\_get\(9F\)](#) and [pci\\_param\\_get\\_ioctl\(9F\)](#). This function must be called by the device driver after obtaining the device parameters using the *param* handle.
- Return Values** DDI\_SUCCESS Successful completion.  
Non-zero value The *param* handle passed is invalid.
- Context** The pci\_param\_free() function can be called from kernel non-interrupt context.
- Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_param\\_get\(9F\)](#), [pci\\_param\\_get\\_ioctl\(9F\)](#)

**Name** pci\_param\_get – get the name-value pair list from the backend store

**Synopsis** #include <sys/sunddi.h>

```
int pci_param_get(dev_info_t *dip, pci_param_t *php);
```

**Parameters** *dip* A pointer to a dev\_info structure

*php* A pointer to a (void \*)param handle

On successful return a non—null handle is returned.

**Description** The param handle pointer (*php*) returned by pci\_param\_get() contains the name-value information of both PF (Physical Function) and its configured VF (Virtual Function) devices. The caller should use [pci\\_plist\\_get\(9F\)](#) to get the param list of the PF device and [pci\\_plist\\_getvf\(9F\)](#) to obtain the param lists for the configured VFs.

The name-value pairs are defined on a per device basis. There is one set of name-value pairs for the PF and each of the configured VFs. The name-value pairs are optional and may be absent for any or all of the devices.

The param handle is invalid after the device is detached.

The caller is responsible to free the *php* (param handle pointer) after each call to pci\_param\_get() using [pci\\_param\\_free\(9F\)](#).

**Return Values** DDI\_SUCCESS A valid param handle pointer *php* is returned.

Non-zero value No name-value pairs defined for devices.

**Usage** Use the DDI pci\_plist\_lookup\_xxx() on the param list of the PF/VF devices to obtain the name-value pairs. Validate all parameters as a whole; that is, both PF parameters and VF parameters. If the current configuration cannot be satisfied, the driver needs to fail the attach. The validation needs to be done before VFs are configured using [pciv\\_vf\\_config\(9F\)](#) interface. Apply these parameters and corresponding action for PF and VFs.

**Examples** EXAMPLE1 pci\_param\_get() example

```
pci_param_t    my_params;
pci_plist_t    pf_plist;
pci_plist_t    vf_plist[8];

labelp = NULL;
rval = pci_param_get(dip, &my_params);
if (rval || (my_params == NULL)) {
    cmn_err(CE_NOTE, "No params available\n");
    goto continue_with_attach;
}
rval = pci_plist_get(my_params, &pf_plist);
```

**EXAMPLE 1** pci\_param\_get() example (Continued)

```

if (rval || (pf_plist == NULL)) {
    cmn_err(CE_NOTE, "No params for PF \n");
    goto continue_with_attach;
}
for (i = 0; i < 8; i++) {
    rval = pci_plist_getvf(my_params, i, &vf_plist[i]);
    if (rval || (vf_plist[i] == NULL)) {
        cmn_err(CE_WARN, "No params for VF %d\n", i);
        continue;
    }
}
pci_param_free(my_params);
/*
 * Validate the PF and VF params lists.
 * Fail the attach if the params are incompatible or exceed the
 * resources available.
 */
continue_with_attach:

```

**Context** The `pci_param_get()` function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_param\\_free\(9F\)](#), [pci\\_plist\\_get\(9F\)](#), [pci\\_plist\\_getvf\(9F\)](#), [pciv\\_vf\\_config\(9F\)](#)

**Name** pci\_param\_get\_ioctl – extract the params for PF and VF drivers

**Synopsis** #include <sys/sunddi.h>

```
int pci_param_get_ioctl(dev_info_t *dip, intptr_t arg, int mode,
    pci_param_t *php);
```

**Parameters** *dip* Pointer to dev\_info structure.

*arg, mode* The device driver is expected to pass the arguments they receive in their ioctl call directly to this interface.

*php* A pointer to a (void \*) param handle.

**Description** The pci\_param\_get\_ioctl() function extracts the params from the *arg* argument and returns a pci\_param\_t handle in *php*. Drivers should call [pci\\_param\\_free\(9F\)](#) to free the param handle returned in this call after retrieving params from the handle. Drivers are expected to use [pci\\_plist\\_get\(9F\)](#) and [pci\\_plist\\_getvf\(9F\)](#) on the param handle to get the name-value list for the PF and VF devices.

**Return Values** DDI\_SUCCESS A valid param handle pointer *php* is returned

Non-zero value The call failed to extract params from *arg*.

**Context** The pci\_param\_get() function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_param\\_free\(9F\)](#), [pci\\_param\\_get\(9F\)](#), [pci\\_plist\\_get\(9F\)](#), [pci\\_plist\\_getvf\(9F\)](#)



**Name** pci\_plist\_get – get the name-value pair list for the PF device

**Synopsis** #include <sys/sunddi.h>

```
int pci_plist_get(pci_param_t param, pci_plist_t *plist_p);
```

**Parameters** *param* A handle obtained from [pci\\_param\\_get\(9F\)](#) or [pci\\_param\\_get\\_ioctl\(9F\)](#).

*plist\_p* A pointer to pci\_plist\_t where a non—null *plist\_p* is returned on successful return

**Description** The *plist\_p* returned by pci\_plist\_get() is similar to nvlist\_t and contains the name-value pairs for the PF (Physical Function) device. The pci\_plist\_t supports the following data types:

```
int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t,
uint64_t, char *, pci_plist_t
```

Arrays of the above data types are supported except pci\_plist\_t.

**Return Values** DDI\_SUCCESS On success.

Non-zero value There is no name-value list for PF device.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_param\\_free\(9F\)](#), [pci\\_param\\_get\(9F\)](#), [pci\\_plist\\_getvf\(9F\)](#)

**Name** pci\_plist\_getvf – get the name-value pair list for the VF device

**Synopsis** #include <sys/sunddi.h>

```
int pci_plist_getvf(pci_param_t param, uint16_t vf_index,
    pci_plist_t *vplist_p);
```

**Parameters**

- param* A handle obtained from [pci\\_param\\_get\(9F\)](#) or [pci\\_param\\_get\\_ioctl\(9F\)](#).
- vf\_index* 0 through #VFS -1
- plist\_p* A pointer to `pci_param_plist_t` where a non—null *plist\_p* is returned on successful return

**Description** The *vplist\_p* returned by `pci_plist_getvf()` is similar to `nvlist_t` and contains the name-value pairs for the VF (Virtual Function) device. The `pci_plist_t` supports the following data types:

`uchar`, `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`, `char *`, `pci_plist_t`

Arrays of the above data types are supported.

**Return Values**

- `DDI_SUCCESS` On success.
- Non-zero value There is no name-value list for VF device.

**Context** The `pci_param_get()` function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_param\\_free\(9F\)](#), [pci\\_param\\_get\(9F\)](#), [pci\\_param\\_get\\_ioctl\(9F\)](#), [pci\\_plist\\_get\(9F\)](#)

**Name** pci\_plist\_lookup, pci\_plist\_lookup\_int8, pci\_plist\_lookup\_uint8, pci\_plist\_lookup\_int16, pci\_plist\_lookup\_uint16, pci\_plist\_lookup\_int32, pci\_plist\_lookup\_uint32, pci\_plist\_lookup\_int64, pci\_plist\_lookup\_uint64, pci\_plist\_lookup\_string, pci\_plist\_lookup\_plist, pci\_plist\_lookup\_int8\_array, pci\_plist\_lookup\_uint8\_array, pci\_plist\_lookup\_int16\_array, pci\_plist\_lookup\_uint16\_array, pci\_plist\_lookup\_int32\_array, pci\_plist\_lookup\_uint32\_array, pci\_plist\_lookup\_int64\_array, pci\_plist\_lookup\_uint64\_array, pci\_plist\_lookup\_string\_array – name-value pair lookup functions

**Synopsis** #include <sys/sunddi.h>

```
int pci_plist_lookup_int8(pci_plist_t plist, const char *name,
    int8_t *val);

int pci_plist_lookup_uint8(pci_plist_t plist, const char *name,
    uint8_t *val);

int pci_plist_lookup_int16(pci_plist_t plist, const char *name,
    int16_t *val);

int pci_plist_lookup_uint16(pci_plist_t plist, const char *name,
    uint16_t *val);

int pci_plist_lookup_int32(pci_plist_t plist, const char *name,
    int32_t *val);

int pci_plist_lookup_uint32(pci_plist_t plist, const char *name,
    uint32_t *val);

int pci_plist_lookup_int64(pci_plist_t plist, const char *name,
    int64_t *val);

int pci_plist_lookup_uint64(pci_plist_t plist, const char *name,
    uint64_t *val);

int pci_plist_lookup_string(pci_plist_t plist, const char *name,
    char **val);

int pci_plist_lookup_plist(pci_plist_t plist, const char *name,
    pci_plist_t **val);

int pci_plist_lookup_int8_array(pci_plist_t plist, const char *name,
    int8_t *val, uint_t *nelem);

int pci_plist_lookup_uint8_array(pci_plist_t plist, const char *name,
    uint8_t *val, uint_t *nelem);

int pci_plist_lookup_int16_array(pci_plist_t plist, const char *name,
    int16_t *val, uint_t *nelem);

int pci_plist_lookup_uint16_array(pci_plist_t plist, const char *name,
    uint16_t *val, uint_t *nelem);

int pci_plist_lookup_int32_array(pci_plist_t plist, const char *name,
    int32_t *val, uint_t *nelem);
```

```

int pci_plist_lookup_uint32_array(pci_plist_t plist, const char *name,
    uint32_t *val, uint_t *nelem);

int pci_plist_lookup_int64_array(pci_plist_t plist, const char *name,
    int64_t *val, uint_t *nelem);

int pci_plist_lookup_uint64_array(pci_plist_t plist, const char *name,
    uint64_t *val, uint_t *nelem);

int pci_plist_lookup_string_array(pci_plist_t plist, const char *name,
    char **val, uint_t *nelem);

```

**Parameters**

- plist*     A handle obtained from [pci\\_plist\\_get\(9F\)](#) or [pci\\_plist\\_getvf\(9F\)](#).
- name*     Name of the name-value pair to search
- nelem*    Address to store the number of elements in value
- val*      Address to store the starting address of the value

**Description** These functions find the nvpair (name-value pair) that matches the name and type as indicated by the interface name. If one is found, *nelem* and *val* are modified to contain the number of elements in value and the starting address of data, respectively.

**Return Values** These functions return 0 on success and an error value on failure.

**Errors** These functions will fail if:

DDI_EINVAL	There is an invalid argument.
ENOENT	No matching name-value pair is found
ENOTSUP	An encode/decode method is not supported.

**Context** These functions can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Uncommitted

**See Also** [attributes\(5\)](#), [pci\\_plist\\_get\(9F\)](#), [pci\\_plist\\_getvf\(9F\)](#), [pciv\\_vf\\_config\(9F\)](#), [pci\\_param\\_get\\_ioctl\(9F\)](#)

- 
- Name** pci\_report\_pmcap – Report Power Management capability of a PCI device
- Synopsis**
- ```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pci_report_pmcap(dev_info_t *dip, int cap, void *arg);
```
- Interface Level** Solaris DDI specific (Solaris DDI)
- Parameters**
- dip* Pointer to the device's dev\_info structure
  - cap* Power management capability
  - arg* Argument for the capability
- Description** Some PCI devices provide power management capabilities in addition to those provided by the PCI Power Management Specification. The pci\_report\_pmcap() function reports those Power Management capabilities of the PCI device to the framework. Framework supports dynamic changing of the capability by allowing pci\_report\_pmcap() to be called multiple times. Following are the supported capabilities as indicated by the cap:
- PCI\_PM\_IDLE SPEED — The PCI\_PM\_IDLE SPEED value indicates the lowest PCI clock speed that a device can tolerate when idle, and is applicable only to 33 MHz PCI bus. arg represents the lowest possible idle speed in KHz (1 KHz is 1000 Hz). The integer value representing the speed should be cast to (void \*) before passing as arg to pci\_report\_pmcap().
- The special values of arg are:
- PCI\_PM\_IDLE SPEED\_ANY The device can tolerate any idle clock speed.
  - PCI\_PM\_IDLE SPEED\_NONE The device cannot tolerate slowing down of PCI clock even when idle.
- If the driver doesn't make this call, PCI\_PM\_IDLE SPEED\_NONE is assumed. In this case, one offending device can keep the entire bus from being power managed.
- Return Values** The pci\_report\_pmcap() function returns:
- DDI\_SUCCESS Successful reporting of the capability
  - DDI\_FAILURE Failure to report capability because of invalid argument(s)
- Context** The pci\_report\_pmcap() function can be called from user, kernel and interrupt context.
- Examples**
1. A device driver knows that the device it controls works with any clock between DC and 33 MHz as specified in *Section 4.2.3.1: Clock Specification of the PCI Bus Specification Revision 2.1*. The device driver makes the following call from its [attach\(9E\)](#):

```
if (pci_report_pmcap(dip, PCI_PM_IDLE SPEED, PCI_PM_IDLE SPEED_ANY) !=
    DDI_SUCCESS)
    cmn_err(CE_WARN, "%s%d: pci_report_pmcap failed\n",
```

```
ddi_driver_name(dip), ddi_get_instance(dip));
```

2. A device driver controls a 10/100 Mb Ethernet device which runs the device state machine on the chip from the PCI clock. For the device state machine to receive packets at 100 Mb, the PCI clock cannot drop below 4 MHz. The driver makes the following call whenever it negotiates a 100 Mb Ethernet connection:

```
if (pci_report_pmcap(dip, PCI_PM_IDLE SPEED, (void *)4000) !=
    DDI_SUCCESS)
    cmn_err(CE_WARN, "%s%d: pci_report_pmcap failed\n",
           ddi_driver_name(dip), ddi_get_instance(dip));
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** *Writing Device Drivers*

*PCI Bus Power Management Interface Specification Version 1.1*

*PCI Bus Specification Revision 2.1*

**Name** pci\_save\_config\_regs, pci\_restore\_config\_regs – save and restore the PCI configuration registers

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int pci_save_config_regs(dev_info_t *dip);
int pci_restore_config_regs(dev_info_t *dip);
```

**Interface Level** Solaris DDI-specific (Solaris DDI).

**Arguments** *dip* Pointer to the device's dev\_info structure.

**Description** pci\_save\_config\_regs() saves the current configuration registers on persistent system memory. pci\_restore\_config\_regs() restores configuration registers previously saved by pci\_save\_config\_regs().

pci\_save\_config\_regs() should be called by the driver's power() entry point before powering a device off (to PCI state D3). Likewise, pci\_restore\_config\_regs() should be called after powering a device on (from PCI state D3), but before accessing the device. See [power\(9E\)](#).

**Return Values** pci\_save\_config\_regs() and pci\_restore\_config\_regs() return:

DDI\_SUCCESS Operation completed successfully.

DDI\_FAILURE Operation failed to complete successfully.

**Context** Both these functions can be called from user or kernel context.

**Examples** EXAMPLE 1 Invoking the save and restore functions

```
static int
xx_power(dev_info_t *dip, int component, int level) {
    struct xx *xx;
    int rval = DDI_SUCCESS;

    xx = ddi_get_soft_state(xx_softcstate, ddi_get_instance(dip));
    if (xx == NULL) {
        return (DDI_FAILURE);
    }

    mutex_enter(&xx->x_mutex);

    switch (level) {
    case PM_LEVEL_D0:
        XX_POWER_ON(xx);
        if (pci_restore_config_regs(dip) == DDI_FAILURE) {
            /*
```

**EXAMPLE 1** Invoking the save and restore functions *(Continued)*

```

        * appropriate error path handling here
        */
        ...
        rval = DDI_FAILURE;
    }
    break;

case PM_LEVEL_D3:
    if (pci_save_config_regs(dip) == DDI_FAILURE) {
        /*
         * appropriate error path handling here
         */
        ...
        rval = DDI_FAILURE;
    }
    else {
        XX_POWER_OFF(xx);
    }
    break;

default:
    rval = DDI_FAILURE;
    break;
}

mutex_exit(&xx->x_mutex);
return (rval);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTETYPE       | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [power\(9E\)](#)

*Writing Device Drivers*

*PCI Bus Power Management Interface Specification Version 1.1*

*PCI Bus Specification Revision 2.1*



**Name** pciv\_send – enable communication between PCI SR-IOV PF and VF drivers

**Synopsis** #include <sys/sunddi.h>

```
int pciv_send(dev_info_t *dip, pciv_pvp_req_t *req);
```

**Parameters** *dip* Pointer to the dev\_info structure.

*req* Pointer to pciv\_pvp\_req structure.

```
typedef struct pciv_pvp_req {
    int                pvp_dstfunc;
    caddr_t            pvp_buf;
    size_t             pvp_nbyte;
    buf_cb_t           pvp_cb;
    caddr_t            pvp_cb_arg;
    uint_t             pvp_flag;
} pciv_pvp_req_t
```

*pvp\_dstfunc* VF index ranges from 1 to num\_vf if called by PF driver. PCIV\_PF if the caller is a VF driver.

*pvp\_buf* Buffer address of caller's buffer to be sent.

*pvp\_nbyte* Number of bytes to be transmitted, which must be less than 8k.

*pvp\_cb* Call back function pointer if the *pvp\_flag* is set as PCIV\_NOWAIT.

*pvp\_cb\_arg* Call back input argument for *pvp\_cb* if the *pvp\_flag* is set as PCIV\_NOWAIT.

*pvp\_flag* Must be one of the following:

PCIV\_NOWAIT Do not wait for receiver's acknowledgment response.

PCIV\_WAIT Wait until receiver acknowledges the transmission (default).

**Description** The pciv\_send() function is used by SR-IOV (Single-Root IO Virtualization)-capable PF (Physical Function) and VF (Virtual Function) drivers to communicate with each other. A PF driver can communicate with any of its VF drivers while a VF driver can only communicate with its PF driver. If *pvp\_flag* is set to PCIV\_NOWAIT, the call returns immediately and the callback routine in *pvp\_cb* is called when data in *pvp\_buf* has been transmitted to the destination. The caller is then allowed to free the buffer in its callback routine.

```
typedef void (*buf_cb_t)(int rc, caddr_t buf, size_t size, caddr_t cb_arg);
```

*rc* DDI return code for the transmission.

*buf* Buffer address of caller's buffer to be sent.

*size* Number of bytes to be transmitted.

*cb\_arg* Input argument the caller set when calling `pciv_send()`.

**Return Values** The `pciv_send()` function returns:

|                |                                                                                                                   |
|----------------|-------------------------------------------------------------------------------------------------------------------|
| DDI_SUCCESS    | The buffer has been sent successfully.                                                                            |
| DDI_ENOTSUP    | The device/driver does not support this operation. The caller may use other mechanisms, such as hardware mailbox. |
| DDI_EINVAL     | The <i>pvp_nbyte</i> or <i>pvp_dstfunc</i> argument is invalid.                                                   |
| ENOMEM         | The operation failed due to lack of resources.                                                                    |
| DDI_ETRANSPORT | The remote end did not register a call back to handle incoming transmission.                                      |
| DDI_FAILURE    | The call failed due unspecified reasons.                                                                          |

**Context** The `pciv_send()` function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Uncommitted     |
| MT-Level            | Unsafe          |

**See Also** [attributes\(5\)](#), [ddi\\_cb\\_register\(9F\)](#)

**Name** `pciv_vf_config` – obtain and configure VF configuration parameters for a device driver instance

**Synopsis** `#include <sys/sunddi.h>`

```
int pciv_vf_config(dev_info_t *dip, pciv_config_vf_t *vfcfg_p);
```

**Parameters** *dip*

pointer to a `dev_info` structure

*vfcfg\_p*

pointer to a `pciv_config_vf` structure

```
typedef enum {
    PCIV_VFCFG_PARAM,
    PCIV_VF_ENABLE,
    PCIV_EVT_VFENABLE_PRE,
    PCIV_EVT_VFENABLE_POST,
    PCIV_EVT_VFDISABLE_PRE,
    PCIV_EVT_VFDISABLE_POST
} pciv_vf_config_cmd;

typedef struct pciv_config_vf {
    int                version;
    pciv_vf_config_cmd cmd;
    uint16_t          num_vf;
    uint16_t          first_vf_offset;
    uint16_t          vf_stride;
    boolean_t         ari_cap;
    uint32_t          page_size;
} pciv_config_vf_t;
```

*cmd* pre/post VF Enable/Disable

*num\_vf* number of VFs to be used

*first\_vf\_offset* offset between 1st VF and PF

*vf\_stride* distance between VFs

*ari\_cap* ARI-capable hierarchy

*page\_size* system page size

**Description** The `pciv_vf_config()` function is used by SR-IOV (Single-Root IO Virtualization) device PF (Physical Function) drivers to obtain VF (Virtual Function) configuration parameters and to configure their VFs. When the *cmd* field is set as `PCIV_VFCFG_PARAM` the call returns with appropriate values in the rest of fields of `pciv_config_vf_t`.

The *num\_vf* field indicates the number of VFs that is defined in the backend store such as MD or the `/etc/pci.conf` file for this device. The *first\_vf\_offset*, *vf\_stride*, *ari\_cap*, and *page\_size* are the corresponding values retrieved from the SRIOV capability structure of the PF device.

After first calling `pciv_vf_config()` with `cmd` set to `PCIV_VFCFG_PARAM`, the caller can then call the interface again with `cmd` set to `PCIV_VF_ENABLE` to enable its VF devices, without modifying any of the `num_vf`, `first_vf_offset`, `vf_stride`, `ari_cap`, or `page_size` values that were returned when `cmd` was set as `PCIV_VFCFG_PARAM`.

**Return Values** The `pciv_vf_config()` function returns:

- `DDI_SUCCESS` The request was accepted and this PF's VFs are enabled.
- `DDI_INVALID` The command was other than `PCIV_VFCFG_PARAM` or `PCIV_VF_ENABLE`, the device is not IOV capable, `num_vf` was set to 0, and the call was made to enable the VF; or parameters of `PCIV_VF_ENABLE` do not correspond to the values returned by `PCIV_VFCFG_PARAM`.
- `DDI_FAILURE` Implementation-specific failures occurred.

**Context** The `pciv_vf_config()` function can be called from kernel non-interrupt context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTETYPE       | ATTRIBUTEVALUE |
|---------------------|----------------|
| Interface Stability | Uncommitted    |
| MT-Level            | Unsafe         |

**See Also** [attributes\(5\)](#), [ddi\\_cb\\_register\(9F\)](#), [pci\\_param\\_get\(9F\)](#)

**Name** physio, minphys – perform physical I/O

**Synopsis** #include <sys/types.h>  
#include <sys/buf.h>  
#include <sys/uio.h>

```
int physio(int(*strat)(struct buf*), struct buf *bp, dev_t dev,
           int rw, void (*mincnt)(struct buf*), struct uio *uio);

void minphys(struct buf *bp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

### Parameters

|           |               |                                                                                                                                                                                               |
|-----------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| physio()  | <i>strat</i>  | Pointer to device strategy routine.                                                                                                                                                           |
|           | <i>bp</i>     | Pointer to a <a href="#">buf(9S)</a> structure describing the transfer. If <i>bp</i> is set to NULL then <code>physio()</code> allocates one which is automatically released upon completion. |
|           | <i>dev</i>    | The device number.                                                                                                                                                                            |
|           | <i>rw</i>     | Read/write flag. This is either <code>B_READ</code> when reading from the device, or <code>B_WRITE</code> when writing to the device.                                                         |
|           | <i>mincnt</i> | Routine which bounds the maximum transfer unit size.                                                                                                                                          |
|           | <i>uio</i>    | Pointer to the <code>uio</code> structure which describes the user I/O request.                                                                                                               |
| minphys() | <i>bp</i>     | Pointer to a <code>buf</code> structure.                                                                                                                                                      |

**Description** `physio()` performs unbuffered I/O operations between the device *dev* and the address space described in the `uio` structure.

Prior to the start of the transfer `physio()` verifies the requested operation is valid by checking the protection of the address space specified in the `uio` structure. It then locks the pages involved in the I/O transfer so they can not be paged out. The device strategy routine, `strat()`, is then called one or more times to perform the physical I/O operations. `physio()` uses [biowait\(9F\)](#) to block until `strat()` has completed each transfer. Upon completion, or detection of an error, `physio()` unlocks the pages and returns the error status.

`physio()` uses `mincnt()` to bound the maximum transfer unit size to the system, or device, maximum length. `minphys()` is the system `mincnt()` routine for use with `physio()` operations. Drivers which do not provide their own local `mincnt()` routines should call `physio()` with `minphys()`.

`minphys()` limits the value of `bp->b_bcount` to a sensible default for the capabilities of the system. Drivers that provide their own `mincnt()` routine should also call `minphys()` to make sure they do not exceed the system limit.

**Return Values** `physio()` returns:

0                Upon success.

non-zero        Upon failure.

**Context** `physio()` can be called from user context only.

**See Also** [strategy\(9E\)](#), [biodone\(9F\)](#), [biowait\(9F\)](#), [buf\(9S\)](#), [uio\(9S\)](#)

*Writing Device Drivers*

**Warnings** Since `physio()` calls `biowait()` to block until each buf transfer is complete, it is the drivers responsibility to call [biodone\(9F\)](#) when the transfer is complete, or `physio()` will block forever.

**Name** pm\_busy\_component, pm\_idle\_component – control device component availability for Power Management

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_busy_component(dev_info_t *dip, int component);
int pm_idle_component(dev_info_t *dip, int component);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

*dip* Pointer to the device's dev\_info structure.

*component* The number of the component to be power-managed.

**Description** The pm\_busy\_component() function sets *component* of *dip* to be busy. Calls to pm\_busy\_component() are stacked, requiring a corresponding number of calls to pm\_idle\_component() to make the component idle again. When a device is busy it will not be power-managed by the system.

The pm\_idle\_component() function marks *component* idle, recording the time that *component* went idle. This function must be called once for each call to pm\_busy\_component(). A component which is idle is available to be power-managed by the system. The pm\_idle\_component() function has no effect if the component is already idle, except to update the system's notion of when the device went idle.

If these functions are called as a result of entry into the driver's [attach\(9E\)](#), [detach\(9E\)](#) or [power\(9E\)](#) entry point, these functions must be called from the same thread which entered [attach\(9E\)](#), [detach\(9E\)](#) or [power\(9E\)](#).

**Return Values** The pm\_busy\_component() and pm\_idle\_component() functions return:

DDI\_SUCCESS Successfully set the indicated component busy or idle.

DDI\_FAILURE Invalid component number *component* or the device has no components.

**Context** These functions can be called from user or kernel context. These functions may also be called from interrupt context, providing they are not the first Power Management function called by the driver.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [pm\(7D\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [power\(9E\)](#), [pm\\_raise\\_power\(9F\)](#), [pm\(9P\)](#), [pm-components\(9P\)](#)

*Writing Device Drivers*



- Name** pm\_power\_has\_changed – Notify Power Management framework of autonomous power level change
- Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

int pm_power_has_changed(dev_info_t *dip, int component, int level);
```
- Interface Level** Solaris DDI specific (Solaris DDI)
- Parameters** *dip* Pointer to the device dev\_info structure  
*component* Number of the component that has changed power level  
*level* Power level to which the indicated component has changed
- Description** The pm\_power\_has\_changed(9) function notifies the Power Management framework that the power level of component of *dip* has changed to *level*.
- Normally power level changes are initiated by the Power Management framework due to device idleness, or through a request to the framework from the driver via pm\_raise\_power(9F) or pm\_lower\_power(9F), but some devices may change power levels on their own. For the framework to track the power level of the device under these circumstances, the framework must be notified of autonomous power level changes by a call to pm\_power\_has\_changed().
- Because of the asynchronous nature of these events, the Power Management framework might have called power(9E) between the device's autonomous power level change and the driver calling pm\_power\_has\_changed(), or the framework may be in the process of changing the power level when pm\_power\_has\_changed() is called. To handle these situations correctly, the driver should verify that the device is indeed at the level or set the device to the level if it does not support the inquiry of power levels, before calling pm\_power\_has\_changed(). In addition, the driver should prevent a power(9E) entry point from running in parallel with pm\_power\_has\_changed().
- If this function is called as a result of entry into the driver's [attach\(9E\)](#), [detach\(9E\)](#) or [power\(9E\)](#) entry point, this function must be called from the same thread which entered [attach\(9E\)](#), [detach\(9E\)](#) or [power\(9E\)](#).
- Return Values** The pm\_power\_has\_changed() function returns:
- DDI\_SUCCESS The power level of component was successfully updated to *level*.  
DDI\_FAILURE Invalid component *component* or power level *level*.
- Context** This function can be called from user or kernel context. This function can also be called from interrupt context, providing that it is not the first Power Management function called by the driver.

**Examples** A hypothetical driver might include this code to handle `pm_power_has_changed(9)`:

```
static int
xxusb_intr(struct buf *bp)
{
    ...

    /*
     * At this point the device has informed us that it has
     * changed power level on its own. Inform this to framework.
     * We need to take care of the case when framework has
     * already called power() entry point and changed power level
     * before we were able to inform framework of this change.
     * Handle this by comparing the informed power level with
     * the actual power level and only doing the call if they
     * are same. In addition, make sure that power() doesn't get
     * run in parallel with this code by holding the mutex.
     */
    ASSERT(mutex_owned(&xsp->lock));
    if (level_informed == *(xsp->level_reg_addr)) {
        if (pm_power_has_changed(xsp->dip, XXUSB_COMPONENT,
            level_informed) != DDI_SUCCESS) {
            mutex_exit(&xsp->lock);
            return(DDI_INTR_UNCLAIMED);
        }
    }

    ....
}

xxdisk_power(dev_info *dip, int comp, int level)
{
    mutex_enter( xsp->lock);

    ...

    ...
}
```

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [pm\(7D\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [power\(9E\)](#), [pm\\_busy\\_component\(9F\)](#), [pm\\_idle\\_component\(9F\)](#), [pm\\_raise\\_power\(9F\)](#), [pm\\_lower\\_power\(9F\)](#), [pm\(9P\)](#), [pm-components\(9P\)](#)

*Writing Device Drivers*

**Name** pm\_raise\_power, pm\_lower\_power – Raise or lower power of components

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>

```
int pm_raise_power(dev_info_t *dip, int component, int level);  
int pm_lower_power(dev_info_t *dip, int component, int level);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

#### Parameters

|                |                  |                                                                              |
|----------------|------------------|------------------------------------------------------------------------------|
| pm_raise_power | <i>dip</i>       | Pointer to the device's dev_info structure                                   |
|                | <i>component</i> | The number of the <i>component</i> for which a power level change is desired |
|                | <i>level</i>     | The power level to which the indicated <i>component</i> will be raised       |
| pm_lower_power | <i>dip</i>       | Pointer to the device's dev_info structure                                   |
|                | <i>component</i> | Number of the <i>component</i> for which a power level change is desired     |
|                | <i>level</i>     | Power level to which the indicated <i>component</i> will be lowered          |

**Description** The pm\_raise\_power(9F) function requests the Power Management framework to raise the power level of *component* of *dip* to at least *level*.

The state of the device should be examined before each physical access. The pm\_raise\_power(9F) function should be called to set a *component* to the required power level if the operation to be performed requires the *component* to be at a power level higher than its current power level.

When pm\_raise\_power(9F) returns with success, the *component* is guaranteed to be at least at the requested power level. All devices that depend on this will be at their full power level. Since the actual device power level may be higher than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from pm\_raise\_power(9F).

The pm\_raise\_power(9F) function may cause re-entry of the driver [power\(9E\)](#) to raise the power level. Deadlock may result if the driver locks are held across the call to pm\_raise\_power(9F).

The pm\_lower\_power(9F) function requests the Power Management framework to lower the power level of *component* of *dip* to at most *level*.

Normally, transitions to lower power levels are initiated by the Power Management framework based on *component* idleness. However, when detaching, the driver should also initiate reduced power levels by setting the power level of all device components to their

lowest levels. The `pm_lower_power(9F)` function is intended for this use only, and will return `DDI_FAILURE` if the driver is not detaching at the time of the call.

If automatic Power Management is disabled (see `dtpower(1M)`), `pm_lower_power(9F)` returns `DDI_SUCCESS` without changing the power level of the component. Otherwise, when `pm_lower_power(9F)` returns with success, the *component* is guaranteed to be at most at the requested power level. Since the actual device power level may be lower than requested by the driver, the driver should not make any assumption about the absolute power level on successful return from `pm_lower_power(9F)`.

The `pm_lower_power(9F)` function may cause re-entry of the driver `power(9E)` to lower the power level. Deadlock may result if the driver locks are held across the call to `pm_lower_power(9F)`.

**Note** – If these functions are called as a result of entry into the driver's `attach(9E)`, `detach(9E)` or `power(9E)` entry point, these functions must be called from the same thread which entered `attach(9E)`, `detach(9E)` or `power(9E)`.

**Return Values** The `pm_raise_power(9F)` function returns:

- `DDI_SUCCESS`     *Component* is now at the requested power level or higher.
- `DDI_FAILURE`    *Component* or *level* is out of range, or the framework was unable to raise the power level of the component to the requested level.

The `pm_lower_power(9F)` function returns:

- `DDI_SUCCESS`     *Component* is now at the requested power level or lower, or automatic Power Management is disabled.
- `DDI_FAILURE`    *Component* or *level* is out of range, or the framework was unable to lower the power level of the component to the requested level, or the device is not detaching.

**Examples** A hypothetical disk driver might include this code to handle `pm_raise_power(9F)`:

```
static int
xxdisk_strategy(struct buf *bp)
{
    ...

    /*
     * At this point we have determined that we need to raise the
     * power level of the device. Since we have to drop the
     * mutex, we need to take care of case where framework is
     * lowering power at the same time we are raising power.
     * We resolve this by marking the device busy and failing
     * lower power in power() entry point when device is busy.
```

```

    */

    ASSERT(mutex_owned(xsp->lock));
    if (xsp->pm_busycnt < 1) {
/*
    * Component is not already marked busy
    */
    if (pm_busy_component(xsp->dip,
        XXDISK_COMPONENT) != DDI_SUCCESS) {
        bioerror(bp,EIO);
        biodone(bp);
        return (0);
    }
    xsp->pm_busycnt++;
}
mutex_exit(xsp->lock);
if (pm_raise_power(xsp->dip,
    XXDISK_COMPONENT, XXPOWER_SPUN_UP) != DDI_SUCCESS) {
    bioerror(bp,EIO);
    biodone(bp);
    return (0);
}
    mutex_enter(xsp->lock);

    ....
}

xxdisk_power(dev_info *dip, int comp, int level)
{
    ...

    /*
    * We fail the power() entry point if the device is busy and
    * request is to lower the power level.
    */

    ASSERT(mutex_owned( xsp->lock));
    if (xsp->pm_busycnt >= 1) {
        if (level < xsp->cur_level) {
            mutex_exit( xsp->lock);
            return (DDI_FAILURE);
        }
    }
}

```

```

...
}

```

**Context** These functions can be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attribute:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [pm\(7D\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [power\(9E\)](#), [pm\\_busy\\_component\(9F\)](#), [pm\\_idle\\_component\(9F\)](#), [pm\(9P\)](#), [pm-components\(9P\)](#)

*Writing Device Drivers*

**Name** pm\_trans\_check – Device power cycle advisory check

**Synopsis** #include <sys/sunddi.h>

```
int pm_trans_check(struct pm_trans_data *datap, time_t *intervalp);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *datap* Pointer to a pm\_trans\_data structure

*intervalp* Pointer to time difference when next power cycle will be advised

**Description** The pm\_trans\_check() function checks if a power-cycle is currently advised based on data in the pm\_trans\_data structure. This function is provided to prevent damage to devices from excess power cycles; drivers for devices that are sensitive to the number of power cycles should call pm\_trans\_check() from their power(9E) function before powering-off a device. If pm\_trans\_check() indicates that the device should not be power cycled, the driver should not attempt to power cycle the device and should fail the call to power(9E) entry point.

If pm\_trans\_check() returns that it is not advised to power cycle the device, it attempts to calculate when the next power cycle is advised, based on the supplied parameters. In such case, intervalp returns the time difference (in seconds) from the current time to when the next power cycle is advised. If the time for the next power cycle cannot be determined, intervalp indicates 0.

To avoid excessive calls to the power(9E) entry point during a period when power cycling is not advised, the driver should mark the corresponding device component busy for the intervalp time period (if interval is not 0). Conveniently, the driver can utilize the fact that calls to pm\_busy\_component(9F) are stacked. If power cycling is not advised, the driver can call pm\_busy\_component(9F) and issue a timeout(9F) for the intervalp time. The timeout() handler can issue the corresponding pm\_idle\_component(9F) call.

The format field of pm\_trans\_data accepts either DC\_SCSI\_FORMAT or DC\_SMART\_FORMAT. If the caller provides information from a SCSI Log Page, it should set the format field to DC\_SCSI\_FORMAT, and provide valid data in svc\_date[], lifemax, ncycles and flag in the pm\_scsi\_cycles structure. If the caller provides information from a SMART feature attribute, it should set the format field to DC\_SMART\_FORMAT and provide valid data in allowed, usedup and flag in the pm\_smart\_count structure, where allowed indicates the normalized cycle count before reaching the borderline threshold cycle count, and usedup indicates the normalized consumed cycle count.

The flag field in both pm\_scsi\_cycles and pm\_smart\_count structures is reserved for future use and must be set to 0.

```
struct pm_trans_data {
    int format;           /* data format */
    union {
        struct pm_scsi_cycles scsi_cycles;
```



```

        struct pm_smart_count smart_count;
    } un;
};

```

- Return Values**
- 1 Power cycle is advised.
  - 0 Power cycle is not advised.
  - 1 Error due to invalid argument.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [power\(9E\)](#)

*Writing Device Drivers*

*Using Power Management*

**Name** pollwakeup – inform a process that an event has occurred

**Synopsis** `#include <sys/poll.h>`

```
void pollwakeup(struct pollhead *php, short event);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *php* Pointer to a pollhead structure.

*event* Event to notify the process about.

**Description** The `pollwakeup()` function wakes a process waiting on the occurrence of an event. It should be called from a driver for each occurrence of an event. The `pollhead` structure will usually be associated with the driver's private data structure associated with the particular minor device where the event has occurred. See [chpoll\(9E\)](#) and [poll\(2\)](#) for more detail.

**Context** The `pollwakeup()` function can be called from user, interrupt, or kernel context.

**See Also** [poll\(2\)](#), [chpoll\(9E\)](#)

*Writing Device Drivers*

**Notes** Driver defined locks should not be held across calls to this function.

**Name** priv\_getbyname – map a privilege name to a number

**Synopsis** #include <sys/cred.h>

```
int priv_getbyname(const char *priv, int flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *priv* name of the privilege

*flags* flags, must be zero or PRIV\_ALLOC

**Description** The priv\_getbyname() function maps a privilege name to a privilege number for use with the priv\_\*( ) kernel interfaces.

If PRIV\_ALLOC is passed as a flag parameter, an attempt is made to allocate a privilege if it is not yet defined. The newly allocated privilege number is returned.

Privilege names can be specified with an optional priv\_ prefix, which is stripped.

Privilege names are case insensitive but allocated privileges preserve case.

Allocated privileges can be at most {PRIVNAME\_MAX} characters long and can contain only alphanumeric characters and the underscore character.

**Return Values** This function returns the privilege number, which is greater than or equal to 0, if it succeeds. It returns a negative error number if an error occurs.

**Errors** EINVAL This might be caused by any of the following

- The *flags* parameter is invalid.
- The specified privilege does not exist.
- The *priv* parameter contains invalid characters.

ENOMEM There is no room to allocate another privilege.

ENAMETOOLONG An attempt was made to allocate a privilege that was longer than {PRIVNAME\_MAX} characters.

**Context** This functions can be called from user and kernel contexts.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Architecture        | All             |
| Interface Stability | Committed       |

**See Also** [attributes\(5\)](#), [privileges\(5\)](#)

*Writing Device Drivers*

**Name** priv\_policy, priv\_policy\_only, priv\_policy\_choice – check, report, and audit privileges

**Synopsis** #include <sys/policy.h>

```
int priv_policy(const cred_t *cr, int priv, boolean_t flag,
               int err, const char *msg);

boolean_t priv_policy_only(const cred_t *cr, int priv,
                          boolean_t flag);

boolean_t priv_policy_choice(const cred_t *cr, int priv,
                            boolean_t flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- cr*        The credential to be checked.
- priv*      The integer value of the privilege to test.
- flag*      All zone privileges flag. Set to B\_FALSE for most tests or B\_TRUE if the operation requires the caller to have all available privileges in the zone.
- err*        The error code to return.
- msg*        String that is added to the privilege debugging message if one is generated. NULL if no additional information is needed. Because the function name is included in the output, NULL is usually the best value to pass as a parameter.

**Description** These functions aid in privilege checking and privilege debugging.

The `priv_policy()`, `priv_policy_only()`, and `priv_policy_choice()` functions all check whether *priv* is asserted in the effective set of the credential. The special value `PRIV_ALL` tests for all privileges.

The `priv_policy()` function updates the ASU accounting flag and records the privilege used on success in the audit trail if the required privilege was not a basic privilege.

The `priv_policy_only()` function checks whether a privilege is asserted and has no side effects.

The `priv_policy_choice()` function behaves like `priv_policy_only()` but records the successfully used non-basic privileges in the audit trail.

**Return Values** On success, `priv_policy()` return 0. On failure it returns its parameter *err*.

On success, `priv_policy_choice()` and `priv_policy_only()` return 1, on failure both return 0.

**Errors** `EINVAL`            The *flags* parameter is invalid, the specified privilege does not exist, or the *priv* parameter contains invalid characters.

ENOMEM            There is no room to allocate another privilege.  
ENAMETOOLONG    An attempt was made to allocate a privilege that was longer than  
                  {PRIVNAME\_MAX} characters.

**Context**    This functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [acct\(3HEAD\)](#), [attributes\(5\)](#), [privileges\(5\)](#)

*Writing Device Drivers*

**Name** proc\_signal, proc\_ref, proc\_unref – send a signal to a process

**Synopsis** #include <sys/ddi.h>  
#include <sys/sunddi.h>  
#include <sys/signal.h>

```
void *proc_ref(void)

void proc_unref(void *pref);

int proc_signal(void *pref, int sig);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *pref* A handle for the process to be signalled.  
*sig* Signal number to be sent to the process.

**Description** This set of routines allows a driver to send a signal to a process. The routine `proc_ref()` is used to retrieve an unambiguous reference to the process for signalling purposes. The return value can be used as a unique handle on the process, even if the process dies. Because system resources are committed to a process reference, `proc_unref()` should be used to remove it as soon as it is no longer needed. `proc_signal()` is used to send signal *sig* to the referenced process. The following set of signals may be sent to a process from a driver:

|          |                                                 |
|----------|-------------------------------------------------|
| SIGHUP   | The device has been disconnected.               |
| SIGINT   | The interrupt character has been received.      |
| SIGQUIT  | The quit character has been received.           |
| SIGPOLL  | A pollable event has occurred.                  |
| SIGKILL  | Kill the process (cannot be caught or ignored). |
| SIGWINCH | Window size change.                             |
| SIGURG   | Urgent data are available.                      |

See [signal.h\(3HEAD\)](#) for more details on the meaning of these signals.

If the process has exited at the time the signal was sent, `proc_signal()` returns an error code; the caller should remove the reference on the process by calling `proc_unref()`.

The driver writer must ensure that for each call made to `proc_ref()`, there is exactly one corresponding call to `proc_unref()`.

**Return Values** The `proc_ref()` returns the following:

*pref* An opaque handle used to refer to the current process.

The `proc_signal()` returns the following:

- 0 The process existed before the signal was sent.
- 1 The process no longer exists; no signal was sent.

**Context** The `proc_unref()` and `proc_signal()` functions can be called from user, interrupt, or kernel context. The `proc_ref()` function should be called only from user context.

**See Also** [signal.h\(3HEAD\)](#), [putnextctl1\(9F\)](#)

*Writing Device Drivers*



- Name** ptob – convert size in pages to size in bytes
- Synopsis** `#include <sys/ddi.h>`
- ```
unsigned long ptob(unsigned long numpages);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *numpages* Size in number of pages to convert to size in bytes.
- Description** This function returns the number of bytes that are contained in the specified number of pages. For example, if the page size is 2048, then `ptob(2)` returns 4096. `ptob(0)` returns 0.
- Return Values** The return value is always the number of bytes in the specified number of pages. There are no invalid input values, and no checking will be performed for overflow in the case of a page count whose corresponding byte count cannot be represented by an unsigned long. Rather, the higher order bits will be ignored.
- Context** The `ptob()` function can be called from user, interrupt, or kernel context.
- See Also** [btop\(9F\)](#), [btopr\(9F\)](#), [ddi\\_ptob\(9F\)](#)
- Writing Device Drivers*

**Name** pullupmsg – concatenate bytes in a message

**Synopsis** #include <sys/stream.h>

```
int pullupmsg(mblk_t *mp, ssize_t len);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the message whose blocks are to be concatenated. *mblk\_t* is an instance of the [msgb\(9S\)](#) structure.

*len* Number of bytes to concatenate.

**Description** The `pullupmsg()` function tries to combine multiple data blocks into a single block. `pullupmsg()` concatenates and aligns the first *len* data bytes of the message pointed to by *mp*. If *len* equals -1, all data are concatenated. If *len* bytes of the same message type cannot be found, `pullupmsg()` fails and returns 0.

**Return Values** On success, 1 is returned; on failure, 0 is returned.

**Context** The `pullupmsg()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `pullupmsg()`

This is a driver write [srv\(9E\)](#) (service) routine for a device that does not support scatter/gather DMA. For all `M_DATA` messages, the data will be transferred to the device with DMA. First, try to pull up the message into one message block with the `pullupmsg()` function (line 12). If successful, the transfer can be accomplished in one DMA job. Otherwise, it must be done one message block at a time (lines 19–22). After the data has been transferred to the device, free the message and continue processing messages on the queue.

```
1 xxxwsrv(q)
2     queue_t *q;
3 {
4     mblk_t *mp;
5     mblk_t *tmp;
6     caddr_t dma_addr;
7     ssize_t dma_len;
8
9     while ((mp = getq(q)) != NULL) {
10        switch (mp->b_datap->db_type) {
11            case M_DATA:
12                if (pullupmsg(mp, -1)) {
13                    dma_addr = vtop(mp->b_rptr);
14                    dma_len = mp->b_wptr - mp->b_rptr;
15                    xxx_do_dma(dma_addr, dma_len);
16                    freemsg(mp);
17                    break;
```

**EXAMPLE 1** Using `pullupmsg()` (Continued)

```

18             }
19             for (tmp = mp; tmp; tmp = tmp->b_cont) {
20                 dma_addr = vtop(tmp->b_rptr);
21                 dma_len = tmp->b_wptr - tmp->b_rptr;
22                 xxx_do_dma(dma_addr, dma_len);
23             }
24             freemsg(mp);
25             break;
26         . . .
27     }
28 }

```

**See Also** [srv\(9E\)](#), [allocb\(9F\)](#), [msgpullup\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The `pullupmsg()` function is not included in the DKI and will be removed from the system in a future release. Device driver writers are strongly encouraged to use [msgpullup\(9F\)](#) instead of `pullupmsg()`.

**Name** put – call a STREAMS put procedure

**Synopsis** #include <sys/stream.h>  
#include <sys/ddi.h>

```
void put(queue_t *q, mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to a STREAMS queue.  
*mp* Pointer to message block being passed into queue.

**Description** put () calls the put procedure ([put\(9E\)](#) entry point) for the STREAMS queue specified by *q*, passing it the message block referred to by *mp*. It is typically used by a driver or module to call its own put procedure.

**Context** put () can be called from a STREAMS module or driver put or service routine, or from an associated interrupt handler, timeout, bufcall, or esballoc call-back. In the latter cases, the calling code must guarantee the validity of the *q* argument.

Since put () may cause re-entry of the module (as it is intended to do), mutexes or other locks should not be held across calls to it, due to the risk of single-party deadlock ([put\(9E\)](#), [putnext\(9F\)](#), [putctl\(9F\)](#), [qreply\(9F\)](#)). This function is provided as a DDI/DKI conforming replacement for a direct call to a put procedure.

**See Also** [put\(9E\)](#), [freezestr\(9F\)](#), [putctl\(9F\)](#), [putctl1\(9F\)](#), [putnext\(9F\)](#), [putnextctl\(9F\)](#), [putnextctl1\(9F\)](#), [qprocson\(9F\)](#), [qreply\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The caller cannot have the stream frozen when calling this function. See [freezestr\(9F\)](#).

DDI/DKI conforming modules and drivers are no longer permitted to call put procedures directly, but must call through the appropriate STREAMS utility function, for example, [put\(9E\)](#), [putnext\(9F\)](#), [putctl\(9F\)](#), and [qreply\(9F\)](#). This function is provided as a DDI/DKI conforming replacement for a direct call to a put procedure.

The put () and putnext () functions should be called only after qprocson () is finished.

- 
- Name** putbq – place a message at the head of a queue
- Synopsis** `#include <sys/stream.h>`
- ```
int putbq(queue_t *q, mblk_t *bp);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the queue.  
*bp* Pointer to the message block.
- Description** The `putbq()` function places a message at the beginning of the appropriate section of the message queue. There are always sections for high priority and ordinary messages. If other priority bands are used, each will have its own section of the queue, in priority band order, after high priority messages and before ordinary messages. `putbq()` can be used for ordinary, priority band, and high priority messages. However, unless precautions are taken, using `putbq()` with a high priority message is likely to lead to an infinite loop of putting the message back on the queue, being rescheduled, pulling it off, and putting it back on.
- This function is usually called when `bcanput(9F)` or `canput(9F)` determines that the message cannot be passed on to the next stream component. The flow control parameters are updated to reflect the change in the queue's status. If `QNOENB` is not set, the service routine is enabled.
- Return Values** The `putbq()` function returns 1 upon success and 0 upon failure.
- Upon failure, the caller should call `freemsg(9F)` to free the pointer to the message block.
- Context** The `putbq()` function can be called from user, interrupt, or kernel context.
- Examples** See the `bufcall(9F)` function page for an example of `putbq()`.
- See Also** `bcanput(9F)`, `bufcall(9F)`, `canput(9F)`, `getq(9F)`, `putq(9F)`
- Writing Device Drivers*
- STREAMS Programming Guide*

**Name** putctl – send a control message with a one-byte parameter to a queue

**Synopsis** #include <sys/stream.h>

```
int putctl(queue_t *q, int type, int p);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Queue to which the message is to be sent.

*type* Type of message.

*p* One-byte parameter.

**Description** The `putctl()` function, like [putctl\(9F\)](#), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an `M_DELAY` message is being sent. `putctl()` fails if *type* is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or if a message block cannot be allocated. If successful, `putctl()` calls the [put\(9E\)](#) routine of the queue pointed to by *q* with the newly allocated and initialized message.

**Return Values** On success, 1 is returned. 0 is returned if *type* is a data type, or if a message block cannot be allocated.

**Context** The `putctl()` function can be called from user, interrupt, or kernel context.

**Examples** See the [putctl\(9F\)](#) function page for an example of `putctl()`.

**See Also** [put\(9E\)](#), [allocb\(9F\)](#), [datamsg\(9F\)](#), [putctl\(9F\)](#), [putnextctl\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- Name** putctl – send a control message to a queue
- Synopsis** #include <sys/stream.h>
- ```
int putctl(queue_t *q, int type);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Queue to which the message is to be sent.  
*type* Message type (must be control, not data type).
- Description** The putctl() function tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. putctl() fails if *type* is M\_DATA, M\_PROTO, or M\_PCPROTO, or if a message block cannot be allocated. If successful, putctl() calls the put(9E) routine of the queue pointed to by *q* with the newly allocated and initialized messages.
- Return Values** On success, 1 is returned. If *type* is a data type, or if a message block cannot be allocated, 0 is returned.
- Context** The putctl() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using putctl()

The send\_ctl() routine is used to pass control messages downstream. M\_BREAK messages are handled with putctl() (line 11). putctl1(9F) (line 16) is used for M\_DELAY messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 12, 17). If an invalid message type is detected, cmn\_err(9F) panics the system (line 21).

```
1 void
2 send_ctl(wrq, type, parm)
3     queue_t *wrq;
4     uchar_t type;
5     uchar_t parm;
6 {
7     extern int num_alloc_fail;
8
9     switch (type) {
10        case M_BREAK:
11            if (!putctl(wrq->q_next, M_BREAK))
12                num_alloc_fail++;
13            break;
14
15        case M_DELAY:
16            if (!putctl1(wrq->q_next, M_DELAY, parm))
17                num_alloc_fail++;
18            break;
19
```

**EXAMPLE 1** Using putctl() (Continued)

```
20     default:
21         cmn_err(CE_PANIC, "send_ctl: bad message type passed");
22         break;
23     }
24 }
```

**See Also** [put\(9E\)](#), [cmn\\_err\(9F\)](#), [datamsg\(9F\)](#), [putctl1\(9F\)](#), [putnextctl\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*



**Name** putnext – send a message to the next queue

**Synopsis** #include <sys/stream.h>  
#include <sys/ddi.h>

```
void putnext(queue_t *q, mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue from which the message *mp* will be sent.  
*mp* Message to be passed.

**Description** The putnext() function is used to pass a message to the [put\(9E\)](#) routine of the next queue in the stream.

**Return Values** None.

**Context** The putnext() function can be called from user, interrupt, or kernel context.

**Examples** See [allocb\(9F\)](#) for an example of using putnext().

**See Also** [put\(9E\)](#), [allocb\(9F\)](#), [put\(9F\)](#), [qprocson\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The put() and putnext() functions should be called only after qprocson() is finished.

**Name** putnextctl1 – send a control message with a one-byte parameter to a queue

**Synopsis** #include <sys/stream.h>

```
int putnextctl1(queue_t *q, int type, int p);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Queue to which the message is to be sent.

*type* Type of message.

*p* One-byte parameter.

**Description** The `putnextctl1()` function, like [putctl1\(9F\)](#), tests the *type* argument to make sure a data type has not been specified, and attempts to allocate a message block. The *p* parameter can be used, for example, to specify how long the delay will be when an `M_DELAY` message is being sent. `putnextctl1()` fails if *type* is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or if a message block cannot be allocated. If successful, `putnextctl1()` calls the [put\(9E\)](#) routine of the queue pointed to by *q* with the newly allocated and initialized message.

A call to `putnextctl1(q,type,p)` is an atomic equivalent of `putctl1(q->q_next, type, p)`. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its `q_next` field and then invoking [putctl1\(9F\)](#) proceeds without interference from other threads.

The `putnextctl1()` function should always be used in preference to [putctl1\(9F\)](#)

**Return Values** On success, 1 is returned. 0 is returned if *type* is a data type, or if a message block cannot be allocated.

**Context** The `putnextctl1()` function can be called from user, interrupt, or kernel context.

**Examples** See the [putnextctl1\(9F\)](#) function page for an example of `putnextctl1()`.

**See Also** [put\(9E\)](#), [alloca\(9F\)](#), [datamsg\(9F\)](#), [putctl1\(9F\)](#), [putnextctl\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- Name** putnextctl – send a control message to a queue
- Synopsis** #include <sys/stream.h>
- ```
int putnextctl(queue_t *q, int type);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Queue to which the message is to be sent.  
*type* Message type (must be control, not data type).
- Description** The `putnextctl()` function tests the *type* argument to make sure a data type has not been specified, and then attempts to allocate a message block. `putnextctl()` fails if *type* is `M_DATA`, `M_PROTO`, or `M_PCPROTO`, or if a message block cannot be allocated. If successful, `putnextctl()` calls the [put\(9E\)](#) routine of the queue pointed to by *q* with the newly allocated and initialized messages.
- A call to `putnextctl(q, type)` is an atomic equivalent of `putctl(q->q_next, type)`. The STREAMS framework provides whatever mutual exclusion is necessary to insure that dereferencing *q* through its `q_next` field and then invoking [putctl\(9F\)](#) proceeds without interference from other threads.
- The `putnextctl()` function should always be used in preference to [putctl\(9F\)](#).
- Return Values** On success, 1 is returned. If *type* is a data type, or if a message block cannot be allocated, 0 is returned.
- Context** The `putnextctl()` function can be user, interrupt, or kernel context.
- Examples** The `send_ctl` routine is used to pass control messages downstream. `M_BREAK` messages are handled with `putnextctl()` (line 8). [putnextctl1\(9F\)](#) (line 13) is used for `M_DELAY` messages, so that *parm* can be used to specify the length of the delay. In either case, if a message block cannot be allocated a variable recording the number of allocation failures is incremented (lines 9, 14). If an invalid message type is detected, [cmn\\_err\(9F\)](#) panics the system (line 18).

```
1 void
2 send_ctl(queue_t *wrq, uchar_t type, uchar_t parm)
3 {
4     extern int num_alloc_fail;
5
6     switch (type) {
7     case M_BREAK:
8         if (!putnextctl(wrq, M_BREAK))
9             num_alloc_fail++;
10        break;
11
12    case M_DELAY:
13        if (!putnextctl1(wrq, M_DELAY, parm))
```

```
14                 num_alloc_fail++;
15             break;
16
17         default:
18             cmn_err(CE_PANIC, "send_ctl: bad message type passed");
19             break;
20     }
21 }
```

**See Also** [put\(9E\)](#), [cmn\\_err\(9F\)](#), [datamsg\(9F\)](#), [putctl\(9F\)](#), [putnextctl1\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** putq – put a message on a queue
- Synopsis** `#include <sys/stream.h>`
- ```
int putq(queue_t *q, mblk_t *bp);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the queue to which the message is to be added.  
*bp* Message to be put on the queue.
- Description** The `putq()` function is used to put messages on a driver's queue after the module's `put` routine has finished processing the message. The message is placed after any other messages of the same priority, and flow control parameters are updated. If `QNOENB` is not set, the service routine is enabled. If no other processing is done, `putq()` can be used as the module's `put` routine.
- Return Values** The `putq()` function returns 1 on success and 0 on failure.  
Upon failure, the caller should call [freemsg\(9F\)](#) to free the pointer to the message block.
- Context** The `putq()` function can be called from user, interrupt, or kernel context.
- Examples** See the [datamsg\(9F\)](#) function page for an example of `putq()`.
- See Also** [datamsg\(9F\)](#), [putbq\(9F\)](#), [qenable\(9F\)](#), [rmvq\(9F\)](#)  
*Writing Device Drivers*  
*STREAMS Programming Guide*

**Name** qassociate – associate STREAMS queue with driver instance

**Synopsis**

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

```
int qassociate(queue_t *q, int instance
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** `queue_t *q` Pointer to a [queue\(9S\)](#) structure. Either the read or write queue can be used.  
`int instance` Driver instance number or -1.

**Description** The `qassociate()` function must be used by DLPI style 2 device drivers to manage the association between STREAMS queues and device instances. The [gld\(7D\)](#) does this automatically on behalf of drivers based on it. It is recommended that the [gld\(7D\)](#) be used for network device drivers whenever possible.

The `qassociate()` function associates the specified STREAMS queue with the specified instance of the bottom driver in the queue. Upon successful return, the stream is associated with the instance with any prior association dissolved.

A DLPI style 2 driver calls `qassociate()` while processing the `DL_ATTACH_REQ` message. The driver is also expected to call this interface while performing stream associations through other means, such as [nnd\(1M\)](#) ioctl commands.

If `instance` is -1, the stream is left unassociated with any hardware instance.

If the interface returns failure, the stream is not associated with the specified instance. Any prior association is left untouched.

The interface typically fails because of failure to locate and attach the device instance. The interface never fails if the specified instance is -1.

**Context** The `qassociate()` function can be called from the stream's [put\(9E\)](#) entry point.

**Return Values** `0` Success.  
`-1` Failure.

**Examples** DLPI style 2 network driver `DL_ATTACH_REQ` code specifies:

```
if (qassociate(q, instance) != 0)
    goto fail;
```

The association prevents Dynamic Reconfiguration (DR) from detaching the instance.

DLPI style 2 network driver `DL_DETACH` code specifies:

```
(void) qassociate(q, -1);
```

This dissolves the queue's association with any device instance.

DLPI style 2 network driver `open(9E)` code must call:

```
qassociate(q, -1);
```

This informs the framework that this driver has been modified to be DDI-compliant.

**See Also** [dlpi\(7P\)](#), [gld\(7D\)](#), [open\(9E\)](#), [put\(9E\)](#), [ddi\\_no\\_info\(9F\)](#), [queue\(9S\)](#)

**Name** qbufcall – call a function when a buffer becomes available

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>
```

```
bufcall_id_t qbufcall(queue_t *q, size_t size, uint_t pri,
    void(*func)(void *arg), void *arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- q* Pointer to STREAMS queue structure.
- size* Number of bytes required for the buffer.
- pri* Priority of the [allocb\(9F\)](#) allocation request (not used).
- func* Function or driver routine to be called when a buffer becomes available.
- arg* Argument to the function to be called when a buffer becomes available.

**Description** The `qbufcall()` function serves as a [qtimeout\(9F\)](#) call of indeterminate length. When a buffer allocation request fails, `qbufcall()` can be used to schedule the routine *func* to be called with the argument *arg* when a buffer becomes available. *func* may call `allocb()` or it may do something else.

The `qbufcall()` function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See [mt-streams\(9F\)](#).) `qbufcall()` schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to `qbufcall()`. All outstanding timeouts and bufcalls must be cancelled (using, respectively, [qtimeout\(9F\)](#) and [qunbufcall\(9F\)](#)) before a driver close routine can block and before the close routine calls [qprocsoff\(9F\)](#).

[qprocson\(9F\)](#) must be called before calling either `qbufcall()` or [qtimeout\(9F\)](#).

**Return Values** If successful, the `qbufcall()` function returns a `qbufcall` ID that can be used in a call to [qunbufcall\(9F\)](#) to cancel the request. If the `qbufcall()` scheduling fails, *func* is never called and 0 is returned.

**Context** The `qbufcall()` function can be called from user, interrupt, or kernel context.

**See Also** [allocb\(9F\)](#), [mt-streams\(9F\)](#), [qprocson\(9F\)](#), [qtimeout\(9F\)](#), [qunbufcall\(9F\)](#), [qtimeout\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** Even when *func* is called by `qbufcall()`, [allocb\(9F\)](#) can fail if another module or driver had allocated the memory before *func* was able to call [allocb\(9F\)](#).



**Name** qenable – enable a queue

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>
```

```
void qenable(queue_t *q);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue to be enabled.

**Description** The `qenable()` function adds the queue pointed to by *q* to the list of queues whose service routines are ready to be called by the STREAMS scheduler.

**Context** The `qenable()` function can be called from user, interrupt, or kernel context.

**Examples** See the [dupb\(9F\)](#) function page for an example of the `qenable()`.

**See Also** [dupb\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** qprocson, qprocsoff – enable, disable put and service routines

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

void qprocson(queue_t *q);
void qprocsoff(queue_t *q);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the RD side of a streams queue pair.

**Description** The `qprocson()` enables the put and service routines of the driver or module whose read queue is pointed to by *q*. Threads cannot enter the module instance through the put and service routines while they are disabled.

The `qprocson()` function must be called by the open routine of a driver or module before returning, and after any initialization necessary for the proper functioning of the put and service routines.

The `qprocson()` function must be called before calling [put\(9F\)](#), [putnext\(9F\)](#), [qbufcall\(9F\)](#), [qtimeout\(9F\)](#), [qwait\(9F\)](#), or [qwait\\_sig\(9F\)](#).

The `qprocsoff()` function must be called by the close routine of a driver or module before returning, and before deallocating any resources necessary for the proper functioning of the put and service routines. It also removes the queue's service routines from the service queue, and blocks until any pending service processing completes.

The module or driver instance is guaranteed to be single-threaded before `qprocson()` is called and after `qprocsoff()` is called, except for threads executing asynchronous events such as interrupt handlers and callbacks, which must be handled separately.

**Context** These routines can be called from user, interrupt, or kernel context.

**See Also** [close\(9E\)](#), [open\(9E\)](#), [put\(9E\)](#), [srv\(9E\)](#), [put\(9F\)](#), [putnext\(9F\)](#), [qbufcall\(9F\)](#), [qtimeout\(9F\)](#), [qwait\(9F\)](#), [qwait\\_sig\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The caller may not have the stream frozen during either of these calls.

**Name** qreply – send a message on a stream in the reverse direction

**Synopsis** #include <sys/stream.h>

```
void qreply(queue_t *q, mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue.

*mp* Pointer to the message to be sent in the opposite direction.

**Description** The qreply() function sends messages in the reverse direction of normal flow. That is, qreply(*q*, *mp*) is equivalent to putnext(OTHERQ(*q*), *mp*).

**Context** The qreply() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Canonical Flushing Code for STREAMS Drivers.

This example depicts the canonical flushing code for STREAMS drivers. Assume that the driver has service procedures so that there may be messages on its queues. See [srv\(9E\)](#). Its write-side put procedure handles M\_FLUSH messages by first checking the FLUSHW bit in the first byte of the message, then the write queue is flushed (line 8) and the FLUSHW bit is turned off (line 9). See [put\(9E\)](#). If the FLUSHR bit is on, then the read queue is flushed (line 12) and the message is sent back up the read side of the stream with the qreply() function (line 13). If the FLUSHR bit is off, then the message is freed (line 15). See the example for [flushq\(9F\)](#) for the canonical flushing code for modules.

```
1 xxxwput(q, mp)
2   queue_t *q;
3   mblk_t *mp;
4   {
5       switch(mp->b_datap->db_type) {
6       case M_FLUSH:
7           if (*mp->b_rptr & FLUSHW) {
8               flushq(q, FLUSHALL);
9               *mp->b_rptr &= ~FLUSHW;
10          }
11          if (*mp->b_rptr & FLUSHR) {
12              flushq(RD(q), FLUSHALL);
13              qreply(q, mp);
14          } else {
15              freemsg(mp);
16          }
17          break;
18          . . .
19  }
```

**See Also** [put\(9E\)](#), [srv\(9E\)](#), [flushq\(9F\)](#), [OTHERQ\(9F\)](#), [putnext\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- Name** qsize – find the number of messages on a queue
- Synopsis** `#include <sys/stream.h>`
- ```
int qsize(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Queue to be evaluated.
- Description** The `qsize()` function evaluates the queue *q* and returns the number of messages it contains.
- Return Values** If there are no message on the queue, `qsize()` returns 0. Otherwise, it returns the integer representing the number of messages on the queue.
- Context** The `qsize()` function can be called from user, interrupt, or kernel context.
- See Also** *Writing Device Drivers*  
*STREAMS Programming Guide*

**Name** qtimeout – execute a function after a specified length of time

**Synopsis** #include <sys/stream.h>  
#include <sys/ddi.h>

```
timeout_id_t qtimeout(queue_t *q, void (*func)(void *),  
void *arg, clock_t ticks);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *q* Pointer to STREAMS queue structure.  
*func* Kernel function to invoke when the time increment expires.  
*arg* Argument to the function.  
*ticks* Number of clock ticks to wait before the function is called. Use [drv\\_usecstohz\(9F\)](#) to convert microseconds to clock ticks.

**Description** The `qtimeout()` function schedules the specified function *func* to be called after a specified time interval. *func* is called with *arg* as a parameter. Control is immediately returned to the caller. This is useful when an event is known to occur within a specific time frame, or when you want to wait for I/O processes when an interrupt is not available or might cause problems. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.

The `qtimeout()` function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See [mt-streams\(9F\)](#).) `qtimeout()` schedules the specified function to execute after entering the perimeters associated with the queue passed in as the first parameter to `qtimeout()`. All outstanding timeouts and bufcalls must be cancelled (using, respectively, [quntimeout\(9F\)](#) and [qunbufcall\(9F\)](#)) before a driver close routine can block and before the close routine calls [qprocsoff\(9F\)](#).

The [qprocson\(9F\)](#) function must be called before calling `qtimeout()`.

**Return Values** The `qtimeout()` function returns an opaque non-zero timeout identifier that can be passed to [quntimeout\(9F\)](#) to cancel the request. Note: No value is returned from the called function.

**Context** The `qtimeout()` function can be called from user, interrupt, or kernel context.

**See Also** [drv\\_usecstohz\(9F\)](#), [mt-streams\(9F\)](#), [qbufcall\(9F\)](#), [qprocson\(9F\)](#), [qunbufcall\(9F\)](#), [quntimeout\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** queue, SLIST\_HEAD, SLIST\_HEAD\_INITIALIZER, SLIST\_ENTRY, SLIST\_INIT, SLIST\_INSERT\_AFTER, SLIST\_INSERT\_HEAD, SLIST\_REMOVE\_HEAD, SLIST\_REMOVE, SLIST\_FOREACH, SLIST\_EMPTY, SLIST\_FIRST, SLIST\_NEXT, SIMPLEQ\_HEAD, SIMPLEQ\_HEAD\_INITIALIZER, SIMPLEQ\_ENTRY, SIMPLEQ\_INIT, SIMPLEQ\_INSERT\_HEAD, SIMPLEQ\_INSERT\_TAIL, SIMPLEQ\_INSERT\_AFTER, SIMPLEQ\_REMOVE\_HEAD, SIMPLEQ\_REMOVE, SIMPLEQ\_FOREACH, SIMPLEQ\_EMPTY, SIMPLEQ\_FIRST, SIMPLEQ\_NEXT, STAILQ\_HEAD, STAILQ\_HEAD\_INITIALIZER, STAILQ\_ENTRY, STAILQ\_INIT, STAILQ\_INSERT\_HEAD, STAILQ\_INSERT\_TAIL, STAILQ\_INSERT\_AFTER, STAILQ\_REMOVE\_HEAD, STAILQ\_REMOVE, STAILQ\_FOREACH, STAILQ\_EMPTY, STAILQ\_FIRST, STAILQ\_NEXT, STAILQ\_CONCAT, LIST\_HEAD, LIST\_HEAD\_INITIALIZER, LIST\_ENTRY, LIST\_INIT, LIST\_INSERT\_AFTER, LIST\_INSERT\_BEFORE, LIST\_INSERT\_HEAD, LIST\_REMOVE, LIST\_FOREACH, LIST\_EMPTY, LIST\_FIRST, LIST\_NEXT, TAILQ\_HEAD, TAILQ\_HEAD\_INITIALIZER, TAILQ\_ENTRY, TAILQ\_INIT, TAILQ\_INSERT\_HEAD, TAILQ\_INSERT\_TAIL, TAILQ\_INSERT\_AFTER, TAILQ\_INSERT\_BEFORE, TAILQ\_REMOVE, TAILQ\_FOREACH, TAILQ\_FOREACH\_REVERSE, TAILQ\_EMPTY, TAILQ\_FIRST, TAILQ\_NEXT, TAILQ\_LAST, TAILQ\_PREV, TAILQ\_CONCAT, CIRCLEQ\_HEAD, CIRCLEQ\_HEAD\_INITIALIZER, CIRCLEQ\_ENTRY, CIRCLEQ\_INIT, CIRCLEQ\_INSERT\_AFTER, CIRCLEQ\_INSERT\_BEFORE, CIRCLEQ\_INSERT\_HEAD, CIRCLEQ\_INSERT\_TAIL, CIRCLEQ\_REMOVE, CIRCLEQ\_FOREACH, CIRCLEQ\_FOREACH\_REVERSE, CIRCLEQ\_EMPTY, CIRCLEQ\_FIRST, CIRCLEQ\_LAST, CIRCLEQ\_NEXT, CIRCLEQ\_PREV, CIRCLEQ\_LOOP\_NEXT, CIRCLEQ\_LOOP\_PREV – implementations of singly-linked lists, simple queues, lists, tail queues, and circular queues

**Synopsis** #include <sys/queue.h>

```

SLIST_HEAD(HEADNAME, TYPE);
SLIST_HEAD_INITIALIZER(head);
SLIST_ENTRY(TYPE);
SLIST_INIT(SLIST_HEAD *head)
SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);
SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME)
SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);
SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);
SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);
int SLIST_EMPTY(SLIST_HEAD *head);
TYPE *SLIST_FIRST(SLIST_HEAD *head);
TYPE *SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);

```

```
SIMPLEQ_HEAD(HEADNAME, TYPE);
SIMPLEQ_HEAD_INITIALIZER(head);
SIMPLEQ_ENTRY(TYPE);
SIMPLEQ_INIT(SIMPLEQ_HEAD *head);
SIMPLEQ_INSERT_HEAD(SIMPLEQ_HEAD *head, TYPE *elm, SIMPLEQ_ENTRY NAME);
SIMPLEQ_INSERT_TAIL(SIMPLEQ_HEAD *head, TYPE *elm, SIMPLEQ_ENTRY NAME);
SIMPLEQ_INSERT_AFTER(SIMPLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
    SIMPLEQ_ENTRY NAME);
SIMPLEQ_REMOVE_HEAD(SIMPLEQ_HEAD *head, SIMPLEQ_ENTRY NAME);
SIMPLEQ_REMOVE(SIMPLEQ_HEAD *head, TYPE *elm, TYPE, SIMPLEQ_ENTRY NAME);
SIMPLEQ_FOREACH(TYPE *var, SIMPLEQ_HEAD *head, SIMPLEQ_ENTRY NAME);
int SIMPLEQ_EMPTY(SIMPLEQ_HEAD *head)
TYPE *SIMPLEQ_FIRST(SIMPLEQ_HEAD *head);
TYPE *SIMPLEQ_NEXT(TYPE *elm, SIMPLEQ_ENTRY NAME);
STAILQ_HEAD(HEADNAME, TYPE);
STAILQ_HEAD_INITIALIZER(head);
STAILQ_ENTRY(TYPE);
STAILQ_INIT(STAILQ_HEAD *head);
STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
    STAILQ_ENTRY NAME);
STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY NAME);
STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);
STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);
int STAILQ_EMPTY(STAILQ_HEAD *head);
TYPE *STAILQ_FIRST(STAILQ_HEAD *head);
TYPE *STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_CONCAT(STAILQ_HEAD *head1, STAILQ_HEAD *head2);
LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD_INITIALIZER(head);
LIST_ENTRY(TYPE);
```



```
LIST_INIT(LIST_HEAD *head);
LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME);
LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);
LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
int LIST_EMPTY(LIST_HEAD *head);
TYPE *LIST_FIRST(LIST_HEAD *head);
TYPE *LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);
TAILQ_HEAD(HEADNAME, TYPE);
TAILQ_HEAD_INITIALIZER(head);
TAILQ_ENTRY(TYPE);
TAILQ_INIT(TAILQ_HEAD *head);
TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)
TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
    TAILQ_ENTRY NAME);
TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);
TAILQ_FOREACH_REVERSE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME);
int TAILQ_EMPTY(TAILQ_HEAD *head);
TYPE *TAILQ_FIRST(TAILQ_HEAD *head);
TYPE *TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);
TYPE *TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);
TYPE *TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);
TAILQ_CONCAT(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TAILQ_ENTRY NAME);
CIRCLEQ_HEAD(HEADNAME, TYPE);
CIRCLEQ_HEAD_INITIALIZER(head);
CIRCLEQ_ENTRY(TYPE);
CIRCLEQ_INIT(CIRCLEQ_HEAD *head);
```

```
CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
    CIRCLEQ_ENTRY NAME);
CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm,
    CIRCLEQ_ENTRY NAME);
CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
CIRCLEQ_REMOVE(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
CIRCLEQ_FOREACH(TYPE *var, CIRCLEQ_HEAD *head, CIRCLEQ_ENTRY NAME);
CIRCLEQ_FOREACH_REVERSE(TYPE *var, CIRCLEQ_HEAD *head,
    CIRCLEQ_ENTRY NAME);
int CIRCLEQ_EMPTY(CIRCLEQ_HEAD *head);
TYPE *CIRCLEQ_FIRST(CIRCLEQ_HEAD *head);
TYPE *CIRCLEQ_LAST(CIRCLEQ_HEAD *head);
TYPE *CIRCLEQ_NEXT(TYPE *elm, CIRCLEQ_ENTRY NAME);
TYPE *CIRCLEQ_PREV(TYPE *elm, CIRCLEQ_ENTRY NAME);
TYPE *CIRCLEQ_LOOP_NEXT(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
TYPE *CIRCLEQ_LOOP_PREV(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME);
```

**Interface Level** Architecture independent level 11 (DDI/DKI).

**Description** These macros define and operate on five types of data structures: singly-linked lists, simple queues, lists, tail queues, and circular queues. All five structures support the following functionality:

1. Insertion of a new entry at the head of the list.
2. Insertion of a new entry before or after any element in the list.
3. Removal of any entry in the list.
4. Forward traversal through the list.

Singly-linked lists are the simplest of the five data structures and support only the above functionality. Singly-linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue.

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. Entries may not be added before any element in the list.
2. All list insertions and removals must specify the head of the list.
3. Each head entry requires two pointers rather than one.

---

Simple queues are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

All doubly linked types of data structures (lists, tail queues, and circle queues) additionally allow:

1. Insertion of a new entry before any element in the list.
2.  $O(1)$  removal of any entry in the list.

However:

1. Each element requires two pointers rather than one.
2. Code size and execution time of operations (except for removal) is about twice that of the singly-linked data structures

Linked lists are the simplest of the doubly linked data structures and support only the above functionality over singly-linked lists.

Tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be concatenated.

However:

1. All list insertions and removals, except insertion before another element, must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than lists.

Circular queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. The termination condition for traversal is more complex.
4. Code size is about 40% greater and operations run about 45% slower than lists.

In the macro definitions, *TYPE* is the name of a user defined structure, that must contain a field of type *LIST\_ENTRY*, *SIMPLEQ\_ENTRY*, *SLIST\_ENTRY*, *TAILQ\_ENTRY*, or *CIRCLEQ\_ENTRY*, named *NAME*. The argument *HEADNAME* is the name of a user defined structure that must

be declared using the macros `LIST_HEAD()`, `SIMPLEQ_HEAD()`, `SLIST_HEAD()`, `TAILQ_HEAD()`, or `CIRCLEQ_HEAD()`. See the examples below for further explanation of how these macros are used.

Summary of Operations The following table summarizes the supported macros for each type of data structure.

|                              | SLIST | LIST | SIMPLEQ | STAILQ | TAILQ | CIRCLEQ |
|------------------------------|-------|------|---------|--------|-------|---------|
| <code>EMPTY</code>           | +     | +    | +       | +      | +     | +       |
| <code>FIRST</code>           | +     | +    | +       | +      | +     | +       |
| <code>FOREACH</code>         | +     | +    | +       | +      | +     | +       |
| <code>FOREACH_REVERSE</code> | -     | -    | -       | -      | +     | +       |
| <code>INSERT_AFTER</code>    | +     | +    | +       | +      | +     | +       |
| <code>INSERT_BEFORE</code>   | -     | +    | -       | -      | +     | +       |
| <code>INSERT_HEAD</code>     | +     | +    | +       | +      | +     | +       |
| <code>INSERT_TAIL</code>     | -     | -    | +       | +      | +     | +       |
| <code>LAST</code>            | -     | -    | -       | -      | +     | +       |
| <code>LOOP_NEXT</code>       | -     | -    | -       | -      | -     | +       |
| <code>LOOP_PREV</code>       | -     | -    | -       | -      | -     | +       |
| <code>NEXT</code>            | +     | +    | +       | +      | +     | +       |
| <code>PREV</code>            | -     | -    | -       | -      | +     | +       |
| <code>REMOVE</code>          | +     | +    | +       | +      | +     | +       |
| <code>REMOVE_HEAD</code>     | +     | -    | +       | +      | -     | -       |
| <code>CONCAT</code>          | -     | -    | -       | +      | +     | -       |

**Singly-linked Lists** A singly-linked list is headed by a structure defined by the `SLIST_HEAD()` macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of  $O(n)$  removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An `SLIST_HEAD` structure is declared as follows:

```
SLIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

The names *head* and *headp* are user selectable.

The macro `SLIST_HEAD_INITIALIZER()` evaluates to an initializer for the list head

The macro `SLIST_EMPTY()` evaluates to true if there are no elements in the list.

The macro `SLIST_ENTRY()` declares a structure that connects the elements in the list.

The macro `SLIST_FIRST()` returns the first element in the list or `NULL` if the list is empty.

The macro `SLIST_FOREACH()` traverses the list referenced by `head` in the forward direction, assigning each element in turn to `var`.

The macro `SLIST_INIT()` initializes the list referenced by `head`.

The macro `SLIST_INSERT_HEAD()` inserts the new element `elm` at the head of the list.

The macro `SLIST_INSERT_AFTER()` inserts the new element `elm` after the element `listelm`.

The macro `SLIST_NEXT()` returns the next element in the list.

The macro `SLIST_REMOVE()` removes the element `elm` from the list.

The macro `SLIST_REMOVE_HEAD()` removes the first element from the head of the list. For optimum efficiency, elements being removed from the head of the list should explicitly use this macro instead of the generic `SLIST_REMOVE()` macro.

```

Singly-linked List Example
SLIST_HEAD(slisthead, entry) head =
    SLIST_HEAD_INITIALIZER(head);
struct slisthead *headp;          /* Singly-linked List head. */
struct entry {
    ...
    SLIST_ENTRY(entry) entries;   /* Singly-linked List. */
    ...
} *n1, *n2, *n3, *np;

SLIST_INIT(&head);                /* Initialize the list. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
SLIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
SLIST_INSERT_AFTER(n1, n2, entries);

SLIST_REMOVE(&head, n2, entry, entries); /* Deletion. */
free(n2);

n3 = SLIST_FIRST(&head);
SLIST_REMOVE_HEAD(&head, entries); /* Deletion from the head. */
free(n3);

/* Forward traversal. */
SLIST_FOREACH(np, &head, entries)
    np-> ...

while (!SLIST_EMPTY(&head)) { /* List Deletion. */
    n1 = SLIST_FIRST(&head);
    SLIST_REMOVE_HEAD(&head, entries);
}

```

```
        free(n1);  
    }
```

**Simple Queues** A simple queue is headed by a structure defined by the `SIMPLEQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the simple queue and the other to the last element in the simple queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of  $O(n)$  removal for arbitrary elements. New elements can be added to the queue after an existing element, at the head of the queue, or at the end of the queue. A `SIMPLEQ_HEAD` structure is declared as follows:

```
SIMPLEQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the simple queue. A pointer to the head of the simple queue can later be declared as:

```
struct HEADNAME *headp;
```

The names *head* and *headp* are user selectable.

The macro `SIMPLEQ_ENTRY()` declares a structure that connects the elements in the simple queue.

The macro `SIMPLEQ_HEAD_INITIALIZER()` provides a value which can be used to initialize a simple queue head at compile time, and is used at the point that the simple queue head variable is declared, like:

```
struct HEADNAME head = SIMPLEQ_HEAD_INITIALIZER(head);
```

The macro `SIMPLEQ_INIT()` initializes the simple queue referenced by *head*.

The macro `SIMPLEQ_INSERT_HEAD()` inserts the new element *elm* at the head of the simple queue.

The macro `SIMPLEQ_INSERT_TAIL()` inserts the new element *elm* at the end of the simple queue.

The macro `SIMPLEQ_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `SIMPLEQ_REMOVE()` removes *elm* from the simple queue.

The macro `SIMPLEQ_REMOVE_HEAD()` removes the first element from the head of the simple queue. For optimum efficiency, elements being removed from the head of the queue should explicitly use this macro instead of the generic `SIMPLQ_REMOVE()` macro.

The macro `SIMPLEQ_EMPTY()` return true if the simple queue head has no elements.

The macro `SIMPLEQ_FIRST()` returns the first element of the simple queue head.

The macro `SIMPLEQ_FOREACH()` traverses the tail queue referenced by `head` in the forward direction, assigning each element in turn to `var`.

The macro `SIMPLEQ_NEXT()` returns the element after the element `elm`.

The macros prefixed with “STAILQ\_” (`STAILQ_HEAD()`, `STAILQ_HEAD_INITIALIZER()`, `STAILQ_ENTRY()`, `STAILQ_INIT()`, `STAILQ_INSERT_HEAD()`, `STAILQ_INSERT_TAIL()`, `STAILQ_INSERT_AFTER()`, `STAILQ_REMOVE_HEAD()`, `STAILQ_REMOVE()`, `STAILQ_FOREACH()`, `STAILQ_EMPTY()`, `STAILQ_FIRST()`, and `STAILQ_NEXT()`) are functionally identical to these simple queue functions, and are provided for compatibility with FreeBSD.

```
Simple Queue Example SIMPLEQ_HEAD(simplehead, entry) head;
struct simplehead *headp;          /* Simple queue head. */
struct entry {
    ...
    SIMPLEQ_ENTRY(entry) entries;  /* Simple queue. */
    ...
} *n1, *n2, *np;

SIMPLEQ_INIT(&head);                /* Initialize the queue. */

n1 = malloc(sizeof(struct entry));  /* Insert at the head. */
SIMPLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry));  /* Insert at the tail. */
SIMPLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry));  /* Insert after. */
SIMPLEQ_INSERT_AFTER(&head, n1, n2, entries);

/* Forward traversal. */
SIMPLEQ_FOREACH(np, &head, entries)
    np-> ...

/* Delete. */
while (SIMPLEQ_FIRST(&head) != NULL)
    SIMPLEQ_REMOVE_HEAD(&head, entries);
if (SIMPLEQ_EMPTY(&head))          /* Test for emptiness. */
    printf("nothing to do\n");
```

**Lists** A list is headed by a structure defined by the `LIST_HEAD()` macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A `LIST_HEAD` structure is declared as follows:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

where `HEADNAME` is the name of the structure to be defined, and `TYPE` is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

The names *head* and *headp* are user selectable.

The macro `LIST_ENTRY()` declares a structure that connects the elements in the list.

The macro `LIST_HEAD_INITIALIZER()` provides a value which can be used to initialize a list head at compile time, and is used at the point that the list head variable is declared, like:

```
struct HEADNAME head = LIST_HEAD_INITIALIZER(head);
```

The macro `LIST_INIT()` initializes the list referenced by *head*.

The macro `LIST_INSERT_HEAD()` inserts the new element *elm* at the head of the list.

The macro `LIST_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `LIST_INSERT_BEFORE()` inserts the new element *elm* before the element *listelm*.

The macro `LIST_REMOVE()` removes the element *elm* from the list.

The macro `LIST_EMPTY()` returns `true` if the list head has no elements.

The macro `LIST_FIRST()` returns the first element of the list head.

The macro `LIST_FOREACH()` traverses the list referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `LIST_NEXT()` returns the element after the element *elm*.

```
ListExample LIST_HEAD(listhead, entry) head;
struct listhead *headp;          /* List head. */
struct entry {
    ...
    LIST_ENTRY(entry) entries;    /* List. */
    ...
} *n1, *n2, *np;

LIST_INIT(&head);                /* Initialize the list. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);

n2 = malloc(sizeof(struct entry)); /* Insert before. */
LIST_INSERT_BEFORE(n1, n2, entries);

/* Forward traversal. */
```



```

LIST_FOREACH(np, &head, entries)
    np-> ...
                                /* Delete. */
while (LIST_FIRST(&head) != NULL)
    LIST_REMOVE(LIST_FIRST(&head), entries);
if (LIST_EMPTY(&head))          /* Test for emptiness. */
    printf("nothing to do\n");

```

**Tail Queues** A tail queue is headed by a structure defined by the `TAILQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end the queue. A `TAILQ_HEAD` structure is declared as follows:

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

The names *head* and *headp* are user selectable.

The macro `TAILQ_ENTRY()` declares a structure that connects the elements in the tail queue.

The macro `TAILQ_HEAD_INITIALIZER()` provides a value which can be used to initialize a tail queue head at compile time, and is used at the point that the tail queue head variable is declared, like:

```
struct HEADNAME head = TAILQ_HEAD_INITIALIZER(head);
```

The macro `TAILQ_INIT()` initializes the tail queue referenced by *head*.

The macro `TAILQ_INSERT_HEAD()` inserts the new element *elm* at the head of the tail queue.

The macro `TAILQ_INSERT_TAIL()` inserts the new element *elm* at the end of the tail queue.

The macro `TAILQ_INSERT_AFTER()` inserts the new element *elm* after the element *listelm*.

The macro `TAILQ_INSERT_BEFORE()` inserts the new element *elm* before the element *listelm*.

The macro `TAILQ_REMOVE()` removes the element *elm* from the tail queue.

The macro `TAILQ_EMPTY()` return true if the tail queue head has no elements.

The macro `TAILQ_FIRST()` returns the first element of the tail queue head.

The macro `TAILQ_FOREACH()` traverses the tail queue referenced by *head* in the forward direction, assigning each element in turn to *var*.

The macro `TAILQ_FOREACH_REVERSE()` traverses the tail queue referenced by *head* in the reverse direction, assigning each element in turn to *var*.

The macro `TAILQ_NEXT()` returns the element after the element *elm*.

The macro `TAILQ_CONCAT()` concatenates the tail queue headed by *head2* onto the end of the one headed by *head1* removing all entries from the former.

```
Tail Queue Example  TAILQ_HEAD(tailhead, entry) head;
                    struct tailhead *headp;          /* Tail queue head. */
                    struct entry {
                        ...
                        TAILQ_ENTRY(entry) entries;    /* Tail queue. */
                        ...
                    } *n1, *n2, *np;

                    TAILQ_INIT(&head);                 /* Initialize the queue. */

                    n1 = malloc(sizeof(struct entry));  /* Insert at the head. */
                    TAILQ_INSERT_HEAD(&head, n1, entries);

                    n1 = malloc(sizeof(struct entry));  /* Insert at the tail. */
                    TAILQ_INSERT_TAIL(&head, n1, entries);

                    n2 = malloc(sizeof(struct entry));  /* Insert after. */
                    TAILQ_INSERT_AFTER(&head, n1, n2, entries);

                    n2 = malloc(sizeof(struct entry));  /* Insert before. */
                    TAILQ_INSERT_BEFORE(n1, n2, entries);

                    TAILQ_FOREACH(np, &head, entries)  /* Forward traversal. */
                        np-> ...

                    /* Reverse traversal. */
                    TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
                        np-> ...

                    /* Delete. */
                    while (TAILQ_FIRST(&head) != NULL)
                        TAILQ_REMOVE(&head, TAILQ_FIRST(&head), entries);
                    if (TAILQ_EMPTY(&head))            /* Test for emptiness. */
                        printf("nothing to do\n");
```

**Circular Queues** A circular queue is headed by a structure defined by the `CIRCLEQ_HEAD()` macro. This structure contains a pair of pointers, one to the first element in the circular queue and the other to the last element in the circular queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the queue. New elements can be added

to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A `CIRCLEQ_HEAD` structure is declared as follows:

```
CIRCLEQ_HEAD(HEADNAME, TYPE) head;
```

where `HEADNAME` is the name of the structure to be defined, and `TYPE` is the type of the elements to be linked into the circular queue. A pointer to the head of the circular queue can later be declared as:

```
struct HEADNAME *headp;
```

The names `head` and `headp` are user selectable.

The macro `CIRCLEQ_ENTRY()` declares a structure that connects the elements in the circular queue.

The macro `CIRCLEQ_HEAD_INITIALIZER()` provides a value which can be used to initialize a circular queue head at compile time, and is used at the point that the circular queue head variable is declared, like:

```
struct HEADNAME() head() = CIRCLEQ_HEAD_INITIALIZER(head());
```

The macro `CIRCLEQ_INIT()` initializes the circular queue referenced by `head`.

The macro `CIRCLEQ_INSERT_HEAD()` inserts the new element `elm` at the head of the circular queue.

The macro `CIRCLEQ_INSERT_TAIL()` inserts the new element `elm` at the end of the circular queue.

The macro `CIRCLEQ_INSERT_AFTER()` inserts the new element `elm` after the element `listelm`.

The macro `CIRCLEQ_INSERT_BEFORE()` inserts the new element `elm` before the element `listelm`.

The macro `CIRCLEQ_REMOVE()` removes the element `elm` from the circular queue.

The macro `CIRCLEQ_EMPTY()` return true if the circular queue head has no elements.

The macro `CIRCLEQ_FIRST()` returns the first element of the circular queue head.

The macro `CIRCLEQ_FOREACH()` traverses the circle queue referenced by `head` in the forward direction, assigning each element in turn to `var`. Each element is assigned exactly once.

The macro `CIRCLEQ_FOREACH_REVERSE()` traverses the circle queue referenced by `head` in the reverse direction, assigning each element in turn to `var`. Each element is assigned exactly once.

The macro `CIRCLEQ_LAST()` returns the last element of the circular queue head.

The macro `CIRCLEQ_NEXT()` returns the element after the element *elm*.

The macro `CIRCLEQ_PREV()` returns the element before the element *elm*.

The macro `CIRCLEQ_LOOP_NEXT()` returns the element after the element *elm*. If *elm* was the last element in the queue, the first element is returned.

The macro `CIRCLEQ_LOOP_PREV()` returns the element before the element *elm*. If *elm* was the first element in the queue, the last element is returned.

```
CircularQueue
Example
CIRCLEQ_HEAD(circlem, entry) head;
struct circlem *head;          /* Circular queue head. */
struct entry {
    ...
    CIRCLEQ_ENTRY(entry) entries; /* Circular queue. */
    ...
} *n1, *n2, *np;

CIRCLEQ_INIT(&head);          /* Initialize circular queue. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
CIRCLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry)); /* Insert at the tail. */
CIRCLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry)); /* Insert before. */
CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);
/* Forward traversal. */
CIRCLEQ_FOREACH(np, &head, entries)
    np-> ...
/* Reverse traversal. */
CIRCLEQ_FOREACH_REVERSE(np, &head, entries)
    np-> ...
/* Delete. */
while (CIRCLEQ_FIRST(&head) != (void *)&head)
    CIRCLEQ_REMOVE(&head, CIRCLEQ_FIRST(&head), entries);
if (CIRCLEQ_EMPTY(&head)) /* Test for emptiness. */
    printf("nothing to do\n");
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

---

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [queue\(3EXT\)attributes\(5\)](#)

**Notes** Some of these macros or functions perform no error checking, and invalid usage leads to undefined behavior. In the case of macros or functions that expect their arguments to be elements that are present in the list or queue, passing an element that is not present is invalid.

The queue functions first appeared in 4.4BSD. The SIMPLEQ functions first appeared in NetBSD 1.2. The SLIST and STAILQ functions first appeared in FreeBSD 2.1.5. The CIRCLEQ\_LOOP functions first appeared in NetBSD 4.0.

**Name** qunbufcall – cancel a pending qbufcall request

**Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>
```

```
void qunbufcall(queue_t *q, bufcall_id_t id);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *q* Pointer to STREAMS queue\_t structure.  
*id* Identifier returned from [qbufcall\(9F\)](#).

**Description** The `qunbufcall()` function cancels a pending `qbufcall()` request. The argument *id* is a non-zero identifier of the request to be cancelled. *id* is returned from the `qbufcall()` function used to issue the cancel request.

The `qunbufcall()` function is tailored to be used with the enhanced STREAMS framework interface which is based on the concept of perimeters. (See [mt-streams\(9F\)](#).) `qunbufcall()` returns when the bufcall has been cancelled or finished executing. The bufcall will be cancelled even if it is blocked at the perimeters associated with the queue. All outstanding timeouts and bufcalls must be cancelled before a driver close routine can block and before the close routine calls [qprocsoff\(9F\)](#).

**Context** The `qunbufcall()` function can be called from user, interrupt, or kernel context.

**See Also** [mt-streams\(9F\)](#), [qbufcall\(9F\)](#), [qtimeout\(9F\)](#), [quntimeout\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** quntimeout – cancel previous qtimeout function call
- Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

clock_t quntimeout(queue_t *q, timeout_id_t id);
```
- Interface Level** Solaris DDI specific (Solaris DDI).
- Parameters** *q* Pointer to a STREAMS queue structure.  
*id* Opaque timeout ID a previous [qtimeout\(9F\)](#) call.
- Description** The `quntimeout()` function cancels a pending [qtimeout\(9F\)](#) request. The `quntimeout()` function is tailored to be used with the enhanced STREAMS framework interface, which is based on the concept of perimeters. (See [mt-streams\(9F\)](#).) `quntimeout()` returns when the timeout has been cancelled or finished executing. The timeout will be cancelled even if it is blocked at the perimeters associated with the queue. `quntimeout()` should be executed for all outstanding timeouts before a driver or module close returns. All outstanding timeouts and bufcalls must be cancelled before a driver close routine can block and before the close routine calls [qprocsoff\(9F\)](#).
- Return Values** The `quntimeout()` function returns -1 if the `id` is not found. Otherwise, `quntimeout()` returns a 0 or positive value.
- Context** The `quntimeout()` function can be called from user, interrupt, or kernel context.
- See Also** [mt-streams\(9F\)](#), [qbufcall\(9F\)](#), [qtimeout\(9F\)](#), [qunbufcall\(9F\)](#)  
*Writing Device Drivers*  
*STREAMS Programming Guide*

**Name** qwait, qwait\_sig – STREAMS wait routines

**Synopsis** #include <sys/stream.h>  
#include <sys/ddi.h>

```
void qwait(queue_t *q);
int qwait_sig(queue_t *q);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *qp* Pointer to the queue that is being opened or closed.

**Description** qwait() and qwait\_sig() are used to wait for a message to arrive to the [put\(9E\)](#) or [srv\(9E\)](#) procedures. qwait() and qwait\_sig() can also be used to wait for [qbufcall\(9F\)](#) or [qtimeout\(9F\)](#) callback procedures to execute. These routines can be used in the [open\(9E\)](#) and [close\(9E\)](#) procedures in a STREAMS driver or module.

The thread that calls [close\(\)](#) does not necessarily have the ability to receive signals, particularly when called by [exit\(2\)](#). In this case, qwait\_sig() behaves exactly as qwait(). Driver writers may use [ddi\\_can\\_receive\\_sig\(9F\)](#) to determine when this is the case, and, if so, arrange some means to avoid blocking indefinitely (for example, by using [qtimeout\(9F\)](#)).

qwait() and qwait\_sig() atomically exit the inner and outer perimeters associated with the queue, and wait for a thread to leave the module's [put\(9E\)](#), [srv\(9E\)](#), or [qbufcall\(9F\)](#) / [qtimeout\(9F\)](#) callback procedures. Upon return they re-enter the inner and outer perimeters.

This can be viewed as there being an implicit wakeup when a thread leaves a [put\(9E\)](#) or [srv\(9E\)](#) procedure or after a [qtimeout\(9F\)](#) or [qbufcall\(9F\)](#) callback procedure has been run in the same perimeter.

[procson\(9F\)](#) must be called before calling qwait() or qwait\_sig().

qwait() is not interrupted by a signal, whereas qwait\_sig() is interrupted by a signal. qwait\_sig() normally returns non-zero, and returns zero when the waiting was interrupted by a signal.

qwait() and qwait\_sig() are similar to [cv\\_wait\(\)](#) and [cv\\_wait\\_sig\(\)](#) except that the mutex is replaced by the inner and outer perimeters and the signalling is implicit when a thread leaves the inner perimeter. See [condvar\(9F\)](#).

**Return Values** 0 For qwait\_sig(), indicates that the condition was not necessarily signaled, and the function returned because a signal was pending.

**Context** These functions can only be called from an [open\(9E\)](#) or [close\(9E\)](#) routine.



**Examples** EXAMPLE 1 Using qwait()

The open routine sends down a T\_INFO\_REQ message and waits for the T\_INFO\_ACK. The arrival of the T\_INFO\_ACK is recorded by resetting a flag in the unit structure (WAIT\_INFO\_ACK). The example assumes that the module is D\_MTQPAIR or D\_MTPERMOD.

```

xxopen(qp, . . .)
    queue_t *qp;
{
    struct xxdata *xx;
    /* Allocate xxdata structure */
    qprocson(qp);
    /* Format T_INFO_ACK in mp */
    putnext(qp, mp);
    xx->xx_flags |= WAIT_INFO_ACK;
    while (xx->xx_flags & WAIT_INFO_ACK)
        qwait(qp);
    return (0);
}
xxrput(qp, mp)
    queue_t *qp;
    mblk_t *mp;
{
    struct xxdata *xx = (struct xxdata *)q->q_ptr;

    ...

    case T_INFO_ACK:
        if (xx->xx_flags & WAIT_INFO_ACK) {
            /* Record information from info ack */
            xx->xx_flags &= ~WAIT_INFO_ACK;
            freemsg(mp);
            return;
        }

    ...
}

```

**See Also** [close\(9E\)](#), [open\(9E\)](#), [put\(9E\)](#), [srv\(9E\)](#), [condvar\(9F\)](#), [ddi\\_can\\_receive\\_sig\(9F\)](#), [mt-streams\(9F\)](#), [qbufcall\(9F\)](#), [qprocson\(9F\)](#), [qtimeout\(9F\)](#)

*STREAMS Programming Guide*

*Writing Device Drivers*

**Name** qwriter – asynchronous STREAMS perimeter upgrade

**Synopsis** `#include <sys/stream.h>`  
`#include <sys/ddi.h>`

```
void qwriter(queue_t *qp, mblk_t *mp, void (*func)(), int perimeter);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                  |                                                                                                             |
|------------------|-------------------------------------------------------------------------------------------------------------|
| <i>qp</i>        | Pointer to the queue.                                                                                       |
| <i>mp</i>        | Pointer to a message that will be passed in to the callback function.                                       |
| <i>func</i>      | A function that will be called when exclusive (writer) access has been acquired at the specified perimeter. |
| <i>perimeter</i> | Either PERIM_INNER or PERIM_OUTER.                                                                          |

**Description** `qwriter()` is used to upgrade the access at either the inner or the outer perimeter from shared to exclusive and call the specified callback function when the upgrade has succeeded. See [mt-streams\(9F\)](#). The callback function is called as:

```
(*func)(queue_t *qp, mblk_t *mp);
```

`qwriter()` will acquire exclusive access immediately if possible, in which case the specified callback function will be executed before `qwriter()` returns. If this is not possible, `qwriter()` will defer the upgrade until later and return before the callback function has been executed. Modules should not assume that the callback function has been executed when `qwriter()` returns. One way to avoid dependencies on the execution of the callback function is to immediately return after calling `qwriter()` and let the callback function finish the processing of the message.

When `qwriter()` defers calling the callback function, the STREAMS framework will prevent other messages from entering the inner perimeter associated with the queue until the upgrade has completed and the callback function has finished executing.

**Context** `qwriter()` can only be called from an [put\(9E\)](#) or [srv\(9E\)](#) routine, or from a `qwriter()`, [qtimeout\(9F\)](#), or [qbufcall\(9F\)](#) callback function.

**See Also** [put\(9E\)](#), [srv\(9E\)](#), [mt-streams\(9F\)](#), [qbufcall\(9F\)](#), [qtimeout\(9F\)](#)

*STREAMS Programming Guide*

*Writing Device Drivers*

- 
- Name** RD, rd – get pointer to the read queue
- Synopsis**

```
#include <sys/stream.h>
#include <sys/ddi.h>

queue_t *RD(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the write queue whose read queue is to be returned.
- Description** The RD() function accepts a write queue pointer as an argument and returns a pointer to the read queue of the same module.
- CAUTION: Make sure the argument to this function is a pointer to a write queue. RD() will not check for queue type, and a system panic could result if it is not the right type.
- Return Values** The pointer to the read queue.
- Context** The RD() function can be called from user, interrupt, or kernel context.
- Examples** EXAMPLE 1 Function page reference  
See the [qreply\(9F\)](#) function page for an example of RD().
- See Also** [qreply\(9F\)](#), [WR\(9F\)](#)  
*Writing Device Drivers*  
*STREAMS Programming Guide*

**Name** rmalloc – allocate space from a resource map

**Synopsis**

```
#include <sys/map.h>
#include <sys/ddi.h>
```

```
unsigned long rmalloc(struct map *mp, size_t size);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Resource map from where the resource is drawn.

*size* Number of units of the resource.

**Description** The `rmalloc()` function is used by a driver to allocate space from a previously defined and initialized resource map. The map itself is allocated by calling the function `rmapallocmap(9F)`. `rmalloc()` is one of five functions used for resource map management. The other functions include:

`rmapalloc_wait(9F)` Allocate space from a resource map, wait if necessary.

`rmapfree(9F)` Return previously allocated space to a map.

`rmapallocmap(9F)` Allocate a resource map and initialize it.

`rmapfreemap(9F)` Deallocate a resource map.

The `rmalloc()` function allocates space from a resource map in terms of arbitrary units. The system maintains the resource map by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. The normal return value is an unsigned long set to the value of the index where sufficient free space in the resource was found.

**Return Values** Under normal conditions, `rmalloc()` returns the base index of the allocated space. Otherwise, `rmalloc()` returns a 0 if all resource map entries are already allocated.

**Context** The `rmalloc()` function can be called from user, interrupt, or kernel context.

**Examples** **EXAMPLE 1** Illustrating the principles of map management

The following example is a simple memory map, but it illustrates the principles of map management. A driver allocates and initializes the map by calling both the `rmapallocmap(9F)` and `rmapfree(9F)` functions. `rmapallocmap(9F)` is called to establish the number of slots or entries in the map, and `rmapfree(9F)` to initialize the resource area the map is to manage. The following example is a fragment from a hypothetical `start` routine and illustrates the following procedures:

- Panics the system if the required amount of memory can not be allocated (lines 11–15).
- Uses `rmapallocmap(9F)` to configure the total number of entries in the map, and `rmapfree(9F)` to initialize the total resource area.

**EXAMPLE 1** Illustrating the principles of map management *(Continued)*

```

1  #define XX_MAPSIZE    12
2  #define XX_BUFSIZE   2560
3  static struct map *xx_mp;          /* Private buffer space map */
4  . . .
5  xxstart( )
6  /*
7   * Allocate private buffer.  If insufficient memory,
8   * display message and halt system.
9   */
10 {
11     register caddr_t bp;
12     . . .
13     if ((bp = kmem_alloc(XX_BUFSIZE, KM_NOSLEEP) == 0) {
14         cmn_err(CE_PANIC, "xxstart: kmem_alloc failed before %d buffer"
15                "allocation", XX_BUFSIZE);
16     }
17     /*
18     * Initialize the resource map with number
19     * of slots in map.
20     */
21     xx_mp = rmallocmap(XX_MAPSIZE);
22     . . .
23     /*
24     * Initialize space management map with total
25     * buffer area it is to manage.
26     */
27     . . .
28     rmfree(xx_mp, XX_BUFSIZE, bp);
29     . . .

```

**EXAMPLE 2** Allocating buffers

The `rmalloc()` function is then used by the driver's read or write routine to allocate buffers for specific data transfers. The `uiomove(9F)` function is used to move the data between user space and local driver memory. The device then moves data between itself and local driver memory through DMA.

The next example illustrates the following procedures:

- The size of the I/O request is calculated and stored in the *size* variable (line 10).
- Buffers are allocated through the `rmalloc()` function using the *size* value (line 15). If the allocation fails the system will panic.
- The `uiomove(9F)` function is used to move data to the allocated buffer (line 23).

## EXAMPLE 2 Allocating buffers (Continued)

- If the address passed to [uiomove\(9F\)](#) is invalid, [rmfree\(9F\)](#) is called to release the previously allocated buffer, and an EFAULT error is returned.

```

1  #define XX_BUFSIZE  2560
2  #define XX_MAXSIZE (XX_BUFSIZE / 4)
3
4  static struct map *xx_mp;          /* Private buffer space map */
5  ...
6  xxread(dev_t dev, uio_t *uiop, cred_t *credp)
7  {
8      register caddr_t addr;
9      register int     size;
10     size = min(COUNT, XX_MAXSIZE); /* Break large I/O */
11                                     /* request into small ones */
12     /*
13      * Get buffer.
14      */
15     if ((addr = (caddr_t)rmalloc(xx_mp, size)) == 0)
16         cmn_err(CE_PANIC, "read: rmalloc failed allocation of size %d",
17                 size);
18
19     /*
20      * Move data to buffer.  If invalid address is found,
21      * return buffer to map and return error code.
22      */
23     if (uiomove(addr, size, UIO_READ, uiop) == -1) {
24         rmfree(xx_mp, size, addr);
25         return(EFAULT);
26     }
27 }

```

**See Also** [kmem\\_alloc\(9F\)](#), [rmalloc\\_wait\(9F\)](#), [rmallocmap\(9F\)](#), [rmfree\(9F\)](#), [rmfreemap\(9F\)](#), [uiomove\(9F\)](#)

*Writing Device Drivers*

- 
- Name** rmallocmap, rmallocmap\_wait, rmfreemap – allocate and free resource maps
- Synopsis**
- ```
#include <sys/ddi.h>
#include <sys/sunddi.h>

struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
void rmfreemap(struct map *mp);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters**
- mapsize*      Number of entries for the map.
- mp*              A pointer to the map structure to be deallocated.
- Description** rmallocmap() dynamically allocates a resource map structure. The argument *mapsize* defines the total number of entries in the map. In particular, it is the total number of allocations that can be outstanding at any one time.
- rmallocmap() initializes the map but does not associate it with the actual resource. In order to associate the map with the actual resource, a call to [rmfree\(9F\)](#) is used to make the entirety of the actual resource available for allocation, starting from the first index into the resource. Typically, the call to rmallocmap() is followed by a call to [rmfree\(9F\)](#), passing the address of the map returned from rmallocmap(), the total size of the resource, and the first index into the actual resource.
- The resource map allocated by rmallocmap() can be used to describe an arbitrary resource in whatever allocation units are appropriate, such as blocks, pages, or data structures. This resource can then be managed by the system by subsequent calls to [rmalloc\(9F\)](#), [rmalloc\\_wait\(9F\)](#), and [rmfree\(9F\)](#).
- rmallocmap\_wait() is similar to rmallocmap(), with the exception that it will wait for space to become available if necessary.
- rmfreemap() deallocates a resource map structure previously allocated by rmallocmap() or rmallocmap\_wait(). The argument *mp* is a pointer to the map structure to be deallocated.
- Return Values** Upon successful completion, rmallocmap() and rmallocmap\_wait() return a pointer to the newly allocated map structure. Upon failure, rmallocmap() returns a NULL pointer.
- Context** rmallocmap() and rmfreemap() can be called from user, kernel, or interrupt context.
- rmallocmap\_wait() can only be called from user or kernel context.
- See Also** [rmalloc\(9F\)](#), [rmalloc\\_wait\(9F\)](#), [rmfree\(9F\)](#)
- Writing Device Drivers*

**Name** rmalloc\_wait – allocate space from a resource map, wait if necessary

**Synopsis**

```
#include <sys/map.h>
#include <sys/ddi.h>
```

```
unsigned long rmalloc_wait(struct map *mp, size_t size);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the resource map from which space is to be allocated.  
*size* Number of units of space to allocate.

**Description** The rmalloc\_wait() function requests an allocation of space from a resource map. rmalloc\_wait() is similar to the [rmalloc\(9F\)](#) function with the exception that it will wait for space to become available if necessary.

**Return Values** The rmalloc\_wait() function returns the base of the allocated space.

**Context** This function can be called from user, interrupt, or kernel context. However, in most cases rmalloc\_wait() should not be called from interrupt context.

**See Also** [rmalloc\(9F\)](#), [rmallocmap\(9F\)](#), [rmfree\(9F\)](#), [rmfreemap\(9F\)](#)

*Writing Device Drivers*



- 
- Name** rmfree – free space back into a resource map
- Synopsis** `#include <sys/map.h>`  
`#include <sys/ddi.h>`
- ```
void rmfree(struct map *mp, size_t size, ulong_t index);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *mp* Pointer to the map structure.  
*size* Number of units being freed.  
*index* Index of the first unit of the allocated resource.
- Description** The `rmfree()` function releases space back into a resource map. It is the opposite of [rmalloc\(9F\)](#), which allocates space that is controlled by a resource map structure.
- When releasing resources using `rmfree()` the size and index passed to `rmfree()` must exactly match the size and index values passed to and returned from a previous call to `rmalloc()`. Resources cannot be returned piecemeal.
- Drivers may define resource maps for resource allocation, in terms of arbitrary units, using the [rmallocmap\(9F\)](#) function. The system maintains the resource map structure by size and index, computed in units appropriate for the resource. For example, units may be byte addresses, pages of memory, or blocks. `rmfree()` frees up unallocated space for re-use.
- The `rmfree()` function can also be used to initialize a resource map, in which case the size and index should cover the entire resource area.
- Context** The `rmfree()` function can be called from user, interrupt, or kernel context.
- See Also** [rmalloc\(9F\)](#), [rmalloc\\_wait\(9F\)](#), [rmallocmap\(9F\)](#), [rmfreemap\(9F\)](#)
- Writing Device Drivers*

**Name** rmbv – remove a message block from a message

**Synopsis** #include <sys/stream.h>

```
mblk_t *rmvb(mblk_t *mp, mblk_t *bp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Message from which a block is to be removed. *mblk\_t* is an instance of the [msgb\(9S\)](#) structure.

*bp* Message block to be removed.

**Description** The `rmvb()` function removes a message block (*bp*) from a message (*mp*), and returns a pointer to the altered message. The message block is not freed, merely removed from the message. It is the module or driver's responsibility to free the message block.

**Return Values** If successful, a pointer to the message (minus the removed block) is returned. The pointer is NULL if *bp* was the only block of the message before `rmvb()` was called. If the designated message block (*bp*) does not exist, -1 is returned.

**Context** The `rmvb()` function can be called from user, interrupt, or kernel context.

**Examples** This routine removes all zero-length M\_DATA message blocks from the given message. For each message block in the message, save the next message block (line 10). If the current message block is of type M\_DATA and has no data in its buffer (line 11), then remove it from the message (line 12) and free it (line 13). In either case, continue with the next message block in the message (line 16).

```
1 void
2  xxclean(mp)
3      mblk_t *mp;
4  {
5      mblk_t *tmp;
6      mblk_t *nmp;
7
8      tmp = mp;
9      while (tmp) {
10         nmp = tmp->b_cont;
11         if ((tmp->b_datap->db_type == M_DATA) &&
12             (tmp->b_rptr == tmp->b_wptr)) {
13             (void) rmbv(mp, tmp);
14             freeb(tmp);
15         }
16         tmp = nmp;
17 }
```

**See Also** [freeb\(9F\)](#), [msgb\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** rmvq – remove a message from a queue

**Synopsis** #include <sys/stream.h>

```
void rmvq(queue_t *q, mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Queue containing the message to be removed.

*mp* Message to remove.

**Description** The `rmvq()` function removes a message from a queue. A message can be removed from anywhere on a queue. To prevent modules and drivers from having to deal with the internals of message linkage on a queue, either `rmvq()` or [getq\(9F\)](#) should be used to remove a message from a queue.

**Context** The `rmvq()` function can be called from user, interrupt, or kernel context.

**Examples** This code fragment illustrates how one may flush one type of message from a queue. In this case, only `M_PROTO T_DATA_IND` messages are flushed. For each message on the queue, if it is an `M_PROTO` message (line 8) of type `T_DATA_IND` (line 10), save a pointer to the next message (line 11), remove the `T_DATA_IND` message (line 12) and free it (line 13). Continue with the next message in the list (line 19).

```
1 mblk_t *mp, *nmp;
2 queue_t *q;
3 union T_primitives *tp;
4
5 /* Insert code here to protect queue and message block */
6 mp = q->q_first;
7 while (mp) {
8     if (mp->b_datap->db_type == M_PROTO) {
9         tp = (union T_primitives *)mp->b_rptr;
10        if (tp->type == T_DATA_IND) {
11            nmp = mp->b_next;
12            rmvq(q, mp);
13            freemsg(mp);
14            mp = nmp;
15        } else {
16            mp = mp->b_next;
17        }
18    } else {
19        mp = mp->b_next;
20    }
21 }
22 /* End of region that must be protected */
```

When using `rmvq()`, you must ensure that the queue and the message block is not modified by another thread at the same time. You can achieve this either by using STREAMS functions or by implementing your own locking.

**See Also** [freemsg\(9F\)](#), [getq\(9F\)](#), [insq\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Warnings** Make sure that the message `mp` is linked onto `q` to avoid a possible system panic.

**Name** rwlock, rw\_init, rw\_destroy, rw\_enter, rw\_exit, rw\_tryenter, rw\_downgrade, rw\_tryupgrade, rw\_read\_locked – readers/writer lock functions

**Synopsis** #include <sys/ksynch.h>

```
void rw_init(krwlock_t *rwlp, char *name, krw_type_t type, void *arg);
void rw_destroy(krwlock_t *rwlp);
void rw_enter(krwlock_t *rwlp, krw_t enter_type);
void rw_exit(krwlock_t *rwlp);
int rw_tryenter(krwlock_t *rwlp, krw_t enter_type);
void rw_downgrade(krwlock_t *rwlp);
int rw_tryupgrade(krwlock_t *rwlp);
int rw_read_locked(krwlock_t *rwlp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

|                   |                   |                                                                                                                                                                                                          |
|-------------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Parameters</b> | <i>rwlp</i>       | Pointer to a <code>krwlock_t</code> readers/writer lock.                                                                                                                                                 |
|                   | <i>name</i>       | Descriptive string. This is obsolete and should be NULL. (Non-null strings are legal, but they're a waste of kernel memory.)                                                                             |
|                   | <i>type</i>       | Type of readers/writer lock.                                                                                                                                                                             |
|                   | <i>arg</i>        | Type-specific argument for initialization function.                                                                                                                                                      |
|                   | <i>enter_type</i> | One of the values <code>RW_READER</code> or <code>RW_WRITER</code> , indicating whether the lock is to be acquired non-exclusively ( <code>RW_READER</code> ) or exclusively ( <code>RW_WRITER</code> ). |

**Description** A multiple-readers, single-writer lock is represented by the `krwlock_t` data type. This type of lock will allow many threads to have simultaneous read-only access to an object. Only one thread may have write access at any one time. An object which is searched more frequently than it is changed is a good candidate for a readers/writer lock.

Readers/writer locks are slightly more expensive than mutex locks, and the advantage of multiple read access may not occur if the lock will only be held for a short time.

The `rw_init()` function initializes a readers/writer lock. It is an error to initialize a lock more than once. The *type* argument should be set to `RW_DRIVER`. If the lock is used by the interrupt handler, the type-specific argument, *arg*, should be the interrupt priority returned from [ddi\\_intr\\_get\\_pri\(9F\)](#) or [ddi\\_intr\\_get\\_softint\\_pri\(9F\)](#). Note that *arg* should be the value of the interrupt priority cast by calling the `DDI_INTR_PRI` macro. If the lock is not used by any interrupt handler, the argument should be NULL.

The `rw_destroy()` function releases any resources that might have been allocated by `rw_init()`. It should be called before freeing the memory containing the lock. The lock must not be held by any thread when it is destroyed.

The `rw_enter()` function acquires the lock, and blocks if necessary. If `enter_type` is `RW_READER`, the caller blocks if there is a writer or a thread attempting to enter for writing. If `enter_type` is `RW_WRITER`, the caller blocks if any thread holds the lock.

NOTE: It is a programming error for any thread to acquire an `rwlock` it already holds, even as a reader. Doing so can deadlock the system: if thread R acquires the lock as a reader, then thread W tries to acquire the lock as a writer, W will set write-wanted and block. When R tries to get its second read hold on the lock, it will honor the write-wanted bit and block waiting for W; but W cannot run until R drops the lock. Thus threads R and W deadlock.

The `rw_exit()` function releases the lock and may wake up one or more threads waiting on the lock.

The `rw_tryenter()` function attempts to enter the lock, like `rw_enter()`, but never blocks. It returns a non-zero value if the lock was successfully entered, and zero otherwise.

A thread which holds the lock exclusively (entered with `RW_WRITER`), may call `rw_downgrade()` to convert to holding the lock non-exclusively (as if entered with `RW_READER`). One or more waiting readers may be unblocked.

The `rw_tryupgrade()` function can be called by a thread which holds the lock for reading to attempt to convert to holding it for writing. This upgrade can only succeed if no other thread is holding the lock and no other thread is blocked waiting to acquire the lock for writing.

The `rw_read_locked()` function returns non-zero if the calling thread holds the lock for read, and zero if the caller holds the lock for write. The caller must hold the lock. The system may panic if `rw_read_locked()` is called for a lock that isn't held by the caller.

|                      |          |                                                                                                                              |
|----------------------|----------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Return Values</b> | 0        | <code>rw_tryenter()</code> could not obtain the lock without blocking.                                                       |
|                      | 0        | <code>rw_tryupgrade()</code> was unable to perform the upgrade because of other threads holding or waiting to hold the lock. |
|                      | 0        | <code>rw_read_locked()</code> returns 0 if the lock is held by the caller for write.                                         |
|                      | non-zero | from <code>rw_read_locked()</code> if the lock is held by the caller for read.                                               |
|                      | non-zero | successful return from <code>rw_tryenter()</code> or <code>rw_tryupgrade()</code> .                                          |

**Context** These functions can be called from user, interrupt, or kernel context, except for `rw_init()` and `rw_destroy()`, which can be called from user context only.

**See Also** [condvar\(9F\)](#), [ddi\\_intr\\_alloc\(9F\)](#), [ddi\\_intr\\_add\\_handler\(9F\)](#), [ddi\\_intr\\_get\\_pri\(9F\)](#), [ddi\\_intr\\_get\\_softint\\_pri\(9F\)](#), [mutex\(9F\)](#), [semaphore\(9F\)](#)

*Writing Device Drivers*

**Notes** Compiling with `_LOCKTEST` or `_MPSTATS` defined no longer has any effect. To gather lock statistics, see [lockstat\(1M\)](#).



- 
- Name** SAMESTR, samestr – test if next queue is in the same stream
- Synopsis** `#include <sys/stream.h>`
- ```
int SAMESTR(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the queue.
- Description** The SAMESTR() function is used to see if the next queue in a stream (if it exists) is the same type as the current queue (that is, both are read queues or both are write queues). This function accounts for the twisted queue connections that occur in a STREAMS pipe and should be used in preference to direct examination of the `q_next` field of [queue\(9S\)](#) to see if the stream continues beyond *q*.
- Return Values** The SAMESTR() function returns 1 if the next queue is the same type as the current queue. It returns 0 if the next queue does not exist or if it is not the same type.
- Context** The SAMESTR() function can be called from user, interrupt, context.
- See Also** [OTHERQ\(9F\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*

**Name** `scsi_abort` – abort a SCSI command

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *ap* Pointer to a `scsi_address` structure.

*pkt* Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Description** The `scsi_abort()` function terminates a command that has been transported to the host adapter driver. A NULL *pkt* causes all outstanding packets to be aborted. On a successful abort, the `pkt_reason` is set to `CMD_ABORTED` and `pkt_statistics` is OR'ed with `STAT_ABORTED`.

**Return Values** The `scsi_abort()` function returns:

1 on success.

0 on failure.

**Context** The `scsi_abort()` function can be called from user, interrupt, or kernel context.

**Examples** **EXAMPLE 1** Terminating a command.

```
if (scsi_abort(&devp->sd_address, pkt) == 0) {
    (void) scsi_reset(&devp->sd_address, RESET_ALL);
}
```

**See Also** [tran\\_abort\(9E\)](#), [scsi\\_reset\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** `scsi_alloc_consistent_buf` – allocate an I/O buffer for SCSI DMA

**Synopsis** `#include <sys/scsi/scsi.h>`

```
struct buf *scsi_alloc_consistent_buf(structscsi_address*ap,
    struct buf *bp, size_t datalen, uint_t bflags,
    int (*callback)(caddr_t), caddr_t arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>ap</i>	Pointer to the <a href="#">scsi_address(9S)</a> structure.
<i>bp</i>	Pointer to the <a href="#">buf(9S)</a> structure.
<i>datalen</i>	Number of bytes for the data buffer.
<i>bflags</i>	Flags setting for the allocated buffer header. This should either be <code>B_READ</code> or <code>B_WRITE</code> .
<i>callback</i>	A pointer to a callback function, <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code> .
<i>arg</i>	The callback function argument.

**Description** The `scsi_alloc_consistent_buf()` function returns allocates a buffer header and the associated data buffer for direct memory access (DMA) transfer. This buffer is allocated from the `iobp` space, which is considered consistent memory. For more details, see [ddi\\_dma\\_mem\\_alloc\(9F\)](#) and [ddi\\_dma\\_sync\(9F\)](#).

For buffers allocated via `scsi_alloc_consistent_buf()`, and marked with the `PKT_CONSISTENT` flag via [scsi\\_init\\_pkt\(9F\)](#), the HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver's command completion callback is performed.

If *bp* is `NULL`, a new buffer header will be allocated using [getrbuf\(9F\)](#). In addition, if *datalen* is non-zero, a new buffer will be allocated using [ddi\\_dma\\_mem\\_alloc\(9F\)](#).

*callback* indicates what the allocator routines should do when direct memory access (DMA) resources are not available; the valid values are:

`NULL_FUNC` Do not wait for resources. Return a `NULL` pointer.

`SLEEP_FUNC` Wait indefinitely for resources.

**Other Values** *callback* points to a function that is called when resources may become available. *callback* must return either `0` (indicating that it attempted to allocate resources but failed to do so), in which case it is put back on a list to be called again later, or `1` indicating either success in allocating resources or indicating that it no longer cares for a retry. The last argument *arg* is supplied to the *callback* function when it is invoked.

**Return Values** The `scsi_alloc_consistent_buf()` function returns a pointer to a [buf\(9S\)](#) structure on success. It returns NULL if resources are not available even if *waitfunc* was not SLEEP\_FUNC.

**Context** If *callback* is SLEEP\_FUNC, then this routine may be called only from user-level code. Otherwise, it may be called from user, interrupt, or kernel context. The *callback* function may not block or call routines that block.

**Examples** **EXAMPLE 1** Allocate a request sense packet with consistent DMA resources attached.

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
    SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
rqpkt = scsi_init_pkt(&devp->sd_address,
    NULL, bp, CDB_GROUP0, 1, 0,
    PKT_CONSISTENT, SLEEP_FUNC, NULL);
```

**EXAMPLE 2** Allocate an inquiry packet with consistent DMA resources attached.

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
    SUN_INQSIZE, B_READ, canwait, NULL);
if (bp) {
    pkt = scsi_init_pkt(&devp->sd_address, NULL, bp,
        CDB_GROUP0, 1, PP_LEN, PKT_CONSISTENT,
        canwait, NULL);
}
```

**See Also** [ddi\\_dma\\_mem\\_alloc\(9F\)](#), [ddi\\_dma\\_sync\(9F\)](#), [getrbuf\(9F\)](#), [scsi\\_destroy\\_pkt\(9F\)](#), [scsi\\_init\\_pkt\(9F\)](#), [scsi\\_free\\_consistent\\_buf\(9F\)](#), [buf\(9S\)](#), [scsi\\_address\(9S\)](#)

*Writing Device Drivers*

**Name** scsi\_cname, scsi\_dname, scsi\_mname, scsi\_rname, scsi\_sname – decode a SCSI name

**Synopsis** #include <sys/scsi/scsi.h>

```
char *scsi_cname(uchar_t cmd, char **cmdvec);
char *scsi_dname(int dtype);
char *scsi_mname(uchar_t msg);
char *scsi_rname(uchar_t reason);
char *scsi_sname(uchar_t sense_key);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

<i>cmd</i>	A SCSI command value.
<i>cmdvec</i>	Pointer to an array of command strings.
<i>dtype</i>	Device type.
<i>msg</i>	A message value.
<i>reason</i>	A packet reason value.
<i>sense_key</i>	A SCSI sense key value.

**Description** The `scsi_cname()` function decodes SCSI commands. *cmdvec* is a pointer to an array of strings. The first byte of the string is the command value, and the remainder is the name of the command.

The `scsi_dname()` function decodes the peripheral device type (for example, direct access or sequential access) in the inquiry data.

The `scsi_mname()` function decodes SCSI messages.

The `scsi_rname()` function decodes packet completion reasons.

The `scsi_sname()` function decodes SCSI sense keys.

**Return Values** These functions return a pointer to a string. If an argument is invalid, they return a string to that effect.

**Context** These functions can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Decoding SCSI tape commands.

The `scsi_cname()` function decodes SCSI tape commands as follows:

```
static char *st_cmds[] = {
    "\000test unit ready",
    "\001rewind",
```

EXAMPLE 1 Decoding SCSI tape commands.      (Continued)

```
        "\003request sense",
        "\010read",
        "\012write",
        "\020write file mark",
        "\021space",
        "\022inquiry",
        "\025mode select",
        "\031erase tape",
        "\032mode sense",
        "\033load tape",
        NULL
    };
    ..
    cmn_err(CE_CONT, "st: cmd=%s", scsi_cname(cmd, st_cmds));
```

**See Also** [Writing Device Drivers](#)

**Name** `scsi_destroy_pkt` – free an allocated SCSI packet and its DMA resource

**Synopsis** `#include <sys/scsi/scsi.h>`

```
void scsi_destroy_pkt(struct scsi_pkt *pktp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** `pktp` Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Description** The `scsi_destroy_pkt()` function releases all necessary resources, typically at the end of an I/O transfer. The data is synchronized to memory, then the DMA resources are deallocated and `pktp` is freed.

**Context** The `scsi_destroy_pkt()` function may be called from user, interrupt, or kernel context.

**Examples** **EXAMPLE 1** Releasing resources  

```
scsi_destroy_pkt(un->un_rqs);
```

**See Also** [tran\\_destroy\\_pkt\(9E\)](#), [scsi\\_init\\_pkt\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** scsi\_dmaget, scsi\_dmafree – SCSI dma utility routines

**Synopsis** #include <sys/scsi/scsi.h>

```
struct scsi_pkt *scsi_dmaget(struct scsi_pkt *pkt,
    opaque_t dmatoken, int(*callback)(void));

void scsi_dmafree(struct scsi_pkt *pkt);
```

**Interface Level** These interfaces are obsolete. Use [scsi\\_init\\_pkt\(9F\)](#) instead of `scsi_dmaget()`. Use [scsi\\_destroy\\_pkt\(9F\)](#) instead of `scsi_dmafree()`.

**Parameters**

<i>pkt</i>	A pointer to a <a href="#">scsi_pkt(9S)</a> structure.
<i>dmatoken</i>	Pointer to an implementation dependent object.
<i>callback</i>	Pointer to a callback function, or <code>NULL_FUNC</code> or <code>SLEEP_FUNC</code> .

**Description** The `scsi_dmaget()` function allocates DMA resources for an already allocated SCSI packet. *pkt* is a pointer to the previously allocated SCSI packet (see [scsi\\_pktalloc\(9F\)](#)).

The *dmatoken* parameter is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The *dmatoken* must be a pointer to a [buf\(9S\)](#) structure. If *dmatoken* is `NULL`, no resources are allocated.

The *callback* parameter indicates what `scsi_dmaget()` should do when resources are not available:

`NULL_FUNC` Do not wait for resources. Return a `NULL` pointer.

`SLEEP_FUNC` Wait indefinitely for resources.

**Other Values** *callback* points to a function which is called when resources may have become available. *callback* must return either `0` (indicating that it attempted to allocate resources but failed to do so again), in which case it is put back on a list to be called again later, or `1` indicating either success in allocating resources or indicating that it no longer cares for a retry.

The `scsi_dmafree()` function frees the DMA resources associated with the SCSI packet. The packet itself remains allocated.

**Return Values** The `scsi_dmaget()` function returns a pointer to a `scsi_pkt` on success. It returns `NULL` if resources are not available.

**Context** If *callback* is `SLEEP_FUNC`, then this routine may only be called from user or kernel context. Otherwise, it may be called from user, kernel, or interrupt context. The *callback* function may not block or call routines that block.

The `scsi_dmafree()` function can be called from user, interrupt, or kernel context.



**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Stability Level	Obsolete

**See Also** [attributes\(5\)](#), [scsi\\_pktalloc\(9F\)](#), [scsi\\_pktfree\(9F\)](#), [scsi\\_resalloc\(9F\)](#), [scsi\\_resfree\(9F\)](#), [buf\(9S\)](#), [scsi\\_pkt\(9S\)](#)

### *Writing Device Drivers*

**Notes** The `scsi_dmaget()` and `scsi_dmafree()` functions are obsolete and will be discontinued in a future release. These functions have been replaced by, respectively, [scsi\\_init\\_pkt\(9F\)](#) and [scsi\\_destroy\\_pkt\(9F\)](#).

**Name** scsi\_errmsg – display a SCSI request sense message

**Synopsis** #include <sys/scsi/scsi.h>

```
void scsi_errmsg(struct scsi_device *devp, struct scsi_pkt *pktp,
                char *drv_name, int severity, daddr_t blkno, daddr_t err_blkno,
                struct scsi_key_strings *cmdlist, struct scsi_extended_sense *sensep);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- devp* Pointer to the [scsi\\_device\(9S\)](#) structure.
- pktp* Pointer to a [scsi\\_pkt\(9S\)](#) structure.
- drv\_name* String used by [scsi\\_log\(9F\)](#).
- severity* Error severity level, maps to severity strings below.
- blkno* Requested block number.
- err\_blkno* Error block number.
- cmdlist* An array of SCSI command description strings.
- sensep* A pointer to a [scsi\\_extended\\_sense\(9S\)](#) structure.

**Description** The `scsi_errmsg()` function interprets the request sense information in the *sensep* pointer and generates a standard message that is displayed using [scsi\\_log\(9F\)](#). The first line of the message is always a CE\_WARN, with the continuation lines being CE\_CONT. *sensep* may be NULL, in which case no sense key or vendor information is displayed.

The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The [scsi\\_device\(9S\)](#) structure denoted by *devp* supplies the identification of the device that requested the display. *severity* selects which string is used in the “Error Level:” reporting, according to the following table:

Severity Value:	String:
SCSI_ERR_ALL	All
SCSI_ERR_UNKNOWN	Unknown
SCSI_ERR_INFO	Informational
SCSI_ERR_RECOVERE	Recovered
SCSI_ERR_RETRYABL	Retryable
SCSI_ERR_FATAL	Fatal

*blkno* is the block number of the original request that generated the error. *err\_blkno* is the block number where the error occurred. *cmdlist* is a mapping table for translating the SCSI command code in *pkt* to the actual command string.

The *cmdlist* is described in the structure below:

```
struct scsi_key_strings {
    int key;
    char *message;
};
```

For a basic SCSI disk, the following list is appropriate:

```
static struct scsi_key_strings scsi_cmds[] = {
    0x00, "test unit ready",
    0x01, "rezero/rewind",
    0x03, "request sense",
    0x04, "format",
    0x07, "reassign",
    0x08, "read",
    0x0a, "write",
    0x0b, "seek",
    0x12, "inquiry",
    0x15, "mode select",
    0x16, "reserve",
    0x17, "release",
    0x18, "copy",
    0x1a, "mode sense",
    0x1b, "start/stop",
    0x1e, "door lock",
    0x28, "read(10)",
    0x2a, "write(10)",
    0x2f, "verify",
    0x37, "read defect data",
    0x3b, "write buffer",
    -1, NULL
};
```

**Context** The `scsi_errmsg()` function may be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Generating error information.

This entry:

```
scsi_errmsg(devp, pkt, "sd", SCSI_ERR_INFO, bp->b_blkno,
            err_blkno, sd_cmds, rqsense);
```

Generates:

```
WARNING: /sbus@1,f8000000/esp@0,800000/sd@1,0 (sd1):
Error for Command: read Error Level: Informational
```

**EXAMPLE 1** Generating error information.      *(Continued)*

Requested Block: 23936 Error Block: 23936  
Vendor: QUANTUM Serial Number: 123456  
Sense Key: Unit Attention  
ASC: 0x29 (reset), ASCQ: 0x0, FRU: 0x0

**See Also** [cmn\\_err\(9F\)](#), [scsi\\_log\(9F\)](#), [scsi\\_device\(9S\)](#), [scsi\\_extended\\_sense\(9S\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** `scsi_ext_sense_fields`, `scsi_sense_info_uint64`, `scsi_sense_cmdspecific_uint64` – retrieve fields from SCSI sense data

**Synopsis** `#include <sys/scsi/scsi.h>`

```
void scsi_ext_sense_fields(uint8_t *sense_buffer, int *sense_buf_len,
    uint8_t **information, uint8_t **cmd_spec_info, uint8_t **fru_code,
    uint8_t **sk_specific, uint8_t **stream_flags);

boolean_t scsi_sense_info_uint64(uint8_t *sense_buffer, int sense_buf_len,
    uint64_t *information);

boolean_t scsi_sense_cmdspecific_uint64(uint8_t *sense_buffer,
    int sense_buf_len, uint64_t *cmd_spec_info);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

<b>Parameters</b>	<i>sense_buffer</i>	Pointer to a buffer containing SCSI sense data. The sense data is expected in wire format starting at the response code. It can be in either a fixed or descriptor format.
	<i>information</i>	For <code>scsi_ext_sense_fields()</code> , this is a call-by-reference pointer to <code>uint8_t</code> . A pointer to the INFORMATION field in the sense data may be returned in this parameter.  The <code>scsi_sense_info_uint64()</code> function requires a pointer to <code>uint64_t</code> . The data in the information field is returned as a 64 bit integer. If the sense data information field is 32 bits, fixed format, the most significant 32-bits are 0.
	<i>cmd_spec_info</i>	For <code>scsi_ext_sense_fields()</code> , this is a call-by-reference pointer to <code>uint8_t</code> . A pointer to the COMMAND_SPECIFIC INFORMATION field in the sense data can be returned in this parameter.  The <code>scsi_sense_cmdspecific_uint64()</code> function requires a pointer to <code>uint64_t</code> . The data in the command specific information field is returned as a 64 bit integer. If the sense data command specific information field is 32 bits, fixed format, the most significant 32-bits are 0.
	<i>fru_code</i>	Call-by-reference pointer to <code>uint8_t</code> . A pointer to the FIELD REPLACEABLE UNIT CODE field in the sense data can be returned in this parameter.
	<i>sk_specific</i>	Call-by-reference pointer to <code>uint8_t</code> . A pointer to the SENSE KEY SPECIFIC field in the sense data can be returned in this parameter.
	<i>stream_flags</i>	Call-by-reference pointer to <code>uint8_t</code> . A pointer to the byte containing the ILI, EOM, and FILEMARK flags can be returned in this parameter.

**Description** The `scsi_ext_sense_fields()` function can be used to retrieve any of the extended sense data fields from a sense data buffer, regardless of whether the sense data is in fixed format or in descriptor format.

The *information*, *cmd\_spec\_info*, *fru\_code*, *sk\_specific*, and *stream\_specific* parameters are all call-by-reference output parameters. Each parameter corresponds to one or more of the extended sense data fields. Any of these parameters can be passed as NULL if the caller is not interested in the associated data.

If the requested data is present and valid for each output parameter, the pointer passed in is set to point to the relevant location in the sense buffer. If the data is not present or invalid, the pointer is set to NULL. For example, a caller that requests a pointer to the information field would get NULL when an information descriptor is not present for descriptor format sense data or when the valid bit is not set for fixed format sense data.

The information and command specific information fields can be 4 bytes or 8 bytes in length, depending on whether the sense data is in fixed or descriptor format respectively. Drivers can use [scsi\\_validate\\_sense\(9F\)](#) to determine the sense data format and, by extension, the length of the information and command specific information fields.

A driver can determine whether integer data is included in the information or command specific information fields based on the `asc` and `ascq` sense keys, such as the LBA of a failed disk request. The `scsi_sense_info_uint64()` function retrieves the contents of the information field as a 64 bit integer and the `scsi_sense_cmdspecific_uint64()` retrieves the command specific information field as a 64 bit integer.

Drivers should use [scsi\\_validate\\_sense\(9F\)](#) to ensure that the sense buffer contains valid sense data.

**Return Values** The `scsi_sense_info_uint64()` function returns TRUE if the information field is present and valid. Otherwise it returns FALSE.

The `scsi_sense_cmdspecific_uint64()` function returns TRUE if the command specific information field is present and valid. Otherwise it returns FALSE.

**Context** The `scsi_ext_sense_fields()`, `scsi_sense_info_uint64()` and `scsi_sense_cmdspecific_uint64()` functions can be called from user or interrupt context.

**See Also** [scsi\\_find\\_sense\\_descr\(9F\)](#), [scsi\\_sense\\_asc\(9F\)](#), [scsi\\_sense\\_ascq\(9F\)](#), [scsi\\_sense\\_key\(9F\)](#), [scsi\\_validate\\_sense\(9F\)](#)

**Name** `scsi_find_sense_descr` – find descriptor in SCSI sense data

**Synopsis** `#include <sys/scsi/scsi.h>`

```
uint8_t *scsi_find_sense_descr(uint8_t *sense_buffer, int sense_buf_len,
                               int req_descr_type);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *sense\_buffer* Pointer to a buffer containing SCSI descriptor sense data. The data is expected in wire format starting at the response code.

*sense\_buf\_len* Integer that contains the length of sense buffer in bytes.

*req\_descr\_type* Integer that contains the descriptor type value for the desired sense descriptor.

**Description** The `scsi_find_sense_descr()` function is used to obtain a pointer to a specific descriptor type, specified by *req\_descr\_type*, within a descriptor sense buffer. Before returning the pointer, `scsi_find_sense_descr()` verifies that the entire descriptor is present based on the length provided in *sense\_buf\_len*.

Any value for *req\_descr\_type* can be requested. The following descriptor types are already defined:

```
#define DESCR_INFORMATION      0x00
#define DESCR_COMMAND_SPECIFIC 0x01
#define DESCR_SENSE_KEY_SPECIFIC 0x02
#define DESCR_FRU              0x03
#define DESCR_STREAM_COMMANDS  0x04
#define DESCR_BLOCK_COMMANDS   0x05
#define DESCR_OSD_OID          0x06
#define DESCR_OSD_RESP_INTEGRITY 0x07
#define DESCR_OSD_ATTR_ID      0x08
```

Drivers should use [scsi\\_validate\\_sense\(9F\)](#) to ensure that the sense buffer contains valid descriptor sense data.

**Return Values** The `scsi_find_sense_descr()` function returns a pointer to a sense descriptor of the requested type if a descriptor of that type exists. If no such descriptor exists, `scsi_find_sense_descr()` returns NULL.

**Context** The `scsi_find_sense_descr()` function can be called from user or interrupt context.

**See Also** [scsi\\_ext\\_sense\\_fields\(9F\)](#), [scsi\\_sense\\_asc\(9F\)](#), [scsi\\_sense\\_ascq\(9F\)](#), [scsi\\_sense\\_cmdspecific\\_uint64\(9F\)](#), [scsi\\_sense\\_info\\_uint64\(9F\)](#), [scsi\\_sense\\_key\(9F\)](#), [scsi\\_validate\\_sense\(9F\)](#)

**Name** `scsi_free_consistent_buf` – free a previously allocated SCSI DMA I/O buffer

**Synopsis** `#include <sys/scsi/scsi.h>`

```
void scsi_free_consistent_buf(struct buf *bp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *bp* Pointer to the `buf(9S)` structure.

**Description** The `scsi_free_consistent_buf()` function frees a buffer header and consistent data buffer that was previously allocated using `scsi_alloc_consistent_buf(9F)`.

**Context** The `scsi_free_consistent_buf()` function can be called from user, interrupt, or kernel context.

**See Also** `freerbuf(9F)`, `scsi_alloc_consistent_buf(9F)`, `buf(9S)`

*Writing Device Drivers*

**Warning** The `scsi_free_consistent_buf()` function will call `freerbuf(9F)` to free the `buf(9S)` that was allocated before or during the call to `scsi_alloc_consistent_buf(9F)`.

If consistent memory is bound to a `scsi_pkt(9S)`, the `pkt` should be destroyed before freeing the consistent memory.



**Name** `scsi_get_device_type_scsi_options` – look up per-device-type scsi-options property

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_get_device_type_scsi_options(dev_info_t *dip,
    struct scsi_device *devp, int default_scsi_options);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *dip* Pointer to the device info node for this HBA driver.

*devp* Pointer to a `scsi_device(9S)` structure of the target.

*default\_scsi\_options* Value returned if no match is found.

**Description** The `scsi_get_device_type_scsi_options()` function looks up the property `device-type-scsi-options-list`, which can be specified in the HBA's `driver.conf(4)` file. This property allows specification of `scsi-options` on a per-device-type basis.

The formal syntax is:

```
device-type-scsi-options-list = <duplet> [, <duplet> *];
```

where:

```
<duplet> := <vid+pid>, <scsi-options-property-name>
```

and:

```
<scsi-options-property-name> = <value>;
```

The string `<vid+pid>` is returned by the device on a SCSI inquiry command. This string can contain any character in the range 0x20-0x7e. Characters such as double quote (") or single quote ('), which are not permitted in property value strings, are represented by their octal equivalent (for example, \042 and \047). Trailing spaces can be truncated.

For example:

```
device-type-scsi-options-list=
    "SEAGATE ST32550W", "seagate-options",
    "EXABYTE EXB-2501", "exabyte-options",
    "IBM OEM DFHSS4S", "ibm-options";
```

```
seagate-options = 0x78;
```

```
exabyte-options = 0x58;
```

```
ibm-options = 0x378;
```

The `scsi_get_device_type_scsi_options()` function searches the list of duplets for a matching INQUIRY string. If a match is found, `scsi_get_device_type_scsi_options()` returns the corresponding value.

**Return Values** `scsi_get_device_type_scsi_options()` returns the `scsi-options` value found, or if no match is found the `default_scsi_options` value passed in.

**Context** This function can be called from kernel or interrupt context.

**See Also** *Writing Device Drivers*

**Name** `scsi_get_device_type_string` – look up per-device-type, returning matching information string

**Synopsis** `#include <sys/scsi/scsi.h>`

```
char *scsi_get_device_type_string(char *prop_name,
    dev_info_t *dip, struct scsi_device *devp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** `prop_name` Name of string array property that contains a list of paired device-type information.

`dip` Pointer to the device info node for this HBA driver.

`devp` Pointer to a `scsi_device(9S)` structure of the target.

**Description** The `scsi_get_device_type_string()` function looks up the `prop_name` property, which can be specified in the HBA's `driver.conf(4)` file. This property allows specification of per-device-type information. The syntax for `prop_name` in a `driver.conf(4)` file is:

The formal syntax is:

```
prop_name = <duplet> [, <duplet> *];
```

where:

```
<duplet> := "<vid+pid>", "<information>"
```

The string `<vid+pid>` is returned by the device on a SCSI inquiry command. This string can contain any character in the range 0x20-0x7e. Characters such as double quote (") or single quote ('), which are not permitted in property value strings, are represented by their octal equivalent (for example, \042 and \047). Trailing spaces can be truncated.

For example:

```
scsi_vhci_failover_override =
    "STK    FLEXLINE 400",    "f_asym_lsi",
    "SUN    T4",            "f_tpgs";;
```

The `scsi_get_device_type_string()` function searches the list of duplets for a matching INQUIRY string. If a match is found, `scsi_get_device_type_string` returns a copy of the `<information>` string. The meaning of the returned `<information>` string is determined by the caller.

**Return Values** `scsi_get_device_type_string()` returns a copy of the matching `<information>` string, or if no match is found, NULL is returned. When a non-NULL value is returned, the caller is responsible for freeing the `<information>` string via:

```
kmem_free(info, strlen(info) + 1);
```

**Context** This function can be called from kernel or interrupt context.

**See Also** [driver.conf\(4\)](#)

*Writing Device Drivers*

**Name** scsi\_hba\_attach\_setup, scsi\_hba\_attach, scsi\_hba\_detach – SCSI HBA attach and detach routines

**Synopsis** #include <sys/scsi/scsi.h>

```
int scsi_hba_attach_setup(dev_info_t *dip, ddi_dma_attr_t *hba_dma_attr,
    scsi_hba_tran_t *hba_tran, int hba_flags);

int scsi_hba_attach(dev_info_t *dip, ddi_dma_lim_t *hba_lim,
    scsi_hba_tran_t *hba_tran, int hba_flags, void *hba_options);

int scsi_hba_detach(dev_info_t *dip);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

<b>Parameters</b>	<i>dip</i>	Pointer to the <code>dev_info_t</code> structure that refers to the instance of the HBA device.
	<i>hba_lim</i>	Pointer to a <code>ddi_dma_lim(9S)</code> structure.
	<i>hba_tran</i>	Pointer to a <code>scsi_hba_tran(9S)</code> structure.
	<i>hba_flags</i>	Flag modifiers. The defined flag values are <code>SCSI_HBA_TRAN_CLONE</code> , <code>SCSI_HBA_TRAN_SCB</code> , and <code>SCSI_HBA_TRAN_CDB</code> .
	<i>hba_options</i>	Optional features provided by the HBA driver for future extensions; must be NULL.
	<i>hba_dma_attr</i>	Pointer to a <code>ddi_dma_attr(9S)</code> structure.

**Description** The `scsi_hba_attach_setup()` function is the recommended interface over the `scsi_hba_attach()` function.

`scsi_hba_attach_setup()` The `scsi_hba_attach()` function registers the *hba\_lim* DMA limits and the *hba\_tran* transport vectors of each instance of the HBA device defined by *dip*. The `scsi_hba_attach_setup()` function registers the *hba\_dma\_attr* DMA attributes and the *hba\_tran* transport vectors of each instance of the HBA device defined by *dip*. The HBA driver can pass different DMA limits or DMA attributes and the transport vectors for each instance of the device to support any constraints imposed by the HBA itself.

The `scsi_hba_attach()` and `scsi_hba_attach_setup()` functions use the `dev_bus_ops` field in the `dev_ops(9S)` structure. The HBA driver should initialize this field to NULL before calling `scsi_hba_attach()` or `scsi_hba_attach_setup()`.

If `SCSI_HBA_TRAN_CLONE` is requested in *hba\_flags*, the `hba_tran` structure is cloned once for each target that is attached to the HBA. The structure is cloned before the `tran_tgt_init(9E)` entry point is called to initialize a target. At all subsequent HBA entry points, including `tran_tgt_init(9E)`, the `scsi_hba_tran_t` structure passed as an argument or found in a `scsi_address` structure is the cloned `scsi_hba_tran_t` structure, which allows the HBA to use the `tran_tgt_private` field in the `scsi_hba_tran_t` structure to point to per-target data.

The HBA should free only the same `scsi_hba_tran_t` structure allocated when the HBA detaches. All cloned `scsi_hba_tran_t` structures that are allocated by the system are freed by the system.

The flags `SCSI_HBA_TRAN_CDB` and `SCSI_HBA_TRAN_SCB` are only valid when `tran_setup_pkt()` is used. See [tran\\_setup\\_pkt\(9E\)](#) for information on using these flags.

The `scsi_hba_attach()` and `scsi_hba_attach_setup()` functions attach a number of integer-valued properties to *dip*, unless properties of the same name are already attached to the node. An HBA driver should retrieve these configuration parameters via [ddi\\_prop\\_get\\_int\(9F\)](#), and respect any settings for features provided the HBA.

#### scsi-options

#### Optional SCSI configuration bits

<code>SCSI_OPTIONS_DR</code>	If not set, the HBA should not grant Disconnect privileges to target devices.
<code>SCSI_OPTIONS_TAG</code>	If not set, the HBA should not operate in Command Tagged Queueing mode.
<code>SCSI_OPTIONS_PARITY</code>	If not set, the HBA should not operate in parity mode.
<code>SCSI_OPTIONS_QAS</code>	If not set, the HBA should not make use of the Quick Arbitration Select feature. Consult your Sun hardware documentation to determine whether your machine supports QAS.
<code>SCSI_OPTIONS_FAST</code>	If not set, the HBA should not operate the bus in FAST SCSI mode.
<code>SCSI_OPTIONS_FAST20</code>	If not set, the HBA should not operate the bus in FAST20 SCSI mode.
<code>SCSI_OPTIONS_FAST40</code>	If not set, the HBA should not operate the bus in FAST40 SCSI mode.
<code>SCSI_OPTIONS_FAST80</code>	If not set, the HBA should not operate the bus in FAST80 SCSI mode.
<code>SCSI_OPTIONS_FAST160</code>	If not set, the HBA should not operate the bus in FAST160 SCSI mode.
<code>SCSI_OPTIONS_FAST320</code>	If not set, the HBA should not operate the bus in FAST320 SCSI mode.
<code>SCSI_OPTIONS_WIDE</code>	If not set, the HBA should not operate the bus in WIDE SCSI mode.
<code>SCSI_OPTIONS_SYNC</code>	If not set, the HBA should not operate the bus in synchronous transfer mode.

---

<code>scsi-reset-delay</code>	SCSI bus or device reset recovery time, in milliseconds.
<code>scsi-selection-timeout</code>	Default SCSI selection phase timeout value, in milliseconds. Please refer to individual HBA man pages for any HBA-specific information

`scsi_hba_detach()` The `scsi_hba_detach()` function removes the reference to the DMA limits or attributes structure and the transport vector for the given instance of an HBA driver.

**Return Values** The `scsi_hba_attach()`, `scsi_hba_attach_setup()`, and `scsi_hba_detach()` functions return `DDI_SUCCESS` if the function call succeeds, and return `DDI_FAILURE` on failure.

**Context** The `scsi_hba_attach()` and `scsi_hba_attach_setup()` functions should be called from [attach\(9E\)](#). The `scsi_hba_detach()` function should be called from [detach\(9E\)](#).

**See Also** [attach\(9E\)](#), [detach\(9E\)](#), [tran\\_setup\\_pkt\(9E\)](#), [tran\\_tgt\\_init\(9E\)](#), [ddi\\_prop\\_get\\_int\(9F\)](#), [ddi\\_dma\\_attr\(9S\)](#), [ddi\\_dma\\_lim\(9S\)](#), [dev\\_ops\(9S\)](#), [scsi\\_address\(9S\)](#), [scsi\\_hba\\_tran\(9S\)](#)

#### *Writing Device Drivers*

**Notes** It is the HBA driver's responsibility to ensure that no more transport requests will be taken on behalf of any SCSI target device driver after `scsi_hba_detach()` is called.

The `scsi_hba_attach()` function is obsolete and will be discontinued in a future release. This function is replaced by `scsi_hba_attach_setup()`.

**Name** `scsi_hba_init`, `scsi_hba_fini` – SCSI Host Bus Adapter system initialization and completion routines

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_hba_init(struct modlinkage *modlp);  
void scsi_hba_fini(struct modlinkage *modlp);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters** `modlp` Pointer to the Host Bus Adapters module linkage structure.

### Description

`scsi_hba_init()` `scsi_hba_init()` is the system-provided initialization routine for SCSI HBA drivers. The `scsi_hba_init()` function registers the HBA in the system and allows the driver to accept configuration requests on behalf of SCSI target drivers. The `scsi_hba_init()` routine must be called in the HBA's `_init(9E)` routine before `mod_install(9F)` is called. If `mod_install(9F)` fails, the HBA's `_init(9E)` should call `scsi_hba_fini()` before returning failure.

`scsi_hba_fini()` `scsi_hba_fini()` is the system provided completion routine for SCSI HBA drivers. `scsi_hba_fini()` removes all of the system references for the HBA that were created in `scsi_hba_init()`. The `scsi_hba_fini()` routine should be called in the HBA's `_fini(9E)` routine if `mod_remove(9F)` is successful.

**Return Values** `scsi_hba_init()` returns 0 if successful, and a non-zero value otherwise. If `scsi_hba_init()` fails, the HBA's `_init()` entry point should return the value returned by `scsi_hba_init()`.

**Context** `scsi_hba_init()` and `scsi_hba_fini()` should be called from `_init(9E)` or `_fini(9E)`, respectively.

**See Also** `_fini(9E)`, `_init(9E)`, `mod_install(9F)`, `mod_remove(9F)`, `scsi_pktalloc(9F)`, `scsi_pktfree(9F)`, `scsi_hba_tran(9S)`

### *Writing Device Drivers*

**Notes** The HBA is responsible for ensuring that no DDI request routines are called on behalf of its SCSI target drivers once `scsi_hba_fini()` is called.



**Name** `scsi_hba_lookup_capstr` – return index matching capability string

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_hba_lookup_capstr(char *capstr);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters** `capstr` Pointer to a string

**Description** The `scsi_hba_lookup_capstr()` function attempts to match `capstr` against a known set of capability strings. If found, the defined index for the matched capability is returned.

The following indices are defined for the capability strings listed below.

<code>SCSI_CAP_DMA_MAX</code>	“dma-max” or “dma_max”
<code>SCSI_CAP_MSG_OUT</code>	“msg-out” or “msg_out”
<code>SCSI_CAP_DISCONNECT</code>	“disconnect”
<code>SCSI_CAP_SYNCHRONOUS</code>	“synchronous”
<code>SCSI_CAP_WIDE_XFER</code>	“wide-xfer” or “wide_xfer”
<code>SCSI_CAP_PARITY</code>	“parity”
<code>SCSI_CAP_INITIATOR_ID</code>	“initiator-id”
<code>SCSI_CAP_UNTAGGED_QING</code>	“untagged-qing”
<code>SCSI_CAP_TAGGED_QING</code>	“tagged-qing”
<code>SCSI_CAP_ARQ</code>	“auto-rqsense”
<code>SCSI_CAP_LINKED_CMDS</code>	“linked-cmds”
<code>SCSI_CAP_SECTOR_SIZE</code>	“sector-size”
<code>SCSI_CAP_TOTAL_SECTORS</code>	“total-sectors”
<code>SCSI_CAP_GEOMETRY</code>	“geometry”
<code>SCSI_CAP_RESET_NOTIFICATION</code>	“reset-notification”
<code>SCSI_CAP_QFULL_RETRIES</code>	“qfull-retries”
<code>SCSI_CAP_QFULL_RETRY_INTERVAL</code>	“qfull-retry-interval”
<code>SCSI_CAP_LUN_RESET</code>	“lun-reset”
<code>SCSI_CAP_CDB_LEN</code>	“max-cdb-length”
<code>SCSI_CAP_TRAN_LAYER_RETRIES</code>	“tran-layer-retries”

**Return Values** The `scsi_hba_lookup_capstr()` function returns a non-negative index value that corresponds to the capability string. If the string does not match a known capability, `-1` is returned.

**Context** The `scsi_hba_lookup_capstr()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [tran\\_getcap\(9E\)](#), [tran\\_setcap\(9E\)](#), [scsi\\_ifgetcap\(9F\)](#), [scsi\\_ifsetcap\(9F\)](#), [scsi\\_reset\\_notify\(9F\)](#)

*Writing Device Drivers*

**Name** `scsi_hba_pkt_alloc`, `scsi_hba_pkt_free` – allocate and free a `scsi_pkt` structure

**Synopsis** `#include <sys/scsi/scsi.h>`

```
struct scsi_pkt *scsi_hba_pkt_alloc(dev_info_t *dip,
    struct scsi_address *ap, int cmdlen, int statuslen,
    int tgtlen, int hbalen,
    int (*callback)(caddr_t arg), caddr_t arg);

void scsi_hba_pkt_free(struct scsi_address *ap, struct scsi_pkt *pkt);
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters**

<i>dip</i>	Pointer to a <code>dev_info_t</code> structure, defining the HBA driver instance.				
<i>ap</i>	Pointer to a <a href="#">scsi_address(9S)</a> structure, defining the target instance.				
<i>cmdlen</i>	Length in bytes to be allocated for the SCSI command descriptor block (CDB).				
<i>statuslen</i>	Length in bytes to be allocated for the SCSI status completion block (SCB).				
<i>tgtlen</i>	Length in bytes to be allocated for a private data area for the target driver's exclusive use.				
<i>hbalen</i>	Length in bytes to be allocated for a private data area for the HBA driver's exclusive use.				
<i>callback</i>	Indicates what <code>scsi_hba_pkt_alloc()</code> should do when resources are not available: <table> <tr> <td><code>NULL_FUNC</code></td> <td>Do not wait for resources. Return a NULL pointer.</td> </tr> <tr> <td><code>SLEEP_FUNC</code></td> <td>Wait indefinitely for resources.</td> </tr> </table>	<code>NULL_FUNC</code>	Do not wait for resources. Return a NULL pointer.	<code>SLEEP_FUNC</code>	Wait indefinitely for resources.
<code>NULL_FUNC</code>	Do not wait for resources. Return a NULL pointer.				
<code>SLEEP_FUNC</code>	Wait indefinitely for resources.				
<i>arg</i>	Must be NULL.				
<i>pkt</i>	A pointer to a <a href="#">scsi_pkt(9S)</a> structure.				

**Description** For `scsi_hba_pkt_alloc()`:

The `scsi_hba_pkt_alloc()` function allocates space for a `scsi_pkt` structure. HBA drivers must use this interface when allocating a `scsi_pkt` from their [tran\\_init\\_pkt\(9E\)](#) entry point.

If *callback* is `NULL_FUNC`, `scsi_hba_pkt_alloc()` may not sleep when allocating resources, and callers should be prepared to deal with allocation failures.

The `scsi_hba_pkt_alloc()` function copies the [scsi\\_address\(9S\)](#) structure pointed to by *ap* to the `pkt_address` field in the [scsi\\_pkt\(9S\)](#).

The `scsi_hba_pkt_alloc()` function also allocates memory for these [scsi\\_pkt\(9S\)](#) data areas, and sets these fields to point to the allocated memory:

pkt\_ha\_private     HBA private data area.  
pkt\_private        Target driver private data area.  
pkt\_scbp           SCSI status completion block.  
pkt\_cdbp           SCSI command descriptor block.

For `scsi_hba_pkt_free()`:

The `scsi_hba_pkt_free()` function frees the space allocated for the `scsi_pkt(9S)` structure.

**Return Values** The `scsi_hba_pkt_alloc()` function returns a pointer to the `scsi_pkt` structure, or `NULL` if no space is available.

**Context** The `scsi_hba_pkt_alloc()` function can be called from user, interrupt, or kernel context. Drivers must not allow `scsi_hba_pkt_alloc()` to sleep if called from an interrupt routine.

The `scsi_hba_pkt_free()` function can be called from user, interrupt, or kernel context.

**See Also** [tran\\_init\\_pkt\(9E\)](#), [scsi\\_address\(9S\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** `scsi_hba_pkt_comp` – `scsi_pkt` completion routine

**Synopsis** `#include <sys/scsi/scsi.h>`

```
void scsi_hba_pkt_comp(struct scsi_pkt *pkt);
```

**Parameters** *pkt* Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** After filling in [scsi\\_pkt\(9S\)](#) fields with packet completion information, an HBA driver should call the `scsi_hba_pkt_comp()` function. This function is the recommended way for an HBA driver to signal completion of a `scsi_pkt` structure. Use is mandatory for HBA drivers that use [tran\\_setup\\_pkt\(9E\)](#).

Calling the `scsi_hba_pkt_comp()` function allows SCSI to observe, and possibly react to, the completion of a `scsi_pkt` request.

A call to the `scsi_hba_pkt_comp()` function will always result in a call to the `pkt_comp()` callback function defined in [scsi\\_pkt\(9S\)](#). This `pkt_comp()` callback may, however, occur after return from `scsi_hba_pkt_comp()`, and may occur from a different thread executing on a different CPU.

**Context** The `scsi_hba_pkt_comp()` function can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [tran\\_setup\\_pkt\(9E\)](#), [scsi\\_pkt\(9S\)](#)

**Notes** HBA driver calls `scsi_hba_pkt_comp()` instead of calling [scsi\\_pkt\(9S\)](#) `pkt_comp` directly.

**Name** `scsi_hba_probe` – default SCSI HBA probe function

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_hba_probe(struct scsi_device *sd, int(*waitfunc)(void));
```

**Interface Level** Solaris architecture specific (Solaris DDI).

**Parameters** `sd` Pointer to a [scsi\\_device\(9S\)](#) structure describing the target.  
`waitfunc` `NULL_FUNC` or `SLEEP_FUNC`.

**Description** `scsi_hba_probe()` is a function providing the semantics of [scsi\\_probe\(9F\)](#). An HBA driver may call `scsi_hba_probe()` from its [tran\\_tgt\\_probe\(9E\)](#) entry point, to probe for the existence of a target on the SCSI bus, or the HBA may set [tran\\_tgt\\_probe\(9E\)](#) to point to `scsi_hba_probe` directly.

**Return Values** See [scsi\\_probe\(9F\)](#) for the return values from `scsi_hba_probe()`.

**Context** `scsi_hba_probe()` should only be called from the HBA's [tran\\_tgt\\_probe\(9E\)](#) entry point.

**See Also** [tran\\_tgt\\_probe\(9E\)](#), [scsi\\_probe\(9F\)](#), [scsi\\_device\(9S\)](#)

*Writing Device Drivers*

- 
- Name** `scsi_hba_tran_alloc`, `scsi_hba_tran_free` – allocate and free transport structures
- Synopsis** `#include <sys/scsi/scsi.h>`
- ```
scsi_hba_tran_t *scsi_hba_tran_alloc(dev_info_t *dip, int flags);
void scsi_hba_tran_free(scsi_hba_tran_t *hba_tran);
```
- Interface Level** Solaris architecture specific (Solaris DDI).
- Parameters**
- |                 |                                                                                                               |
|-----------------|---------------------------------------------------------------------------------------------------------------|
| <i>dip</i>      | Pointer to a <code>dev_info</code> structure, defining the HBA driver instance.                               |
| <i>flag</i>     | Flag modifiers. The only possible flag value is <code>SCSI_HBA_CANSLEEP</code> (memory allocation may sleep). |
| <i>hba_tran</i> | Pointer to a <code>scsi_hba_tran(9S)</code> structure.                                                        |
- Description** For `scsi_hba_tran_alloc()`:
- The `scsi_hba_tran_alloc()` function allocates a `scsi_hba_tran(9S)` structure for a HBA driver. The HBA must use this structure to register its transport vectors with the system by using `scsi_hba_attach_setup(9F)`.
- If the flag `SCSI_HBA_CANSLEEP` is set in *flags*, `scsi_hba_tran_alloc()` may sleep when allocating resources; otherwise it may not sleep, and callers should be prepared to deal with allocation failures.
- For `scsi_hba_tran_free()`:
- The `scsi_hba_tran_free()` function is used to free the `scsi_hba_tran(9S)` structure allocated by `scsi_hba_tran_alloc()`.
- Return Values** The `scsi_hba_tran_alloc()` function returns a pointer to the allocated transport structure, or `NULL` if no space is available.
- Context** The `scsi_hba_tran_alloc()` function can be called from user, interrupt, or kernel context. Drivers must not allow `scsi_hba_tran_alloc()` to sleep if called from an interrupt routine.
- The `scsi_hba_tran_free()` function can be called from user, interrupt, or kernel context context.
- See Also** `scsi_hba_attach_setup(9F)`, `scsi_hba_tran(9S)`
- Writing Device Drivers*

**Name** scsi\_ifgetcap, scsi\_ifsetcap – get/set SCSI transport capability

**Synopsis** #include <sys/scsi/scsi.h>

```
int scsi_ifgetcap(struct scsi_address *ap, char *cap, int whom);
int scsi_ifsetcap(struct scsi_address *ap, char *cap, int value,
    int whom);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

- ap* Pointer to the `scsi_address` structure.
- cap* Pointer to the string capability identifier.
- value* Defines the new state of the capability.
- whom* Determines if all targets or only the specified target is affected.

**Description** The `scsi_ifsetcap()` function is used by target drivers to set the capabilities of the host adapter driver. The *cap* pointer is a name-value pair identified by a null-terminated character string and the integer value of the *cap*. The current value of the capability can be retrieved with the `scsi_ifgetcap()` function. If the *whom* value is 0, all target drivers are affected. Otherwise, the `scsi_address` structure pointed to by *ap* is the only target that is affected.

The driver should confirm that `scsi_ifsetcap()` and `scsi_ifsetcap()` functions are called with a *cap* that points to a capability which is supported by the device.

The following capabilities have been defined:

|                           |                                                                                                                                                                                                                                                                                                                                  |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dma-max</code>      | Maximum dma transfer size that is supported by the host adapter.                                                                                                                                                                                                                                                                 |
| <code>dma-max-arch</code> | Maximum dma transfer size that is supported by system. Takes the host adapter and system architecture into account. This is useful for target drivers which do not support partial DMAs on systems which do not have an IOMMU. In this case, the DMA can also be limited by the host adapters “scatter/gather” list constraints. |
|                           | The “ <code>dma-max-arch</code> ” capability can not be set. It is implemented with this command and does not rely on a <a href="#">tran_getcap(9E)</a> response from the HBA.                                                                                                                                                   |
| <code>msg-out</code>      | Message out capability that is supported by the host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                                             |
| <code>disconnect</code>   | Disconnect capability that is supported by the host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                                              |
| <code>synchronous</code>  | Synchronous data transfer capability that is supported by the host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                               |



|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>wide-xfer</code>          | Wide transfer capability that is supported by the host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>parity</code>             | Parity checking capability that is supported by host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>initiator-id</code>       | Host bus address that is returned.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>untagged-queueing</code>  | Host adapter capability that supports internal queueing of commands without tagged queueing: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>tagged-queueing</code>    | Host adapter capability that supports queueing: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>auto-rqsense</code>       | Host adapter capability that supports auto request sense on check conditions: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>sector-size</code>        | Capability that is set by the target driver to inform the HBA of the granularity, in bytes, of the DMA breakup. The HBA DMA limit structure is set to reflect the byte total of this setting. See <a href="#">ddi_dma_lim_sparc(9S)</a> or <a href="#">ddi_dma_lim_x86(9S)</a> . The <code>sector-size</code> should be set to the size of the physical disk sector. The capability defaults to 512 bytes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>total-sectors</code>      | Capability that is set by the target driver to inform the HBA of the total number of sectors on the device returned by the SCSI <code>get capacity</code> command. This capability must be set before the target driver “gets” the geometry capability.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>geometry</code>           | <p>Capability that returns the HBA geometry of a target disk. The target driver sets the <code>total-sectors</code> capability before “getting” the geometry capability. The geometry is returned as a 32-bit value. The upper 16 bits represent the number of heads per cylinder. The lower 16 bits represent the number of sectors per track. The geometry capability cannot be “set”.</p> <p>If geometry is not relevant or appropriate for the target disk, <code>scsi_ifgetcap()</code> can return -1 to indicate that the geometry is not defined. For example, if the HBA BIOS supports Logical Block Addressing for the target disk, <code>scsi_ifgetcap()</code> returns -1. Attempts to retrieve the “virtual geometry” from the target driver, such as the <code>DKIOCG_VIRTGEOM</code> ioctl, will fail. See <a href="#">dkio(7I)</a> for more information about <code>DKIOCG_VIRTGEOM</code>.</p> |
| <code>reset-notification</code> | Host adapter capability that supports bus reset notification: 0 disables, 1 enables. See <a href="#">scsi_reset_notify(9F)</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>linked-cmds</code>        | Host adapter capability that supports linked commands: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>qfull-retries</code>        | Capability that enables or disables QUEUE FULL handling. If 0, the HBA will not retry a command when a QUEUE FULL status is returned. If the value is greater than 0, the HBA driver retries the command a specified number of times at an interval determined by the <code>qfull-retry-interval</code> . The range for <code>qfull-retries</code> is 0-255.                                                                                                                                                                                                                                                                                                 |
| <code>qfull-retry-interval</code> | Capability that sets the retry interval in milliseconds (ms) for commands completed with a QUEUE FULL status. The range for <code>qfull-retry-intervals</code> is 0-1000 ms.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>lun-reset</code>            | Capability that is created with a value of zero by HBA drivers that support the RESET_LUN flag in the <code>tran_reset(9E)</code> function. If it exists, the <code>lun-reset</code> value can be set to 1 by target drivers to allow the use of LOGICAL UNIT RESET on a specific target instance. If <code>lun-reset</code> does not exist or has a value of zero, <code>scsi_reset(9F)</code> is prevented from passing the RESET_LUN flag to <code>tran_reset()</code> function of the HBA driver. If <code>lun-reset</code> exists and has a value of 1, the <code>tran_reset()</code> function of the HBA driver can be called with the RESET_LUN flag. |
| <code>interconnect-type</code>    | Capability held in the <code>tran_interconnect_type</code> element of struct <code>scsi_hba_tran</code> that indicates the HBA transport interconnect type. The integer value of the interconnect type of the transport is defined in the <code>services.h</code> header file.                                                                                                                                                                                                                                                                                                                                                                               |
| <code>max-cdb-length</code>       | Host adapter capability of the maximum supported CDB (Command Descriptor Block) length. The target driver asks for the capability at attach time. If the HBA driver supports the capability, the maximum length of the CDB is returned in bytes. The target driver can then use that value to determine which CDB is used for the HBA.<br><br>If the HBA driver does not support the <code>max-cdb-length</code> capability, the default value of the target driver is used for the CDB determination.                                                                                                                                                       |
| <code>tran-layer-retries</code>   | Transport Layer retries is support by the host adapter: 0 disables, 1 enables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

**Return Values** The `scsi_ifsetcap()` function returns:

- 1 If the capability was successfully set to the new value.
- 0 If the capability is not variable.
- 1 If the capability was not defined, or setting the capability to a new value failed.

The `scsi_ifgetcap()` function returns the current value of a capability, or:

-1 If the capability was not defined.

**Examples** EXAMPLE 1 Using `scsi_ifgetcap()`

```
if (scsi_ifgetcap(&sd->sd_address, "auto-rqsense", 1) == 1) {
    un->un_arq_enabled = 1;
} else {
    un->un_arq_enabled =
        ((scsi_ifsetcap(&sd->sd_address, "auto-rqsense", 1, 1) == 1) ?
         1 : 0);
}

if (scsi_ifsetcap(&devp->sd_address, "tagged-qing", 1, 1) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = MAX_THROTTLE;
} else if (scsi_ifgetcap(&devp->sd_address, "untagged-qing", 0) == 1) {
    un->un_dp->options |= SD_QUEUEING;
    un->un_throttle = 3;
} else {
    un->un_dp->options &= ~SD_QUEUEING;
    un->un_throttle = 1;
}
```

**Context** These functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [tran\\_reset\(9E\)](#), [scsi\\_hba\\_lookup\\_capstr\(9F\)](#), [scsi\\_reset\(9F\)](#), [scsi\\_reset\\_notify\(9F\)](#), [ddi\\_dma\\_lim\\_sparc\(9S\)](#), [ddi\\_dma\\_lim\\_x86\(9S\)](#), [scsi\\_address\(9S\)](#), [scsi\\_arq\\_status\(9S\)](#)

*Writing Device Drivers*

**Name** scsi\_init\_pkt – prepare a complete SCSI packet

**Synopsis** #include <sys/scsi/scsi.h>

```
struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap,
    struct scsi_pkt *pktp, struct buf *bp, int cmdlen, int statuslen,
    int privatelen, int flags, int (*callback)(caddr_t), caddr_t arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *ap*

Pointer to a [scsi\\_address\(9S\)](#) structure.

*pktp*

A pointer to a [scsi\\_pkt\(9S\)](#) structure.

*bp*

Pointer to a [buf\(9S\)](#) structure.

*cmdlen*

The required length for the SCSI command descriptor block (CDB) in bytes.

*statuslen*

The required length for the SCSI status completion block (SCB) in bytes. Valid values are:

0

No status back.

1

Return SCSI status byte.

sizeof(scsi\_arq\_status)

Return status information in a [scsi\\_arq\\_status](#) structure. This will include up to 20 bytes of sense data. Please refer to [scsi\\_arq\\_status\(9S\)](#) for more information.

For extra sense packets (PKT\_XARQ flag asserted), set *statuslen* to be a greater number like,  $(N + \text{sizeof}(\text{struct scsi\_arq\_status}))$  where *N* is the number of extra bytes beyond the default 20. For example, *N*=1 requests 21 bytes of sense, *N*=235 asks for 255 bytes.

*privatelen*

The required length for the *pkt\_private* area.

*flags*

Flags modifier.

*callback*

A pointer to a callback function, NULL\_FUNC, or SLEEP\_FUNC.

*arg*

The *callback* function argument.

**Description** Target drivers use `scsi_init_pkt()` to request the transport layer to allocate and initialize a packet for a SCSI command which possibly includes a data transfer. If `pkt` is `NULL`, a new `scsi_pkt(9S)` is allocated using the HBA driver's packet allocator. The `bp` is a pointer to a `buf(9S)` structure. If `bp` is non-`NULL` and contains a valid byte count, the `buf(9S)` structure is also set up for DMA transfer using the HBA driver DMA resources allocator. When `bp` is allocated by `scsi_alloc_consistent_buf(9F)`, the `PKT_CONSISTENT` bit must be set in the `flags` argument to ensure proper operation. If `privatelen` is non-zero then additional space is allocated for the `pkt_private` area of the `scsi_pkt(9S)`. On return `pkt_private` points to this additional space. Otherwise `pkt_private` is a pointer that is typically used to store the `bp` during execution of the command. In this case `pkt_private` is `NULL` on return.

The `flags` argument is a set of bit flags. Possible bits include:

#### PKT\_CONSISTENT

This must be set if the DMA buffer was allocated using `scsi_alloc_consistent_buf(9F)`. In this case, the HBA driver will guarantee that the data transfer is properly synchronized before performing the target driver's command completion callback.

#### PKT\_DMA\_PARTIAL

This may be set if the driver can accept a partial DMA mapping. If set, `scsi_init_pkt()` will allocate DMA resources with the `DDI_DMA_PARTIAL` bit set in the `dma_flag` element of the `ddi_dma_req(9S)` structure. The `pkt_resid` field of the `scsi_pkt(9S)` structure may be returned with a non-zero value, which indicates the number of bytes for which `scsi_init_pkt()` was unable to allocate DMA resources. In this case, a subsequent call to `scsi_init_pkt()` may be made for the same `pkt` and `bp` to adjust the DMA resources to the next portion of the transfer. This sequence should be repeated until the `pkt_resid` field is returned with a zero value, which indicates that with transport of this final portion the entire original request will have been satisfied.

#### PKT\_XARQ

Setting this flag requests that the HBA return extra sense data for this `scsi_pkt(9S)`. The default auto request sense mechanism returns up to 20 bytes. More than 20 bytes of sense data can be requested by setting this flag and setting the `statuslen` correctly. Set the `statuslen` to be the `sizeof(struct scsi_arq_status)` plus the number of sense bytes needed beyond 20. For example, set `statuslen` to be `(sizeof(struct scsi_arq_status) + 5)` for 25 bytes of sense.

When calling `scsi_init_pkt()` to move already-allocated DMA resources, the `cmdlen`, `statuslen`, and `privatelen` fields are ignored.

The last argument `arg` is supplied to the `callback` function when it is invoked.

`callback` indicates what the allocator routines should do when resources are not available:

|                         |                                                                |
|-------------------------|----------------------------------------------------------------|
| <code>NULL_FUNC</code>  | Do not wait for resources. Return a <code>NULL</code> pointer. |
| <code>SLEEP_FUNC</code> | Wait indefinitely for resources.                               |

**Other Values** *callback* points to a function which is called when resources may have become available. *callback* must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

When allocating DMA resources, `scsi_init_pkt()` returns the `scsi_pkt` field `pkt_resid` as the number of residual bytes for which the system was unable to allocate DMA resources. A `pkt_resid` of 0 means that all necessary DMA resources were allocated.

**Return Values** The `scsi_init_pkt()` function returns NULL if the packet or DMA resources could not be allocated. Otherwise, it returns a pointer to an initialized `scsi_pkt(9S)`. If *pktp* was not NULL the return value will be *pktp* on successful initialization of the packet.

**Context** If *callback* is SLEEP\_FUNC, then this routine can be called only from user or kernel context. Otherwise, it can be called from user, interrupt, or kernel context. The *callback* function may not block or call routines that block.

**Examples** **EXAMPLE 1** Allocating a Packet Without DMA Resources Attached

To allocate a packet without DMA resources attached, use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, NULL, CDB_GROUP1,
                  1, sizeof (struct my_pkt_private *), 0,
                  sd_runout, sd_unit);
```

**EXAMPLE 2** Allocating a Packet With DMA Resources Attached

To allocate a packet with DMA resources attached use:

```
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP1,
                  sizeof(struct scsi_arq_status), 0, 0, NULL_FUNC, NULL);
```

**EXAMPLE 3** Attaching DMA Resources to a Preallocated Packet

To attach DMA resources to a preallocated packet, use:

```
pkt = scsi_init_pkt(&devp->sd_address, old_pkt, bp, 0,
                  0, 0, 0, sd_runout, (caddr_t) sd_unit);
```

**EXAMPLE 4** Allocating a Packet with Consistent DMA Resources Attached

Since the packet is already allocated, the *cmdlen*, *statuslen* and *privatelen* are 0. To allocate a packet with consistent DMA resources attached, use:

```
bp = scsi_alloc_consistent_buf(&devp->sd_address, NULL,
                              SENSE_LENGTH, B_READ, SLEEP_FUNC, NULL);
pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,
                  sizeof(struct scsi_arq_status), sizeof (struct my_pkt_private *),
                  PKT_CONSISTENT, SLEEP_FUNC, NULL);
```

**EXAMPLE 5** Allocating a Packet with Partial DMA Resources Attached

To allocate a packet with partial DMA resources attached, use:

```
my_pkt = scsi_init_pkt(&devp->sd_address, NULL, bp, CDB_GROUP0,  
    1, sizeof (struct buf *), PKT_DMA_PARTIAL,  
    SLEEP_FUNC, NULL);
```

**See Also** [scsi\\_alloc\\_consistent\\_buf\(9F\)](#), [scsi\\_destroy\\_pkt\(9F\)](#), [scsi\\_dmaget\(9F\)](#),  
[scsi\\_pktalloc\(9F\)](#), [buf\(9S\)](#), [ddi\\_dma\\_req\(9S\)](#), [scsi\\_address\(9S\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Notes** If a DMA allocation request fails with `DDI_DMA_NOMAPPING`, the `B_ERROR` flag will be set in `bp`, and the `b_error` field will be set to `EFAULT`.

If a DMA allocation request fails with `DDI_DMA_TOOBIG`, the `B_ERROR` flag will be set in `bp`, and the `b_error` field will be set to `EINVAL`.

**Name** `scsi_log` – display a SCSI-device-related message

**Synopsis** `#include <sys/scsi/scsi.h>`  
`#include <sys/cmn_err.h>`

```
void scsi_log(dev_info_t *dip, char *drv_name, uint_t level,
              const char *fmt, ...);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <i>dip</i>      | Pointer to the <code>dev_info</code> structure. |
| <i>drv_name</i> | String naming the device.                       |
| <i>level</i>    | Error level.                                    |
| <i>fmt</i>      | Display format.                                 |

**Description** The `scsi_log()` function is a utility function that displays a message via the [cmn\\_err\(9F\)](#) routine. The error levels that can be passed in to this function are `CE_PANIC`, `CE_WARN`, `CE_NOTE`, `CE_CONT`, and `SCSI_DEBUG`. The last level is used to assist in displaying debug messages to the console only. *drv\_name* is the short name by which this device is known; example disk driver names are `sd` and `cmdk`. If the *dev\_info\_t* pointer is `NULL`, then the *drv\_name* will be used with no unit or long name.

If the first character in *format* is:

- An exclamation mark (!), the message goes only to the system buffer.
- A caret (^), the message goes only to the console.
- A question mark (?) and *level* is `CE_CONT`, the message is always sent to the system buffer, but is written to the console only when the system has been booted in verbose mode. See [kernel\(1M\)](#). If neither condition is met, the ? character has no effect and is simply ignored.

All formatting conversions in use by `cmn_err()` also work with `scsi_log()`.

**Context** The `scsi_log()` function may be called from user, interrupt, or kernel context.

**See Also** [kernel\(1M\)](#), [sd\(7D\)](#), [cmn\\_err\(9F\)](#), [scsi\\_errmsg\(9F\)](#)

*Writing Device Drivers*



**Name** scsi\_pktalloc, scsi\_realloc, scsi\_pktfree, scsi\_resfree – SCSI packet utility routines

**Synopsis** #include <sys/scsi/scsi.h>

```
struct scsi_pkt *scsi_pktalloc (struct scsi_address* ap, int cmdlen,
                               int statuslen, int (*callback)(void));

struct scsi_pkt *scsi_realloc (struct scsi_address* ap, int cmdlen,
                               int statuslen, opaque_t dmatoken, int (*callback)(void));

void scsi_pktfree (struct scsi_pkt* pkt);

void scsi_resfree (struct scsi_pkt* pkt);
```

**Interface Level** The `scsi_pktalloc()`, `scsi_pktfree()`, `scsi_realloc()`, and `scsi_resfree()` functions are obsolete. The `scsi_pktalloc()` and `scsi_realloc()` functions have been replaced by [scsi\\_init\\_pkt\(9F\)](#). The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by [scsi\\_destroy\\_pkt\(9F\)](#).

**Parameters**

- ap* Pointer to a `scsi_address` structure.
- cmdlen* The required length for the SCSI command descriptor block (CDB) in bytes.
- statuslen* The required length for the SCSI status completion block (SCB) in bytes.
- dmatoken* Pointer to an implementation-dependent object.
- callback* A pointer to a callback function, or `NULL_FUNC` or `SLEEP_FUNC`.
- pkt* Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Description** The `scsi_pktalloc()` function requests the host adapter driver to allocate a command packet. For commands that have a data transfer associated with them, `scsi_realloc()` should be used.

*ap* is a pointer to a `scsi_address` structure. Allocator routines use it to determine the associated host adapter.

The *cmdlen* parameter is the required length for the SCSI command descriptor block. This block is allocated such that a kernel virtual address is established in the `pkt_cdbp` field of the allocated `scsi_pkt` structure.

*statuslen* is the required length for the SCSI status completion block. The address of the allocated block is placed into the `pkt_scbp` field of the `scsi_pkt` structure.

The *dmatoken* parameter is a pointer to an implementation dependent object which defines the length, direction, and address of the data transfer associated with this SCSI packet (command). The *dmatoken* must be a pointer to a [buf\(9S\)](#) structure. If *dmatoken* is `NULL`, no DMA resources are required by this SCSI command, so none are allocated. Only one transfer direction is allowed per command. If there is an unexpected data transfer phase (either no data

transfer phase expected, or the wrong direction encountered), the command is terminated with the `pkt_reason` set to `CMD_DMA_DERR`. *dmatoken* provides the information to determine if the transfer count is correct.

*callback* indicates what the allocator routines should do when resources are not available:

|                         |                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>NULL_FUNC</code>  | Do not wait for resources. Return a NULL pointer.                                                                                                                                                                                                                                                                                                                                              |
| <code>SLEEP_FUNC</code> | Wait indefinitely for resources.                                                                                                                                                                                                                                                                                                                                                               |
| Other Values            | <i>callback</i> points to a function which is called when resources may have become available. <i>callback</i> must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry. |

The `scsi_pktfree()` function frees the packet.

The `scsi_resfree()` function free all resources held by the packet and the packet itself.

**Return Values** Both allocation routines return a pointer to a `scsi_pkt` structure on success, or NULL on failure.

**Context** If *callback* is `SLEEP_FUNC`, then this routine can be called only from user or kernel context. Otherwise, it can be called from user, kernel, or interrupt context. The *callback* function may not block or call routines that block. Both deallocation routines can be called from user, kernel, or interrupt context.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [scsi\\_dmafree\(9F\)](#), [scsi\\_dmaget\(9F\)](#), [buf\(9S\)](#), [scsi\\_pkt\(9S\)](#)

### *Writing Device Drivers*

**Notes** The `scsi_pktalloc()`, `scsi_pktfree()`, `scsi_realloc()`, and `scsi_resfree()` functions are obsolete and will be discontinued in a future release. The `scsi_pktalloc()` and `scsi_realloc()` functions have been replaced by [scsi\\_init\\_pkt\(9F\)](#). The `scsi_pktfree()` and `scsi_resfree()` functions have been replaced by [scsi\\_destroy\\_pkt\(9F\)](#).

**Name** `scsi_poll` – run a polled SCSI command on behalf of a target driver

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_poll(struct scsi_pkt *pkt);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** `pkt` Pointer to the [scsi\\_pkt\(9S\)](#) structure.

**Description** The `scsi_poll()` function requests the host adapter driver to run a polled command. Unlike [scsi\\_transport\(9F\)](#) which runs commands asynchronously, `scsi_poll()` runs commands to completion before returning. If the `pkt_time` member of `pkt` is 0, the value of `pkt_time` is defaulted to `SCSI_POLL_TIMEOUT` to prevent an indefinite hang of the system.

**Return Values** The `scsi_poll()` function returns:

0 command completed successfully.

-1 command failed.

**Context** The `scsi_poll()` function can be called from user, interrupt, or kernel context. This function should not be called when the caller is executing [timeout\(9F\)](#) in the context of a thread.

**See Also** [makecom\(9F\)](#), [scsi\\_transport\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Warnings** Since `scsi_poll()` runs commands to completion before returning, it may require more time than is desirable when called from interrupt context. Therefore, calling `scsi_poll` from interrupt context is not recommended.

**Name** scsi\_probe – utility for probing a scsi device

**Synopsis** #include <sys/scsi/scsi.h>

```
int scsi_probe(struct scsi_device *devp, int (*waitfunc);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *devp* Pointer to a [scsi\\_device\(9S\)](#) structure

*waitfunc* NULL\_FUNC or SLEEP\_FUNC

**Description** `scsi_probe()` determines whether a target/lun is present and sets up the `scsi_device` structure with inquiry data.

`scsi_probe()` uses the SCSI Inquiry command to test if the device exists. It can retry the Inquiry command as appropriate. If `scsi_probe()` is successful, it will allocate space for the `scsi_inquiry` structure and assign the address to the `sd_inq` member of the [scsi\\_device\(9S\)](#) structure. `scsi_probe()` will then fill in this [scsi\\_inquiry\(9S\)](#) structure and return `SCSI_PROBE_EXISTS`. If `scsi_probe()` is unsuccessful, it returns `SCSI_PROBE_NOMEM` in spite of callback set to `SLEEP_FUNC`.

[scsi\\_unprobe\(9F\)](#) is used to undo the effect of `scsi_probe()`.

If the target is a non-CCS device, `SCSI_PROBE_NONCCS` will be returned.

*waitfunc* indicates what the allocator routines should do when resources are not available; the valid values are:

`NULL_FUNC` Do not wait for resources. Return `SCSI_PROBE_NOMEM` or `SCSI_PROBE_FAILURE`

`SLEEP_FUNC` Wait indefinitely for resources.

**Return Values** `scsi_probe()` returns:

`SCSI_PROBE_BUSY` Device exists but is currently busy.

`SCSI_PROBE_EXISTS` Device exists and inquiry data is valid.

`SCSI_PROBE_FAILURE` Polled command failure.

`SCSI_PROBE_NOMEM` No space available for structures.

`SCSI_PROBE_NOMEM_CB` No space available for structures but callback request has been queued.

`SCSI_PROBE_NONCCS` Device exists but inquiry data is not valid.

`SCSI_PROBE_NORESP` Device does not respond to an INQUIRY.

**Context** `scsi_probe()` is normally called from the target driver's [probe\(9E\)](#) or [attach\(9E\)](#) routine. In any case, this routine should not be called from interrupt context, because it can sleep waiting for memory to be allocated.

**Examples** EXAMPLE 1 Using `scsi_probe()`

```
switch (scsi_probe(devp, NULL_FUNC)) {
default:
case SCSIPROBE_NORESP:
case SCSIPROBE_NONCCS:
case SCSIPROBE_NOMEM:
case SCSIPROBE_FAILURE:
case SCSIPROBE_BUSY:
    break;
case SCSIPROBE_EXISTS:
    switch (devp->sd_inq->inq_dtype) {
    case DTYPE_DIRECT:
        rval = DDI_PROBE_SUCCESS;
        break;
    case DTYPE_RODIRECT:
        rval = DDI_PROBE_SUCCESS;
        break;
    case DTYPE_NOTPRESENT:
    default:
        break;
    }
}
scsi_unprobe(devp);
```

**See Also** [attach\(9E\)](#), [probe\(9E\)](#), [scsi\\_slave\(9F\)](#), [scsi\\_unprobe\(9F\)](#), [scsi\\_unslave\(9F\)](#), [scsi\\_device\(9S\)](#), [scsi\\_inquiry\(9S\)](#)

*ANSI Small Computer System Interface-2 (SCSI-2)*

*Writing Device Drivers*

**Notes** A *waitfunc* function other than `NULL_FUNC` or `SLEEP_FUNC` is not supported and may have unexpected results.

**Name** `scsi_reset` – reset a SCSI bus or target

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_reset(struct scsi_address *ap, int level);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *ap* Pointer to the `scsi_address` structure.

*level* The level of reset required.

**Description** The `scsi_reset()` function asks the host adapter driver to reset the SCSI bus or a SCSI target as specified by *level*. If *level* equals `RESET_ALL`, the SCSI bus is reset. If it equals `RESET_TARGET`, *ap* is used to determine the target to be reset. If it equals `RESET_LUN`, *ap* is used to determine the logical unit to be reset.

When given the `RESET_LUN` level, `scsi_reset()` can return failure if the LOGICAL UNIT RESET message is not supported by the target device, or if the underlying HBA driver does not implement the ability to issue a LOGICAL UNIT RESET message.

Note that, at the point when `scsi_reset()` resets the logical unit (case `RESET_LUN`), or the target (case `RESET_TARGET`), or the bus (case `RESET_ALL`), there might be one or more command packets outstanding. That is, packets have been passed to `scsi_transport()`, and queued or possibly transported, but the commands have not been completed and the target completion routine has not been called for those packets.

The successful call to `scsi_reset()` has the side effect that any such commands currently outstanding are aborted, at which point the packets are marked with `pkt_reason` set to `CMD_RESET`, and the appropriate bit -- either `STAT_BUS_RESET` or `STAT_DEV_RESET` -- is set in `pkt_statistics`. Once thus appropriately marked, the aborted command packets are passed to the target driver command completion routine.

Also note that, at the moment that a thread executing `scsi_reset()` actually resets the target or the bus, it is possible that a second thread may have already called `scsi_transport()`, but not yet queued or transported its command. In this case the HBA will not yet have received the second thread's packet and this packet will not be aborted.

**Return Values** The `scsi_reset()` function returns:

1 Upon success.

0 Upon failure.

**Context** The `scsi_reset()` function can be called from user, interrupt, or kernel context.

**See Also** [tran\\_reset\(9E\)](#), [tran\\_reset\\_notify\(9E\)](#), [scsi\\_abort\(9F\)](#)

*Writing Device Drivers*

**Name** scsi\_reset\_notify – notify target driver of bus resets

**Synopsis** #include <sys/scsi/scsi.h>

```
void scsi_reset_notify(struct scsi_address *ap, int flag,
    void (*callback)(caddr_t), caddr_t arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                 |                                                                             |
|-----------------|-----------------------------------------------------------------------------|
| <i>ap</i>       | Pointer to the <code>scsi_address</code> structure.                         |
| <i>flag</i>     | A flag indicating registration or cancellation of the notification request. |
| <i>callback</i> | A pointer to the target driver's reset notification function.               |
| <i>arg</i>      | The callback function argument.                                             |

**Description** The `scsi_reset_notify()` function is used by a target driver when it needs to be notified of a bus reset. The bus reset could be issued by the transport layer (e.g. the host bus adapter (HBA) driver or controller) or by another initiator.

The argument *flag* is used to register or cancel the notification. The supported values for *flag* are as follows:

|                   |                                                                                    |
|-------------------|------------------------------------------------------------------------------------|
| SCSI_RESET_NOTIFY | Register <i>callback</i> as the reset notification function for the target driver. |
| SCSI_RESET_CANCEL | Cancel the reset notification request.                                             |

Target drivers can find out whether the HBA driver and controller support reset notification by checking the reset-notification capability using the [scsi\\_ifgetcap\(9F\)](#) function.

**Return Values** If *flag* is SCSI\_RESET\_NOTIFY, `scsi_reset_notify()` returns:

|             |                                                                                           |
|-------------|-------------------------------------------------------------------------------------------|
| DDI_SUCCESS | The notification request has been accepted.                                               |
| DDI_FAILURE | The transport layer does not support reset notification or could not accept this request. |

If *flag* is SCSI\_RESET\_CANCEL, `scsi_reset_notify()` returns:

|             |                                             |
|-------------|---------------------------------------------|
| DDI_SUCCESS | The notification request has been canceled. |
| DDI_FAILURE | No notification request was registered.     |

**Context** The `scsi_reset_notify()` function can be called from user, interrupt, or kernel context.

**See Also** [scsi\\_address\(9S\)](#), [scsi\\_ifgetcap\(9F\)](#)

*Writing Device Drivers*

**Name** `scsi_sense_key`, `scsi_sense_asc`, `scsi_sense_ascq` – retrieve fields from SCSI sense data

**Synopsis** `#include <sys/scsi/scsi.h>`

```
uint8_t scsi_sense_key(uint8_t *sense_buffer);
```

```
uint8_t scsi_sense_asc(uint8_t *sense_buffer);
```

```
uint8_t scsi_sense_ascq(uint8_t *sense_buffer);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *sense\_buffer* Pointer to a buffer containing SCSI sense data. The sense data is expected in wire format starting at the response code. It can be in either fixed or descriptor format.

**Description** The `scsi_sense_key()` function returns the sense key value from the sense data contained in the `sense_buffer`.

The `scsi_sense_asc()` function returns the additional sense code (ASC) value from the sense data contained in the `sense_buffer`.

The `scsi_sense_ascq()` function returns the additional sense code qualifier (ASCQ) value from the sense data contained in the `sense_buffer`.

The `scsi_sense_key()`, `scsi_sense_asc()`, and `scsi_sense_ascq()` functions are used to retrieve values from SCSI sense data, regardless of whether the sense data is in fixed format or descriptor format.

Drivers should use [scsi\\_validate\\_sense\(9F\)](#) to ensure that valid sense key, asc, and ascq values are present in the sense data.

**Return Values** The `scsi_sense_key()` function returns the sense key value from the sense buffer. The `scsi_sense_asc()` function returns the additional sense code (ASC) from the sense buffer and the `scsi_sense_ascq()` function returns the additional sense code qualifier (ASCQ) from the sense buffer.

**Context** The `scsi_sense_key()`, `scsi_sense_asc()`, and `scsi_sense_ascq()` functions can be called from user or interrupt context.

**See Also** [scsi\\_ext\\_sense\\_fields\(9F\)](#), [scsi\\_find\\_sense\\_descr\(9F\)](#), [scsi\\_sense\\_cmdspecific\\_uint64\(9F\)](#), [scsi\\_sense\\_info\\_uint64\(9F\)](#), [scsi\\_validate\\_sense\(9F\)](#)



**Name** `scsi_setup_cdb` – setup SCSI command descriptor block (CDB)

**Synopsis** `int scsi_setup_cdb(union scsi_cdb *cdbp, uchar_t cmd, uint_t addr, uint_t cnt, uint_t othr_cdb_data);`

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|                            |                                                                                                                                                                                             |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>cdbp</code>          | Pointer to command descriptor block.                                                                                                                                                        |
| <code>cmd</code>           | The first byte of the SCSI group 0, 1, 2, 4, or 5 CDB.                                                                                                                                      |
| <code>addr</code>          | Pointer to the location of the data.                                                                                                                                                        |
| <code>cnt</code>           | Data transfer length in units defined by the SCSI device type. For sequential devices <code>cnt</code> is the number of bytes. For block devices, <code>cnt</code> is the number of blocks. |
| <code>othr_cdb_data</code> | Additional CDB data.                                                                                                                                                                        |

**Description** The `scsi_setup_cdb()` function initializes a group 0, 1, 2, 4, or 5 type of command descriptor block pointed to by `cdbp` using `cmd`, `addr`, `cnt`, `othr_cdb_data`.

`addr` should be set to 0 for commands having no addressing information (for example, group 0 READ command for sequential access devices). `othr_cdb_data` should be additional CDB data for Group 4 commands; otherwise, it should be set to 0.

The `scsi_setup_cdb()` function does not set the LUN bits in `CDB[1]` as the [makecom\(9F\)](#) functions do. Also, the fixed bit for sequential access device commands is not set.

**Return Values** The `scsi_setup_cdb()` function returns:

- 1 Upon success.
- 0 Upon failure.

**Context** These functions can be called from a user, interrupt, or kernel context.

**See Also** [makecom\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

*American National Standard Small Computer System Interface-2 (SCSI-2)*

*American National Standard SCSI-3 Primary Commands (SPC)*

**Name** `scsi_slave` – utility for SCSI target drivers to establish the presence of a target

**Synopsis** `#include <sys/scsi/scsi.h>`

```
int scsi_slave(struct scsi_device *devp, int (*callback)(void));
```

**Interface Level** The `scsi_slave()` function is obsolete. This function has been replaced by [scsi\\_probe\(9F\)](#).

**Parameters** *devp* Pointer to a [scsi\\_device\(9S\)](#) structure.

*callback* Pointer to a callback function, `NULL_FUNC` or `SLEEP_FUNC`.

**Description** `scsi_slave()` checks for the presence of a SCSI device. Target drivers may use this function in their [probe\(9E\)](#) routines. `scsi_slave()` determines if the device is present by using a Test Unit Ready command followed by an Inquiry command. If `scsi_slave()` is successful, it will fill in the `scsi_inquiry` structure, which is the `sd_inq` member of the [scsi\\_device\(9S\)](#) structure, and return `SCSI_PROBE_EXISTS`. This information can be used to determine if the target driver has probed the correct SCSI device type. *callback* indicates what the allocator routines should do when DMA resources are not available:

`NULL_FUNC` Do not wait for resources. Return a NULL pointer.

`SLEEP_FUNC` Wait indefinitely for resources.

**Other Values** *callback* points to a function which is called when resources may have become available. *callback* must return either 0 (indicating that it attempted to allocate resources but again failed to do so), in which case it is put back on a list to be called again later, or 1 indicating either success in allocating resources or indicating that it no longer cares for a retry.

**Return Values** `scsi_slave()` returns:

`SCSI_PROBE_NOMEM` No space available for structures.

`SCSI_PROBE_EXISTS` Device exists and inquiry data is valid.

`SCSI_PROBE_NONCCS` Device exists but inquiry data is not valid.

`SCSI_PROBE_FAILURE` Polled command failure.

`SCSI_PROBE_NORESP` No response to TEST UNIT READY.

**Context** `scsi_slave()` is normally called from the target driver's [probe\(9E\)](#) or [attach\(9E\)](#) routine. In any case, this routine should not be called from interrupt context, because it can sleep waiting for memory to be allocated.

**Attributes** See [attributes\(5\)](#) for a description of the following attributes:

---

| ATTRIBUTE TYPE  | ATTRIBUTE VALUE |
|-----------------|-----------------|
| Stability Level | Obsolete        |

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [probe\(9E\)](#), [ddi\\_iopb\\_alloc\(9F\)](#), [makecom\(9F\)](#), [scsi\\_dmaget\(9F\)](#), [scsi\\_ifgetcap\(9F\)](#), [scsi\\_pktalloc\(9F\)](#), [scsi\\_poll\(9F\)](#), [scsi\\_probe\(9F\)](#), [scsi\\_device\(9S\)](#)

*ANSI Small Computer System Interface-2 (SCSI-2)*

*Writing Device Drivers*

**Notes** The `scsi_slave()` function is obsolete and will be discontinued in a future release. This function has been replaced by [scsi\\_probe\(9F\)](#).

**Name** scsi\_sync\_pkt – synchronize CPU and I/O views of memory

**Synopsis** #include <sys/scsi/scsi.h>

```
void scsi_sync_pkt(struct scsi_pkt *pkt);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *pkt* Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Description** The `scsi_sync_pkt()` function is used to selectively synchronize a CPU's or device's view of the data associated with the SCSI packet that has been mapped for I/O. This may involve operations such as flushes of CPU or I/O caches, as well as other more complex operations such as stalling until hardware write buffers have drained.

This function need only be called under certain circumstances. When a SCSI packet is mapped for I/O using [scsi\\_init\\_pkt\(9F\)](#) and destroyed using [scsi\\_destroy\\_pkt\(9F\)](#), then an implicit `scsi_sync_pkt()` will be performed. However, if the memory object has been modified by either the device or a CPU after the mapping by [scsi\\_init\\_pkt\(9F\)](#), then a call to `scsi_sync_pkt()` is required.

If the same `scsi_pkt` is reused for a data transfer from memory to a device, then `scsi_sync_pkt()` must be called before calling [scsi\\_transport\(9F\)](#). If the same packet is reused for a data transfer from a device to memory `scsi_sync_pkt()` must be called after the completion of the packet but before accessing the data in memory.

**Context** The `scsi_sync_pkt()` function may be called from user, interrupt, or kernel context.

**See Also** [tran\\_sync\\_pkt\(9E\)](#), [ddi\\_dma\\_sync\(9F\)](#), [scsi\\_destroy\\_pkt\(9F\)](#), [scsi\\_init\\_pkt\(9F\)](#), [scsi\\_transport\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** scsi\_transport – request by a SCSI target driver to start a command

**Synopsis** #include <sys/scsi/scsi.h>

```
int scsi_transport(struct scsi_pkt *pkt);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *pkt* Pointer to a [scsi\\_pkt\(9S\)](#) structure.

**Description** Target drivers use `scsi_transport()` to request the host adapter driver to transport a command to the SCSI target device specified by *pkt*. The target driver must obtain resources for the packet using [scsi\\_init\\_pkt\(9F\)](#) prior to calling this function. The packet may be initialized using one of the [makecom\(9F\)](#) functions. `scsi_transport()` does not wait for the SCSI command to complete. See [scsi\\_poll\(9F\)](#) for a description of polled SCSI commands. Upon completion of the SCSI command the host adapter calls the completion routine provided by the target driver in the `pkt_comp` member of the `scsi_pkt` pointed to by *pkt*.

**Return Values** The `scsi_transport()` function returns:

|                  |                                                                                                                                                                            |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TRAN_ACCEPT      | The packet was accepted by the transport layer.                                                                                                                            |
| TRAN_BUSY        | The packet could not be accepted because there was already a packet in progress for this target/lun, the host adapter queue was full, or the target device queue was full. |
| TRAN_BADPKT      | The DMA count in the packet exceeded the DMA engine's maximum DMA size.                                                                                                    |
| TRAN_FATAL_ERROR | A fatal error has occurred in the transport layer.                                                                                                                         |

**Context** The `scsi_transport()` function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE1 Using `scsi_transport()`

```
if ((status = scsi_transport(rqpkt)) != TRAN_ACCEPT) {
    scsi_log(devp, sd_label, CE_WARN,
            "transport of request sense pkt fails (0x%x)\n", status);
}
```

**See Also** [tran\\_start\(9E\)](#), [makecom\(9F\)](#), [scsi\\_init\\_pkt\(9F\)](#), [scsi\\_pktalloc\(9F\)](#), [scsi\\_poll\(9F\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

**Name** `scsi_unprobe`, `scsi_unslave` – free resources allocated during initial probing

**Synopsis** `#include <sys/scsi/scsi.h>`

```
void scsi_unslave(struct scsi_device *devp);
```

```
void scsi_unprobe(struct scsi_device *devp);
```

**Interface Level** Solaris DDI specific (Solaris DDI). The `scsi_unslave()` interface is obsolete. Use `scsi_unprobe()` instead.

**Parameters** *devp* Pointer to a [scsi\\_device\(9S\)](#) structure.

**Description** `scsi_unprobe()` and `scsi_unslave()` are used to free any resources that were allocated on the driver's behalf during [scsi\\_slave\(9F\)](#) and [scsi\\_probe\(9F\)](#) activity.

**Context** `scsi_unprobe()` and `scsi_unslave()` must not be called from an interrupt context.

**See Also** [scsi\\_probe\(9F\)](#), [scsi\\_slave\(9F\)](#), [scsi\\_device\(9S\)](#)

*Writing Device Drivers*

**Notes** The `scsi_unslave()` function is obsolete and will be discontinued in a future release. This function has been replaced by `scsi_unprobe()`.

---

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|-------------------------------|-----------------------------------------------------------------------------------------|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Name</b>                   | scsi_validate_sense – find descriptor in SCSI sense data                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Synopsis</b>               | #include <sys/scsi/scsi.h>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|                               | <pre>int scsi_validate_sense(uint8_t *sense_buffer, int sense_buf_len,                         int *flags);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Interface Level</b>        | Solaris DDI specific (Solaris DDI).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Parameters</b>             | <i>sense_buffer</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Pointer to a buffer containing SCSI sense data. The sense data is expected in wire format starting at the response code.        |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|                               | <i>sense_buf_len</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | Length of sense buffer in bytes.                                                                                                |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|                               | <i>flags</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Returns additional properties of the sense data.                                                                                |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Description</b>            | <p>The <code>scsi_validate_sense()</code> function returns the format of the sense data contained in the provided sense buffer. If the response code field in the sense data is not recognized or if there is not enough sense data to include sense key, asc, and ascq then <code>scsi_validate_sense()</code> returns <code>SENSE_UNUSABLE</code>. If the buffer contains usable sense data in fixed format, the function returns <code>SENSE_FIXED_FORMAT</code>. If the buffer contains usable sense data in descriptor format, the function returns <code>SENSE_DESCR_FORMAT</code>.</p> <p>The following flags may be set as appropriate depending on the sense data:</p> <table> <tr> <td><code>SNS_BUF_OVERFLOW</code></td> <td>The sense data buffer provided for the request is too small to hold all the sense data.</td> </tr> <tr> <td><code>SNS_BUF_DEFERRED</code></td> <td>The sense data contained in the buffer relates to an error that has occurred during the processing of a successfully completed command, such as a cached write that could not be committed to the media.</td> </tr> </table> |                                                                                                                                 | <code>SNS_BUF_OVERFLOW</code> | The sense data buffer provided for the request is too small to hold all the sense data. | <code>SNS_BUF_DEFERRED</code> | The sense data contained in the buffer relates to an error that has occurred during the processing of a successfully completed command, such as a cached write that could not be committed to the media. |
| <code>SNS_BUF_OVERFLOW</code> | The sense data buffer provided for the request is too small to hold all the sense data.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <code>SNS_BUF_DEFERRED</code> | The sense data contained in the buffer relates to an error that has occurred during the processing of a successfully completed command, such as a cached write that could not be committed to the media.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Return Values</b>          | <code>SENSE_UNUSABLE</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | The response code from the sense data is unrecognized or not enough sense data present to provide the sense key, asc, and ascq. |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|                               | <code>SENSE_FIXED_FORMAT</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | The sense data in the buffer is in “fixed format”.                                                                              |                               |                                                                                         |                               |                                                                                                                                                                                                          |
|                               | <code>SENSE_DESCR_FORMAT</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | The sense data in the buffer is in “descriptor format”.                                                                         |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>Context</b>                | The <code>scsi_validate_sense()</code> function can be called from user or interrupt context.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |
| <b>See Also</b>               | <a href="#">scsi_ext_sense_fields(9F)</a> , <a href="#">scsi_find_sense_descr(9F)</a> , <a href="#">scsi_sense_asc(9F)</a> ,<br><a href="#">scsi_sense_ascq(9F)</a> , <a href="#">scsi_sense_cmdspecific_uint64(9F)</a> ,<br><a href="#">scsi_sense_info_uint64(9F)</a> , <a href="#">scsi_sense_key(9F)</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                 |                               |                                                                                         |                               |                                                                                                                                                                                                          |

**Name** scsi\_vu\_errmsg – display a SCSI request sense message

**Synopsis** #include <sys/scsi/scsi.h>

```
void scsi_vu_errmsg(struct scsi_pkt *pkt, char *drv_name, int severity,
    int err_blkno, struct scsi_key_strings *cmdlist,
    struct scsi_extended_sense *sensep,
    struct scsi_asq_key_strings *asc_list,
    char **decode_fru struct scsi_device*, char *, int, char);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** The following parameters are supported:

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>devp</i>       | Pointer to the <a href="#">scsi_device(9S)</a> structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <i>pkt</i>        | Pointer to a <a href="#">scsi_pkt(9S)</a> structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <i>drv_name</i>   | String used by <a href="#">scsi_log(9F)</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <i>severity</i>   | Error severity level, maps to severity strings below.                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>blkno</i>      | Requested block number.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <i>err_blkno</i>  | Error block number.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <i>cmdlist</i>    | An array of SCSI command description strings.                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <i>sensep</i>     | A pointer to a <a href="#">scsi_extended_sense(9S)</a> structure.                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <i>asc_list</i>   | A pointer to a array of asc and ascq message list. The list must be terminated with -1 asc value.                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <i>decode_fru</i> | This is a function pointer that will be called after the entire sense information has been decoded. The parameters will be the <a href="#">scsi_device</a> structure to identify the device. Second argument will be a pointer to a buffer of length specified by third argument. The fourth argument will be the FRU byte. <i>decode_fru</i> might be NULL if no special decoding is required. <i>decode_fru</i> is expected to return pointer to a char string if decoding possible and NULL if no decoding is possible. |

**Description** This function is very similar to [scsi\\_errmsg\(9F\)](#) but allows decoding of vendor-unique ASC/ASCQ and FRU information.

The `scsi_vu_errmsg()` function interprets the request sense information in the *sensep* pointer and generates a standard message that is displayed using [scsi\\_log\(9F\)](#). It first searches the list array for a matching vendor unique code if supplied. If it does not find one in the list then the standard list is searched. The first line of the message is always a CE\_WARN, with the continuation lines being CE\_CONT. *sensep* may be NULL, in which case no sense key or vendor information is displayed.



The driver should make the determination as to when to call this function based on the severity of the failure and the severity level that the driver wants to report.

The `scsi_device(9S)` structure denoted by `devp` supplies the identification of the device that requested the display. `severity` selects which string is used in the Error Level: reporting, according to the table below:

| Severity Value:    | String:     |
|--------------------|-------------|
| SCSI_ERR_ALL       | All         |
| SCSI_ERR_UNKNOWN   | Unknown     |
| SCSI_ERR_INFO      | Information |
| SCSI_ERR_RECOVERED | Recovered   |
| SCSI_ERR_RETRYABLE | Retryable   |
| SCSI_ERR_FATAL     | Fatal       |

`blkno` is the block number of the original request that generated the error. `err_blkno` is the block number where the error occurred. `cmdlist` is a mapping table for translating the SCSI command code in `pktp` to the actual command string.

The `cmdlist` is described in the structure below:

```
struct scsi_key_strings {
    int key;
    char *message;
};
```

For a basic SCSI disk, the following list is appropriate:

```
static struct scsi_key_strings scsi_cmds[] = {
    0x00, "test unit ready",
    0x01, "rezero/rewind",
    0x03, "request sense",
    0x04, "format",
    0x07, "reassign",
    0x08, "read",
    0x0a, "write",
    0x0b, "seek",
    0x12, "inquiry",
    0x15, "mode select",
    0x16, "reserve",
    0x17, "release",
    0x18, "copy",
    0x1a, "mode sense",
    0x1b, "start/stop",
    0x1e, "door lock",
    0x28, "read(10)",
    0x2a, "write(10)",
    0x2f, "verify",
    0x37, "read defect data",
```

```
        0x3b, "write buffer",
        -1, NULL
    };
```

**Context** The `scsi_vu_errmsg()` function may be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 Using `scsi_vu_errmsg()`

```
struct scsi_asc_key_strings cd_slist[] = {
    0x81, 0, "Logical Unit is inaccessible",
    -1, 0, NULL,
};

scsi_vu_errmsg(devp, pkt, "sd",
               SCSI_ERR_INFO, bp->b_blkno, err_blkno,
               sd_cmds, rqsense, cd_list,
               my_decode_fru);
```

This generates the following console warning:

```
WARNING: /sbus@1,f8000000/esp@0,8000000/sd@1,0 (sd1):
  Error for Command: read          Error Level: Informational
  Requested Block: 23936          Error Block: 23936
  Vendor: XYZ                      Serial Number: 123456
  Sense Key: Unit Attention
  ASC: 0x81 (Logical Unit is inaccessible), ASCQ: 0x0
  FRU: 0x11 (replace LUN 1, located in slot 1)
```

**See Also** [cmn\\_err\(9F\)](#), [scsi\\_errmsg\(9F\)](#), [scsi\\_log\(9F\)](#), [scsi\\_errmsg\(9F\)](#),  
[scsi\\_asc\\_key\\_strings\(9S\)](#), [scsi\\_device\(9S\)](#), [scsi\\_extended\\_sense\(9S\)](#), [scsi\\_pkt\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** semaphore, sema\_init, sema\_destroy, sema\_p, sema\_p\_sig, sema\_v, sema\_try – semaphore functions

**Synopsis** #include <sys/ksynch.h>

```
void sema_init(ksema_t *sp, uint_t val, char *name, ksema_type_t type,
              void *arg);

void sema_destroy(ksema_t *sp);

void sema_p(ksema_t *sp);

void sema_v(ksema_t *sp);

int sema_p_sig(ksema_t *sp);

int sema_try(ksema_t *sp);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

|             |                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------|
| <i>sp</i>   | A pointer to a semaphore, type <code>ksema_t</code> .                                                                         |
| <i>val</i>  | Initial value for semaphore.                                                                                                  |
| <i>name</i> | Descriptive string. This is obsolete and should be NULL. (Non-NULL strings are legal, but they are a waste of kernel memory.) |
| <i>type</i> | Variant type of the semaphore. Currently, only <code>SEMA_DRIVER</code> is supported.                                         |
| <i>arg</i>  | Type-specific argument; should be NULL.                                                                                       |

**Description** These functions implement counting semaphores as described by Dijkstra. A semaphore has a value which is atomically decremented by `sema_p()` and atomically incremented by `sema_v()`. The value must always be greater than or equal to zero. If `sema_p()` is called and the value is zero, the calling thread is blocked until another thread performs a `sema_v()` operation on the semaphore.

Semaphores are initialized by calling `sema_init()`. The argument, `val`, gives the initial value for the semaphore. The semaphore storage is provided by the caller but more may be dynamically allocated, if necessary, by `sema_init()`. For this reason, `sema_destroy()` should be called before deallocating the storage containing the semaphore.

The `sema_p_sig()` function decrements the semaphore, as does `sema_p()`. However, if the semaphore value is zero, `sema_p_sig()` will return without decrementing the value if a signal (that is, from `kill(2)`) is pending for the thread.

The `sema_try()` function will decrement the semaphore value only if it is greater than zero, and will not block.

**Return Values** 0 `sema_try()` could not decrement the semaphore value because it was zero.

- 1 `sema_p_sig()` was not able to decrement the semaphore value and detected a pending signal.

**Context** These functions can be called from user, interrupt, or kernel context, except for `sema_init()` and `sema_destroy()`, which can be called from user or kernel context only. None of these functions can be called from a high-level interrupt context. In most cases, `sema_v()` and `sema_p()` should not be called from any interrupt context.

If `sema_p()` is used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the `sema_v()` becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the `sema_v()` may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on semaphores or condition variables in an interrupt context.

**See Also** [kill\(2\)](#), [condvar\(9F\)](#), [mutex\(9F\)](#)

*Writing Device Drivers*

**Name** `sof_bypass` – stop receiving callbacks

**Synopsis** `#include <sys/socketfilter.h>`

```
void sof_bypass(sof_handle_t hdl);
```

**Parameters** *hdl* per-socket filter handle

**Description** The `sof_bypass()` function informs the framework that the filter instance no longer wants to receive callbacks. No guarantee is made that there will not be any active callbacks when `sof_bypass()` returns, only that no new callbacks will be triggered.

The only callback that will triggered for a filter that has called `sof_bypass()` is [sofop\\_detach\(9E\)](#).

A typical use of `sof_bypass()` is for a filter that only needs to be active during a certain period of a socket's lifetime. Calling `sof_bypass()` will minimize the performance impact of the inactive filter.

**Context** The `sof_bypass()` function can be called from any context

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [attributes\(5\)](#), [sofop\\_detach\(9E\)](#), [sof\\_ops\(9S\)](#)

**Name** `sof_flowctrl_data_in`, `sof_flowctrl_data_out` – assert flow control

**Synopsis** `#include <sys/socketfilter.h>`

```
void sof_flowctrl_data_in(sof_handle_t hdl, boolean_t on);
```

```
void sof_flowctrl_data_out(sof_handle_t hdl, boolean_t on);
```

**Parameters** *hdl* per-socket filter handle

*on* set to `B_TRUE` or `B_FALSE` to assert or clear flow control, respectively

**Description** The `sof_flowctrl_data_in()` and `sof_flowctrl_data_out()` functions are used to enforce flow control on the transmit and receive paths, respectively. Unlike send side flow control, which is enforced by the socket layer, the behavior on the receive side is dependent on the transport protocol. Asserting flow control for a protocol that supports it at the protocol level, such as TCP, will cause the peer to stop transmitting data. Other protocols, like UDP, will start dropping packets as they are received.

Neither `sof_flowctrl_data_in()` nor `sof_flowctrl_data_out()` can be used to clear a flow control condition that was not asserted by the filter. For example, a socket may be flow controlled as a result of an application not consuming data, and clearing the filter's flow control condition will not change the socket's state.

Flow control may not take effect right away, and during this transient period both the socket and protocol may generate data that will call the data entry points. It is also possible that another filter inject data while the socket is flow controlled.

**Context** The `sof_flowctrl_data_in()` and `sof_flowctrl_data_out()` functions can be called from any context

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [attributes\(5\)](#), [sofop\\_data\\_in\(9E\)](#), [sof\\_inject\\_data\\_in\(9F\)](#)

**Name** `sof_inject_data_in`, `sof_inject_data_out` – inject data

**Synopsis** `#include <sys/socketfilter.h>`

```
int sof_inject_data_in(sof_handle_t hdl, mblk_t *mp, boolean_t *stop);
int sof_inject_data_out(sof_handle_t hdl, mblk_t *mp, boolean_t *stop);
```

**Parameters**

- hdl* per-socket filter handle
- mp* pointer to a [msgb\(9S\)](#) structure containing data to inject. Multiple `mblk` structures can be chained together via *b\_cont*. The first `mblk` can contain address and/or control information, which can be constructed from a message header structure using [sof\\_mblk\\_create\(9F\)](#).
- stop* result parameter indicating whether the socket is flow controlled. The filter should stop injecting any more data if *stop* is `B_TRUE`, and only resume once the `SOF_EV_FLOWCTRL_CLR` ([sofop\\_notify\(9E\)](#)) event is observed.

**Description** The `sof_inject_data_in()` function is used to enqueue data in socket's receive buffer, which can then be consumed using [recv\(3SOCKET\)](#).

The `sof_inject_data_out()` function is used to transmit data from the socket to its peer.

The framework takes ownership of the message chain on both successful and failed inject attempts.

The filter can check flow control state by calling `sof_inject_data_in()` and `sof_inject_data_out()` with a `NULL` message, which will update *stop*.

**Return Values** The `sof_inject_data_in()` and `sof_inject_data_out()` functions returns 0 if data was successfully injected, and a non-zero error value, as described in [Intro\(2\)](#), on failure.

**Context** The `sof_inject_data_in()` and `sof_inject_data_out()` functions can be called from user and kernel context

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [Intro\(2\)](#), [recv\(3SOCKET\)](#), [attributes\(5\)](#), [sofop\\_notify\(9E\)](#), [sof\\_flowctrl\\_data\\_in\(9F\)](#), [sof\\_mblk\\_create\(9F\)](#), [msgb\(9S\)](#), [sof\\_ops\(9S\)](#)

**Name** sof\_mblk\_msghdr, sof\_mblk\_create – access and create control messages

**Synopsis** #include <sys/socketfilter.h>

```
boolean_t sof_mblk_msghdr(sof_handle_t hdl, mblk_t *mp,
    struct msghdr *msg, int flags);

mbblk_t *sof_mblk_create(sof_handle_t hdl, struct msghdr *msg,
    int flags);
```

**Parameters**

- hdl* per-socket filter handle
- msg* pointer to a message header structure, as defined in `socket.h(3HEAD)`
- mp* pointer to a `msgb(9S)` structure
- flags* either 0 or `SOF_MBLK_CMSG`

**Description** The `sof_mblk_msghdr()` function is used to obtain message header information from a message block. On success, *msg\_name* will point to a location within the message block containing address information, and *msg\_namelen* will reflect the size of the address. Any changes made to the content pointed to by *msg\_name* will be made to the underlying message block. The content pointed to by *msg\_control* should be treated as opaque data unless `SOF_MBLK_CMSG` is specified. If `SOF_MBLK_CMSG` is specified, then the caller must also provide a buffer, pointed to by *msg\_control*, where ancillary data will be stored. *msg\_controllen* is updated on return to reflect the amount of ancillary data. If there is not enough room in the provided buffer to copy out all control data, then none of it will be copied, and `MSG_CTRUNC` will be set on *msg\_flags*.

The `sof_mblk_create()` function tries to allocate a message block that is large enough to contain the address and control data provided in the message header, and copies the information into the newly created message block. If `SOF_MBLK_CMSG` is not specified, then *msg\_control* should either be `NULL`, or point to the opaque data returned in *msg\_control* by a previous call to `sof_mblk_msghdr()`. If `SOF_MBLK_CMSG` is specified, then *msg\_control* points to a buffer containing `cmsghdr` structures.

**Return Values** The `sof_mblk_msghdr()` function returns `B_TRUE` if a message header was found, and `B_FALSE` otherwise.

If successful, `sof_mblk_create()` returns a pointer to the allocated message block. On failure, `sof_mblk_create()` returns a null pointer.

**Examples** EXAMPLE 1 Access and modify address information

```
mbblk_t *xx_data_in(sof_handle_t hdl, void *cookie, mblk_t *mp)
{
    xxpriv_t *priv = cookie;
    struct msghdr msg;
```



**EXAMPLE 1** Access and modify address information *(Continued)*

```

/*
 * Extract name information. The filter is not interested in
 * control data.
 */
if (sof_mblk_msghdr(hdl, mp, &msg, 0)) {
    (void) strncpy(msg.msg_name, priv->xx_addr,
        msg.msg_namelen);
}

return (mp);
}

```

**EXAMPLE 2** Update ancillary data

```

mblk_t *xx_data_in(sof_handle_t hdl, void *cookie, mblk_t *mp)
{
    xxpriv_t *priv = cookie;
    struct msghdr msg;
    char buf[XX_BUFLEN];

    msg.msg_control = buf;
    msg.msg_controllen = sizeof (buf);

    /* Process any ancillary data. */
    if (sof_mblk_msghdr(hdl, mp, &msg, SOF_MBLK_CMSG)) {
        mblk_t *nmp;

        xx_proc_ctrl(&msg);

        /* msghdr has cmsghdr control data. */
        nmp = sof_mblk_create(hdl, &msg, SOF_MBLK_CMSG);
        if (nmp != NULL) {
            nmp->b_cont = mp->b_cont;
            freeb(mp);
            mp = nmp;
        }
    }

    return (mp);
}

```

**Context** The `sof_mblk_msghdr()` and `sof_mblk_create()` functions can be called from any context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [socket.h\(3HEAD\)](#), [attributes\(5\)](#), [sofop\\_data\\_in\(9E\)](#), [sof\\_inject\\_data\\_in\(9F\)](#), [msgb\(9S\)](#)

**Name** `sof_newconn_ready`, `sof_newconn_close` – accept or discard a deferred connection

**Synopsis** `#include <sys/socketfilter.h>`

```
void sof_newconn_ready(sof_handle_t hdl);
void sof_newconn_close(sof_handle_t hdl);
```

**Parameters** *hdl* per-socket filter handle

**Description** The `sof_newconn_ready()` function informs the framework that the deferred connection should be made available to the application via [accept\(3SOCKET\)](#). In the presence of multiple filters, the connection will only be made available once all filters have called `sof_newconn_ready()`.

The `sof_newconn_close()` function causes the connection to be closed immediately. The normal close sequence will be executed, and filters will therefore observe a `SOF_EV_CLOSING` event (see [sofop\\_notify\(9E\)](#)) followed by a call to [sofop\\_detach\(9E\)](#).

Calling either `sof_newconn_ready()` or `sof_newconn_close()` on a connection that is not deferred has no effect.

**Context** The `sof_newconn_ready()` and `sof_newconn_close()` functions can be called from any context

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [accept\(3SOCKET\)](#), [attributes\(5\)](#), [sofop\\_attach\\_passive\(9E\)](#), [sofop\\_detach\(9E\)](#), [sofop\\_notify\(9E\)](#),

**Name** `sof_register`, `sof_unregister` – register or unregister a socket filter module

**Synopsis** `#include <sys/socketfilter.h>`

```
int sof_register(int version, const char *name,
                const sof_ops_t *ops, int flags);

int sof_unregister(const char *name);
```

**Parameters**

- version* the socket filter version; should always be set to `SOF_VERSION`.
- name* a name that uniquely identifies this filter, and must match the name (not module name) used when configuring the filter via [soconfig\(1M\)](#).
- ops* a pointer to an instance of the [sof\\_ops\(9S\)](#) structure.
- flags* unused; should always be 0.

**Description** The `sof_register()` function, typically called from a module's `_init(9E)` entry point, is used to register a module with the socket filter framework.

The `sof_unregister()` function is used to unregister a module from the socket filter framework. If `sof_unregister()` is called from a module's `_fini(9E)` entry point, then the module must return `EBUSY` if `sof_unregister()` fails.

**Return Values** The `sof_register()` and `sof_unregister()` functions return 0 on success, and a non-zero error, as described in [Intro\(2\)](#), on failure.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Availability        | system/kernel   |
| Interface Stability | Uncommitted     |

**See Also** [soconfig\(1M\)](#), [Intro\(2\)](#), [attributes\(5\)](#), [\\_fini\(9E\)](#), [\\_init\(9E\)](#), [sof\\_ops\(9S\)](#)

**Name** sprintf, snprintf – format characters in memory

**Synopsis** #include <sys/ddi.h>

```
char *sprintf(char *buf, const char *fmt...);
size_t snprintf(char *buf, size_t n, const char *fmt...);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *buf* Pointer to a character string.  
*fmt* Pointer to a character string.

**Description** The `sprintf()` function builds a string in *buf* under the control of the format *fmt*. The format is a character string with either plain characters, which are simply copied into *buf*, or conversion specifications, each of which converts zero or more arguments, again copied into *buf*. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user's responsibility to ensure that enough storage is available for *buf*.

The `snprintf()` function is identical to `sprintf()` with the addition of the argument *n*, which specifies the size of the buffer referred to by *buf*. The buffer is always terminated with the null byte.

**Conversion Specifications** Each conversion specification is introduced by the % character, after which the following appear in sequence:

An optional value specifying a minimum field width for numeric conversion. The converted value will be right-justified and, if it has fewer characters than the minimum, is padded with leading spaces unless the field width is an octal value, then it is padded with leading zeroes.

An optional `l` (`ll`) specifying that a following `d`, `D`, `o`, `O`, `x`, `X`, or `u` conversion character applies to a long (long long) integer argument. An `l` (`ll`) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

`d,D,o,O,x,X,u` The integer argument is converted to signed decimal (`d`, `D`), unsigned octal (`o`, `O`), unsigned hexadecimal (`x`, `X`) or unsigned decimal (`u`), respectively, and copied. The letters `abcdef` are used for `x` conversion. The letters `ABCDEF` are used for `X` conversion.

`c` The character value of argument is copied.

`b` This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form `<base>[<arg> . . . ]`. The base supplies the conversion base for the first argument as a binary value; `\10` gives

octal, \20 gives hexadecimal. Each subsequent *<arg>* is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.

A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by < and >. Thus, the following function call would generate `reg=3<BitTwo, BitOne>\n` in *buf*.

```
printf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```

- p The argument is taken to be a pointer; the value of the pointer is displayed in unsigned hexadecimal. The display format is equivalent to %lx. To avoid lint warnings, cast pointers to type void \* when using the %p format specifier.
- s The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL, the string `<null string>` is used in its place.
- % Copy a %; no argument is converted.

**Return Values** The `printf()` function returns its first argument, *buf*.

The `snprintf()` function returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. If the value of *n* is less than or equal to 0 on a call to `snprintf()`, the function simply returns the number of characters formatted.

**Context** The `printf()` and `snprintf()` functions and can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)

**Name** stoi, numtos – convert between an integer and a decimal string

**Synopsis** #include <sys/ddi.h>

```
int stoi(char **str);  
void numtos(unsigned long num, char *s);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *str* Pointer to a character string to be converted.  
*num* Decimal number to be converted to a character string.  
*s* Character buffer to hold converted decimal number.

### Description

*stoi()* The *stoi()* function returns the integer value of a string of decimal numeric characters beginning at *\*\*str*. No overflow checking is done. *\*str* is updated to point at the last character examined.

*numtos()* The *numtos()* function converts a long into a null-terminated character string. No bounds checking is done. The caller must ensure there is enough space to hold the result.

**Return Values** The *stoi()* function returns the integer value of the string *str*.

**Context** The *stoi()* function can be called from user, interrupt, or kernel context.

**See Also** [Writing Device Drivers](#)

**Notes** The *stoi()* function handles only positive integers; it does not handle leading minus signs.

**Name** string, strcasecmp, strncasecmp, strncat, strlcat, strchr, strrchr, strcmp, strncmp, strcpy, strncpy, strlcpy, strfree, strspn, strdup, ddi\_strdup, strlen, strnlen, strstr, strcasestr – string operations

**Synopsis** #include <sys/ddi.h>

```
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
char *strncat(char *s1, const char *s2, size_t n);
size_t strlcat(char *dst, const char *src, size_t dstsize);
char *strchr(const char *str, int chr);
char *strrchr(const char *str, int chr);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strcpy(char *dst, const char *src);
char *strncpy(char *dst, const char *src, size_t n);
size_t strlcpy(char *dst, const char *src, size_t dstsize);
void strfree(char *s);
size_t strspn(const char *s1, const char *s2);
char *strdup(const char *s1);
char *ddi_strdup(const char *s1, int flag);
size_t strlen(const char *s);
size_t strnlen(const char *s, size_t n);
char *strstr(const char *s1, const char *s2);
char *strcasestr(const char *s1, const char *s2);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Description** The arguments *s*, *s1*, and *s2* point to strings (arrays of characters terminated by a null character). The `strcat()`, `strncat()`, `strlcat()`, `strcpy()`, `strncpy()`, `strlcpy()`, and `strfree()` functions all alter their first argument. Additionally, the `strcpy()` function does not check for overflow of the array.

`strcasecmp()`,  
`strncasecmp()` The `strcasecmp()` and `strncasecmp()` functions are case-insensitive versions of `strcmp()` and `strncmp()` respectively, described below. They assume the ASCII character set and ignore differences in case when comparing lower and upper case characters.



`strncat()`, `strlcat()` The `strncat()` function appends at most  $n$  characters of string  $s2$ , including the terminating null character, to the end of string  $s1$ . It returns a pointer to the null-terminated result. The initial character of  $s2$  overrides the null character at the end of  $s1$ . If copying takes place between objects that overlap, the behavior of `strncat()` and `strlcat()` is undefined.

The `strlcat()` function appends at most  $(dstsize - strlen(dst) - 1)$  characters of  $src$  to  $dst$  ( $dstsize$  being the size of the string buffer  $dst$ ). If the string pointed to by  $dst$  contains a null-terminated string that fits into  $dstsize$  bytes when `strlcat()` is called, the string pointed to by  $dst$  will be a null-terminated string that fits in  $dstsize$  bytes (including the terminating null character) when it completes, and the initial character of  $src$  will override the null character at the end of  $dst$ . If the string pointed to by  $dst$  is longer than  $dstsize$  bytes when `strlcat()` is called, the string pointed to by  $dst$  will not be changed. The function returns  $\min\{dstsize, strlen(dst)\} + strlen(src)$ . Buffer overflow can be checked as follows:

```
if (strlcat(dst, src, dstsize) >= dstsize)
    return -1;
```

`strchr()`, `strrchr()` The `strchr()` function returns a pointer to the first occurrence of  $c$  (converted to a `char`) in string  $s$ , or a null pointer if  $c$  does not occur in the string. The `strrchr()` function returns a pointer to the last occurrence of  $c$ . The null character terminating a string is considered to be part of the string.

`strcmp()`, `strncmp()` The `strcmp()` function compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by  $s1$  is greater than, equal to, or less than the string pointed to by  $s2$  respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared. The `strncmp()` function makes the same comparison but looks at a maximum of  $n$  bytes. Bytes following a null byte are not compared.

`strcpy()`, `strncpy()`, `strlcpy()` The `strcpy()` function copies string  $s2$  to  $s1$ , including the terminating null character, stopping after the null character has been copied. The `strncpy()` function copies exactly  $n$  bytes, truncating  $s2$  or adding null characters to  $s1$  if necessary. The result will not be null-terminated if the length of  $s2$  is  $n$  or more. Each function returns  $s1$ . If copying takes place between objects that overlap, the behavior of `strcpy()`, `strncpy()`, and `strlcpy()` is undefined.

The `strlcpy()` function copies at most  $dstsize - 1$  characters ( $dstsize$  being the size of the string buffer  $dst$ ) from  $src$  to  $dst$ , truncating  $src$  if necessary. The result is always null-terminated. The function returns  $strlen(src)$ . Buffer overflow can be checked as follows:

```
if (strlcpy(dst, src, dstsize) >= dstsize)
    return -1;
```

`strfree()` The `strfree()` function frees the memory associated with the string pointed to by `s`. This memory pointed to by `s` must be of size `strlen(s)+1`, and must have been allocated (either directly or indirectly) by `kmem_alloc(9F)` or `kmem_zalloc(9F)`.

`strspn()` The `strspn()` function returns the length of the initial segment of string `s1` that consists entirely of characters from string `s2`.

`strdup()`,  
`ddi_strdup()` The `ddi_strdup()` function returns a pointer to a new string that is a duplicate of the string pointed to by `s1`. The returned pointer can be passed to `strfree()` or `kmem_free(9F)`. The space for the new string is obtained using `kmem_alloc()`. `flag` can be either `KM_SLEEP` or `KM_NOSLEEP`, and determines whether the caller can sleep for memory. `KM_SLEEP` allocations may sleep but are guaranteed to succeed. `KM_NOSLEEP` allocations are guaranteed not to sleep but may fail (return `NULL`) if no memory is currently available.

The `strdup()` function behaves the same as the `ddi_strdup()` when called with the `KM_SLEEP` flag. This means that `strdup()` can sleep until memory is available and will always succeed.

`strlen()`, `strnlen()` The `strlen()` function returns the number of bytes in `s`, not including the terminating null character.

The `strnlen()` function returns the smaller of `n` or the number of bytes in `s`, not including the terminating null character. The `strnlen()` function never examines more than `n` bytes of the string pointed to by `s`.

`strstr()` The `strcasestr()` function locates the first occurrence of the string `s2` (excluding the terminating null character) in string `s1` and returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length (that is, the string ""), the function returns `s1`.

`strcasestr()` The `strcasestr()` function is a case-insensitive version of `strstr()`. It assumes the ASCII character set and ignores differences in case when comparing lower and upper case characters.

**Context** The `strdup()` and `ddi_strdup()` functions can be called from user or kernel context.

The `ddi_strdup()` function can be called from interrupt context only if the `KM_NOSLEEP` flag is set.

All the other string manipulation functions can be called from user, interrupt, or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

| ATTRIBUTE TYPE      | ATTRIBUTE VALUE |
|---------------------|-----------------|
| Interface Stability | Committed       |

**See Also** [string\(3C\)](#), [attributes\(5\)](#), [bcopy\(9F\)](#), [ddi\\_copyin\(9F\)](#), [kmem\\_alloc\(9F\)](#)

*Writing Device Drivers*

**Notes** If copying takes place between objects that overlap, the behavior of `strlcat()`, `strncat()`, `strcpy()`, `strncpy()`, and `strncpy()` is undefined.

**Name** strlog – submit messages to the log driver

**Synopsis**

```
#include <sys/stream.h>
#include <sys/strlog.h>
#include <sys/log.h>
```

```
int strlog(short mid, short sid, char level,
           unsigned short flags, char *fmt, ...);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters**

*mid* Identification number of the module or driver submitting the message (in the case of a module, its `mi_idnum` value from [module\\_info\(9S\)](#)).

*sid* Identification number for a particular minor device.

*level* Tracing level for selective screening of low priority messages. Larger values imply less important information.

*flags* Valid flag values are:

|                         |                                               |
|-------------------------|-----------------------------------------------|
| <code>SL_ERROR</code>   | Message is for error logger.                  |
| <code>SL_TRACE</code>   | Message is for trace.                         |
| <code>SL_NOTIFY</code>  | Mail copy of message to system administrator. |
| <code>SL_CONSOLE</code> | Log message to console.                       |
| <code>SL_FATAL</code>   | Error is fatal.                               |
| <code>SL_WARN</code>    | Error is a warning.                           |
| <code>SL_NOTE</code>    | Error is a notice.                            |

*fmt* [printf\(3C\)](#) style format string. `%e`, `%g`, and `%G` formats are not allowed but `%s` is supported.

**Description** The `strlog()` function expands the [printf\(3C\)](#) style format string passed to it, that is, the conversion specifiers are replaced by the actual argument values in the format string. The 32-bit representations of the arguments (up to `NLOGARGS`) follow the string starting at the next 32-bit boundary following the string. Note that the 64-bit argument will be truncated to 32-bits here but will be fully represented in the string.

The messages can be retrieved with the [getmsg\(2\)](#) system call. The *flags* argument specifies the type of the message and where it is to be sent. [strace\(1M\)](#) receives messages from the log driver and sends them to the standard output. [strerr\(1M\)](#) receives error messages from the log driver and appends them to a file called `/var/adm/streams/error.mm-dd`, where *mm-dd* identifies the date of the error message.

**Return Values** The `strlog()` function returns 0 if it fails to submit the message to the `log(7D)` driver and 1 otherwise.

**Context** The `strlog()` function can be called from user, interrupt, or kernel context.

**Files** `/var/adm/streams/error.mm-dd` Error messages dated `mm-dd` appended by `strerr(1M)` from the `log` driver

**See Also** `strace(1M)`, `strerr(1M)`, `getmsg(2)`, `log(7D)`, `module_info(9S)`

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** strqget – get information about a queue or band of the queue

**Synopsis** #include <sys/stream.h>

```
int strqget(queue_t *q, qfields_t what, unsigned char pri, void *valp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *q* Pointer to the queue.

*what* Field of the queue structure for (or the specified priority band) to return information about. Valid values are one of:

QHIWAT High water mark.

QLOWAT Low water mark.

QMAXPSZ Largest packet accepted.

QMINPSZ Smallest packet accepted.

QCOUNT Approximate size (in bytes) of data.

QFIRST First message.

QLAST Last message.

QFLAG Status.

*pri* Priority band of interest.

*valp* The address of where to store the value of the requested field.

**Description** The `strqget()` function gives drivers and modules a way to get information about a queue or a particular band of a queue without directly accessing STREAMS data structures, thus insulating them from changes in the implementation of these data structures from release to release.

**Return Values** On success, 0 is returned and the value of the requested field is stored in the location pointed to by *valp*. An error number is returned on failure.

**Context** The `strqget()` function can be called from user, interrupt, or kernel context.

**See Also** [strqset\(9F\)](#), [queue\(9S\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** strqset – change information about a queue or band of the queue
- Synopsis** #include <sys/stream.h>
- ```
int strqset(queue_t *q, qfields_t what, unsigned char pri, intptr_t val);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters**
- |             |  |
|-------------|--|
| <i>q</i>    | Pointer to the queue.  |
| <i>what</i> | Field of the queue structure (or the specified priority band) to return information about. Valid values are one of: <ul style="list-style-type: none"> <li>QHIWAT High water mark.</li> <li>QLOWAT Low water mark.</li> <li>QMAXPSZ Largest packet accepted.</li> <li>QMINPSZ Smallest packet accepted.</li> </ul> |
| <i>pri</i>  | Priority band of interest.   |
| <i>val</i>  | The value for the field to be changed.   |
- Description** The `strqset()` function gives drivers and modules a way to change information about a queue or a particular band of a queue without directly accessing STREAMS data structures.
- Return Values** On success, 0 is returned. EINVAL is returned if an undefined attribute is specified.
- Context** The `strqset()` function can be called from user, interrupt, or kernel context.
- See Also** [strqget\(9F\)](#), [queue\(9S\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*
- Notes** When lowering existing values, set QMINPSZ before setting QMAXPSZ; when raising existing values, set QMAXPSZ before setting QMINPSZ.

**Name** STRUCT\_DECL, SIZEOF\_PTR, SIZEOF\_STRUCT, STRUCT\_BUF, STRUCT\_FADDR, STRUCT\_FGET, STRUCT\_FGETP, STRUCT\_FSET, STRUCT\_FSETP, STRUCT\_HANDLE, STRUCT\_INIT, STRUCT\_SIZE, STRUCT\_SET\_HANDLE – 32-bit application data access macros

**Synopsis**

```
#include <sys/ddi.h>
#include <sys/sunddi.h>

STRUCT_DECL(structname, handle);
STRUCT_HANDLE(structname, handle);

void STRUCT_INIT(handle, model_t umodel);
void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr);
STRUCT_FGET(handle, field);
STRUCT_FGETP(handle, field);
STRUCT_FSET(handle, field, val);
STRUCT_FSETP(handle, field, val);
<typeof field> *STRUCT_FADDR(handle, field);
struct structname *STRUCT_BUF(handle);
size_t SIZEOF_STRUCT(structname, umodel);
size_t SIZEOF_PTR(umodel);
size_t STRUCT_SIZE(handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** The macros take the following parameters:

<i>structname</i>	The structure name that appears <i>after</i> the C keyword struct of the native form.
<i>umodel</i>	A bit field that contains either the ILP32 model bit (DATAMODEL_ILP32), or the LP64 model bit (DATAMODEL_LP64). In an <a href="#">ioctl(9E)</a> , these bits are present in the flag parameter. In a <a href="#">devmap(9E)</a> , the bits are present in the model parameter <a href="#">mmap(9E)</a> . The <a href="#">ddi_mmap_get_model(9F)</a> can be called to get the data model of the current thread.
<i>handle</i>	The variable name used to refer to a particular instance of a structure which is handled by these macros.
<i>field</i>	The field name within the structure that can contain substructures. If the structures contain substructures, unions, or arrays, the <i>field</i> can be whatever complex expression would naturally follow the first . or ->.



<b>Description</b>	The above macros allow a device driver to access data consumed from a 32-bit application regardless whether the driver was compiled to the ILP32 or LP64 data model. These macros effectively hide the difference between the data model of the user application and the driver.	
	The macros can be broken up into two main categories described in the following sections.	
Declaration and Initialization Macros	The macros <code>STRUCT_DECL()</code> and <code>STRUCT_HANDLE()</code> declare structure handles on the stack, whereas the macros <code>STRUCT_INIT()</code> and <code>STRUCT_SET_HANDLE()</code> initialize the structure handles to point to an instance of the native form structure.	
	The macros <code>STRUCT_HANDLE()</code> and <code>STRUCT_SET_HANDLE()</code> are used to declare and initialize a structure handle to an existing data structure, for example, <code>ioctl</code> s within a <code>STREAMS</code> module.	
	The macros <code>STRUCT_DECL()</code> and <code>STRUCT_INIT()</code> , on the other hand, are used in modules which declare and initialize a structure handle to a data structure allocated by <code>STRUCT_DECL()</code> , that is, any standard character or block device driver <code>ioctl(9E)</code> routine that needs to copy in data from a user-mode program.	
	<pre>STRUCT_DECL(structname, handle)</pre> <p>Declares a structure handle for a struct and allocates an instance of its native form on the stack. It is assumed that the native form is larger than or equal to the ILP32 form. <i>handle</i> is a variable name and is declared as a variable by this macro.</p>	
	<pre>void STRUCT_INIT(handle, model_t umodel)</pre> <p>Initializes <i>handle</i> to point to the instance allocated by <code>STRUCT_DECL()</code>. It also sets data model for <i>handle</i> to <i>umodel</i> and it must be called before any access is made through the macros that operate on these structures. When used in an <code>ioctl(9E)</code> routine, <i>umodel</i> is the flag parameter. In a <code>devmap(9E)</code> routine, <i>umodel</i> is the model parameter. In a <code>mmap(9E)</code> routine, <i>umodel</i> is the return value of <code>ddi_mmap_get_model(9F)</code>. This macro is intended only for handles created with <code>STRUCT_DECL()</code>.</p>	
	<pre>STRUCT_HANDLE(structname, handle)</pre> <p>Declares a structure handle <i>handle</i> but, unlike <code>STRUCT_DECL()</code>, it does not allocate an instance of “struct”.</p>	
	<pre>void STRUCT_SET_HANDLE(handle, model_t umodel, void *addr)</pre> <p>Initializes <i>handle</i> to point to the native form instance at <i>addr</i>. It also sets the data model for <i>handle</i> to <i>umodel</i>. This is intended for handles created with <code>STRUCT_HANDLE()</code>. Fields cannot be referenced via the <i>handle</i> until this macro has been invoked. Typically, <i>addr</i> is the address of the native form structure containing the user-mode programs data. When used in an <code>ioctl(9E)</code>, <i>umodel</i> is the flag parameter. In a <code>devmap(9E)</code> routine, <i>umodel</i> is the model parameter. In an <code>mmap(9E)</code> routine, <i>umodel</i> is the return value of <code>ddi_mmap_get_model(9F)</code>.</p>	
Operation Macros	<code>size_t STRUCT_SIZE(handle)</code>	Returns size of the structure referred to by <i>handle</i> , depending on the data model associated with <i>handle</i> . If the data model stored by <code>STRUCT_INIT()</code> or

		STRUCT_SET_HANDLE() is DATAMODEL_ILP32, the size of the ILP32 form is returned. Otherwise, the size of the native form is returned.
	STRUCT_FGET(handle, field)	Returns the contents of <i>field</i> in the structure described by <i>handle</i> according to the data model associated with <i>handle</i> .
	STRUCT_FGETP(handle, field)	This is the same as STRUCT_FGET() except that the <i>field</i> in question is a pointer of some kind. This macro casts <code>caddr32_t</code> to a <code>(void *)</code> when it is accessed. Failure to use this macro for a pointer leads to compiler warnings or failures.
	STRUCT_FSET(handle, field, val)	Assigns <i>val</i> to the (non-pointer) in the structure described by <i>handle</i> . It should not be used within another expression, but only as a statement.
	STRUCT_FSETP(handle, field, val)	This is the equivalent of STRUCT_FGETP() for STRUCT_FGET(), with the same exceptions. Like STRUCT_FSET, STRUCT_FSETP should not be used within another expression, but only as a statement.
	struct structname *STRUCT_BUF(handle)	Returns a pointer to the native mode instance of the structure described by <i>handle</i> .
Macros Not Using Handles	size_t SIZEOF_STRUCT(structname, umodel)	Returns size of <i>structname</i> based on <i>umodel</i> .
	size_t SIZEOF_PTR(umodel)	Returns the size of a pointer based on <i>umodel</i> .

### Examples EXAMPLE 1 Copying a Structure

The following example uses an [ioctl\(9E\)](#) on a regular character device that copies a data structure that looks like this into the kernel:

```
struct opdata {
    size_t size;
    uint_t flag;
};
```

### EXAMPLE 2 Defining a Structure

This data structure definition describes what the [ioctl\(9E\)](#) would look like in a 32-bit application using fixed width types.

**EXAMPLE 2** Defining a Structure *(Continued)*

```
#if defined(_MULTI_DATAMODEL)
struct opdata32 {
    size32_t    size;
    uint32_t    flag;
};
#endif
```

**EXAMPLE 3** Using STRUCT\_DECL() and STRUCT\_INIT()

Note: This example uses the STRUCT\_DECL() and STRUCT\_INIT() macros to declare and initialize the structure handle.

```
int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p)
{
    STRUCT_DECL(opdata, op);

    if (cmd != OPONE)
        return (ENOTTY);

    STRUCT_INIT(op, mode);

    if (copyin((void *)data,
              STRUCT_BUF(op), STRUCT_SIZE(op)))
        return (EFAULT);

    if (STRUCT_FGET(op, flag) != FACTIVE ||
        STRUCT_FGET(op, size) > sizeof (device_state))
        return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```

This piece of code is an excerpt from a STREAMS module that handles [ioctl\(9E\)](#) data (M\_IOCTLDATA) messages and uses the data structure defined above. This code has been written to run in the ILP32 environment only.

**EXAMPLE 4** Using STRUCT\_HANDLE() and STRUCT\_SET\_HANDLE()

The next example illustrates the use of the STRUCT\_HANDLE() and STRUCT\_SET\_HANDLE() macros which declare and initialize the structure handle to point to an already existing instance of the structure.

The above code example can be converted to run in the LP64 environment using the STRUCT\_HANDLE() and STRUCT\_SET\_HANDLE() as follows:

EXAMPLE 4 Using STRUCT\_HANDLE() and STRUCT\_SET\_HANDLE() (Continued)

```

struct strbuf {
int maxlen;    /* no. of bytes in buffer */
int len;      /* no. of bytes returned */
caddr_t buf;  /* pointer to data */
};

static void
wput_iocdata(queue_t *q, mblk_t *msgp)
{
    struct copyresp *cp = (struct copyresp *)msgp->b_rptr;
    STRUCT_HANDLE(strbuf, sb);

    if (msgp->b_cont->b_cont != NULL) {
        msgp->b_cont = msgpullup(msgp->b_cont, -1);
        if (msgp->b_cont == NULL) {
            miocnak(q, msgp, 0, ENOSR);
            return;
        }
    }
    STRUCT_SET_HANDLE(sb, cp->cp_flag, (void *)msgp->b_cont->b_rptr);
    if (STRUCT_FGET(sb, maxlen) < (int)sizeof (ipa_t)) {
        miocnak(q, msgp, 0, ENOSR);
        return;
    }
    ...
    miocack(q, msgp, 0, 0);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [devmap\(9E\)](#), [ioctl\(9E\)](#), [mmap\(9E\)](#), [ddi\\_mmap\\_get\\_model\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

- 
- Name** `swab` – swap bytes in 16-bit halfwords
- Synopsis** `#include <sys/sunddi.h>`
- ```
void swab(const void *src, void *dst, ssize_t nbytes);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters**
- src* A pointer to the buffer containing the bytes to be swapped.
  - dst* A pointer to the destination buffer where the swapped bytes will be written. If *dst* is the same as *src* the buffer will be swapped in place.
  - nbytes* Number of bytes to be swapped, rounded down to the nearest half-word.
- Description** The `swab()` function copies the bytes in the buffer pointed to by *src* to the buffer pointer to by *dst*, swapping the order of adjacent bytes in half-word pairs as the copy proceeds. A total of *nbytes* bytes are copied, rounded down to the nearest half-word.
- Context** The `swab()` function can be called from user, interrupt, or kernel context.
- See Also** [Writing Device Drivers](#)
- Notes** Since `swab()` operates byte-by-byte, it can be used on non-aligned buffers.

**Name** taskq, ddi\_taskq\_create, ddi\_taskq\_destroy, ddi\_taskq\_dispatch, ddi\_taskq\_wait, ddi\_taskq\_suspend, taskq\_suspended, ddi\_taskq\_resume – Kernel task queue operations

**Synopsis** #include <sys/sunddi.h>

```

ddi_taskq_t *ddi_taskq_create(dev_info_t *dip, const char *name,
    int nthreads, pri_t pri, uint_t cflags);

void ddi_taskq_destroy(ddi_taskq_t *tq);

int ddi_taskq_dispatch(ddi_taskq_t *tq, void (*func)(void *),
    void *arg, uint_t dflags);

void ddi_taskq_wait(ddi_taskq_t *tq);

void ddi_taskq_suspend(ddi_taskq_t *tq);

boolean_t ddi_taskq_suspended(ddi_taskq_t *tq);

void ddi_taskq_resume(ddi_taskq_t *tq);

```

**Interface Level** Solaris DDI specific (Solaris DDI)

|                   |                                                            |                                                                                                                                                                                                                                                                        |           |                                                      |             |                                                            |
|-------------------|------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|------------------------------------------------------|-------------|------------------------------------------------------------|
| <b>Parameters</b> | <i>dip</i>                                                 | Pointer to the device's dev_info structure. May be NULL for kernel modules that do not have an associated dev_info structure.                                                                                                                                          |           |                                                      |             |                                                            |
|                   | <i>name</i>                                                | Descriptive string. Only alphanumeric characters can be used in name and spaces are not allowed. The name should be unique.                                                                                                                                            |           |                                                      |             |                                                            |
|                   | <i>nthreads</i>                                            | Number of threads servicing the task queue. Note that the request ordering is guaranteed (tasks are processed in the order scheduled) if the taskq is created with a single servicing thread.                                                                          |           |                                                      |             |                                                            |
|                   | <i>pri</i>                                                 | Priority of threads servicing the task queue. Drivers and modules should specify TASKQ_DEFAULTPRI.                                                                                                                                                                     |           |                                                      |             |                                                            |
|                   | <i>cflags</i>                                              | Should pass 0 as flags.                                                                                                                                                                                                                                                |           |                                                      |             |                                                            |
|                   | <i>func</i>                                                | Callback function to call.                                                                                                                                                                                                                                             |           |                                                      |             |                                                            |
|                   | <i>arg</i>                                                 | Argument to the callback function.                                                                                                                                                                                                                                     |           |                                                      |             |                                                            |
|                   | <i>dflags</i>                                              | Possible <i>dflags</i> are: <table style="margin-left: 2em;"> <tr> <td>DDI_SLEEP</td> <td>Allow sleeping (blocking) until memory is available.</td> </tr> <tr> <td>DDI_NOSLEEP</td> <td>Return DDI_FAILURE immediately if memory is not available.</td> </tr> </table> | DDI_SLEEP | Allow sleeping (blocking) until memory is available. | DDI_NOSLEEP | Return DDI_FAILURE immediately if memory is not available. |
| DDI_SLEEP         | Allow sleeping (blocking) until memory is available.       |                                                                                                                                                                                                                                                                        |           |                                                      |             |                                                            |
| DDI_NOSLEEP       | Return DDI_FAILURE immediately if memory is not available. |                                                                                                                                                                                                                                                                        |           |                                                      |             |                                                            |
|                   | <i>tq</i>                                                  | Pointer to a task queue (ddi_taskq_t *).                                                                                                                                                                                                                               |           |                                                      |             |                                                            |
|                   | <i>tp</i>                                                  | Pointer to a thread structure.                                                                                                                                                                                                                                         |           |                                                      |             |                                                            |

**Description** A kernel task queue is a mechanism for general-purpose asynchronous task scheduling that enables tasks to be performed at a later time by another thread. There are several reasons why you may utilize asynchronous task scheduling:

1. You have a task that isn't time-critical, but a current code path that is.
2. You have a task that may require grabbing locks that a thread already holds.
3. You have a task that needs to block (for example, to wait for memory), but a have a thread that cannot block in its current context.
4. You have a code path that can't complete because of a specific condition, but also can't sleep or fail. In this case, the task is immediately queued and then is executed after the condition disappears.
5. A task queue is just a simple way to launch multiple tasks in parallel.

A task queue consists of a list of tasks, together with one or more threads to service the list. If a task queue has a single service thread, all tasks are guaranteed to execute in the order they were dispatched. Otherwise they can be executed in any order. Note that since tasks are placed on a list, execution of one task and should not depend on the execution of another task or a deadlock may occur. A `taskq` created with a single servicing thread guarantees that all the tasks are serviced in the order in which they are scheduled.

The `ddi_taskq_create()` function creates a task queue instance.

The `ddi_taskq_dispatch()` function places `taskq` on the list for later execution. The *dflag* argument specifies whether it is allowed sleep waiting for memory. `DDI_SLEEP` dispatches can sleep and are guaranteed to succeed. `DDI_NOSLEEP` dispatches are guaranteed not to sleep but may fail (return `DDI_FAILURE`) if resources are not available.

The `ddi_taskq_destroy()` function waits for any scheduled tasks to complete, then destroys the `taskq`. The caller should guarantee that no new tasks are scheduled for the closing `taskq`.

The `ddi_taskq_wait()` function waits for all previously scheduled tasks to complete. Note that this function does not stop any new task dispatches.

The `ddi_taskq_suspend()` function suspends all task execution until `ddi_taskq_resume()` is called. Although `ddi_taskq_suspend()` attempts to suspend pending tasks, there are no guarantees that they will be suspended. The only guarantee is that all tasks dispatched after `ddi_taskq_suspend()` will not be executed. Because it will trigger a deadlock, the `ddi_taskq_suspend()` function should never be called by a task executing on a `taskq`.

The `ddi_taskq_suspended()` function returns `B_TRUE` if `taskq` is suspended, and `B_FALSE` otherwise. It is intended to `ASSERT` that the task queue is suspended.

The `ddi_taskq_resume()` function resumes task queue execution.

**Return Values** The `ddi_taskq_create()` function creates an opaque handle that is used for all other `taskq` operations. It returns a `taskq` pointer on success and `NULL` on failure.

The `ddi_taskq_dispatch()` function returns `DDI_FAILURE` if it can't dispatch a task and returns `DDI_SUCCESS` if dispatch succeeded.

The `ddi_taskq_suspended()` function returns `B_TRUE` if `taskq` is suspended. Otherwise `B_FALSE` is returned.

**Context** All functions may be called from the user or kernel contexts.

Additionally, the `ddi_taskq_dispatch` function may be called from the interrupt context only if the `DDI_NOSLEEP` flag is set.



- Name** testb – check for an available buffer
- Synopsis** #include <sys/stream.h>
- ```
int testb(size_t size, uint_t pri);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *size* Size of the requested buffer.  
*pri* Priority of the allocb request.
- Description** The testb() function checks to see if an allocb(9F) call is likely to succeed if a buffer of size bytes at priority pri is requested. Even if testb() returns successfully, the call to allocb(9F) can fail. The pri argument is no longer used, but is retained for compatibility.
- Return Values** Returns 1 if a buffer of the requested size is available, and 0 if one is not.
- Context** The testb() function can be called user, interrupt, or kernel context.

**Examples** EXAMPLE 1 testb() example

In a service routine, if copymsg(9F) fails (line 6), the message is put back on the queue (line 7) and a routine, tryagain, is scheduled to be run in one tenth of a second. Then the service routine returns.

When the timeout(9F) function runs, if there is no message on the front of the queue, it just returns. Otherwise, for each message block in the first message, check to see if an allocation would succeed. If the number of message blocks equals the number we can allocate, then enable the service procedure. Otherwise, reschedule tryagain to run again in another tenth of a second. Note that tryagain is merely an approximation. Its accounting may be faulty. Consider the case of a message comprised of two 1024-byte message blocks. If there is only one free 1024-byte message block and no free 2048-byte message blocks, then testb() will still succeed twice. If no message blocks are freed of these sizes before the service procedure runs again, then the copymsg(9F) will still fail. The reason testb() is used here is because it is significantly faster than calling copymsg. We must minimize the amount of time spent in a timeout() routine.

```

1  xxxsrv(q)
2      queue_t *q;
3  {
4      mblk_t *mp;
5      mblk_t *nmp;
6      . . .
7      if ((nmp = copymsg(mp)) == NULL) {
8          putbq(q, mp);
9          timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
10         return;
11     }

```

**EXAMPLE 1** testb() example (Continued)

```
. . .
11 }
12
13 tryagain(q)
14     queue_t *q;
15 {
16     register int can_alloc = 0;
17     register int num_blks = 0;
18     register mblk_t *mp;
19
20     if (!q->q_first)
21         return;
22     for (mp = q->q_first; mp; mp = mp->b_cont) {
23         num_blks++;
24         can_alloc += testb((mp->b_datap->db_lim -
25             mp->b_datap->db_base), BPRI_MED);
26     }
27     if (num_blks == can_alloc)
28         qenable(q);
29     else
30         timeout(tryagain, (intptr_t)q, drv_usectohz(100000));
31 }
```

**See Also** [allocb\(9F\)](#), [bufcall\(9F\)](#), [copymsg\(9F\)](#), [timeout\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Notes** The *pri* argument is provided for compatibility only. Its value is ignored.

- 
- Name** timeout – execute a function after a specified length of time
- Synopsis** `#include <sys/types.h>`  
`#include <sys/conf.h>`
- ```
timeout_id_t timeout(void (*func)(void *), void *arg,
                    clock_t ticks);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *func* Kernel function to invoke when the time increment expires.
- arg* Argument to the function.
- ticks* Number of clock ticks to wait before the function is called. Use [drv\\_usectohz\(9F\)](#) to convert microseconds to clock ticks.
- Description** The `timeout()` function schedules the specified function to be called after a specified time interval. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation.
- The function called by `timeout()` must adhere to the same restrictions as a driver soft interrupt handler.
- The [delay\(9F\)](#) function calls `timeout()`. Because `timeout()` is subject to priority inversion, drivers waiting on behalf of processes with real-time constraints should use [cv\\_timedwait\(9F\)](#) rather than `delay()`.
- Return Values** The `timeout()` function returns an opaque non-zero timeout identifier that can be passed to [untimeout\(9F\)](#) to cancel the request.
- Context** The `timeout()` function can be called from user, interrupt, or kernel context.
- Examples** **EXAMPLE 1** Using `timeout()`
- In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.
- ```
static void
xxtimeout_handler(void *arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->lock);
    cv_signal(&xsp->cv);
    xsp->flags |= TIMED_OUT;
    mutex_exit(&xsp->lock);
    xsp->timeout_id = 0;
}
```

**EXAMPLE 1** Using timeout() (Continued)

```

static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    .
    .
    .
    mutex_enter(&xsp->lock);
    /* Service interrupt */
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->lock);
    if (xsp->timeout_id != 0) {
        (void) untimeout(xsp->timeout_id);
        xsp->timeout_id = 0;
    }
    return(DDI_INTR_CLAIMED);
}

static void
xxcheckcond(struct xxstate *xsp)
{
    .
    .
    .
    xsp->timeout_id = timeout(xtimeout_handler,
        xsp, (5 * drv_usectohz(1000000)));
    mutex_enter(&xsp->lock);
    while (/* Waiting for interrupt or timeout */)
        cv_wait(&xsp->cv, &xsp->lock);
    if (xsp->flags & TIMED_OUT)
        cmn_err(CE_WARN, "Device not responding");
    .
    .
    .
    mutex_exit(&xsp->lock);
    .
    .
    .
}

```

**See Also** [bufcall\(9F\)](#), [cv\\_timedwait\(9F\)](#), [ddi\\_in\\_panic\(9F\)](#), [delay\(9F\)](#), [drv\\_usectohz\(9F\)](#), [untimeout\(9F\)](#)

*Writing Device Drivers*

**Name** tree, SPLAY\_PROTOTYPE, SPLAY\_GENERATE, SPLAY\_ENTRY, SPLAY\_HEAD, SPLAY\_INITIALIZER, SPLAY\_ROOT, SPLAY\_EMPTY, SPLAY\_NEXT, SPLAY\_MIN, SPLAY\_MAX, SPLAY\_FIND, SPLAY\_LEFT, SPLAY\_RIGHT, SPLAY\_FOREACH, SPLAY\_INIT, SPLAY\_INSERT, SPLAY\_REMOVE, RB\_PROTOTYPE, RB\_PROTOTYPE\_STATIC, RB\_GENERATE, RB\_GENERATE\_STATIC, RB\_ENTRY, RB\_HEAD, RB\_INITIALIZER, RB\_ROOT, RB\_EMPTY, RB\_NEXT, RB\_PREV, RB\_MIN, RB\_MAX, RB\_FIND, RB\_NFIND, RB\_LEFT, RB\_RIGHT, RB\_PARENT, RB\_FOREACH, RB\_FOREACH\_SAFE, RB\_FOREACH\_REVERSE, RB\_FOREACH\_REVERSE\_SAFE, RB\_INIT, RB\_INSERT, RB\_REMOVE – implementations of splay and red-black trees

**Synopsis** #include <sys/tree.h>

```
SPLAY_PROTOTYPE(NAME, TYPE, FIELD, CMP);
SPLAY_GENERATE(NAME, TYPE, FIELD, CMP);
SPLAY_ENTRY(TYPE);
SPLAY_HEAD(HEADNAME, TYPE);
struct TYPE *SPLAY_INITIALIZER(SPLAY_HEAD *head);
SPLAY_ROOT(SPLAY_HEAD *head);
int SPLAY_EMPTY(SPLAY_HEAD *head);
struct TYPE *SPLAY_NEXT(NAME, SPLAY_HEAD *head, struct TYPE *elm);
struct TYPE *SPLAY_MIN(NAME, SPLAY_HEAD *head);
struct TYPE *SPLAY_MAX(NAME, SPLAY_HEAD *head);
struct TYPE *SPLAY_FIND(NAME, SPLAY_HEAD *head, struct TYPE *elm);
struct TYPE *SPLAY_LEFT(struct TYPE *elm, SPLAY_ENTRY NAME);
struct TYPE *SPLAY_RIGHT(struct TYPE *elm, SPLAY_ENTRY NAME);
SPLAY_FOREACH(VARNAME, NAME, SPLAY_HEAD *head);
void SPLAY_INIT(SPLAY_HEAD *head);
struct TYPE *SPLAY_INSERT(NAME, SPLAY_HEAD *head, struct TYPE *elm);
struct TYPE *SPLAY_REMOVE(NAME, SPLAY_HEAD *head, struct TYPE *elm);
RB_PROTOTYPE(NAME, TYPE, FIELD, CMP);
RB_PROTOTYPE_STATIC(NAME, TYPE, FIELD, CMP);
RB_GENERATE(NAME, TYPE, FIELD, CMP);
RB_GENERATE_STATIC(NAME, TYPE, FIELD, CMP);
RB_ENTRY(TYPE);
RB_HEAD(HEADNAME, TYPE);
```

```

RB_INITIALIZER(RB_HEAD *head);
struct TYPE *RB_ROOT(RB_HEAD *head);
int RB_EMPTY(RB_HEAD *head);
struct TYPE *RB_NEXT(NAME, RB_HEAD *head, struct TYPE *elm);
struct TYPE *RB_PREV(NAME, RB_HEAD *head, struct TYPE *elm);
struct TYPE *RB_MIN(NAME, RB_HEAD *head);
struct TYPE *RB_MAX(NAME, RB_HEAD *head);
struct TYPE *RB_FIND(NAME, RB_HEAD *head, struct TYPE *elm);
struct TYPE *RB_NFIND(NAME, RB_HEAD *head, struct TYPE *elm);
struct TYPE *RB_LEFT(struct TYPE *elm, RB_ENTRY NAME);
struct TYPE *RB_RIGHT(struct TYPE *elm, RB_ENTRY NAME);
struct TYPE *RB_PARENT(struct TYPE *elm, RB_ENTRY NAME);
RB_FOREACH(VARNAME, NAME, RB_HEAD *head);
RB_FOREACH_SAFE(VARNAME, NAME, RB_HEAD *head, TEMP_VARNAME);
RB_FOREACH_REVERSE(VARNAME, NAME, RB_HEAD *head);
RB_FOREACH_REVERSE_SAFE(VARNAME, NAME, RB_HEAD *head, TEMP_VARNAME);
void RB_INIT(RB_HEAD *head);
struct TYPE *RB_INSERT(NAME, RB_HEAD *head, struct TYPE *elm);
struct TYPE *RB_REMOVE(NAME, RB_HEAD *head, struct TYPE *elm);

```

**Description** These macros define data structures for different types of trees: splay trees and red-black trees.

In the macro definitions, *TYPE* is the name tag of a user-defined structure that must contain a field named *FIELD*, of type *SPLAY\_ENTRY* or *RB\_ENTRY*. The argument *HEADNAME* is the name tag of a user defined structure that must be declared using the macros *SPLAY\_HEAD()* or *RB\_HEAD()*. The argument *NAME* has to be a unique name prefix for every tree that is defined.

**SplayTrees** A splay tree is a self-organizing data structure. Every operation on the tree causes a splay to happen. The splay moves the requested node to the root of the tree and partly rebalances it. This has the benefit that request locality causes faster lookups as the requested nodes move to the top of the tree. On the other hand, every lookup causes memory writes

The Balance Theorem bounds the total access time for *m* operations and *n* inserts on an initially empty tree as  $O((m + n)\lg n)$ . The amortized cost for a sequence of *m* accesses to a splay tree is  $O(\lg n)$ .

A splay tree is headed by a structure defined by the `SPLAY_HEAD()` macro. A `SPLAY_HEAD` structure is declared as follows:

```
SPLAY_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and `struct TYPE` is the type of the elements to be inserted into the tree.

The `SPLAY_ENTRY()` macro declares a structure that allows elements to be connected in the tree.

To use the functions that manipulate the tree structure, their prototypes need to be declared with the `SPLAY_PROTOTYPE()` macro, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by `SPLAY_ENTRY()`.

The function bodies are generated with the `SPLAY_GENERATE()` macro. It takes the same arguments as the `SPLAY_PROTOTYPE()` macro, but should be used only once.

The *CMP* argument is the name of a function used to compare trees' nodes with each other. The function takes two arguments of type `struct TYPE*`. If the first argument is smaller than the second, the function returns a value smaller than zero. If they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The `SPLAY_INIT()` macro initializes the tree referenced by `head`.

The splay tree can also be initialized statically by using the `SPLAY_INITIALIZER()` macro as follows:

```
SPLAY_HEAD(HEADNAME, TYPE) head = SPLAY_INITIALIZER(&head);
```

The `SPLAY_INSERT()` macro inserts the new element *elm* into the tree. Upon success, `NULL` is returned. If a matching element already exists in the tree, the insertion is aborted, and a pointer to the existing element is returned.

The `SPLAY_REMOVE()` macro removes the element *elm* from the tree pointed by `head`. Upon success, a pointer to the removed element is returned. `NULL` is returned if *elm* is not present in the tree.

The `SPLAY_FIND()` macro can be used to find a particular element in the tree.

```
struct TYPE find, *res;
find.key = 30;
res = SPLAY_FIND(NAME, &head, &find);
```

The `SPLAY_ROOT()`, `SPLAY_MIN()`, `SPLAY_MAX()`, and `SPLAY_NEXT()` macros can be used to traverse the tree

```
for (np = SPLAY_MIN(NAME, &head); np != NULL; np = SPLAY_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the `SPLAY_FOREACH()` macro:

```
SPLAY_FOREACH(np, NAME, &head)
```

The `SPLAY_EMPTY()` macro should be used to check whether a splay tree is empty.

**Red-Black Trees** A red-black tree is a binary search tree with the node color as an extra attribute. It fulfills a set of conditions:

1. every search path from the root to a leaf consists of the same number of black nodes,
2. each red node (except for the root) has a black parent,
3. each leaf node is black.

Every operation on a red-black tree is bounded as  $O(\lg n)$ . The maximum height of a red-black tree is  $2\lg(n+1)$ .

A red-black tree is headed by a structure defined by the `RB_HEAD()` macro. A `RB_HEAD` structure is declared as follows:

```
RB_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *struct TYPE* is the type of the elements to be inserted into the tree.

The `RB_ENTRY()` macro declares a structure that allows elements to be connected in the tree.

To use the functions that manipulate the tree structure, their prototypes need to be declared with the `RB_PROTOTYPE()` or `RB_PROTOTYPE_STATIC()` macros, where *NAME* is a unique identifier for this particular tree. The *TYPE* argument is the type of the structure that is being managed by the tree. The *FIELD* argument is the name of the element defined by `RB_ENTRY()`.

The function bodies are generated with the `RB_GENERATE()` or `RB_GENERATE_STATIC()` macros. These macros take the same arguments as the `RB_PROTOTYPE()` and `RB_PROTOTYPE_STATIC()` macros, but should be used only once.

The *CMP* argument is the name of a function used to compare trees' nodes with each other. The function takes two arguments of type `struct TYPE*`. If the first argument is smaller they are equal, the function returns zero. Otherwise, it should return a value greater than zero. The compare function defines the order of the tree elements.

The `RB_INIT()` macro initializes the tree referenced by `head`.

The red-black tree can also be initialized statically by using the `RB_INITIALIZER()` macro as follows:

```
RB_HEAD(HEADNAME, TYPE) head = RB_INITIALIZER(&head)
```



The `RB_INSERT()` macro inserts the new element *elm* into the tree. Upon success, `NULL` is returned. If a matching element already exists in the tree, the insertion is aborted, and a pointer to the existing element is returned.

The `RB_REMOVE()` macro removes the element *elm* from the tree pointed by `head`. `RB_REMOVE()` returns *elm*.

The `RB_FIND()` and `RB_NFIND()` macros can be used to find a particular element in the tree. `RB_FIND()` finds the node with the same key as *elm*. `RB_NFIND()` finds the first node greater than or equal to the search key.

```
struct TYPE find, *res;
find.key = 30;
res = RB_FIND(NAME, &head, &find)
```

The `RB_ROOT()`, `RB_MIN()`, `RB_MAX()`, `RB_NEXT()`, and `RB_PREV()` macros can be used to traverse the tree:

```
for (np = RB_MIN(NAME, &head); np != NULL; np = RB_NEXT(NAME, &head, np))
```

Or, for simplicity, one can use the `RB_FOREACH()` or `RB_FOREACH_REVERSE()` macros:

```
RB_FOREACH(np, NAME, &head)
```

The macros `RB_FOREACH_SAFE()` and `RB_FOREACH_REVERSE_SAFE()` traverse the tree referenced by `head` in a forward or reverse direction respectively, assigning each element in turn to *np*. However, unlike their unsafe counterparts, they permit both the removal of *np* as well as freeing it from within the loop safely without interfering with the traversal.

The `RB_EMPTY()` macro should be used to check whether a red-black tree is empty.

**Examples** **EXAMPLE 1** Declare a red-black tree holding integers.

The following example demonstrates how to declare a red-black tree holding integers. Values are inserted into it and the contents of the tree are printed in order. Lastly, the internal structure of the tree is printed.

```
#include <sys/tree.h>
#include    struct node {
            RB_ENTRY(node) entry;
            int i;
        };

int
intcmp(struct node *e1, struct node *e2)
{
    return (e1->i < e2->i ? -1 : e1->i > e2->i);
}
```

EXAMPLE 1 Declare a red-black tree holding integers. (Continued)

```
RB_HEAD(inttree, node) head = RB_INITIALIZER(&head);
RB_GENERATE(inttree, node, entry, intcmp)

int testdata[] = {
    20, 16, 17, 13, 3, 6, 1, 8, 2, 4, 10, 19, 5, 9, 12, 15, 18,
    7, 11, 14
};

void
print_tree(struct node *n)
{
    struct node *left, *right;

    if (n == NULL) {
        printf("nil");
        return;
    }
    left = RB_LEFT(n, entry);
    right = RB_RIGHT(n, entry);
    if (left == NULL && right == NULL)
        printf("%d", n->i);
    else {
        printf("%d(", n->i);
        print_tree(left);
        main()
    }

    int i;
    struct node *n;

    for (i = 0; i < sizeof(testdata) / sizeof(testdata[0]); i++) {
        if ((n = malloc(sizeof(struct node))) == NULL)
            err(1, NULL);
        n->i = testdata[i];
        RB_INSERT(inttree, &head, n);
    }

    RB_FOREACH(n, inttree, &head) {
        printf("%d\n", n->i);
    }
    print_tree(RB_ROOT(&head));
    printf("\n");
    return (0);
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [attributes\(5\)](#)

**Notes** Trying to free a tree in the following way is a common error:

```
SPLAY_FOREACH(var, NAME, &head) {
    SPLAY_REMOVE(NAME, &head, var);
    free(var);
}
free(head);
```

Since *var* is freed, the FOREACH() macro refers to a pointer that may have been reallocated already. Proper code needs a second variable.

```
for (var = SPLAY_MIN(NAME, &head); var != NULL; var = nxt) {
    nxt = SPLAY_NEXT(NAME, &head, var);
    SPLAY_REMOVE(NAME, &head, var);
    free(var);
}
```

**Authors** The author of the tree macros is Niels Provos.

**Name** u8\_strcmp – UTF-8 string comparison function

**Synopsis** #include <sys/sunddi.h>

```
int u8_strcmp(const char *s1, const char *s2, size_t n,
              int flag, size_t unicode_version, int *errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

<b>Parameters</b>	<i>s1, s2</i>	Pointers to null-terminated UTF-8 strings
	<i>n</i>	The maximum number of bytes to be compared. If 0, the comparison is performed until either or both of the strings are examined to the string terminating null byte.
	<i>flag</i>	The possible comparison options constructed by a bit-wise-inclusive-OR of the following values: <ul style="list-style-type: none"> <li>U8_STRCMP_CS Perform case-sensitive string comparison. This is the default.</li> <li>U8_STRCMP_CI_UPPER Perform case-insensitive string comparison based on Unicode upper case converted results of <i>s1</i> and <i>s2</i>.</li> <li>U8_STRCMP_CI_LOWER Perform case-insensitive string comparison based on Unicode lower case converted results of <i>s1</i> and <i>s2</i>.</li> <li>U8_STRCMP_NFD Perform string comparison after <i>s1</i> and <i>s2</i> have been normalized by using Unicode Normalization Form D.</li> <li>U8_STRCMP_NFC Perform string comparison after <i>s1</i> and <i>s2</i> have been normalized by using Unicode Normalization Form C.</li> <li>U8_STRCMP_NFKD Perform string comparison after <i>s1</i> and <i>s2</i> have been normalized by using Unicode Normalization Form KD.</li> <li>U8_STRCMP_NFKC Perform string comparison after <i>s1</i> and <i>s2</i> have been normalized by using Unicode Normalization Form KC.</li> </ul> <p>Only one case-sensitive or case-insensitive option is allowed. Only one Unicode Normalization option is allowed.</p>
	<i>unicode_version</i>	The version of Unicode data that should be used during comparison. The following values are supported:

**U8\_UNICODE\_320**

Use Unicode 3.2.0 data during comparison.

**U8\_UNICODE\_500**

Use Unicode 5.0.0 data during comparison.

**U8\_UNICODE\_LATEST**

Use the latest Unicode version data available, which is Unicode 5.0.0.

*errno*

A non-zero value indicates that an error has occurred during comparison. The following values are supported:

- EBADF**      The specified option values are conflicting and cannot be supported.
- EILSEQ**     There was an illegal character at *s1*, *s2*, or both.
- EINVAL**     There was an incomplete character at *s1*, *s2*, or both.
- ERANGE**     The specified Unicode version value is not supported.

**Description** After proper pre-processing, the `u8_strcmp()` function compares two UTF-8 strings byte-by-byte, according to the machine ordering defined by the corresponding version of the Unicode Standard.

When multiple comparison options are specified, Unicode Normalization is performed after case-sensitive or case-insensitive processing is performed.

**Return Values** The `u8_strcmp()` function returns an integer greater than, equal to, or less than 0 if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

When `u8_strcmp()` detects an illegal or incomplete character, such character causes the function to set *errno* to indicate the error. Afterward, the comparison is still performed on the resultant strings and a value based on byte-by-byte comparison is always returned.

**Context** The `u8_strcmp()` function can be called from user or interrupt context.

**Examples** **EXAMPLE 1** Perform simple default string comparison.

```
#include <sys/sunddi.h>

int
docmp_default(const char *u1, const char *u2) {
    int result;
    int ;

    result = u8_strcmp(u1, u2, 0, 0, U8_UNICODE_LATEST, &errno);
    if (errno == EILSEQ)
        return (-1);
    if (errno == EINVAL)
```

**EXAMPLE 1** Perform simple default string comparison. *(Continued)*

```

        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);

```

**EXAMPLE 2** Perform upper case based case-insensitive comparison with Unicode 3.2.0 date.

```

#include <sys/sunddi.h>

int
docmp_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errno;

    result = u8_strcmp(u1, u2, 0, U8_STRCMP_CI_UPPER,
        U8_UNICODE_320, &errno);
    if (errno == EILSEQ)
        return (-1);
    if (errno == EINVAL)
        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);

    return (result);
}

```

**EXAMPLE 3** Perform Unicode Normalization Form D.

Perform Unicode Normalization Form D and uppercase-based case-insensitive comparison with Unicode 3.2.0 date.

```

#include <sys/sunddi.h>

int
docmp_nfd_caseinsensitive_u320(const char *u1, const char *u2) {
    int result;
    int errno;

    result = u8_strcmp(u1, u2, 0,
        (U8_STRCMP_NFD|U8_STRCMP_CI_UPPER), U8_UNICODE_320,
        &errno);
    if (errno == EILSEQ)
        return (-1);
}

```

**EXAMPLE 3** Perform Unicode Normalization Form D. *(Continued)*

```

    if (errno == EINVAL)
        return (-2);
    if (errno == EBADF)
        return (-3);
    if (errno == ERANGE)
        return (-4);

    return (result);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [u8\\_validate\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [attributes\(5\)](#), [u8\\_textprep\\_str\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#)

The Unicode Standard (<http://www.unicode.org>)

**Name** u8\_textprep\_str – string-based UTF-8 text preparation function

**Synopsis** #include <sys/types.h>  
#include <sys/errno.h>  
#include <sys/sunddi.h>

```
size_t u8_textprep_str(char *inarray, size_t *inlen,
                      char *outarray, size_t *outlen, int flag,
                      size_t unicode_version, int *errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *inarray* A pointer to a byte array containing a sequence of UTF-8 character bytes to be prepared.

*inlen* As input argument, the number of bytes to be prepared in *inarray*. As output argument, the number of bytes in *inarray* still not consumed.

*outarray* A pointer to a byte array where prepared UTF-8 character bytes can be saved.

*outlen* As input argument, the number of available bytes at *outarray* where prepared character bytes can be saved. As output argument, after the conversion, the number of bytes still available at *outarray*.

*flag* The possible preparation options constructed by a bitwise-inclusive-OR of the following values:

**U8\_TEXTPREP\_IGNORE\_NULL**  
Normally `u8_textprep_str()` stops the preparation if it encounters null byte even if the current *inlen* is pointing to a value bigger than zero.

With this option, null byte does not stop the preparation and the preparation continues until *inlen* specified amount of *inarray* bytes are all consumed for preparation or an error happened.

**U8\_TEXTPREP\_IGNORE\_INVALID**  
Normally `u8_textprep_str()` stops the preparation if it encounters illegal or incomplete characters with corresponding *errno* values.

When this option is set, `u8_textprep_str()` does not stop the preparation and instead treats such characters as no need to do any preparation.

**U8\_TEXTPREP\_Toupper**  
Map lowercase characters to uppercase characters if applicable.

**U8\_TEXTPREP\_Tolower**  
Map uppercase characters to lowercase characters if applicable.



`U8_TEXTPREP_NFD`  
Apply Unicode Normalization Form D.

`U8_TEXTPREP_NFC`  
Apply Unicode Normalization Form C.

`U8_TEXTPREP_NFKD`  
Apply Unicode Normalization Form KD.

`U8_TEXTPREP_NFKC`  
Apply Unicode Normalization Form KC.

Only one case folding option is allowed. Only one Unicode Normalization option is allowed.

When a case folding option and a Unicode Normalization option are specified together, UTF-8 text preparation is done by doing case folding first and then Unicode Normalization.

If no option is specified, no processing occurs except the simple copying of bytes from input to output.

*unicode\_version* The version of Unicode data that should be used during UTF-8 text preparation. The following values are supported:

`U8_UNICODE_320`  
Use Unicode 3.2.0 data during comparison.

`U8_UNICODE_500`  
Use Unicode 5.0.0 data during comparison.

`U8_UNICODE_LATEST`  
Use the latest Unicode version data available which is Unicode 5.0.0 currently.

*errno* The error value when preparation is not completed or fails. The following values are supported:

`E2BIG` Text preparation stopped due to lack of space in the output array.

`EBADF` Specified option values are conflicting and cannot be supported.

`EILSEQ` Text preparation stopped due to an input byte that does not belong to UTF-8.

`EINVAL` Text preparation stopped due to an incomplete UTF-8 character at the end of the input array.

`ERANGE` The specified Unicode version value is not a supported version.

**Description** The `u8_textprep_str()` function prepares the sequence of UTF-8 characters in the array specified by *inarray* into a sequence of corresponding UTF-8 characters prepared in the array specified by *outarray*. The *inarray* argument points to a character byte array to the first character in the input array and *inlen* indicates the number of bytes to the end of the array to be converted. The *outarray* argument points to a character byte array to the first available byte in the output array and *outlen* indicates the number of the available bytes to the end of the array. Unless *flag* is `U8_TEXTPREP_IGNORE_NULL`, `u8_textprep_str()` normally stops when it encounters a null byte from the input array regardless of the current *inlen* value.

If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and a sequence of input bytes does not form a valid UTF-8 character, preparation stops after the previous successfully prepared character. If *flag* is `U8_TEXTPREP_IGNORE_INVALID` and the input array ends with an incomplete UTF-8 character, preparation stops after the previous successfully prepared bytes. If the output array is not large enough to hold the entire prepared text, preparation stops just prior to the input bytes that would cause the output array to overflow. The value pointed to by *inlen* is decremented to reflect the number of bytes still not prepared in the input array. The value pointed to by *outlen* is decremented to reflect the number of bytes still available in the output array.

**Return Values** The `u8_textprep_str()` function updates the values pointed to by *inlen* and *outlen* arguments to reflect the extent of the preparation. When `U8_TEXTPREP_IGNORE_INVALID` is specified, `u8_textprep_str()` returns the number of illegal or incomplete characters found during the text preparation. When `U8_TEXTPREP_IGNORE_INVALID` is not specified and the text preparation is successful, the function returns 0. If the entire string in the input array is prepared, the value pointed to by *inlen* will be 0. If the text preparation is stopped due to any conditions mentioned above, the value pointed to by *inlen* will be non-zero and *errno* is set to indicate the error. If such and any other error occurs, `u8_textprep_str()` returns `(size_t)-1` and sets *errno* to indicate the error.

**Context** The `u8_textprep_str()` function can be called from user or interrupt context.

**Examples** EXAMPLE 1 Simple UTF-8 text preparation

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
size_t ret;
char ib[MAXPATHLEN];
char ob[MAXPATHLEN];
size_t il, ol;
int err;
.
.
.
```

**EXAMPLE 1** Simple UTF-8 text preparation (Continued)

```

/*
 * We got a UTF-8 pathname from somewhere.
 *
 * Calculate the length of input string including the terminating
 * NULL byte and prepare other arguments.
 */
(void) strncpy(ib, pathname, MAXPATHLEN);
il = strlen(ib) + 1;
ol = MAXPATHLEN;

/*
 * Do toupper case folding, apply Unicode Normalization Form D,
 * ignore NULL byte, and ignore any illegal/incomplete characters.
 */
ret = u8_textprep_str(ib, &il, ob, &ol,
    (U8_TEXTPREP_IGNORE_NULL|U8_TEXTPREP_IGNORE_INVALID|
    U8_TEXTPREP_Toupper|U8_TEXTPREP_NFD), U8_UNICODE_LATEST, &err);
if (ret == (size_t)-1) {
    if (err == E2BIG)
        return (-1);
    if (err == EBADF)
        return (-2);
    if (err == ERANGE)
        return (-3);
    return (-4);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [u8\\_strcmp\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [attributes\(5\)](#), [u8\\_strcmp\(9F\)](#), [u8\\_validate\(9F\)](#), [uconv\\_u16tou32\(9F\)](#)

The Unicode Standard (<http://www.unicode.org>)

**Name** u8\_validate – validate UTF-8 characters and calculate the byte length

**Synopsis** #include <sys/types.h>  
#include <sys/errno.h>  
#include <sys/sunddi.h>

```
int u8_validate(char *u8str, size_t n, char **list, int flag,
               int *errno);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

*u8str* The UTF-8 string to be validated.

*n* The maximum number of bytes in *u8str* that can be examined and validated.

*list* A list of null-terminated character strings in UTF-8 that must be additionally checked against as invalid characters. The last string in *list* must be null to indicate there is no further string.

*flag* Possible validation options constructed by a bitwise-inclusive-OR of the following values:

**U8\_VALIDATE\_ENTIRE**  
By default, `u8_validate()` looks at the first character or up to *n* bytes, whichever is smaller in terms of the number of bytes to be consumed, and returns with the result.

When this option is used, `u8_validate()` will check up to *n* bytes from *u8str* and possibly more than a character before returning the result.

**U8\_VALIDATE\_CHECK\_ADDITIONAL**  
By default, `u8_validate()` does not use *list* supplied.

When this option is supplied with a list of character strings, `u8_validate()` additionally validates *u8str* against the character strings supplied with *list* and returns EBADF in *errno* if *u8str* has any one of the character strings in *list*.

**U8\_VALIDATE\_UCS2\_RANGE**  
By default, `u8_validate()` uses the entire Unicode coding space of U+0000 to U+10FFFF.

When this option is specified, the valid Unicode coding space is smaller to U+0000 to U+FFFF.

*errno* An error occurred during validation. The following values are supported:

**EBADF** Validation failed because *list*-specified characters were found in the string pointed to by *u8str*.

**EILSEQ** Validation failed because an illegal byte was found in the string pointed to by *u8str*.

- EINVAL** Validation failed because an incomplete byte was found in the string pointed to by *ustr*.
- ERANGE** Validation failed because character bytes were encountered that are outside the range of the Unicode coding space.

**Description** The `u8_validate()` function validates *ustr* in UTF-8 and determines the number of bytes constituting the character(s) pointed to by *ustr*.

**Return Values** If *ustr* is a null pointer, `u8_validate()` returns 0. Otherwise, `u8_validate()` returns either the number of bytes that constitute the characters if the next *n* or fewer bytes form valid characters, or -1 if there is an validation failure, in which case it may set *errno* to indicate the error.

**Examples** **EXAMPLE 1** Determine the length of the first UTF-8 character.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int errno;
.
.
.
len = u8_validate(u8, 4, (char **)NULL, 0, &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVALID);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

**EXAMPLE 2** Check if there are any invalid characters in the entire string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int n;
```

**EXAMPLE 2** Check if there are any invalid characters in the entire string. *(Continued)*

```
int errno;
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, (char **)NULL, U8_VALIDATE_ENTIRE, &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVALID);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
            return (MYFS4_ERR_BADCHAR);
        default:
            return (-10);
    }
}
```

**EXAMPLE 3** Check if there is any invalid character, including prohibited characters, in the entire string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

char u8[MAXPATHLEN];
int n;
int errno;
char *prohibited[4] = {
    ".", "..", "\\", NULL
};
.
.
.
n = strlen(u8);
len = u8_validate(u8, n, prohibited,
    (U8_VALIDATE_ENTIRE|U8_VALIDATE_CHECK_ADDITIONAL), &errno);
if (len == -1) {
    switch (errno) {
        case EILSEQ:
        case EINVAL:
            return (MYFS4_ERR_INVALID);
        case EBADF:
            return (MYFS4_ERR_BADNAME);
        case ERANGE:
```

**EXAMPLE 3** Check if there is any invalid character, including prohibited characters, in the entire string. *(Continued)*

```

        return (MYFS4_ERR_BADCHAR);
    default:
        return (-10);
    }
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [u8\\_strcmp\(3C\)](#), [u8\\_textprep\\_str\(3C\)](#), [u8\\_validate\(3C\)](#), [attributes\(5\)](#), [u8\\_strcmp\(9F\)](#), [u8\\_textprep\\_str\(9F\)](#), [uconv\\_u16tou32\(9F\)](#)

The Unicode Standard (<http://www.unicode.org>)

**Name** uconv\_u16tou32, uconv\_u16tou8, uconv\_u32tou16, uconv\_u32tou8, uconv\_u8tou16, uconv\_u8tou32 – Unicode encoding conversion functions

**Synopsis**

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>

int uconv_u16tou32(const uint16_t *utf16str, size_t *utf16len,
                  uint32_t *utf32str, size_t *utf32len, int flag);

int uconv_u16tou8(const uint16_t *utf16str, size_t *utf16len,
                  uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u32tou16(const uint32_t *utf32str, size_t *utf32len,
                  uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u32tou8(const uint32_t *utf32str, size_t *utf32len,
                  uchar_t *utf8str, size_t *utf8len, int flag);

int uconv_u8tou16(const uchar_t *utf8str, size_t *utf8len,
                  uint16_t *utf16str, size_t *utf16len, int flag);

int uconv_u8tou32(const uchar_t *utf8str, size_t *utf8len,
                  uint32_t *utf32str, size_t *utf32len, int flag);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters**

*utf16str* A pointer to a UTF-16 character string.

*utf16len* As an input parameter, the number of 16-bit unsigned integers in *utf16str* as UTF-16 characters to be converted or saved.

As an output parameter, the number of 16-bit unsigned integers in *utf16str* consumed or saved during conversion.

*utf32str* A pointer to a UTF-32 character string.

*utf32len* As an input parameter, the number of 32-bit unsigned integers in *utf32str* as UTF-32 characters to be converted or saved.

As an output parameter, the number of 32-bit unsigned integers in *utf32str* consumed or saved during conversion.

*utf8str* A pointer to a UTF-8 character string.

*utf8len* As an input parameter, the number of bytes in *utf8str* as UTF-8 characters to be converted or saved.

As an output parameter, the number of bytes in *utf8str* consumed or saved during conversion.

*flag* The possible conversion options that are constructed by a bitwise-inclusive-OR of the following values:



**UCONV\_IN\_BIG\_ENDIAN**

The input parameter is in big endian byte ordering.

**UCONV\_OUT\_BIG\_ENDIAN**

The output parameter should be in big endian byte ordering.

**UCONV\_IN\_SYSTEM\_ENDIAN**

The input parameter is in the default byte ordering of the current system.

**UCONV\_OUT\_SYSTEM\_ENDIAN**

The output parameter should be in the default byte ordering of the current system.

**UCONV\_IN\_LITTLE\_ENDIAN**

The input parameter is in little endian byte ordering.

**UCONV\_OUT\_LITTLE\_ENDIAN**

The output parameter should be in little endian byte ordering.

**UCONV\_IGNORE\_NULL**

The null or U+0000 character should not stop the conversion.

**UCONV\_IN\_ACCEPT\_BOM**

If the Byte Order Mark (BOM, U+FEFF) character exists as the first character of the input parameter, interpret it as the BOM character.

**UCONV\_OUT\_EMIT\_BOM**

Start the output parameter with Byte Order Mark (BOM, U+FEFF) character to indicate the byte ordering if the output parameter is in UTF - 16 or UTF - 32.

**Description** The `uconv_u16tou32()` function reads the given *utf16str* in UTF - 16 until U+0000 (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The UTF - 16 characters that are read are converted into UTF - 32 and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as UTF - 32 characters.

The `uconv_u16tou8()` function reads the given *utf16str* in UTF - 16 until U+0000 (zero) in *utf16str* is encountered as a character or until the number of 16-bit unsigned integers specified in *utf16len* is read. The UTF - 16 characters that are read are converted into UTF - 8 and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as UTF - 8 characters.

The `uconv_u32tou16()` function reads the given *utf32str* in UTF - 32 until U+0000 (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified in *utf32len* is read. The UTF - 32 characters that are read are converted into UTF - 16 and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as UTF - 16 characters.

The `uconv_u32tou8()` function reads the given *utf32str* in UTF - 32 until U+0000 (zero) in *utf32str* is encountered as a character or until the number of 32-bit unsigned integers specified

in *utf32len* is read. The UTF-32 characters that are read are converted into UTF-8 and the result is saved at *utf8str*. After the successful conversion, *utf8len* contains the number of bytes saved at *utf8str* as UTF-8 characters.

The `uconv_u8tou16()` function reads the given *utf8str* in UTF-8 until the null ('\0') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The UTF-8 characters that are read are converted into UTF-16 and the result is saved at *utf16str*. After the successful conversion, *utf16len* contains the number of 16-bit unsigned integers saved at *utf16str* as UTF-16 characters.

The `uconv_u8tou32()` function reads the given *utf8str* in UTF-8 until the null ('\0') byte in *utf8str* is encountered or until the number of bytes specified in *utf8len* is read. The UTF-8 characters that are read are converted into UTF-32 and the result is saved at *utf32str*. After the successful conversion, *utf32len* contains the number of 32-bit unsigned integers saved at *utf32str* as UTF-32 characters.

During the conversion, the input and the output parameters are treated with byte orderings specified in the *flag* parameter. When not specified, the default byte ordering of the system is used. The byte ordering *flag* value that is specified for UTF-8 is ignored.

When `UCONV_IN_ACCEPT_BOM` is specified as the *flag* and the first character of the string pointed to by the input parameter is the BOM character, the value of the BOM character dictates the byte ordering of the subsequent characters in the string pointed to by the input parameter, regardless of the supplied input parameter byte ordering option *flag* values. If the `UCONV_IN_ACCEPT_BOM` is not specified, the BOM as the first character is treated as a regular Unicode character: Zero Width No Break Space (ZWNBS) character.

When `UCONV_IGNORE_NULL` is specified, regardless of whether the input parameter contains `U+0000` or null byte, the conversion continues until the specified number of input parameter elements at *utf16len*, *utf32len*, or *utf8len* are entirely consumed during the conversion.

As output parameters, *utf16len*, *utf32len*, and *utf8len* are not changed if conversion fails for any reason.

**Context** The `uconv_u16tou32()`, `uconv_u16tou8()`, `uconv_u32tou16()`, `uconv_u32tou8()`, `uconv_u8tou16()`, and `uconv_u8tou32()` functions can be called from user or interrupt context.

**Return Values** Upon successful conversion, the functions return 0. Upon failure, the functions return one of the following `errno` values:

- EILSEQ** The conversion detected an illegal or out of bound character value in the input parameter.
- E2BIG** The conversion cannot finish because the size specified in the output parameter is too small.
- EINVAL** The conversion stops due to an incomplete character at the end of the input string.

EBADF Conflicting byte-ordering option *flag* values are detected.

**Examples** EXAMPLE 1 Convert a UTF-16 string in little-endian byte ordering into UTF-8 string.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
uint16_t u16s[MAXNAMELEN + 1];
uchar_t u8s[MAXNAMELEN + 1];
size_t u16len, u8len;
int ret;
.
.
.
u16len = u8len = MAXNAMELEN;
ret = uconv_u16tou8(u16s, &u16len, u8s, &u8len,
    UCONV_IN_LITTLE_ENDIAN);
if (ret != 0) {
    /* Conversion error occurred. */
    return (ret);
}
.
.
.
```

EXAMPLE 2 Convert a UTF-32 string in big endian byte ordering into little endian UTF-16.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
/*
 * An UTF-32 character can be mapped to an UTF-16 character with
 * two 16-bit integer entities as a "surrogate pair."
 */
uint32_t u32s[101];
uint16_t u16s[101];
int ret;
size_t u32len, u16len;
.
.
.
u32len = u16len = 100;
```

**EXAMPLE 2** Convert a UTF-32 string in big endian byte ordering into little endian UTF-16.  
(Continued)

```
ret = uconv_u32tou16(u32s, &u32len, u16s, &u16len,
    UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
if (ret == 0) {
    return (0);
} else if (ret == E2BIG) {
    /* Use bigger output parameter and try just one more time. */
    uint16_t u16s2[201];

    u16len = 200;
    ret = uconv_u32tou16(u32s, &u32len, u16s2, &u16len,
        UCONV_IN_BIG_ENDIAN | UCONV_OUT_LITTLE_ENDIAN);
    if (ret == 0)
        return (0);
}

/* Otherwise, return -1 to indicate an error condition. */
return (-1);
```

**EXAMPLE 3** Convert a UTF-8 string into UTF-16 in little-endian byte ordering.

Convert a UTF-8 string into UTF-16 in little-endian byte ordering with a Byte Order Mark (BOM) character at the beginning of the output parameter.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
.
.
.
uchar_t u8s[MAXNAMELEN + 1];
uint16_t u16s[MAXNAMELEN + 1];
size_t u8len, u16len;
int ret;
.
.
.
u8len = u16len = MAXNAMELEN;
ret = uconv_u8tou16(u8s, &u8len, u16s, &u16len,
    UCONV_IN_LITTLE_ENDIAN | UCONV_EMIT_BOM);
if (ret != 0) {
    /* Conversion error occurred. */
    return (ret);
}
.
.
```

**EXAMPLE 3** Convert a UTF-8 string into UTF-16 in little-endian byte ordering. *(Continued)*

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

**See Also** [uconv\\_u16tou32\(3C\)](#), [attributes\(5\)](#)

The Unicode Standard (<http://www.unicode.org>)

**Notes** Each UTF-16 or UTF-32 character maps to an UTF-8 character that might need one to maximum of four bytes.

One UTF-32 or UTF-8 character can yield two 16-bit unsigned integers as a UTF-16 character, which is a surrogate pair if the Unicode scalar value is bigger than U+FFFF.

Ill-formed UTF-16 surrogate pairs are seen as illegal characters during the conversion.

**Name** uiomove – copy kernel data using uio structure

**Synopsis**

```
#include <sys/types.h>
#include <sys/uio.h>
```

```
int uiomove(caddr_t address, size_t nbytes, enum uio_rw rwflag,
            uio_t *uio_p);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters**

- address* Source/destination kernel address of the copy.
- nbytes* Number of bytes to copy.
- rwflag* Flag indicating read or write operation. Possible values are UIO\_READ and UIO\_WRITE.
- uio\_p* Pointer to the uio structure for the copy.

**Description** The `uiomove()` function copies *nbytes* of data to or from the space defined by the uio structure (described in [uio\(9S\)](#)) and the driver.

The `uio_segflg` member of the [uio\(9S\)](#) structure determines the type of space to or from which the transfer is being made. If it is set to `UIO_SYSSPACE`, the data transfer is between addresses in the kernel. If it is set to `UIO_USERSPACE`, the transfer is between a user program and kernel space.

*rwflag* indicates the direction of the transfer. If `UIO_READ` is set, the data will be transferred from *address* to the buffer(s) described by *uio\_p*. If `UIO_WRITE` is set, the data will be transferred from the buffer(s) described by *uio\_p* to *address*.

In addition to moving the data, `uiomove()` adds the number of bytes moved to the `iov_base` member of the [iovec\(9S\)](#) structure, decreases the `iov_len` member, increases the `uio_offset` member of the [uio\(9S\)](#) structure, and decreases the `uio_resid` member.

This function automatically handles page faults. *nbytes* does not have to be word-aligned.

**Return Values** The `uiomove()` function returns 0 upon success or `EFAULT` on failure.

**Context** User context only, if `uio_segflg` is set to `UIO_USERSPACE`. User, interrupt, or kernel context, if `uio_segflg` is set to `UIO_SYSSPACE`.

**See Also** [ureadc\(9F\)](#), [uwritec\(9F\)](#), [iovec\(9S\)](#), [uio\(9S\)](#)

*Writing Device Drivers*

**Warnings** If `uio_segflg` is set to `UIO_SYSSPACE` and *address* is selected from user space, the system may panic.

---

**Name** unbufcall – cancel a pending bufcall request

**Synopsis** #include <sys/stream.h>

```
void unbufcall(bufcall_id_t id);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *id* Identifier returned from [bufcall\(9F\)](#) or [esbbcall\(9F\)](#).

**Description** The unbufcall function cancels a pending bufcall() or esbbcall() request. The argument *id* is a non-zero identifier for the request to be cancelled. *id* is returned from the bufcall() or esbbcall() function used to issue the request. unbufcall() will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to unbufcall() or deadlock may result.

**Return Values** None.

**Context** The unbufcall function can be called from user, interrupt, or kernel context.

**See Also** [bufcall\(9F\)](#), [esbbcall\(9F\)](#)

*Writing Device Drivers*

*STREAMS Programming Guide*

**Name** unlinkb – remove a message block from the head of a message

**Synopsis** #include <sys/stream.h>

```
mblk_t *unlinkb(mblk_t *mp);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *mp* Pointer to the message.

**Description** The unlinkb() function removes the first message block from the message pointed to by *mp*. A new message, minus the removed message block, is returned.

**Return Values** If successful, the unlinkb() function returns a pointer to the message with the first message block removed. If there is only one message block in the message, NULL is returned.

**Context** The unlinkb() function can be called from user, interrupt, or kernel context.

**Examples** EXAMPLE 1 unlinkb() example

The routine expects to get passed an M\_PROTO T\_DATA\_IND message. It will remove and free the M\_PROTO header and return the remaining M\_DATA portion of the message.

```
1 mblk_t *
2 makedata(mp)
3     mblk_t *mp;
4 {
5     mblk_t *nmp;
6
7     nmp = unlinkb(mp);
8     freeb(mp);
9     return(nmp);
10 }
```

**See Also** linkb(9F)

*Writing Device Drivers*

*STREAMS Programming Guide*



**Name** untimeout – cancel previous timeout function call

**Synopsis** `#include <sys/types.h>`  
`#include <sys/conf.h>`

```
clock_t untimeout(timeout_id_t id);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *id* Opaque timeout ID from a previous `timeout(9F)` call.

**Description** The `untimeout()` function cancels a pending `timeout(9F)` request. `untimeout()` will not return until the pending callback is cancelled or has run. Because of this, locks acquired by the callback routine should not be held across the call to `untimeout()` or a deadlock may result.

Since no mutex should be held across the call to `untimeout()`, there is a race condition between the occurrence of an expected event and the execution of the timeout handler. In particular, it should be noted that no problems will result from calling `untimeout()` for a timeout which is either running on another CPU, or has already completed. Drivers should be structured with the understanding that the arrival of both an interrupt and a timeout for that interrupt can occasionally occur, in either order.

**Return Values** The `untimeout()` function returns -1 if the *id* is not found. Otherwise, it returns an integer value greater than or equal to 0.

**Context** The `untimeout()` function can be called from user, interrupt, or kernel context.

**Examples** In the following example, the device driver has issued an IO request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

```
static void
xxtimeout_handler(void *arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->lock);
    cv_signal(&xsp->cv);
    xsp->flags |= TIMED_OUT;
    mutex_exit(&xsp->lock);
    xsp->timeout_id = 0;
}

static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    .
    .
    .
```

```
    mutex_enter(&xsp->lock);
    /* Service interrupt */
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->lock);
    if (xsp->timeout_id != 0) {
        (void) untimeout(xsp->timeout_id);
        xsp->timeout_id = 0;
    }
    return(DDI_INTR_CLAIMED);
}
static void
xxcheckcond(struct xxstate *xsp)
{
    .
    .
    .
    xsp->timeout_id = timeout(xtimeout_handler,
        xsp, (5 * drv_usectohz(1000000)));
    mutex_enter(&xsp->lock);
    while (/* Waiting for interrupt or timeout */)
        cv_wait(&xsp->cv, &xsp->lock);
    if (xsp->flags & TIMED_OUT)
        cmn_err(CE_WARN, "Device not responding");
    .
    .
    .
    mutex_exit(&xsp->lock);
    .
    .
    .
}
```

**See Also** [open\(9E\)](#), [cv\\_signal\(9F\)](#), [cv\\_wait\\_sig\(9F\)](#), [delay\(9F\)](#), [timeout\(9F\)](#)

*Writing Device Drivers*

**Name** ureadc – add character to a uio structure

**Synopsis**

```
#include <sys/uio.h>
#include <sys/types.h>
```

```
int ureadc(int c, uio_t *uio_p);
```

**Interface Level** Architecture independent level 1 (DDI/DKI).

**Parameters** *c*            The character added to the [uio\(9S\)](#) structure.  
*uio\_p*            Pointer to the [uio\(9S\)](#) structure.

**Description** The `ureadc()` function transfers the character *c* into the address space of the [uio\(9S\)](#) structure pointed to by *uio\_p*, and updates the uio structure as for [uiomove\(9F\)](#).

**Return Values** 0 is returned on success and EFAULT on failure.

**Context** The `ureadc()` function can be called from user, interrupt, or kernel context.

**See Also** [uiomove\(9F\)](#), [uwritec\(9F\)](#), [iovec\(9S\)](#), [uio\(9S\)](#)

*Writing Device Drivers*

**Name** usb\_alloc\_request, usb\_alloc\_ctrl\_req, usb\_free\_ctrl\_req, usb\_alloc\_bulk\_req, usb\_free\_bulk\_req, usb\_alloc\_intr\_req, usb\_free\_intr\_req, usb\_alloc\_isoc\_req, usb\_free\_isoc\_req – Allocate and free USB transfer requests

**Synopsis** #include <sys/usb/usba.h>

```
usb_ctrl_req_t *usb_alloc_ctrl_req(dev_info_t *dip, size_t len,
    usb_flags_t flags);

void usb_free_ctrl_req(usb_ctrl_req_t *request);

usb_bulk_req_t *usb_alloc_bulk_req(dev_info_t dip, size_t len,
    usb_flags_t flags);

void usb_free_bulk_req(usb_bulk_req_t *request);

usb_intr_req_t *usb_alloc_intr_req(dev_info_t *dip, size_t len,
    usb_flags_t flags);

void usb_free_intr_req(usb_intr_req_t *request);

usb_isoc_req_t *usb_alloc_isoc_req(dev_info_t *dip,
    uint_t isoc_pkts_count, size_t len, usb_flags_t flags);

void usb_free_isoc_req(usb_isoc_req_t *request);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_alloc\_ctrl\_req(), usb\_alloc\_bulk\_req() and usb\_alloc\_intr\_req():

*dip* Pointer to the device's dev\_info structure.

*len* Length of *data* for this request.

*flags* Only USB\_FLAGS\_SLEEP is recognized. Wait for resources if not immediately available.

For usb\_alloc\_isoc\_req():

*dip* Pointer to the device's dev\_info structure.

*isoc\_pkts\_count* Number of isochronous packet descriptors to associate with this request. Must be greater than zero.

*len* Length of *data* for this isochronous request.

*flags* Only USB\_FLAGS\_SLEEP is recognized. Wait for resources if not immediately available.

For usb\_free\_ctrl\_req(), usb\_free\_bulk\_req(), usb\_free\_intr\_req() and usb\_free\_isoc\_req():

*request* Pointer to the request structure to be freed. Can be NULL.

**Description** The `usb_alloc_ctrl_req()`, `usb_alloc_bulk_req()`, `usb_alloc_intr_req()`, and `usb_alloc_isoc_req()` functions allocate control, bulk, interrupt, or isochronous requests. Optionally, these functions can also allocate an mblk of the specified length to pass data associated with the request. (For guidelines on mblk data allocation, see the manpage for the relevant transfer function).

The `usb_alloc_isoc_req()` function also allocates a number of isochronous packet descriptors (`usb_isoc_pkt_descr_t`) specified by `isoc_pkts_count` to the end of the request proper (`usb_isoc_req_t`). See [usb\\_isoc\\_request\(9S\)](#) for more information on isochronous packet descriptors.

These functions always succeed when the `USB_FLAGS_SLEEP` flag is set, provided that they are given valid args and are not called from interrupt context.

The `usb_free_ctrl_req()`, `usb_free_bulk_req()`, `usb_free_intr_req()`, and `usb_free_isoc_req()` functions free their corresponding request. If the request's data block pointer is non-zero, the data block is also freed. For isoc requests, the array of packet descriptors is freed.

**Return Values** For `usb_alloc_ctrl_req()`, `usb_alloc_bulk_req()`, `usb_alloc_intr_req()` and `usb_alloc_isoc_req()`:

On success: returns a pointer to the appropriate `usb_xxx_request_t`.

On failure: returns `NULL`. Fails because the `dip` argument is invalid, `USB_FLAGS_SLEEP` is not set and memory is not available or because `USB_FLAGS_SLEEP` is set but the call was made in interrupt context.

For `usb_free_ctrl_req()`, `usb_free_bulk_req()`, `usb_free_intr_req()` and `usb_free_isoc_req()`: None.

**Context** The allocation routines can always be called from kernel and user context. They may be called from interrupt context only if `USB_FLAGS_SLEEP` is not specified.

The free routines may be called from kernel, user, and interrupt context.

**Examples**

```

/* This allocates and initializes an asynchronous control
 * request which will pass no data. Asynchronous requests
 * are used when they cannot block the calling thread.
 */

usb_ctrl_req_t *ctrl_req;

if ((ctrl_req = usb_alloc_ctrl_req(dip, 0, 0)) == NULL) {
    return (FAILURE);
}

```

```

/* Now initialize. */
ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_DEV_TO_HOST |
    USB_DEV_REQ_STANDARD | USB_DEV_REQ_RCPT_DEV;

ctrl_req->ctrl_bRequest      = (uint8_t)USB_REQ_GET_STATUS;
...
...
ctrl_req->ctrl_callback      = normal_callback;
ctrl_req->ctrl_exc_callback  = exception_callback;
...
...

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_current\\_frame\\_number\(9F\)](#),  
[usb\\_get\\_max\\_pkts\\_per\\_isoc\\_request\(9F\)](#), [usb\\_pipe\\_get\\_max\\_bulk\\_transfer\\_size\(9F\)](#),  
[usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#),  
[usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_bulk\\_request\(9S\)](#), [usb\\_ctrl\\_request\(9S\)](#),  
[usb\\_intr\\_request\(9S\)](#), [usb\\_isoc\\_request\(9S\)](#)

**Name** usb\_client\_attach, usb\_client\_detach – USB framework registration of client USB drivers

**Synopsis**

```
#define USBDRV_MAJOR_VER    <major>
#define USBDRV_MINOR_VER    <minor>
#include <sys/usb/usba.h>

int usb_client_attach(dev_info_t *dip,
    uint_t version, usb_flags_t flags);

void usb_client_detach(dev_info_t *dip,
    usb_client_dev_data_t *dev_data);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For `usb_client_attach()`:

*dip* Pointer to the device's `dev_info` structure.

*version* Must be set to `USBDRV_VERSION`. (See below.)

*flags* Not used.

For `usb_client_detach()`:

*dip* Pointer to the device's `dev_info` structure.

*dev\_data* Pointer to a `usb_client_dev_data_t` to free. Can be NULL.

**Description** The `usb_client_attach()` function registers a driver with the USB framework and must be called before any other USB function. Usually, `usb_client_attach()` is followed by a call to [usb\\_get\\_dev\\_data\(9F\)](#).

The `usb_client_detach()` function unregisters a driver with the USB framework. The `usb_client_detach()` function releases memory for all strings, descriptors and trees set up by [usb\\_get\\_dev\\_data\(9F\)](#) when its `dev_data` argument is non-NULL. The `usb_client_detach()` function is the last USB function a client calls before completing [detach\(9E\)](#). It is not necessary to call `usb_client_detach()` during a suspend operation.

**VERSIONING** `USBDRV_VERSION` is a macro which creates a version number based on the `USBDRV_MAJOR_VER` and `USBDRV_MINOR_VER` definitions. It must be passed as the version argument.

For drivers version 2.0 or greater, the value of `USBDRV_MAJOR_VERSION` must match its corresponding `USBA_MAJOR_VER` value in `<sys/usb/usbai.h>`, and the value of `USBDRV_MINOR_VERSION` must not be greater than its corresponding `USBA_MINOR_VER` value also in `<sys/usb/usbai.h>`.

Version 0.8 drivers from previous releases are binary compatible and run on Solaris 10, but are not compilable. Version 0.8 binary compatibility will not be supported in subsequent Solaris OS releases.

Definitions of USBDRV\_MAJOR\_VERSION and USBDRV\_MINOR\_VERSION must appear in the client driver above the reference to <sys/usb/usba.h>. Note that different releases have different USBA\_[MAJOR|MINOR]\_VER numbers.

**Return Values** For `usb_client_attach()`:

USB_SUCCESS	Registration is successful.
USB_INVALID_ARGS	<i>dip</i> is NULL.
USB_INVALID_CONTEXT	Called from interrupt context. Not called from an attach routine context.
USB_INVALID_VERSION	Version passed in <i>version</i> is invalid.
USB_FAILURE	Other internal error.

For `usb_client_detach()`:

USB_INVALID_ARGS	<i>dip</i> is NULL.
USB_INVALID_CONTEXT	Not called from an attach routine context.

**Context** The `usb_client_attach()` function may only be called from [attach\(9E\)](#).

The `usb_client_detach()` function may be called only from [attach\(9E\)](#) or [detach\(9E\)](#).

**Examples**

```

if (usb_client_attach(dip, USBDRV_VERSION, 0) != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Couldn't register USB device",
            ddi_driver_name(dip), ddi_get_instance(dip));

    return (USB_FAILURE);
}

if (usb_get_dev_data(dip, &dev_data, USB_PARSE_LVL_IF, 0) !=
    USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Couldn't get device descriptor data.",
            ddi_driver_name(dip), ddi_get_instance(dip));

    return (USB_FAILURE);
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed



ATTRIBUTETYPE	ATTRIBUTE VALUE
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [usb\\_get\\_dev\\_data\(9F\)](#)

**Name** usb\_clr\_feature – Clear feature of USB device, interface or endpoint

**Synopsis** #include <sys/usb/usba.h>

```
int usb_clr_feature(dev_info_t *dip,
    uint_t request_type, uint_t feature,
    uint_t which, usb_flags_t flags,
    void (*callback) (usb_pipe_handle_t pipe_handle,
    usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
    usb_opaque_t callback_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's dev\_info structure.

*pipe\_handle* Pipe handle to device, device interface or endpoint.

*request\_type* bmRequestType to be used. One of the following:

USB\_DEV\_REQ\_RCPT\_DEV - Clear feature on device.

USB\_DEV\_REQ\_RCPT\_IF - Clear feature on interface.

USB\_DEV\_REQ\_RCPT\_EP - Clear feature on endpoint.

*feature* Feature to be cleared. Can be any device-defined device-, interface-, or endpoint-specific feature, including the following which are defined in the USB 2.0 specification:

USB\_EP\_HALT - Clear a HALT on an endpoint.

USB\_DEV\_REMOTE\_WAKEUP - Clear REMOTE\_WAKEUP on a device.

USB\_DEV\_TEST\_MODE - Clear TEST\_MODE on a device.

*which* Device, interface or endpoint on which to clear the feature. One of:

Interface number, for interfaces.

Endpoint number, for endpoints.

0 for devices.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for completion and do not call callback.

*callback* Callback handler to notify of asynchronous completion.

*callback\_arg* Second argument passed to callback handler.

**Description** The `usb_clr_feature()` function clears a specific feature of a device, interface or endpoint. This function always blocks and waits for resources if not available, regardless of the flags argument.

This call blocks for completion if `USB_FLAGS_SLEEP` is set in flags. It returns immediately and calls the callback upon completion if `USB_FLAGS_SLEEP` is not set.

**Return Values**

<code>USB_SUCCESS</code>	Feature was successfully cleared.
<code>USB_INVALID_ARGS</code>	<i>dip</i> argument is NULL.
<code>USB_INVALID_PIPE</code>	<i>pipe_handle</i> argument is NULL
<code>USB_INVALID_CONTEXT</code>	Called from interrupt context with <code>USB_FLAGS_SLEEP</code> flag set.
<code>USB_FAILURE</code>	Clearing of feature was unsuccessful.

**Context** May always be called from user or kernel context. May be called from interrupt context only if `USB_FLAGS_SLEEP` is not set in flags.

If the `USB_CB_ASYNC_REQ_FAILED` bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

**Examples**

```
if (usb_clr_feature(dip, pipe_handle, USB_DEV_REQ_RCPT_EP,
    USB_EP_HALT, data_endpoint_num, 0) == USB_FAILURE) {
    cmn_err (CE_WARN,
        "%s%d: Error clearing halt condition on data endpoint %d.",
        ddi_driver_name(dip), ddi_get_instance(dip),
        data_endpoint_num);
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_callback\\_flags\(9S\)](#)

**Name** usb\_create\_pm\_components – Create power management components for USB devices

**Synopsis** #include <sys/usb/usba.h>

```
int usb_create_pm_components(dev_info_t *dip, uint_t *pwrstates);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's dev\_info structure.  
*pwrstates* Address into which a mask which lists power states capable by device is returned. This is a bitmask containing zero or more of the following values:

USB\_DEV\_PWRMASK\_D0 Corresponds to USB\_DEV\_OS\_PWR\_3 or full power.

USB\_DEV\_PWRMASK\_D1 Corresponds to USB\_DEV\_OS\_PWR\_2.

USB\_DEV\_PWRMASK\_D2 Corresponds to USB\_DEV\_OS\_PWR\_1.

USB\_DEV\_PWRMASK\_D3 Corresponds to USB\_DEV\_OS\_PWR\_0 or no power.

**Description** The usb\_create\_pm\_components() function creates pm component properties that assume the standard USB D0-D3 powerlevels (USB\_DEV\_PWR\_D0 - USB\_DEV\_PWR\_D3). See the device's relevant USB descriptor to determine the device's power management capabilities and account for bus-powered devices. The usb\_create\_pm\_components() function also updates the pm-components property in the device's dev\_info structure.

Note that these USB power levels are inverse of OS power levels. For example, USB\_DEV\_OS\_PWR\_0 and USB\_DEV\_PWR\_D3 are equivalent levels corresponding to powered-down.

**Return Values** USB\_SUCCESS Power management facilities in device are recognized by system.

USB\_FAILURE An error occurred.

**Context** May be called from user or kernel context.

**Examples** uint\_t \*pwrstates;

```
/* Hook into device's power management. Enable remote wakeup. */
if (usb_create_pm_components(dip, pwrstates) == USB_SUCCESS) {
    usb_handle_remote_wakeup(dip, USB_REMOTE_WAKEUP_ENABLE);
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_clr\\_feature\(9F\)](#), [usb\\_register\\_hotplug\\_cbs\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_handle\\_remote\\_wakeup\(9F\)](#), [pm\\_idle\\_component\(9F\)](#), [pm\\_busy\\_component\(9F\)](#), [pm\\_raise\\_power\(9F\)](#), [pm\\_lower\\_power\(9F\)](#), [usb\\_cfg\\_descr\(9S\)](#)

**Name** `usb_get_addr` – Retrieve device USB address

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_get_addr(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's `dev_info` structure.

**Description** The `usb_get_addr()` function returns the current USB bus address for debugging purposes. The returned address is unique for a specific USB bus, and may be replicated if multiple host controller instances are present on the system.

**Return Values** On success: USB device address.

On failure: returns 0. Fails if *dip* is NULL.

**Context** May be called from user, kernel or interrupt context.

**Examples** `int usb_addr;`

```
usb_addr = usb_get_addr(dip);
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_open\(9F\)](#)

**Name** usb\_get\_alt\_if, usb\_set\_alt\_if, usb\_get\_if\_number, usb\_owns\_device – Get and set alternate interface values

**Synopsis** #include <sys/usb/usba.h>

```
int usb_get_alt_if(dev_info_t *dip, uint_t interface_number,
                  uint_t *alternate_number, usb_flags_t flags);

int usb_set_alt_if(dev_info_t *dip, uint_t interface_number,
                  uint_t alternate_number, usb_flags_t flags,
                  void (*callback)(usb_pipe_handle_t pipe_handle,
                                   usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
                  usb_opaque_t callback_arg);

int usb_get_if_number(dev_info_t *dip);

boolean_t usb_owns_device(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_get\_alt\_if():

*dip* Pointer to device's dev\_info structure.  
*interface\_number* Interface of the desired alternate.  
*alternate\_number* Address where current alternate setting is returned.  
*flags* No flags are recognized. Reserved for future expansion.

For usb\_set\_alt\_if():

*dip* Pointer to device's dev\_info structure.  
*interface\_number* Interface of the desired alternate.  
*alternate\_number* Alternate interface number to be set.  
*flags* Only USB\_FLAGS\_SLEEP is recognized. Wait for completion and do not call callback.  
*callback* Callback handler to notify of asynchronous completion.  
*callback\_arg* Second argument passed to callback handler.

For usb\_get\_if\_number():

*dip* Pointer to device's dev\_info structure.

For usb\_owns\_device():

*dip* Pointer to device's `dev_info` structure.

**Description** USB devices can have multiple configurations, each with many interfaces. Within interfaces are alternate settings, and within alternate settings are endpoints.

Each interface within a configuration may be represented by the kernel as a device node. Only one set of device nodes (interfaces as determined by the configuration) can be active at one time.

Alternates to an interface represent different ways the kernel sees a device node. Only one alternate setting within an interface can be active (or selected) at one time. The functions presented in this man page get or set interface or alternate setting information.

The `usb_get_alt_if()` function requests the device to return the current alternate setting of the given interface. This function ignores the `flags` argument and always blocks.

The `usb_set_alt_if()` function requests the device to set the interface and its alternate setting as specified. Because this call changes the current device's interface and sets the new interface's mode of operation as seen by the system, the driver must insure that all pipes other than the default control pipe are closed and quiescent. To avoid contending with another driver for a different part of the device, the driver must be bound to: the entire device, the interface-association which includes the alternative interface, or to the interface whose number is *interface\_number*.

If `USB_FLAGS_SLEEP` is set in `flags`, `usb_set_alt_if()` blocks until completed. Otherwise, `usb_set_alt_if()` returns immediately and calls the callback handler when completed.

*callback* is the asynchronous callback handler and takes the following arguments:

`usb_pipe_handle_t pipe_handle` Handle of the default control pipe used to perform the request.

`usb_opaque_t callback_arg` `callback_arg` specified to `usb_set_alt_if()`.

`int rval` Request status.

`usb_cb_flags_t callback_flags` Status of the queueing operation. Can be:

- `USB_CB_NO_INFO` - Callback was uneventful.
- `USB_CB_ASYNC_REQ_FAILED` - Error queueing request.
- `USB_CB_NO_RESOURCES` - Error allocating resources.

The `usb_get_if_number()` function returns the interface number, or `USB_COMBINED_NODE` or `USB_DEVICE_NODE` node indicating that the driver is bound to the entire device.



The `usb_owns_device()` function returns `B_TRUE` if the driver of the `dip` argument owns the entire device, or `B_FALSE` if it owns just a particular interface.

**Return Values** For `usb_get_alt_if()`:

`USB_SUCCESS` Interface's alternate setting was successfully obtained.

`USB_INVALID_ARGS` Pointer to `alternate_number` and/or `dip` are `NULL`.

`USB_INVALID_CONTEXT` Called from interrupt context.

`USB_FAILURE` The interface number is invalid.

An access error occurred.

For `usb_set_alt_if()`:

`USB_SUCCESS` Alternate interface was successfully set.

`USB_INVALID_ARGS` `dip` is `NULL`. `USB_FLAGS_SLEEP` is clear and callback is `NULL`.

`USB_INVALID_PERM` `dip` does not own the interface to be set.

`USB_INVALID_CONTEXT` Called from interrupt context with `USB_FLAGS_SLEEP` specified.

`USB_INVALID_PIPE` Pipe handle is `NULL`, invalid, or refers to a pipe that is closing or closed.

`USB_FAILURE` The interface number and/or alternate setting are invalid.

Pipes were open.

An access error occurred.

For `usb_get_if_number()`:

`USB_COMBINED_NODE` if the driver is responsible for the entire active device configuration. The `dip` doesn't correspond to an entire physical device.

`USB_DEVICE_NODE` if the driver is responsible for the entire device. The `dip` corresponds to an entire physical device.

interface number: otherwise.

For `usb_owns_device()`:

`B_TRUE` Driver of the `dip` argument owns the entire device.

`B_FALSE` Driver of the `dip` argument owns only the current interface.

**Context** The `usb_get_if_number()` and `usb_owns_device()` functions may be called from user or kernel context.

The `usb_set_alt_if()` function may always be called from user or kernel context. It may be called from interrupt context only if `USB_FLAGS_SLEEP` is not set in flags. If the `USB_CB_ASYNC_REQ_FAILED` bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. See [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

The `usb_get_alt_if()` function may be called from user or kernel context.

**Examples**

```
/* Change alternate setting of interface 0. Wait for completion. */
if (usb_set_alt_if(
    dip, 0, new_alternate_setting_num, USB_FLAGS_SLEEP, NULL, 0) !=
    USB_SUCCESS) {
    cmn_err (CE_WARN,
        "%S%d: Error setting alternate setting on pipe",
        ddi_driver_name(dip), ddi_get_instance(dip));
    }
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_get\\_string\\_descr\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#)

**Name** `usb_get_cfg`, `usb_set_cfg` – Get and set current USB device configuration

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_get_cfg(dev_info_t *dip, uint_t cfgval, usb_flags_t flags);

int usb_set_cfg(dev_info_t *dip, uint_t cfg_index, usb_flags_t flags,
void (*callback)(usb_pipe_handle_t pipe_handle, usb_opaque_t
callback_arg, int rval, usb_cb_flags_t flags), usb_opaque_t
callback_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For `usb_get_cfg()`:

*dip* Pointer to device's `dev_info` structure.

*cfgval* Pointer to returned configuration value.

*flags* Not used. Always waits for completion.

For `usb_set_cfg()`:

*dip* Pointer to device's `dev_info` structure.

*cfg\_index* Desired device configuration index. Set to `USB_DEV_DEFAULT_CONFIG_INDEX` to restore default configuration.

*flags* Only `USB_FLAGS_SLEEP` is recognized. Wait for completion and do not call callback.

*callback* Callback handler to notify of asynchronous completion.

*callback\_arg* Second argument passed to callback handler.

**Description** The `usb_get_cfg()` function retrieves the current configuration. It ignores the `flags` argument and always blocks while contacting the device.

The `usb_set_cfg()` function sets a new configuration. Because this call changes the device's mode of operation, the device must be quiescent and have all pipes, with the exception of the default control pipe, closed. The driver must have control over the entire device and cannot own just a single interface on a composite device. Additionally, its device node must not be a parent to other device nodes that can be operated by other drivers. The driver must own the device exclusively, otherwise drivers managing other parts of the device would be affected without their knowledge or control.

This call updates all internal USBA framework data structures, whereas issuing a raw `USB_REQ_SET_CFG` device request does not. The `usb_set_cfg()` function is the only supported programmatic way to change device configuration.

This call blocks if `USB_FLAGS_SLEEP` is set in flags. It returns immediately and calls the callback on completion if `USB_FLAGS_SLEEP` is not set.

**Return Values** For `usb_get_cfg()`:

<code>USB_SUCCESS</code>	New configuration is retrieved.
<code>USB_INVALID_ARGS</code>	<i>cfgval</i> or <i>dip</i> is NULL.
<code>USB_FAILURE</code>	Configuration cannot be retrieved.

For `usb_set_cfg()`:

<code>USB_SUCCESS</code>	New configuration is set.
<code>USB_INVALID_ARGS</code>	<i>dip</i> is NULL.
	<code>USB_FLAGS_SLEEP</code> is clear and callback is NULL.
<code>USB_INVALID_CONTEXT</code>	Called from interrupt context with <code>USB_FLAGS_SLEEP</code> specified.
<code>USB_INVALID_PERM</code>	Caller does not own entire device or device is a parent to child devices.
<code>USB_BUSY</code>	One or more pipes other than the default control pipe are open on the device.
<code>USB_INVALID_PIPE</code>	Pipe handle is NULL or invalid, or pipe is closing or closed.
<code>USB_FAILURE</code>	An illegal configuration is specified.
	One or more pipes other than the default control pipe are open on the device.

**Context** The `usb_get_cfg()` function may be called from user or kernel context.

The `usb_set_cfg()` function may be called from user or kernel context always. It may be called from interrupt context only if `USB_FLAGS_SLEEP` is not set in flags.

If the `USB_CB_ASYNC_REQ_FAILED` bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

**Examples** Setting the configuration to the one at index 1 (in the array of `usb_cfg_data_t` configuration nodes as returned by `usb_get_dev_data()`), and verifying what the configuration is at that index. (See `usb_get_dev_data(9F)`).

```
uint_t cfg_index = 1;
```

```

/*
 * Assume all pipes other than the default control pipe
 * are closed and make sure all requests to the default
 * control pipe have completed. /
 */

if (usb_set_cfg(dip, cfg_index, USB_FLAGS_SLEEP, NULL, 0)
    != USB_SUCCESS) {
    cmn_err (CE_WARN,
            "%s%d: Error setting USB device to configuration #%d",
            ddi_driver_name(dip), ddi_get_instance(dip), cfg_index);
}

if (usb_get_cfg(dip, &bConfigurationValue, 0) == USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: USB device active configuration is %d",
            ddi_driver_name(dip), ddi_get_instance(dip),
            bConfigurationValue);
} else {
    ...
    ...
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_alt\\_if\(9F\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_get\\_string\\_descr\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_callback\\_flags\(9S\)](#), [usb\\_cfg\\_descr\(9S\)](#), [usb\\_ep\\_descr\(9S\)](#), [usb\\_if\\_descr\(9S\)](#)

**Name** `usb_get_current_frame_number` – Return current logical usb frame number

**Synopsis** `#include <sys/usb/usba.h>`

```
usb_frame_number_t usb_get_current_frame_number(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's `dev_info` structure.

**Description** The `usb_get_current_frame_number()` function retrieves the current logical USB frame number.

Isochronous requests can be started on a particular numbered frame. An offset number of frames (typically between 4 and 10) can be added to the current logical frame number to specify the number of an upcoming frame to start an isochronous request.

The USB specification requires that the frame frequency (the period between start-of-frame packets) is one millisecond. The Solaris operating environment USB implementation uses a running counter of the number of milliseconds since boot as the current logical frame number.

**Return Values** On success, the `usb_get_current_frame_number()` function returns the current USB frame number. On failure it returns 0. The function fails if *dip* is NULL.

**Context** May be called from user, kernel or interrupt context.

**Examples**

```
usb_pipe_handle_t handle;
usb_frame_number_t offset = 10;
usb_isoc_req_t *isoc_req;

isoc_req = usb_alloc_isoc_req(...);
...
...
isoc_req->isoc_frame_no = usb_get_current_frame_number(dip) + offset;
isoc_req->isoc_attributes = USB_ATTRS_ISOC_START_FRAME;
...
...
if (usb_pipe_isoc_xfer(handle, isoc_req, 0) != USB_SUCCESS) {
    ...
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems

---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_alloc\\_isoc\\_req\(9F\)](#), [usb\\_get\\_max\\_pkts\\_per\\_isoc\\_request\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_get\\_max\\_bulk\\_transfer\\_size\(9F\)](#), [usb\\_isoc\\_request\(9S\)](#)

**Name** `usb_get_dev_data`, `usb_free_dev_data`, `usb_free_descr_tree`, `usb_print_descr_tree` – Retrieve device configuration information

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_get_dev_data(dev_info_t *dip, usb_client_dev_data_t **dev_data,  
    usb_reg_parse_lvl_t parse_level, usb_flags_t flags);  
  
void usb_free_dev_data(dev_info_t *dip, usb_client_dev_data_t *dev_data);  
  
void usb_free_descr_tree(dev_info_t *dip, usb_client_dev_data_t *dev_data);  
  
int usb_print_descr_tree(dev_info_t *dip, usb_client_dev_data_t *dev_data);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For `usb_get_dev_data()`:

*dip* Pointer to device's `dev_info` structure.

*dev\_data* Address in which pointer to info is returned.

*parse\_level* Portion of device represented in the tree of parsed descriptors. See below for possible `usb_reg_parse_lvl_t` values and explanations.

*flags* Not used.

For `usb_free_dev_data()`:

*dip* Pointer to device's `dev_info` structure.

*dev\_data* Pointer to `usb_client_dev_data_t` to be freed.

For `usb_free_descr_tree()`:

*dip* Pointer to device's `dev_info` structure.

*dev\_data* Pointer to `usb_client_dev_data_t` containing the descriptor tree to free.

For `usb_print_descr_tree()`:

*dip* Pointer to device's `dev_info` structure.

*dev\_data* Pointer to `usb_client_dev_data_t` containing the descriptor tree to display on-screen.

**Description** The `usb_get_dev_data()` function interrogates a device and returns its configuration information in a `usb_client_dev_data_t` structure. Most USBA functions require information which comes from a `usb_client_dev_data_t`, and all other functions in this man page operate on this structure. Please see `usb_client_dev_data(9S)` for a full content description. Pass the `usb_client_dev_data_t` structure to `usb_client_detach(9F)` to completely deallocate it.



A descriptor tree is included in the information returned. The `usb_reg_parse_lvl_t` type represents the extent of the device to be represented by the returned tree (2nd arg to `usb_get_dev_data`) or what is actually represented in the returned tree (`dev_parse_level` field of the returned `usb_client_dev_data_t`). It has the following possible values:

<code>USB_PARSE_LVL_NONE</code>	Build no tree. <code>dev_n_cfg</code> returns 0, <code>dev_cfg</code> and <code>dev_curr_cfg</code> are returned NULL, and the <code>dev_curr_xxx</code> fields are invalid.
<code>USB_PARSE_LVL_IF</code>	If configuration number and interface properties are set (as when different interfaces are viewed by the OS as different device instances), parse configured interface only. If an OS device instance is set up to represent an entire physical device, <code>USB_PARSE_LVL_IF</code> works like <code>USB_PARSE_LVL_ALL</code> .
<code>USB_PARSE_LVL_CFG</code>	Parse entire configuration of configured interface only. Behaves similarly to <code>USB_PARSE_LVL_IF</code> , except that entire configuration is returned.
<code>USB_PARSE_LVL_ALL</code>	Parse entire device (all configurations), even when driver is bound to a single interface of a single configuration.

The `usb_free_dev_data()` function undoes what `usb_get_dev_data()` set up. It releases memory for all strings, descriptors, and trees set up by `usb_get_dev_data()`.

The `usb_free_descr_tree()` function frees the descriptor tree of its `usb_client_dev_data_t` argument, while leaving the rest of the information intact. The intent is for drivers to free memory after copying needed descriptor information from the tree. Upon return, the following `usb_client_dev_data_t` fields are modified as follows: `dev_cfg` is NULL, `dev_n_cfg` is zero and `dev_parse_level` is `USB_PARSE_LVL_NONE`. Additionally, `dev_curr_cfg` is NULL and `dev_curr_if` is invalid.

The `usb_print_descr_tree()` function is an easy-to-use diagnostic aid which dumps the descriptor tree to the screen when the system is verbose booted (`boot -v`). Output is spaced with blank lines for readability and provides you with an on-screen look at what a device has to offer.

**Return Values** For `usb_get_dev_data()`:

<code>USB_SUCCESS</code>	Registration is successful.
<code>USB_INVALID_ARGS</code>	<i>dip</i> or <i>dev_data</i> is NULL. <i>parse_level</i> is invalid.
<code>USB_INVALID_CONTEXT</code>	Called from interrupt context.
<code>USB_INVALID_VERSION</code>	<code>usb_client_attach(9F)</code> was not called first.
<code>USB_FAILURE</code>	Bad descriptor info or other internal error.

For `usb_free_dev_data()`: None

For `usb_free_descr_tree()`: None, but no operation occurs if *dip* and/or *dev\_data* are NULL.

For `usb_print_descr_tree()`:

USB_SUCCESS	Descriptor tree dump is successful.
USB_INVALID_ARGS	<i>dev_data</i> or <i>dip</i> are NULL.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_FAILURE	Other error.

**Context** The `usb_get_dev_data()` and `usb_print_descr_tree()` functions may be called from user or kernel context.

The `usb_free_dev_data()` and `usb_free_descr_tree()` functions may be called from user, kernel or interrupt context.

**Examples** In this example, assume a device has the configuration shown below, and the endpoint of config 2, iface 1, alt 1 which supports intr IN transfers needs to be found. Config 2, iface 1 is the "default" config/iface for the current OS device node.

```
config 1
  iface 0
    endpt 0
config 2
  iface 0
  iface 1
    alt 0
      endpt 0
      cv 0
    alt 1
      endpt 0
      endpt 1
      cv 0
      endpt 2
    alt 2
      endpt 0
      cv 0

usb_client_dev_data_t *dev_data;
usb_ep_descr_t ep_descr;
usb_ep_data_t *ep_tree_node;
uint8_t interface = 1;
uint8_t alternate = 1;
uint8_t first_ep_number = 0;
```

```

/*
 * We want default config/iface, so specify USB_PARSE_LVL_IF.
 * Default config will be returned as dev_cfg[0].
 /
 if (usb_get_dev_data(dip, &dev_data,
   USB_PARSE_LVL_IF, 0) != USB_SUCCESS) {
     cmn_err (CE_WARN,
       "%s%d: Couldn't get USB configuration descr tree",
       ddi_driver_name(dip), ddi_get_instance(dip));

     return (USB_FAILURE);
 }

 ep_tree_node = usb_lookup_ep_data(dip, dev_data, interface,
   alternate, first_ep_number, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
 if (ep_tree_node != NULL) {
     ep_descr = ep_tree_node->ep_descr;
 } else {
     cmn_r (CE_WARN,
       "%s%d: Device is missing intr-IN endpoint",
       ddi_driver_name(dip), ddi_get_instance(dip));

     usb_free_descr_tree(dip, &dev_data);

     return (USB_FAILURE);
 }

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_client\\_attach\(9F\)](#), [usb\\_get\\_alt\\_if\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_string\\_descr\(9F\)](#), [usb\\_lookup\\_ep\\_data\(9F\)](#), [usb\\_parse\\_data\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_cfg\\_descr\(9S\)](#), [usb\\_client\\_dev\\_data\(9S\)](#), [usb\\_ep\\_descr\(9S\)](#), [usb\\_if\\_descr\(9S\)](#), [usb\\_string\\_descr\(9S\)](#)

**Name** `usb_get_max_pkts_per_isoc_request` – Get maximum number of packets allowed per isochronous request

**Synopsis** `#include <sys/usb/usba.h>`

```
uint_t usb_get_max_pkts_per_isoc_request(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's `dev_info` structure.

**Description** The `usb_get_max_pkts_per_isoc_request()` function returns the maximum number of isochronous packets per request that the host control driver can support. This number can be used to determine the maximum amount of data which can be handled by a single isochronous request. That length is found by:

```
max = usb_get_max_pkts_per_isoc_request(dip) * endpoint_max_packet_size;
```

where `endpoint_max_packet_size` is the `wMaxPacketSize` field of the isochronous endpoint over which the transfer will take place.

**Return Values** On success, the `usb_get_current_frame_number()` function returns the maximum number of isochronous pkts per request. On failure it returns 0. The function fails if *dip* is NULL.

**Context** May be called from user, kernel or interrupt context.

**Examples**

```
/*
 * Set up to receive periodic isochronous data, requesting
 * the maximum amount for each transfer.
 */

int pkt;
/* Get max packet size from endpoint descriptor. */
uint_t ep_max_pkt_size = ep_descr.wMaxPacketSize;
uint_t isoc_pkts_count = usb_get_max_pkts_per_isoc_request(dip);

/*
 * Allocate an isoc request, specifying the max number of packets
 * and the greatest size transfer possible.
 */
usb_isoc_req_t *isoc_req = usb_alloc_isoc_req(dip,
    isoc_pkts_count,
    isoc_pkts_count * ep_max_pkt_size,
    USB_FLAGS_SLEEP);

/* Init each packet descriptor for maximum size. */
for (pkt = 0; pkt < isoc_pkts_count; pkt++) {
    isoc_req->isoc_pkt_descr[pkt].isoc_pkt_length = ep_max_pkt_size;
```

```

}

/* Set the length of a packet in the request too. */
isoc_req->isoc_pkts_length = ep_max_pkt_size;

/* Other isoc request initialization. */
...
...

if (usb_pipe_isoc_xfer(pipe, isoc_req, USB_FLAGS_NOSLEEP) != USB_SUCCESS) {
    ...
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_get\\_current\\_frame\\_number\(9F\)](#), [usb\\_ep\\_descr\(9S\)](#), [usb\\_isoc\\_request\(9S\)](#)

**Name** `usb_get_status` – Get status of a USB device/endpoint/interface

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_get_status(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
    uint_t request_type, uint_t which, uint16_t *status,
    usb_flags_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to device's <code>dev_info</code> structure.
<i>pipe_handle</i>	Default control pipe handle on which request is made.
<i>request_type</i>	<code>bmRequestType</code> . Either:  <code>USB_DEV_REQ_RCPT_DEV</code> — Get device status.  <code>USB_DEV_REQ_RCPT_IF</code> — Get interface status.  <code>USB_DEV_REQ_RCPT_EP</code> — Get endpoint status.
<i>which</i>	Device, interface or endpoint from which to get status. Either number of interface or endpoint, or 0 if device status requested.
<i>status</i>	Address into which the status is written.
<i>flags</i>	None are recognized.

**Description** The `usb_get_status()` function returns the status of a device, interface or endpoint. All status requests use the default control pipe. Length of data returned is `USB_GET_STATUS_LEN` bytes. Always block and wait for resources if not available, regardless of the `flags` argument.

When the *request\_type* recipient is `USB_DEV_REQ_RCPT_DEV`, device status is requested. Status returned includes bits for `USB_DEV_SLF_PWRD_STATUS` (device is currently self-powered) and `USB_DEV_RWAKEUP_STATUS` (device has remote wakeup enabled). A set bit indicates the corresponding status.

When the *request\_type* is `USB_DEV_REQ_RCPT_EP`, endpoint status is requested. Status returned includes bits for `USB_EP_HALT_STATUS` (endpoint is halted). A set bit indicates the corresponding status.

When the *request\_type* is `USB_DEV_REQ_RCPT_IF`, interface status is requested and `USB_IF_STATUS` (zero) is returned.

**Return Values**

<code>USB_SUCCESS</code>	Status returned successfully in the status argument.
<code>USB_INVALID_ARGS</code>	Status pointer and/or <code>dip</code> argument is <code>NULL</code> .
<code>USB_INVALID_PIPE</code>	Pipe handle is <code>NULL</code> .

USB\_FAILURE                      Status not returned successfully.

**Context**    May be called from user or kernel context.

**Examples**    `uint16_t status;`

```

if (usb_get_status(
    dip, pipe_handle, USB_DEV_REQ_RCPT_DEV, 0 &status, 0) ==
    USB_SUCCESS) {
    if (status & USB_DEV_SLF_PWRD_STATUS) {
        cmn_err (CE_WARN,
            "%s%d: USB device is running on its own power.",
            ddi_driver_name(dip), ddi_get_instance(dip));
    }
}

```

**Attributes**    See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also**    [attributes\(5\)](#), [usb\\_clr\\_feature\(9F\)](#), [usb\\_get\\_alt\\_if\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#),

**Name** usb\_get\_string\_descr – Get string descriptor from device

**Synopsis** #include <sys/usb/usba.h>

```
int usb_get_string_descr(dev_info_t *dip,
    uint16_t langid, uint8_t index,
    char *buf, size_t buflen);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's dev_info structure.
<i>langid</i>	Language ID. Currently only USB_LANG_ID (English ascii) is valid.
<i>index</i>	String index indicating descriptor to retrieve.
<i>buf</i>	Address into which the string descriptor is placed.
<i>buflen</i>	Size of buf in bytes.

**Description** The usb\_get\_string\_descr() function retrieves a parsed string descriptor from a device. *dip* specifies the device, while *index* indicates which descriptor to return.

String descriptors provide information about other descriptors, or information that is encoded in other descriptors, in readable form. Many descriptor types have one or more index fields which identify string descriptors. (See Sections 9.5 and 9.6 of the *USB 2.0* specification.) For example, a configuration descriptor's seventh byte contains the string descriptor index describing a specific configuration.

Retrieved descriptors that do not fit into *buflen* bytes are truncated. All returned descriptors are null-terminated.

**Return Values**

USB_SUCCESS	String descriptor is returned in <i>buf</i> .
USB_INVALID_ARGS	<i>dip</i> or <i>buf</i> are NULL, or <i>index</i> or <i>buflen</i> is 0.
USB_FAILURE	Descriptor cannot be retrieved.

**Context** May be called from user or kernel context.

**Examples**

```
/* Get the first string descriptor. */

char buf[SIZE];

if (usb_get_string_descr(
    dip, USB_LANG_ID, 0, buf, SIZE) == USB_SUCCESS) {
    cmn_err (CE_NOTE, "%s%d: %s",
        ddi_driver_name(dip), ddi_get_instance(dip), buf);
}
```



**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_string\\_descr\(9S\)](#)

**Name** usb\_handle\_remote\_wakeup – Enable or disable remote wakeup on USB devices

**Synopsis** #include <sys/usb/usba.h>

```
int usb_handle_remote_wakeup(dev_info_t *dip, int cmd);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's dev\_info structure.

*cmd* Command. Either USB\_REMOTE\_WAKEUP\_ENABLE or USB\_REMOTE\_WAKEUP\_DISABLE.

**Description** The usb\_handle\_remote\_wakeup() function enables or disables remote wakeup on a USB device. This call can block.

**Return Values** USB\_SUCCESS Remote wakeup is successfully enabled or disabled.

USB\_FAILURE Remote wakeup is not supported by the device. An internal error occurred.

**Context** May be called from user or kernel context.

**Examples** uint\_t \*pwrstates;

```
/* Hook into device's power management. Enable remote wakeup. */
if (usb_create_pm_components(dip, pwrstates) == USB_SUCCESS) {
    usb_handle_remote_wakeup(dip, USB_REMOTE_WAKEUP_ENABLE);
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [pm\\_busy\\_component\(9F\)](#), [pm\\_idle\\_component\(9F\)](#), [pm\\_lower\\_power\(9F\)](#), [pm\\_raise\\_power\(9F\)](#), [usb\\_clr\\_feature\(9F\)](#), [usb\\_create\\_pm\\_components\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_register\\_hotplug\\_cbs\(9F\)](#), [usb\\_cfg\\_descr\(9S\)](#)

**Name** usb\_lookup\_ep\_data – Lookup endpoint information

**Synopsis** #include <sys/usb/usba.h>

```
usb_ep_data_t *usb_lookup_ep_data(dev_info_t *dip,
    usb_client_dev_data_t *dev_datap, uint_t interface,
    uint_t alternate, uint_t skip, uint_t type, uint_t direction);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's dev_info structure.
<i>dev_datap</i>	Pointer to a usb_client_dev_data_t structure containing tree.
<i>interface</i>	Number of interface in which endpoint resides.
<i>alternate</i>	Number of interface alternate setting in which endpoint resides.
<i>skip</i>	Number of endpoints which match the requested type and direction to skip before finding one to retrieve.
<i>type</i>	Type of endpoint. This is one of: USB_EP_ATTR_CONTROL, USB_EP_ATTR_ISOCH, USB_EP_ATTR_BULK, or USB_EP_ATTR_INTR. Please see <a href="#">usb_pipe_open(9F)</a> for more information.
<i>direction</i>	Direction of endpoint, either USB_EP_DIR_OUT or USB_EP_DIR_IN. This argument is ignored for bi-directional control endpoints.

**Description** The usb\_lookup\_ep\_data() function returns endpoint information from the tree embedded in client data returned from usb\_get\_dev\_data. It operates on the current configuration (pointed to by the dev\_curr\_cfg field of the usb\_client\_dev\_data\_t argument). It skips the first <skip> number of endpoints it finds which match the specifications of the other arguments, and then retrieves information on the next matching endpoint it finds. Note that it does not make a copy of the data, but points to the tree itself.

**Return Values** On success: the tree node corresponding to the desired endpoint.

On failure: returns NULL. Fails if *dip* or *dev\_datap* are NULL, if the desired endpoint does not exist in the tree, or no tree is present in dev\_datap.

**Context** May be called from user, kernel or interrupt context.

**Examples** Retrieve the polling interval for the second interrupt endpoint at interface 0, alt 3:

```
uint8_t interval = 0;
usb_ep_data_t *ep_node = usb_lookup_ep_data(
    dip, dev_datap, 0, 3, 1, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
if (ep_node != NULL) {
    interval = ep_node->ep_descr.bInterval;
}
```

Retrieve the maximum packet size for the first control pipe at interface 0, alt 4:

```
uint16_t maxPacketSize = 0;
usb_ep_data_t *ep_node = usb_lookup_ep_data(
    dip, dev_datap, 0, 4, 0, USB_EP_ATTR_CONTROL, 0);
if (ep_node != NULL) {
    maxPacketSize = ep_node->ep_descr.wMaxPacketSize;
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_cfg\\_descr\(9S\)](#), [usb\\_if\\_descr\(9S\)](#), [usb\\_ep\\_descr\(9S\)](#)

- 
- Name** `usb_parse_data` – Tokenize and align the bytes of raw variable-format data
- Synopsis** `#include <sys/usb/usba.h>`
- ```
size_t usb_parse_data(char *format, uchar_t *data,
                    size_t datalen, void *structure, size_t structlen);
```
- Interface Level** Solaris DDI specific (Solaris DDI)
- Parameters**
- |                  |                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>format</i>    | Null terminated string describing the format of the data structure for general-purpose byte swapping. The letters “c,” “s,” “l,” and “L” represent 1, 2, 4 and 8 byte quantities, respectively. A descriptor that consists of a short and two bytes would be described by “scc.” A number preceding a letter serves as a multiplier of that letter. A format equivalent to “scc” is “s2c.” |
| <i>data</i>      | Raw descriptor data to parse.                                                                                                                                                                                                                                                                                                                                                              |
| <i>datalen</i>   | Length, in bytes, of the raw descriptor data buffer.                                                                                                                                                                                                                                                                                                                                       |
| <i>structure</i> | Destination data buffer where parsed data is returned.                                                                                                                                                                                                                                                                                                                                     |
| <i>structlen</i> | Length, in bytes, of the destination data buffer. Parsed result length will not exceed this value.                                                                                                                                                                                                                                                                                         |
- Description** The `usb_parse_data` function parses data such as a variable-format class- or vendor-specific descriptor. The function also tokenizes and aligns the bytes of raw descriptor data into fields of a variable-format descriptor.
- While the USB framework can parse the endpoint, interface, configuration, and string descriptors defined by the *USB 2.0* specification, the format of class- or vendor-specific descriptors cannot be explicitly defined by the specification and will be unique for each. The *format* argument defines how to parse such a descriptor.
- While the USB specification defines bit ordering as little-endian, this routine (like the entire API), converts the data to the endianness of the host.
- The *structlen* parameter defines the size of the destination data buffer. Data is truncated to this size if the destination data buffer is too small.
- Return Values** On success: Returns the size (in bytes) of the parsed data result.
- On failure: Returns 0. (Same as `USB_PARSE_ERROR`).
- Context** May be called from user, kernel or interrupt context.
- Examples**
- ```
/*
 * Parse raw descriptor data in buf, putting result into ret_descr.
 * ret_buf_len holds the size of ret_descr buf; routine returns
 * number of resulting bytes.
 */
```

```

    * Descriptor being parsed has 2 chars, followed by one short,
    * 3 chars and one more short.
    */
    size_t size_of_returned_descr;
    xxx_descr_t ret_descr;

    size_of_returned_descr = usb_parse_data("ccsccc",
        buf, sizeof(buf), (void *)ret_descr, (sizeof)xxx_descr_t));
    if (size_of_returned_descr < (sizeof (xxx_descr_t))) {
        /* Data truncated. */
    }

    or:

    size_of_returned_descr = usb_parse_data("2cs3cs",
        buf, sizeof(buf), (void *)ret_descr, (sizeof)xxx_descr_t));
    if (size_of_returned_descr < (sizeof (xxx_descr_t))) {
        /* Data truncated. */
    }

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_get\\_string\\_descr\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#)

**Name** `usb_pipe_bulk_xfer` – USB bulk transfer function

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_pipe_bulk_xfer(usb_pipe_handle_t pipe_handle,
    usb_bulk_req_t *request, usb_flags_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<code>pipe_handle</code>	Bulk pipe handle on which request is made.
<code>request</code>	Pointer to bulk transfer request.
<code>flags</code>	USB_FLAGS_SLEEP is the only flag recognized. Wait for request to complete.

**Description** The `usb_pipe_bulk_xfer()` function requests the USB framework to perform a transfer through a USB bulk pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request. Requests are synchronous when USB\_FLAGS\_SLEEP has been specified in flags. Calls for synchronous requests will not return until their transaction has completed. Asynchronous requests (made without specifying the USB\_FLAGS\_SLEEP flag) notify the caller of their completion via a callback function.

Requests for bulk transfers must have mblks attached to store data. Allocate an mblk for data when a request is allocated via [usb\\_alloc\\_bulk\\_req\(9F\)](#) by passing a non-negative value for the *len* argument.

<b>Return Values</b>	USB_SUCCESS	Transfer was successful.
	USB_INVALID_ARGS	Request is NULL.
	USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
	USB_INVALID_REQUEST	The request has been freed or otherwise invalidated.
		A set of conflicting attributes were specified. See <a href="#">usb_bulk_request(9S)</a> .
		The normal and/or exception callback was NULL and USB_FLAGS_SLEEP was not set.
		Data space is not provided to a non-zero length bulk request: ( <code>bulk_data == NULL</code> and <code>bulk_len != 0</code> )
	USB_INVALID_PIPE	Pipe handle is NULL or invalid.
		Pipe is closing or closed.

USB_PIPE_ERROR	Pipe handle refers to a pipe which is in the USB_PIPE_STATE_ERROR state.
USB_NO_RESOURCES	Memory, descriptors or other resources are unavailable.
USB_HC_HARDWARE_ERROR	Host controller is in error state.
USB_FAILURE	An asynchronous transfer failed or an internal error occurred.  A bulk request requested too much data: (length > usb_get_max_bulk_xfer size())  The pipe is in a unsuitable state (error, busy, not ready).

Additional status information may be available in the `bulk_completion_reason` and `bulk_cb_flags` fields of the request. Please see [usb\\_completion\\_reason\(9S\)](#) and [usb\\_callback\\_flags\(9S\)](#) for more information.

**Context** May be called from kernel or user context without regard to arguments. May be called from interrupt context only when the `USB_FLAGS_SLEEP` flag is clear.

**Examples**

```
/* Allocate, initialize and issue a synchronous bulk request. */
```

```
usb_bulk_req_t bulk_req;
mblk_t *mblk;

bulk_req = usb_alloc_bulk_req(dip, bp->b_bcount, USB_FLAGS_SLEEP);

bulk_req->bulk_attributes = USB_ATTRS_AUTOCLEARING;
mblk = bulk_req->bulk_data;
bcopy(buffer, mblk->b_wptr, bp->b_bcount);
mblk->b_wptr += bp->b_bcount;

if ((rval = usb_pipe_bulk_xfer(pipe, bulk_req, USB_FLAGS_SLEEP))
    != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Error writing bulk data.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems



---

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_bulk\\_request\(9S\)](#), [usb\\_callback\\_flags\(9S\)](#), [usb\\_completion\\_reason\(9S\)](#), [usb\\_ctrl\\_request\(9S\)](#), [usb\\_intr\\_request\(9S\)](#), [usb\\_isoc\\_request\(9S\)](#)

**Name** usb\_pipe\_close – Close and cleanup a USB device pipe

**Synopsis** #include <sys/usb/usba.h>

```
void usb_pipe_close(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
    usb_flags_t flags,
    void (*callback)(usb_pipe_handle_t pipe_handle,
    usb_opaque_t arg, int rval,
    usb_cb_flags_t flags), usb_opaque_t callback_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's dev_info structure.
<i>pipe_handle</i>	Handle of pipe to close. Cannot be a handle to the default control pipe.
<i>flags</i>	USB_FLAGS_SLEEP is the only flag recognized. Set it to wait for resources, for pipe to become free, and for all pending request callbacks to complete.
<i>callback</i>	This function is called on completion if the USB_FLAGS_SLEEP flag is not specified. Mandatory if the USB_FLAGS_SLEEP flag has not been specified.
<i>callback_arg</i>	Second argument to callback function.

**Description** The usb\_pipe\_close() function closes the pipe pointed to by *pipe\_handle*, releases all related resources and then frees the pipe handle. This function stops polling if the pipe to be closed is an interrupt-IN or isochronous-IN pipe. The default control pipe cannot be closed.

Pipe cleanup includes waiting for the all pending requests in the pipe to finish, and then flushing residual requests remaining after waiting for several seconds. Exception handlers of flushed requests are called with a completion reason of USB\_CR\_FLUSHED.

If USB\_FLAGS\_SLEEP is specified in *flags*, wait for all cleanup operations to complete before calling the callback handler and returning.

If USB\_FLAGS\_SLEEP is not specified in *flags*, an asynchronous close (to be done in a separate thread) is requested. Return immediately. The callback handler is called after all pending operations are completed.

The *callback* parameter is the callback handler and takes the following arguments:

usb_pipe_handle_t pipe_handle	Handle of the pipe to close.	
usb_opaque_t callback_arg	Callback_arg specified to usb_pipe_close().	
int rval	Return value of close operation	
usb_cb_flags_t callback_flags	Status of queueing operation. Can be:	
	USB_CB_NO_INFO	Callback was uneventful.

USB_CB_ASYNC_REQ_FAILED	Error starting asynchronous request.
-------------------------	--------------------------------------

**Return Values** Status is returned to the caller via the callback handler's `rval` argument. Possible callback handler `rval` argument values are:

USB_INVALID_PIPE	Pipe handle specifies a pipe which is closed or closing.
USB_INVALID_ARGS	<i>dip</i> or <i>pipe_handle</i> arguments are NULL.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_INVALID_PERM	Pipe handle specifies the default control pipe.
USB_FAILURE	Asynchronous resources are unavailable. In this case, USB_CB_ASYNC_REQ_FAILED is passed in as the <i>callback_flags</i> arg to the callback handler.

Exception handlers of any queued requests which were flushed are called with a completion reason of USB\_CR\_FLUSHED. Exception handlers of periodic pipe requests which were terminated are called with USB\_CR\_PIPE\_CLOSING.

Note that messages mirroring the above errors are logged to the console logfile on error. (This provides status for calls which otherwise could provide status).

**Context** May be called from user or kernel context regardless of arguments. May not be called from a callback executing in interrupt context. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

If the USB\_CB\_ASYNC\_REQ\_FAILED bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

**Examples**

```

/* Synchronous close of pipe. */
usb_pipe_close(dip, pipe, USB_FLAGS_SLEEP, NULL, NULL);

-----

/* Template callback. */
void close_callback(usb_pipe_handle_t, usb_opaque_t, usb_cb_flags_t);

/* Asynchronous close of pipe. */
usb_pipe_close(dip, pipe, 0, close_callback, callback_arg);

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_drain\\_reqs\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_callback\\_flags\(9S\)](#)

**Name** usb\_pipe\_ctrl\_xfer, usb\_pipe\_ctrl\_xfer\_wait – USB control pipe transfer functions

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_ctrl_xfer(usb_pipe_handle_t pipe_handle,
    usb_ctrl_req_t *request, usb_flags_t flags);
```

```
int usb_pipe_ctrl_xfer_wait(usb_pipe_handle_t pipe_handle,
    usb_ctrl_setup_t *setup, mblk_t **data,
    usb_cr_t * completion_reason, usb_cb_flags_t *cb_flags,
    usb_flags_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_pipe\_ctrl\_xfer():

*pipe\_handle* Control pipe handle on which request is made.

*request* Pointer to control transfer request.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for all pending request callbacks to complete.

For usb\_pipe\_ctrl\_xfer\_wait():

*pipe\_handle* Control pipe handle on which request is made.

*setup* Pointer to setup parameters. (See below.)

*data* Pointer to mblk containing data bytes to transfer with command. Ignored if NULL.

*completion\_reason* Returns overall completion status. Ignored if NULL. Please see [usb\\_callback\\_flags\(9S\)](#) for more information.

*callback\_flags* Returns flags set either during autoclearing or some other callback, which indicate recovery handling done in callback. Ignored if NULL.

*flags* No flags are recognized. Reserved for future expansion.

**Description** The usb\_pipe\_ctrl\_xfer() function requests the USB framework to perform a transfer through a USB control pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request. Requests are synchronous when USB\_FLAGS\_SLEEP is specified in flags; calls for synchronous requests do not return until their transaction is completed. Asynchronous requests (made without specifying the USB\_FLAGS\_SLEEP flag) notifies the caller of their completion via a callback function.



USB_INVALID_ARGS	Request is NULL.
USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_REQUEST	The request has been freed or otherwise invalidated.  A set of conflicting attributes were specified. See <a href="#">usb_request_attributes(9S)</a> .  The normal and/or exception callback is NULL and USB_FLAGS_SLEEP is not set.  Data space not provided to a control request while ctrl_wLength is nonzero.
USB_INVALID_PIPE	Pipe handle is NULL or invalid.  Pipe is closing or closed.
USB_NO_RESOURCES	Memory, descriptors or other resources unavailable.
USB_HC_HARDWARE_ERROR	Host controller is in error state.
USB_FAILURE	An asynchronous transfer failed or an internal error occurred.  The pipe is in an unsuitable state (error, busy, not ready).

Additional status information may be available in the `ctrl_completion_reason` and `ctrl_cb_flags` fields of the request. Please see [usb\\_callback\\_flags\(9S\)](#) and [usb\\_completion\\_reason\(9S\)](#) for more information.

For `usb_pipe_ctrl_xfer_wait()`:

USB_SUCCESS	Request was successful.
USB_INVALID_CONTEXT	Called from interrupt context.
USB_INVALID_ARGS	<i>dip</i> is NULL.

Any error code returned by `usb_pipe_ctrl_xfer()`.

Additional status information may be available in the `ctrl_completion_reason` and `ctrl_cb_flags` fields of the request. Please see [usb\\_callback\\_flags\(9S\)](#) and [usb\\_completion\\_reason\(9S\)](#) for more information.

**Context** The `usb_pipe_ctrl_xfer()` function may be called from kernel or user context without regard to arguments and from the interrupt context only when the `USB_FLAGS_SLEEP` flag is clear.

The `usb_pipe_ctrl_xfer_wait()` function may be called from kernel or user context.

**Examples**

```

/* Allocate, initialize and issue a synchronous control request. */

usb_ctrl_req_t ctrl_req;
void control_pipe_exception_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);

ctrl_req = usb_alloc_ctrl_req(dip, 0, USB_FLAGS_SLEEP);

ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_HOST_TO_DEV |
    USB_DEV_REQ_TYPE_CLASS | USB_DEV_REQ_RCPT_OTHER;

ctrl_req->ctrl_bRequest      = (uint8_t)USB_PRINTER_SOFT_RESET;
ctrl_req->ctrl_exc_cb        = control_pipe_exception_callback;
...
...
if ((rval = usb_pipe_ctrl_xfer(pipe, ctrl_req, USB_FLAGS_SLEEP))
    != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Error issuing USB cmd.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-----

/*
 * Allocate, initialize and issue an asynchronous control request to
 * read a configuration descriptor.
 */

usb_ctrl_req_t *ctrl_req;
void control_pipe_normal_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);
void control_pipe_exception_callback(
    usb_pipe_handle_t, usb_ctrl_req_t*);
struct buf *bp = ...;

ctrl_req =
    usb_alloc_ctrl_req(dip, sizeof(usb_cfg_descr_t), USB_FLAGS_SLEEP);

ctrl_req->ctrl_bmRequestType = USB_DEV_REQ_DEV_TO_HOST |
    USB_DEV_REQ_TYPE_STANDARD | USB_DEV_REQ_RCPT_DEV;

ctrl_req->ctrl_wLength      = sizeof(usb_cfg_descr_t);

```



```

ctrl_req->ctrl_wValue      = USB_DESCR_TYPE_SETUP_CFG | 0;
ctrl_req->ctrl_bRequest    = (uint8_t)USB_REQ_GET_DESCR;
ctrl_req->ctrl_cb          = control_pipe_normal_callback;
ctrl_req->ctrl_exc_cb      = control_pipe_exception_callback;

/* Make buf struct available to callback handler. */
ctrl_req->ctrl_client_private = (usb_opaque_t)bp;
...
...
if ((rval = usb_pipe_ctrl_xfer(pipe, ctrl_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Error issuing USB cmd.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-----

/* Call usb_pipe_ctrl_xfer_wait() to get device status. */

mblk_t *data;
usb_cr_t completion_reason;
usb_cb_flags_t callback_flags;
usb_ctrl_setup_t setup_params = {
    USB_DEV_REQ_DEV_TO_HOST |          /* bmRequestType */
    USB_DEV_REQ_TYPE_STANDARD | USB_DEV_REQ_RCPT_DEV,
    USB_REQ_GET_STATUS,                /* bRequest */
    0,                                  /* wValue */
    0,                                  /* wIndex */
    USB_GET_STATUS_LEN,                /* wLength */
    0                                    /* attributes. */
};

if (usb_pipe_ctrl_xfer_wait(
    pipe,
    &setup_params,
    &data,
    &completion_reason,
    &callback_flags,
    0) != USB_SUCCESS) {
    cmn_err (CE_WARN,
            "%s%d: USB get status command failed: "
            "reason=%d callback_flags=0x%x",
            ddi_driver_name(dip), ddi_get_instance(dip),
            completion_reason, callback_flags);
    return (EIO);
}

```

```

/* Check data length. Should be USB_GET_STATUS_LEN (2 bytes). */
length_returned = data->b_wptr - data->b_rptr;
if (length_returned != USB_GET_STATUS_LEN) {
    cmn_err (CE_WARN,
            "%5d: USB get status command returned %d bytes of data.",
            ddi_driver_name(dip), ddi_get_instance(dip), length_returned);
    return (EIO);
}

/* Retrieve data in endian neutral way. */
status = (*(data->b_rptr + 1) << 8) | *(data->b_rptr);

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_bulk\\_request\(9S\)](#), [usb\\_callback\\_flags\(9S\)](#), [usb\\_ctrl\\_request\(9S\)](#), [usb\\_completion\\_reason\(9S\)](#), [usb\\_intr\\_request\(9S\)](#), [usb\\_isoc\\_request\(9S\)](#)

**Name** usb\_pipe\_drain\_reqs – Allow completion of pending pipe requests

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_drain_reqs(dev_info_t *dip, usb_pipe_handle_t pipe_handle,
    uint_t timeout, usb_flags_t usb_flags,
    void (*callback)(usb_pipe_handle_t pipe_handle,
    usb_opaque_t callback_arg, int rval, usb_cb_flags_t flags),
    usb_opaque_t callback_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's dev_info structure.
<i>pipe_handle</i>	Handle of the pipe containing pending requests.
<i>timeout</i>	Maximum wait time for requests to drain. Must be a non-negative value in seconds. Zero specifies no timeout.
<i>flags</i>	USB_FLAGS_SLEEP is the only flag recognized. Wait for completion and do not call callback.
<i>callback</i>	Callback handler to notify of asynchronous completion.
<i>callback_arg</i>	Second argument passed to callback function.

**Description** The usb\_pipe\_drain\_reqs() function provides waits for pending requests to complete and then provides synchronous or asynchronous notification that all pending requests on a non-shared pipe indicated by pipe\_handle have completed. For a shared pipe (such as the default control pipe used by multiple drivers each managing one interface of a device), this function provides notification that all pending requests on that pipe that are associated with a given dip are completed.

The usb\_pipe\_drain\_reqs() function can be used to notify a close procedure when the default control pipe is clear during device closure, thereby allowing the close procedure to continue safely. Normally, a synchronous call to [usb\\_pipe\\_close\(9F\)](#) allows all requests in a pipe to finish before returning. However, a client driver cannot close the default control pipe.

If USB\_FLAGS\_SLEEP is set in flags, block until all pending requests are completed. Otherwise, return immediately and call the callback handler when all pending requests are completed.

The *callback* parameter accepts the asynchronous callback handler, which takes the following arguments:

usb_pipe_handle_t default_pipe_handle	Handle of the pipe to drain.
usb_opaque_t callback_arg	callback_arg specified to usb_pipe_drain_reqs().
int rval	Request status.

usb_cb_flags_t callback_flags	Status of the queueing operation. Can be:				
	<table> <tr> <td>USB_CB_NO_INFO</td> <td>Callback was uneventful.</td> </tr> <tr> <td>USB_CB_ASYNC_REQ_FAILED</td> <td>Error starting asynchronous request.</td> </tr> </table>	USB_CB_NO_INFO	Callback was uneventful.	USB_CB_ASYNC_REQ_FAILED	Error starting asynchronous request.
USB_CB_NO_INFO	Callback was uneventful.				
USB_CB_ASYNC_REQ_FAILED	Error starting asynchronous request.				

<b>Return Values</b>	USB_SUCCESS	Request is successful.
	USB_INVALID_ARGS	<i>dip</i> argument is NULL. USB_FLAGS_SLEEP is clear and callback is NULL.
	USB_INVALID_CONTEXT	Called from callback context with the USB_FLAGS_SLEEP flag set.
	USB_INVALID_PIPE	Pipe is not open, is closing or is closed.

**Context** May be called from user or kernel context.

If the USB\_CB\_ASYNC\_REQ\_FAILED bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

**Examples**

```

mydev_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    ...
    ...

    mydev_state->pipe_state = CLOSED;

    /* Wait for pending requests of a pipe to finish. Don't timeout. */
    (void) usb_pipe_drain_reqs(
        dip, pipe_handle, 0, USB_FLAGS_SLEEP, NULL, 0);

    /*
     * Dismantle streams and tear down this instance,
     * now that all requests have been sent.
     */
    qprocsoff(q);
    ...
    ...
    ddi_remove_minor_node(dip, NULL);
    ...
    ...
}

```

```
}

```

**Notes** For pipes other than the default control pipe, it is recommended to close the pipe using a synchronous `usb_pipe_close()`. `usb_pipe_close()` with the `USB_FLAGS_SLEEP` flag allows any pending requests in that pipe to complete before returning.

Do not call `usb_pipe_drain_reqs()` while additional requests are being submitted by a different thread. This action can stall the calling thread of `usb_pipe_drain_reqs()` unnecessarily.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_close\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_callback\\_flags\(9S\)](#)

**Name** `usb_pipe_get_max_bulk_transfer_size` – Get maximum bulk transfer size

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_pipe_get_max_bulk_transfer_size(dev_info_t dip, size_t *size);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip* Pointer to the device's `dev_info` structure.

*size* Returns the bulk transfer size.

**Description** The `usb_pipe_get_max_bulk_transfer_size()` function returns the maximum data transfer size in bytes that the host controller driver can support per bulk request. This information can be used to limit or break down larger requests to manageable sizes.

**Return Values** `USB_SUCCESS` Size is returned in *size* argument.

`USB_INVALID_ARGS` *dip* and/or *size* argument is `NULL`.

`USB_FAILURE` Size could not be returned. Zero is returned in *size* arg.

**Context** May be called from user, kernel or interrupt context.

**Examples**

```
int xxx_attach(dev_info_t *dip, int command)
{
    ...
    usb_pipe_get_max_bulk_transfer_size(dip, &state->max_xfer_size);
    ...
}

void xxx_minphys(struct buf bp)
{
    ...
    if (bp->b_bcount > state->max_xfer_size) {
        bp->b_bcount = state->max_xfer_size;
    }
    ...
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_bulk\\_request\(9S\)](#)

**Name** usb\_pipe\_get\_state – Return USB pipe state

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_get_state(usb_pipe_handle_t pipe_handle,
    usb_pipe_state_t *pipe_state, usb_flags_t usb_flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>pipe_handle</i>	Handle of the pipe to retrieve the state.
<i>pipe_state</i>	Pointer to where pipe state is returned.
<i>usb_flags</i>	No flags are recognized. Reserved for future expansion.

**Description** The `usb_pipe_get_state()` function retrieves the state of the pipe referred to by *pipe\_handle* into the location pointed to by *pipe\_state*.

Possible pipe states are:

USB_PIPE_STATE_CLOSED	Pipe is closed.
USB_PIPE_STATE_ACTIVE	Pipe is active and can send/receive data. Polling is active for isochronous and interrupt pipes.
USB_PIPE_STATE_IDLE	Polling is stopped for isochronous and interrupt-IN pipes.
USB_PIPE_STATE_ERROR	An error occurred. Client must call <code>usb_pipe_reset()</code> . Note that this status is not seen by a client driver if <code>USB_ATTRS_AUTOCLEARING</code> is set in the request attributes.
USB_PIPE_STATE_CLOSING	Pipe is being closed. Requests are being drained from the pipe and other cleanup is in progress.

**Return Values**

USB_SUCCESS	Pipe state returned in second argument.
USB_INVALID_ARGS	<i>pipe_state</i> argument is NULL.
USB_INVALID_PIPE	<i>pipe_handle</i> argument is NULL.

**Context** May be called from user, kernel or interrupt context.

**Examples**

```
usb_pipe_handle_t pipe;
usb_pipe_state_t state;

/* Recover if the pipe is in an error state. */
if ((usb_pipe_get_state(pipe, &state, 0) == USB_SUCCESS) &&
    (state == USB_PIPE_STATE_ERROR)) {
    cmn_err (CE_WARN, "%s%d: USB Pipe error.",
        ddi_driver_name(dip), ddi_get_instance(dip));
    do_recovery();
}
```



}

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_clr\\_feature\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_close\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#)

**Name** usb\_pipe\_intr\_xfer, usb\_pipe\_stop\_intr\_polling – USB interrupt transfer and polling functions

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_intr_xfer(usb_pipe_handle_t pipe_handle, usb_intr_req_t *request,
    usb_flags_t flags);

void usb_pipe_stop_intr_polling(usb_pipe_handle_t pipe_handle, usb__flags_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_pipe\_intr\_xfer():

*pipe\_handle* Interrupt pipe handle on which request is made.

*request* Pointer to interrupt transfer request.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for needed resources if unavailable. For requests specifying the USB\_ATTRS\_ONE\_XFER attribute, wait for the request to complete.

For usb\_pipe\_stop\_intr\_polling():

*pipe\_handle* Interrupt pipe handle on which to stop polling for data.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for polling to stop.

**Description** The usb\_pipe\_intr\_xfer() function requests the USB framework to perform a transfer through a USB interrupt pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request.

There are three categories of interrupt transfers: periodic or polled interrupt-IN, single-transfer interrupt-IN, and (single-transfer) interrupt-OUT.

**Periodic Interrupt-IN Transfers** Periodic or polled interrupt-IN transfers execute on input requests which do not have the USB\_ATTRS\_ONE\_XFER attribute set. One request enables repetitive transfers at a periodic rate set by the endpoint's bInterval. There can be only one interrupt-IN request submitted at a time.

Periodic interrupt-IN transfers are always asynchronous. Client driver notification of new data is always via a callback. The USB\_FLAGS\_SLEEP flag is only to wait for resources to become available. Callbacks must always be in place to receive transfer completion notification. Please see [usb\\_callback\\_flags\(9S\)](#) for details on USB callbacks.

Calls made to usb\_pipe\_intr\_xfer() for starting input polling need allocate only one request. The USB framework allocates a new request each time polling has new data to return. (Note that each request returned must be freed via [usb\\_free\\_intr\\_req\(9F\)](#). Specify a

zero length when calling `usb_alloc_intr_req()` to allocate the original request, since it will not be used to return data. Set the `intr_len` in the request to specify how much data can be returned per polling interval.

The original request passed to `usb_pipe_intr_xfer()` is used to return status when polling is terminated, or on an error condition when the `USB_ATTRS_AUTOCLEARING` attribute is set for the request. If autoclearing is not set, the current (non-original) request is returned on error. Call `usb_pipe_reset(9F)` to reset the pipe and get back the original request in this case. The `USB_CR_STOPPED_POLLING` flag is always set for callbacks where the original request is returned.

Single-transfer  
Interrupt-IN Transfers

Interrupt-IN requests which have the `USB_ATTRS_ONE_XFER` attribute perform a single transfer. Such requests are synchronous when the `USB_FLAGS_SLEEP` flag is specified. Calls for synchronous requests do not return until their transaction is complete, and their callbacks are optional. The request is returned to the client through the normal or the exception completion callback to signal either normal completion or an error condition.

Interrupt-OUT Transfers

Interrupt-OUT requests always set up for a single transfer. However, multiple requests can be queued and execute in periodic fashion until depleted.

Interrupt-OUT transfers are synchronous when the `USB_FLAGS_SLEEP` flag is set in the request's flags. Calls for synchronous transfers will not return until their transaction has completed. Calls for asynchronous transfers notify the client driver of transaction completion via a normal callback, or error completion via an exception callback.

The `usb_pipe_stop_intr_polling()` function terminates polling on interrupt-IN pipes and does the following:

1. Cease polling.
2. Allow any requests-in-progress to complete and be returned to the client driver through the normal callback mechanism.
3. Idle the pipe.
4. Return the original polling request to the client driver through an exception callback with a completion reason of `USB_CR_STOPPED_POLLING`.

The client driver may restart polling from an exception callback only if the callback corresponds to an original request. The callback handler checks for the following completion reasons to ensure that a callback corresponds to an original request:

```
USB_CR_STOPPED_POLLING,
USB_CR_PIPE_RESET,
USB_CR_PIPE_CLOSING,
USB_CR_NOT_SUPPORTED
```

The callback handler also checks the request's `intr_data` field to mark original polling requests, when the requests are created with a zero `len` argument. In this case, a NULL `intr_data` field distinguishes a returned original request from a request allocated by the framework during polling.

Mblks for data for interrupt-OUT requests are allocated when a request is allocated via [usb\\_alloc\\_intr\\_req\(9F\)](#) by passing a non-negative value for the `len` argument.

**Return Values** For `usb_pipe_intr_xfer()`

<code>USB_SUCCESS</code>	Transfer was successful.
<code>USB_INVALID_ARGS</code>	Request is NULL.
<code>USB_INVALID_CONTEXT</code>	Called from interrupt context with the <code>USB_FLAGS_SLEEP</code> flag set.
<code>USB_INVALID_REQUEST</code>	The request has been freed or otherwise invalidated.  A set of conflicting attributes was specified. See <a href="#">usb_intr_request(9S)</a> .  The normal and/or exception callback was NULL, <code>USB_FLAGS_SLEEP</code> was not set and <code>USB_ATTRS_ONE_XFER</code> was not set.  An interrupt request was specified with a NULL data and a non-zero <code>intr_len</code> value.  An IN interrupt request was specified with both polling ( <code>USB_ATTRS_ONE_XFER</code> clear in attributes) and non-zero timeout specified.  An IN interrupt request was specified with a non-NULL data argument.  An OUT interrupt request was specified with a NULL data argument.
<code>USB_INVALID_PIPE</code>	Pipe handle is NULL or invalid.  Pipe is closing or closed.
<code>USB_PIPE_ERROR</code>	Pipe handle refers to a pipe which is in the <code>USB_PIPE_STATE_ERROR</code> state.
<code>USB_NO_RESOURCES</code>	Memory, descriptors or other resources unavailable.
<code>USB_HC_HARDWARE_ERROR</code>	Host controller is in error state.

USB_FAILURE	<p>An asynchronous transfer failed or an internal error occurred.</p> <p>An intr polling request is made while polling is already in progress.</p> <p>The pipe is in an unsuitable state (error, busy, not ready).</p> <p>Additional status information may be available in the <code>intr_completion_reason</code> and <code>intr_cb_flags</code> fields of the request. Please see <a href="#">usb_completion_reason(9S)</a> and <a href="#">usb_callback_flags(9S)</a> for more information.</p>
-------------	---

For `usb_pipe_stop_intr_polling()`

None, but fails if called with `USB_FLAGS_SLEEP` specified from interrupt context, pipe handle is invalid, NULL or pertains to a closing or closed pipe, or the pipe is in an error state. Error messages are logged to the console logfile.

Exception handlers' queued requests which are flushed by these commands before execution are returned with completion reason of `USB_CR_FLUSHED`.

**Context** Both of these functions can be called from kernel or user context without regard to arguments, and may be called from interrupt context only when the `USB_FLAGS_SLEEP` flag is clear.

**Examples**

```

/* Start polling on interrupt-IN pipe. */

usb_intr_req_t intr_req;
void intr_pipe_callback(usb_pipe_handle_t, usb_intr_req_t*);
void intr_pipe_exception_callback(
    usb_pipe_handle_t, usb_intr_req_t*);
usb_ep_descr_t *ep_descr;

ep_descr = ...;
intr_req = usb_alloc_intr_req(dip, 0, USB_FLAGS_SLEEP);
...
...
intr_req->intr_attributes = USB_ATTRS_SHORT_XFER_OK;
intr_req->intr_len        = ep_descr->wMaxPacketSize;
...
...
intr_req->intr_cb          = intr_pipe_callback;
intr_req->intr_exc_cb      = intr_pipe_exception_callback;

if ((rval = usb_pipe_intr_xfer(pipe, intr_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Error starting interrupt pipe polling.",

```

```

        ddi_driver_name(dip), ddi_get_instance(dip));
    }

    -----

    /* Stop polling before setting device idle. Wait for polling to stop. */

    usb_pipe_stop_intr_polling(pipe, USB_FLAGS_SLEEP);
    (void) pm_idle_component(dip, 0);

    -----

    /* Allocate, initialize and issue a synchronous intr-OUT request. */

    usb_intr_req_t intr_req;
    mblk_t *mblk;
    usb_ep_descr_t *ep_descr;

    ep_descr = ...;

    intr_req =
        usb_alloc_intr_req(dip, ep_descr->wMaxPacketSize, USB_FLAGS_SLEEP);

    intr_req->intr_attributes = USB_ATTRS_AUTOCLEARING;
    mblk = intr_req->intr_data;
    bcopy(buffer, mblk->b_wptr, ep_descr->wMaxPacketSize);
    mblk->b_wptr += ep_descr->wMaxPacketSize;

    if ((rval = usb_pipe_intr_xfer(pipe, intr_req, USB_FLAGS_SLEEP))
        != USB_SUCCESS) {
        cmn_err (CE_WARN, "%s%d: Error writing intr data.",
            ddi_driver_name(dip), ddi_get_instance(dip));
    }
}

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#),

usb\_bulk\_request(9S),usb\_callback\_flags(9S),usb\_completion\_reason(9S),  
usb\_ctrl\_request(9S),usb\_ep\_descr(9S),usb\_intr\_request(9S),usb\_isoc\_request(9S)

**Name** usb\_pipe\_isoc\_xfer, usb\_pipe\_stop\_isoc\_polling – USB isochronous transfer and polling functions

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_isoc_xfer(usb_pipe_handle_t pipe_handle, usb_isoc_req_t *request,
    usb_flags_t flags);
```

```
void usb_pipe_stop_isoc_polling(usb_pipe_handle_t pipe_handle, usb_flags_t flags);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_pipe\_isoc\_xfer():

*pipe\_handle* Isochronous pipe handle on which request is made.

*request* Pointer to isochronous transfer request.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for needed resources if unavailable.

For usb\_pipe\_stop\_isoc\_polling():

*pipe\_handle* Isochronous pipe handle on which to stop polling for input.

*flags* USB\_FLAGS\_SLEEP is the only flag recognized. Wait for polling to stop.

**Description** The usb\_pipe\_isoc\_xfer() function requests the USB framework to perform a transfer through a USB isochronous pipe. The request is passed to the host controller driver (HCD), which performs the necessary transactions to complete the request.

By their nature, isochronous transfers require several transactions for completion. Each request may contain several packet descriptors. Descriptors correspond to subtransfers to be made in different frames. A request is deemed completed once all packets of that request have been processed. It is illegal to specify the USB\_ATTRS\_ONE\_XFER attribute in an isochronous request. The isochronous polling interval is always one millisecond, the period of a full-speed frame.

All isochronous requests are asynchronous, and will notify the caller of their completion via a callback function. All isochronous requests must specify normal and exception callback handlers.

Requests will wait for needed, unavailable resources when USB\_FLAGS\_SLEEP has been specified in flags. Requests made without USB\_FLAGS\_SLEEP set will fail if needed resources are not readily available.



No errors seen during request processing will result in aborted transfers or exception callbacks. Such errors will instead be logged in the packet descriptor's `isoc_pkt_status` field. These errors can be examined when the completed request is returned through a normal callback.

**Isynchronous-OUT TRANSFERS** Allocate room for data when allocating isochronous-OUT requests via `usb_alloc_isoc_req(9F)`, by passing a positive value for the *len* argument. The data will be divided among the request transactions, each transaction represented by a packet descriptor. (See `usb_isoc_request(9F)`. When all of the data has been sent, regardless of any errors encountered, a normal transfer callback will be made to notify the client driver of completion.

If a request is submitted while other requests are active or queued, and the new request has its `USB_ATTRS_ISOC_XFER_ASAP` attribute set, the host controller driver will queue the request to start on a frame which immediately follows the last frame of the last queued request.

**Isynchronous-IN TRANSFERS** All isochronous-IN transfers start background polling, and require only a single (original) request. The USB framework will allocate a new request each time polling has new data to return. Specify a zero length when calling `usb_alloc_isoc_req()` to allocate the original request, since it will not be used to return data. Set the `isoc_pkts_length` in the request to specify how much data to poll per interval (the length of one packet in the request).

The original request passed to `usb_pipe_isoc_xfer()` will be used to return status when polling termination is requested, or for error condition notification. There can be only one isochronous-IN request submitted at a time.

**CALLBACKS** Isochronous transfer normal-completion callbacks cannot block for any reason since they are called from interrupt context. They will have `USB_CB_INTR_CONTEXT` set in their callback flags to note this.

Isochronous exception callbacks have the following restrictions for blocking:

1. They can block for resources (for example to allocate memory).
2. They cannot block for synchronous completion of a command (for example `usb_pipe_close(9F)`) done on the same pipe. Asynchronous commands can be started, when the pipe's policy `pp_max_async_reqs` field is initialized to accommodate them.
3. They cannot block waiting for another callback to complete.
4. They cannot block waiting for a synchronous transfer request to complete. They can, however, make an asynchronous request (such as restarting polling with a new isochronous-IN transfer).

Please see the section on callbacks in [usb\\_callback\\_flags\(9S\)](#) for more information.

All isochronous transfer exception callbacks signify that polling has stopped. Polling requests are returned with the following completion reasons:

USB\_CR\_STOPPED\_POLLING  
 USB\_CR\_PIPE\_CLOSING

Note: There are no exception callbacks for error conditions.

The `usb_pipe_stop_isoc_polling()` function terminates polling on an isochronous-IN pipe. The `usb_pipe_stop_isoc_polling()` function does the following:

1. Cease polling.
2. Allow any requests-in-progress to complete and be returned to the client driver through the normal callback mechanism.
3. Idle the pipe.
4. Return the original polling request to the client driver through an exception callback with a completion reason of `USB_CR_STOPPED_POLLING`.

**Return Values** For `usb_pipe_isoc_xfer()`:

<code>USB_SUCCESS</code>	Transfer was successful.
<code>USB_INVALID_ARGS</code>	Request is NULL.
<code>USB_INVALID_CONTEXT</code>	Called from interrupt context with the <code>USB_FLAGS_SLEEP</code> flag set.
<code>USB_INVALID_REQUEST</code>	The request has been freed or otherwise invalidated.
	A set of conflicting attributes were specified. See <a href="#">usb_isoc_request(9S)</a> .
	The normal and/or exception callback was NULL, <code>USB_FLAGS_SLEEP</code> was not set and <code>USB_ATTRS_ONE_XFER</code> was not set.
	An isochronous request was specified with a zeroed <code>isoc_pkt_descr</code> , a NULL <code>isoc_pkt_descr</code> , or a NULL data argument.
	An isochronous request was specified with <code>USB_ATTRS_ISOC_XFER_ASAP</code> and a nonzero <code>isoc_frame_no</code> .
<code>USB_NO_FRAME_NUMBER</code>	An isochronous request was not specified with one and only one of <code>USB_ATTRS_ISOC_START_FRAME</code> or <code>USB_ATTRS_ISOC_XFER_ASAP</code> specified.
	An isochronous request was specified with <code>USB_ATTRS_ISOC_START_FRAME</code> and a zero <code>isoc_frame_no</code> .

USB_INVALID_START_FRAME	An isochronous request was specified with an invalid starting frame number (less than current frame number, or zero) and USB_ATTRS_ISOC_START_FRAME specified.
USB_INVALID_PIPE	Pipe handle is NULL or invalid.  Pipe is closing or closed.
USB_PIPE_ERROR	Pipe handle refers to a pipe which is in the USB_PIPE_STATE_ERROR state.
USB_NO_RESOURCES	Memory, descriptors or other resources unavailable.
USB_HC_HARDWARE_ERROR	Host controller is in error state.
USB_FAILURE	An asynchronous transfer failed or an internal error occurred.

An isoch request requested too much data:

```
(length > (usb_get_max_pkts_per_isoc_request() *
endpoint's wMaxPacketSize))
```

The pipe is in an unsuitable state (error, busy, not ready).

Additional status information may be available in the `isoc_completion_reason` and `isoc_cb_flags` fields of the request. Please see [usb\\_completion\\_reason\(9S\)](#) and [usb\\_callback\\_flags\(9S\)](#) for more information.

For `usb_pipe_stop_isoc_polling()`:

None, but will fail if called with `USB_FLAGS_SLEEP` specified from interrupt context; the pipe handle is invalid, NULL or pertains to a closing or closed pipe; or the pipe is in an error state. Messages regarding these errors will be logged to the console logfile.

**Context** Both of these functions may be called from kernel or user context without regard to arguments. May be called from interrupt context only when the `USB_FLAGS_SLEEP` flag is clear.

**Examples**

```
/* Start polling on an isochronous-IN pipe. */
```

```
usb_isoc_req_t isoc_req;
void isoc_pipe_callback(usb_pipe_handle_t, usb_isoc_req_t*);
void isoc_pipe_exception_callback(
    usb_pipe_handle_t, usb_isoc_req_t*);
uint_t pkt_size;
usb_ep_data_t *isoc_ep_tree_node;
usb_ep_descr_t *isoc_ep_descr = ...; /* From usb_lookup_ep_data() */
```

```

isoc_ep_descr = &isoc_ep_tree_node->ep_descr;
pkt_size = isoc_ep_descr->wMaxPacketSize;

isoc_req = usb_alloc_isoc_req(
    dip, num_pkts, NUM_PKTS * pkt_size, USB_FLAGS_SLEEP);
...
...
isoc_req->isoc_attributes = USB_ATTRS_ISOC_XFER_ASAP;
...
...
isoc_req->isoc_cb          = isoc_pipe_callback;
isoc_req->isoc_exc_cb     = isoc_pipe_exception_callback;
...
...
isoc_req->isoc_pkts_length = pkt_size;
isoc_req->isoc_pkts_count  = NUM_PKTS;
for (pkt = 0; pkt < NUM_PKTS; pkt++) {
    isoc_req->isoc_pkt_descr[pkt].isoc_pkt_length = pkt_size;
}

if ((rval = usb_pipe_isoc_xfer(pipe, isoc_req, USB_FLAGS_NOSLEEP))
    != USB_SUCCESS) {
    cmn_err (CE_WARN, "%s%d: Error starting isochronous pipe polling.",
            ddi_driver_name(dip), ddi_get_instance(dip));
}

-----

/* Stop polling before powering off device. Wait for polling to stop. */

usb_pipe_stop_isoc_polling(pipe, USB_FLAGS_SLEEP);
pm_idle_component(dip, 0);

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_alloc\\_request\(9F\)](#), [usb\\_get\\_current\\_frame\\_number\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_max\\_pkts\\_per\\_isoc\\_request\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#),

usb\_bulk\_request(9S),usb\_callback\_flags(9S),usb\_completion\_reason(9S),  
usb\_ctrl\_request(9S),usb\_ep\_descr(9S),usb\_intr\_request(9S),usb\_isoc\_request(9S)

**Name** usb\_pipe\_open – Open a USB pipe to a device

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_open(dev_info_t *dip, usb_ep_descr_t *endpoint,
                 usb_pipe_policy_t *pipe_policy,
                 usb_flags_t flags, usb_pipe_handle_t *pipe_handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's dev_info structure.
<i>endpoint</i>	Pointer to endpoint descriptor.
<i>pipe_policy</i>	Pointer to <i>pipe_policy</i> . <i>pipe_policy</i> provides hints on pipe usage.
<i>flags</i>	USB_FLAGS_SLEEP is only flag that is recognized. Wait for memory resources if not immediately available.
<i>pipe_handle</i>	Address to where new pipe handle is returned. (The handle is opaque.)

**Description** A pipe is a logical connection to an endpoint on a USB device. The `usb_pipe_open()` function creates such a logical connection and returns an initialized handle which refers to that connection.

The *USB 2.0* specification defines four endpoint types, each with a corresponding type of pipe. Each of the four types of pipes uses its physical connection resource differently. They are:

Control pipe	Used for bursty, non-periodic, reliable, host-initiated request/response communication, such as for command/status operations. These are guaranteed to get approximately 10% of frame time and will get more if needed and if available, but there is no guarantee on transfer promptness. Bidirectional.
Bulk pipe	Used for large, reliable, non-time-critical data transfers. These get the bus on a bandwidth-available basis. Unidirectional. Sample uses include printer data.
Interrupt pipe	Used for sending or receiving small amounts of reliable data infrequently but with bounded service periods, as for interrupt handling. Unidirectional.
Isochronous pipe	Used for large, unreliable, time-critical data transfers. Boasts a guaranteed constant data rate as long as there is data, but there are no retries of failed transfers. Interrupt and isochronous data are together guaranteed 90% of frame time as needed. Unidirectional. Sample uses include audio.

The type of endpoint to which a pipe connects (and therefore the pipe type) is defined by the `bmAttributes` field of that pipe's endpoint descriptor. (See [usb\\_ep\\_desc\(9S\)](#)). Opens to interrupt and isochronous pipes can fail if the required bandwidth cannot be guaranteed.

The polling interval for periodic (interrupt or isochronous) pipes, carried by the endpoint argument's `bInterval` field, must be within range. Valid ranges are:

Full speed: range of 1-255 maps to 1-255 ms.

Low speed: range of 10-255 maps to 10-255 ms.

High speed: range of 1-16 maps to  $(2^{bInterval-1}) * 125\mu s$ .

Adequate bandwidth during transfers is guaranteed for all periodic pipes which are opened successfully. Interrupt and isochronous pipes have guaranteed latency times, so bandwidth for them is allocated when they are opened. (Please refer to Sections 5.7 and 5.8 of the *USB 2.0* specification which address isochronous and interrupt transfers.) Opens of interrupt and isochronous pipes fail if inadequate bandwidth is available to support their guaranteed latency time. Because periodic pipe bandwidth is allocated on pipe open, open periodic pipes only when needed.

The bandwidth required by a device varies based on polling interval, the maximum packet size (`wMaxPacketSize`) and the device speed. Unallocated bandwidth remaining for new devices depends on the bandwidth already allocated for previously opened periodic pipes.

The *pipe\_policy* parameter provides a hint as to pipe usage and must be specified. It is a `usb_pipe_policy_t` which contains the following fields:

```

uchar_t      pp_max_async_reqs:
              A hint indicating how many
              asynchronous operations requiring
              their own kernel thread will be
              concurrently in progress, the highest
              number of threads ever needed at one
              time. Allow at least one for
              synchronous callback handling and as
              many as are needed to accommodate the
              anticipated parallelism of asynchronous*
              calls to the following functions:
                usb_pipe_close(9F)
                usb_set_cfg(9F)
                usb_set_alt_if(9F)
                usb_clr_feature(9F)
                usb_pipe_reset(9F)
                usb_pipe_drain_reqs(9F)
                usb_pipe_stop_intr_polling(9F)
                usb_pipe_stop_isoc_polling(9F)

```

Setting to too small a value can deadlock the pipe.

\* Asynchronous calls are calls made without the `USB_FLAGS_SLEEP` flag being passed. Note that a large number of callbacks becomes an issue mainly when blocking functions are called from callback handlers.

The control pipe to the default endpoints (endpoints for both directions with `addr 0`, sometimes called the default control pipe or default pipe) comes pre-opened by the hub. A client driver receives the default control pipe handle through `usb_get_dev_data(9F)`. A client driver cannot open the default control pipe manually. Note that the same control pipe may be shared among several drivers when a device has multiple interfaces and each interface is operated by its own driver.

All explicit pipe opens are exclusive; attempts to open an opened pipe fail.

On success, the `pipe_handle` argument points to an opaque handle of the opened pipe. On failure, it is set to `NULL`.

<b>Return Values</b>	<code>USB_SUCCESS</code>	Open succeeded.
	<code>USB_NO_RESOURCES</code>	Insufficient resources were available.
	<code>USB_NO_BANDWIDTH</code>	Insufficient bandwidth available. (isochronous and interrupt pipes).
	<code>USB_INVALID_CONTEXT</code>	Called from interrupt handler with <code>USB_FLAGS_SLEEP</code> set.
	<code>USB_INVALID_ARGS</code>	<code>dip</code> and/or <code>pipe_handle</code> is <code>NULL</code> . <code>Pipe_policy</code> is <code>NULL</code> .
	<code>USB_INVALID_PERM</code>	Endpoint is <code>NULL</code> , signifying the default control pipe. A client driver cannot open the default control pipe.
	<code>USB_NOT_SUPPORTED</code>	Isochronous or interrupt endpoint with maximum packet size of zero is not supported.
	<code>USB_HC_HARDWARE_ERROR</code>	Host controller is in an error state.
	<code>USB_FAILURE</code>	Pipe is already open. Host controller not in an operational state. Polling interval ( <code>ep_descrbInterval</code> field) is out of range ( <code>intr</code> or <code>isoc</code> pipes).

**Context** May be called from user or kernel context regardless of arguments. May also be called from interrupt context if the `USB_FLAGS_SLEEP` option is not set.



```

Examples      usb_ep_data_t *ep_data;
                  usb_pipe_policy_t policy;
                  usb_pipe_handle_t pipe;
                  usb_client_dev_data_t *reg_data;
                  uint8_t interface = 1;
                  uint8_t alternate = 1;
                  uint8_t first_ep_number = 0;

                  /* Initialize pipe policy. */
                  bzero(policy, sizeof(usb_pipe_policy_t));
                  policy.pp_max_async_requests = 2;

                  /* Get tree of descriptors for device. */
                  if (usb_get_dev_data(
                      dip, USBDRV_VERSION, &reg_data, USB_FLAGS_ALL_DESCR, 0) !=
                      USB_SUCCESS) {
                      ...
                  }

                  /* Get first interrupt-IN endpoint. */
                  ep_data = usb_lookup_ep_data(dip, reg_data, interface, alternate,
                      first_ep_number, USB_EP_ATTR_INTR, USB_EP_DIR_IN);
                  if (ep_data == NULL) {
                      ...
                  }

                  /* Open the pipe. Get handle to pipe back in 5th argument. */
                  if (usb_pipe_open(dip, &ep_data.ep_descr
                      &policy, USB_FLAGS_SLEEP, &pipe) != USB_SUCCESS) {
                      ...
                  }

```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_alt\\_if\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_get\\_dev\\_data\(9F\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_close\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#), [usb\\_pipe\\_set\\_private\(9F\)](#), [usb\\_ep\\_descr\(9S\)](#), [usb\\_callback\\_flags\(9S\)](#)

**Name** `usb_pipe_reset` – Abort queued requests from a USB pipe and reset the pipe

**Synopsis** `#include <sys/usb/usba.h>`

```
void usb_pipe_reset(dev_info_t *dip,
    usb_pipe_handle_t pipe_handle, usb_flags_t usb_flags,
    void (*callback)(usb_pipe_handle_t cb_pipe_handle,
    usb_opaque_t arg, int rval, usb_cb_flags_t flags),
    usb_opaque_t callback_arg);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters**

<i>dip</i>	Pointer to the device's <code>dev_info</code> structure.
<i>pipe_handle</i>	Handle of the pipe to reset. Cannot be the handle to the default control pipe.
<i>usb_flags</i>	<code>USB_FLAGS_SLEEP</code> is the only flag recognized. Wait for completion.
<i>callback</i>	Function called on completion if the <code>USB_FLAGS_SLEEP</code> flag is not specified. If <code>NULL</code> , no notification of completion is provided.
<i>callback_arg</i>	Second argument to callback function.

**Description** Call `usb_pipe_reset()` to reset a pipe which is in an error state, or to abort a current request and clear the pipe. The `usb_pipe_reset()` function can be called on any pipe other than the default control pipe.

A pipe can be reset automatically when requests sent to the pipe have the `USB_ATTRS_AUTOCLEARING` attribute specified. Client drivers see an exception callback with the `USB_CB_STALL_CLEARED` callback flag set in such cases.

Stalls on pipes executing requests without the `USB_ATTRS_AUTOCLEARING` attribute set must be cleared by the client driver. The client driver is notified of the stall via an exception callback. The client driver must then call `usb_pipe_reset()` to clear the stall.

The `usb_pipe_reset()` function resets a pipe as follows:

1. Any polling activity is stopped if the pipe being reset is an interrupt or isochronous pipe.
2. All pending requests are removed from the pipe. An exception callback, if specified beforehand, is executed for each aborted request.
3. The pipe is reset to the idle state.

Requests to reset the default control pipe are not allowed. No action is taken on a pipe which is closing.

If `USB_FLAGS_SLEEP` is specified in *flags*, this function waits for the action to complete before calling the callback handler and returning. If not specified, this function queues the request and returns immediately, and the specified callback is called upon completion.

*callback* is the callback handler. It takes the following arguments:

usb_pipe_handle_t cb_pipe_handle	Handle of the pipe to reset.
usb_opaque_t callback_arg	Callback_arg specified to usb_pipe_reset().
int rval	Return value of the reset call.
usb_cb_flags_t callback_flags	Status of the queuing operation. Can be:  USB_CB_NO_INFO — Callback was uneventful.  USB_CB_ASYNC_REQ_FAILED — Error starting asynchronous request.

**Return Values** Status is returned to the caller via the callback handler's rval argument. Possible callback handler rval argument values are:

USB_SUCCESS	Pipe successfully reset.
USB_INVALID_PIPE	<i>pipe_handle</i> specifies a pipe which is closed or closing.
USB_INVALID_ARGS	<i>dip</i> or <i>pipe_handle</i> arguments are NULL. USB_FLAGS_SLEEP is clear and callback is NULL.
USB_INVALID_CONTEXT	Called from interrupt context with the USB_FLAGS_SLEEP flag set.
USB_INVALID_PERM	<i>pipe_handle</i> specifies the default control pipe.
USB_FAILURE	Asynchronous resources are unavailable. In this case, USB_CB_ASYNC_REQ_FAILED is passed in as the <i>callback_flags</i> arg to the callback handler.

Exception callback handlers of interrupt-IN and isochronous-IN requests which are terminated by these commands are called with a completion reason of USB\_CR\_STOPPED\_POLLING.

Exception handlers of incomplete bulk requests are called with a completion reason of USB\_CR\_FLUSHED.

Exception handlers of unstarted requests are called with USB\_CR\_PIPE\_RESET.

Note that messages mirroring the above errors are logged to the console logfile on error. This provides status for calls which could not otherwise provide status.

**Context** May be called from user or kernel context regardless of arguments. May be called from any callback with the USB\_FLAGS\_SLEEP clear. May not be called from a callback executing in interrupt context if the USB\_FLAGS\_SLEEP flag is set.

If the `USB_CB_ASYNC_REQ_FAILED` bit is clear in `usb_cb_flags_t`, the callback, if supplied, can block because it is executing in kernel context. Otherwise the callback cannot block. Please see [usb\\_callback\\_flags\(9S\)](#) for more information on callbacks.

**Examples**

```
void post_reset_handler(
    usb_pipe_handle_t, usb_opaque_t, int, usb_cb_flags_t);

/*
 * Do an asynchronous reset on bulk_pipe.
 * Execute post_reset_handler when done.
 */
usb_pipe_reset(dip, bulk_pipe, 0, post_reset_handler, arg);

/* Do a synchronous reset on bulk_pipe. */
usb_pipe_reset(dip, bulk_pipe, USB_FLAGS_SLEEP, NULL, NULL);
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_pipe\\_bulk\\_xfer\(9F\)](#), [usb\\_pipe\\_close\(9F\)](#), [usb\\_get\\_status\(9F\)](#), [usb\\_pipe\\_ctrl\\_xfer\(9F\)](#), [usb\\_pipe\\_drain\\_reqs\(9F\)](#), [usb\\_pipe\\_get\\_state\(9F\)](#), [usb\\_pipe\\_intr\\_xfer\(9F\)](#), [usb\\_pipe\\_isoc\\_xfer\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_stop\\_intr\\_polling\(9F\)](#), [usb\\_pipe\\_stop\\_isoc\\_polling\(9F\)](#), [usb\\_callback\\_flags\(9S\)](#)

**Name** usb\_pipe\_set\_private, usb\_pipe\_get\_private – USB user-defined pipe data-field facility

**Synopsis** #include <sys/usb/usba.h>

```
int usb_pipe_set_private(usb_pipe_handle_t pipe_handle, usb_opaque_t data);
usb_opaque_t usb_pipe_get_private (usb_pipe_handle_t pipe_handle);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For usb\_pipe\_set\_private():

*pipe\_handle* Pipe handle into which user-defined data is placed.

*data* Data to store in the pipe handle.

For usb\_pipe\_get\_private():

*pipe\_handle* Pipe handle from which user-defined data is retrieved.

**Description** The usb\_set\_driver\_private() function initializes the user-private data field of the pipe referred to by *pipe\_handle*, using *data*. The user-private data field is used to store any data the client desires and is not used in any way by the USBA or OS framework. Client drivers often store their soft-state here for convenient retrieval by their callback handlers.

The usb\_get\_driver\_private() function retrieves the user-private data stored via usb\_set\_driver\_private(), from the pipe referred to by *pipe\_handle*.

**Return Values** For usb\_pipe\_set\_private():

USB\_SUCCESS Private data has been successfully stored in pipe handle.

USB\_INVALID\_PIPE *pipe\_handle* argument is NULL or invalid.

Pipe is closing or closed.

USB\_INVALID\_PERM The *pipe\_handle* argument refers to the default control pipe.

For usb\_pipe\_get\_private():

On success: usb\_opaque\_t pointer to data being retrieved.

On failure: NULL. Fails if pipe handle is NULL or invalid. Fails if pipe handle is to a pipe which is closing or closed.

**Context** May be called from user, kernel or interrupt context.

**Examples** usb\_pipe\_handle\_t pipe;

```
/* Some driver defined datatype. */
xxx_data_t *data = kmem_zalloc(...);
```

```
usb_pipe_set_private(pipe, data);
```

```
----
```

```
xxx_data_t *xxx_data_ptr = (xxx_data_t *)usb_pipe_get_private(pipe);
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_alloc\\_request\(9F\)](#)

**Name** `usb_register_hotplug_cbs`, `usb_unregister_hotplug_cbs` – Register/unregister for notification of device hotplug events

**Synopsis** `#include <sys/usb/usba.h>`

```
int usb_register_hotplug_cbs(dev_info_t *dip,
    int (*disconnection_event_handler)(dev_info_t *dip,
    int (*reconnection_event_handler)(dev_info_t *dip);

void usb_unregister_hotplug_cbs(dev_info_t *dip);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** For `usb_register_hotplug_cbs()`

*dip* Pointer to the device's `dev_info` structure.

*disconnection\_event\_handler* Called when device is disconnected. This handler takes a `dev_info_t` as an argument (representing the device being disconnected) and always returns `USB_SUCCESS`.

*reconnection\_event\_handler* Called when device is reconnected. This handler takes a `dev_info_t` as an argument (representing the device being reconnected) and always returns `USB_SUCCESS`.

For `usb_unregister_hotplug_cbs()`:

*dip* Pointer to the device's `dev_info` structure.

**Description** The `usb_register_hotplug_cbs()` function registers callbacks to be executed when the USB device represented by *dip* is hotplugged or removed.

The `usb_unregister_hotplug_cbs()` function unregisters or disengages callbacks from executing when the USB device represented by *dip* is hotplugged or removed.

**Return Values** For `usb_register_hotplug_cbs()`:

`USB_SUCCESS` Callbacks were successfully registered.

`USB_FAILURE` One or more arguments were `NULL`.

Callbacks could not be successfully registered.

For `usb_unregister_hotplug_cbs()`: None

**Context** The `usb_register_hotplug_cbs()` function may be called only from [attach\(9E\)](#).

The `usb_unregister_hotplug_cbs()` function may be called only from [detach\(9E\)](#).

Registered callback handlers requiring the use of any DDI (section 9F) function (except `ddi_taskq_*` functions), should launch a separate thread using `ddi_taskq_*` routines for processing their event, to avoid deadlocks. The new thread can then safely call any DDI function it needs to handle the event.

The registered callback handlers execute in kernel context.

### Examples

```
int remove_device(dev_info_t *)
{
    ...
    ...
    return (USB_SUCCESS);
}

int accommodate_device(dev_info_t *)
{
    ...
    ...
    return (USB_SUCCESS);
}

if (usb_register_hotplug_cbs(
    dip, remove_device, accommodate_device) == USB_FAILURE) {
    cmn_err (CE_WARN,
        "%5$d: Could not register hotplug handlers.",
        ddi_driver_name(dip), ddi_get_instance(dip));
}
```

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	PCI-based systems
Interface Stability	Committed
Availability	system/io/usb

**See Also** [attributes\(5\)](#), [attach\(9E\)](#), [detach\(9E\)](#), [usb\\_get\\_status\(9F\)](#)



**Name** `usb_reset_device` – reset a USB device according to the `reset_level`.

**Synopsis** `#include <sys/usb/usba.h`

```
int usb_reset_device (dev_info_t *dip,
                     usb_dev_reset_lvl_t reset_level);
```

**Interface Level** Solaris DDI specific (Solaris DDI)

**Parameters** *dip*

Pointer to the device's `dev_info` structure.

*reset\_level*

The level to which the device is reset. See below for a list of valid `usb_dev_reset_lvl_t` values and explanations.

**Description** The `usb_reset_device()` function provides a client driver to request a hardware reset for a USB device, which may be required in some situations such as:

- Resetting the hardware may help drivers to recover devices from an error state caused by physical or firmware defects.
- Some USB devices need the driver to upload firmware into the device's RAM and initiate a hardware reset in order to activate the new firmware.

The valid values for the *reset\_level* are:

`USB_RESET_LVL_DEFAULT`

The default reset level. The device is reset and any error status is cleared. The device state machines and registers are also cleared and need to be reinitialized in the driver. The current driver remains attached. This reset level applies to hardware error recovery, or firmware download without changing the descriptors.

`USB_RESET_LVL_REATTACH`

The device is reset. The original driver is detached and a new driver attaching process is started according to the updated compatible name. This reset level applies to the firmware download with the descriptors changing, or other situations in which the device needs to be reattached.

The `usb_reset_device()` function creates a new helper thread for reattachment. When called from [attach\(9E\)](#), the new thread sets a timer (1 second), and waits until the driver's [attach\(9E\)](#) completes, after which the thread attempts to reattach the driver. When not called from [attach\(9E\)](#), the new thread attempts to reattach the driver immediately.

If the thread fails to reattach to the driver, an error message is printed in system log with the detailed reason. The driver returns to a stable state, depending on where the failure occurred.

**Return Values** The return values for the `usb_reset_device()` function are:

<code>USB_SUCCESS</code>	If <code>USB_RESET_LVL_DEFAULT</code> is specified, the device was reset successfully. If <code>USB_RESET_LVL_REATTACH</code> is specified, reattaching was started successfully or a previous reset is still in progress.
<code>USB_FAILURE</code>	The state of the device's parent hub is invalid (disconnected or suspended). This is called when the driver being detached. If <code>USB_RESET_LVL_DEFAULT</code> is specified, the device failed to be reset. If <code>USB_RESET_LVL_REATTACH</code> is specified, reattaching failed to start.
<code>USB_INVALID_ARGS</code>	Invalid arguments.
<code>USB_INVALID_PERM</code>	The driver of the dip does not own the entire device.
<code>USB_BUSY</code>	If <code>USB_RESET_LVL_DEFAULT</code> is specified, one or more pipes other than the default control pipe are open on the device.
<code>USB_INVALID_CONTEXT</code>	If <code>USB_RESET_LVL_DEFAULT</code> is specified, called from interrupt context

**Examples** **EXAMPLE 1** Resetting a Device

The following example shows how a device is reset to recover it from an error state:

```
xxx_configure()
{
    xxx_set_parameter1();

    if (xxx_set_parameter2() == USB_FAILURE) {

        /* Close all the opened pipes except the default pipe */
        xxx_close_all_opened_pipes();

        /* Reset the device */
        rval = usb_reset_device(dip, USB_RESET_LVL_DEFAULT);
        if (rval == USB_SUCCESS) {
            /* Re-configure the device */
            xxx_set_parameter1();
            xxx_set_parameter2();

            /* Open the closed pipes if needed */
            ...
        } else {
            /* Failed to reset the device, check the return value for
             further process */
            ...
        }
    }
}
```

**EXAMPLE 1** Resetting a Device *(Continued)*

}

**EXAMPLE 2** Resetting a Device After Downloading Firmware

The following example shows how a device is reset after downloading firmware. The device's descriptors change after the reset:

```
static int xxx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ...
    /* Download the firmware to the RAM of the device */
    if (firmware_download() == USB_SUCCESS) {
        /* Reset the device and re-enumerate it */
        rval = usb_reset_device(dip, USB_RESET_LVL_REATTACH);
        if (rval == USB_SUCCESS) {
            /* The re-enumeration has been started up, just return */
            return (DDI_SUCCESS);
        } else {
            /* Failed to start the re-enumeration, check the return value
             * for further process*/
            ...
            return (DDI_FAILURE);
        }
    }
}
```

**Context** The `usb_reset_device()` function may be called from user or kernel context.

**Attributes** See [attributes\(5\)](#) for descriptions of the following attributes:

ATTRIBUTETYPE	ATTRIBUTEVALUE
Architecture	PCI-based systems
Availability	system/io/usb
Interface Stability	Committed

**See Also** [attributes\(5\)](#), [usb\\_clr\\_feature\(9F\)](#), [usb\\_get\\_cfg\(9F\)](#), [usb\\_pipe\\_close\(9F\)](#), [usb\\_pipe\\_open\(9F\)](#), [usb\\_pipe\\_reset\(9F\)](#),

**Diagnostics** The messages described below may appear on the system console as well as being logged. All messages are formatted in the following manner:

```
WARNING: dev_path hubdinstance_num driver_name Error message ...
```

*driver\_name instance\_num* is under bus power management, cannot be reset. Please disconnect and reconnect this device.

Bus power management is in process when calling the reset function, the device failed to be reset and stayed in the former state. Please disconnect and reconnect it to system.

Time out when resetting the device *driver\_name instance\_num*. Please disconnect and reconnect it to system.

The parent hub/root hub of this device is busy now. The device failed to be reset and stayed in the former state. Please disconnect the device, wait for a while and reconnect it to system.

*driver\_name instance\_num* cannot be reset due to other applications are using it, please first close these applications, then disconnect and reconnect the device.

The device is using by other applications, the related driver failed to be detached. Please make sure to close these applications before calling the reset function.

**Notes** Always close all applications before resetting a device.

- Name** uwritec – remove a character from a uio structure
- Synopsis** #include <sys/uio.h>
- ```
int uwritec(uio_t *uio_p);
```
- Interface Level** Architecture independent level 1 (DDI/DKI)
- Parameters** *uio\_p* Pointer to the [uio\(9S\)](#) structure
- Description** The `uwritec()` function returns a character from the uio structure pointed to by *uio\_p* and updates the uio structure. See [uio\(9S\)](#).
- Return Values** The next character for processing is returned on success, and -1 is returned if uio is empty or if there is an error.
- Context** The `uwritec()` function can be called from user, interrupt, or kernel context.
- See Also** [uio\(9S\)](#), [uio\(9S\)](#), [uio\(9S\)](#), [uio\(9S\)](#)
- Writing Device Drivers*

**Name** va\_arg, va\_start, va\_copy, va\_end – handle variable argument list

**Synopsis** #include <sys/varargs.h>

```
void va_start(va_list pvar, name);
(type *) va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

### Parameters

|            |             |                                                               |
|------------|-------------|---------------------------------------------------------------|
| va_start() | <i>pvar</i> | Pointer to variable argument list.                            |
|            | <i>name</i> | Identifier of rightmost parameter in the function definition. |
| va_arg()   | <i>pvar</i> | Pointer to variable argument list.                            |
|            | <i>type</i> | Type name of the next argument to be returned.                |
| va_copy()  | <i>dest</i> | Destination variable argument list.                           |
|            | <i>src</i>  | Source variable argument list.                                |
| va_end()   | <i>pvar</i> | Pointer to variable argument list.                            |

**Description** This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists but do not use the `varargs` macros are inherently non-portable, as different machines use different argument-passing conventions. Routines that accept a variable argument list can use these macros to traverse the list.

`va_list` is the type defined for the variable used to traverse the list of arguments.

`va_start()` is called to initialize `pvar` to the beginning of the variable argument list. `va_start()` must be invoked before any access to the unnamed arguments. The parameter `name` is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the “, . . .”). If this parameter is declared with the `register` storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

`va_arg()` expands to an expression that has the type and value of the next argument in the call. The parameter `pvar` must be initialized by `va_start()`. Each invocation of `va_arg()` modifies `pvar` so that the values of successive arguments are returned in turn. The parameter `type` is the type name of the next argument to be returned. The type name must be specified in such a way that the type of pointer to an object that has the specified type can be obtained by

postfixing a *\** to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The `va_copy()` macro saves the state represented by the `va_listsrc` in the `va_list dest`. The `va_list` passed as *dest* should not be initialized by a previous call to `va_start()`. It then must be passed to `va_end()` before being reused as a parameter to `va_start()` or as the *dest* parameter of a subsequent call to `va_copy()`. The behavior is undefined if any of these restrictions are not met.

The `va_end()` macro is used to clean up. It invalidates *pvar* for use (unless `va_start()` is invoked again).

Multiple traversals, each bracketed by a call to `va_start()` and `va_end()`, are possible.

### Examples **EXAMPLE 1** Creating a Variable Length Command

The following example uses these routines to create a variable length command. This might be useful for a device that provides for a variable-length command set. `ncmdbytes` is the number of bytes in the command. The new command is written to `cmdp`.

```
static void
xx_write_cmd(uchar_t *cmdp, int ncmdbytes, ...)
{
    va_list ap;
    int i;

    /*
     * Write variable-length command to destination
     */
    va_start(ap, ncmdbytes);
    for (i = 0; i < ncmdbytes; i++) {
        *cmdp++ = va_arg(ap, uchar_t);
    }
    va_end(ap);
}
```

**See Also** [vcmn\\_err\(9F\)](#), [vsprintf\(9F\)](#)

**Notes** It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame.

Specifying a second argument of `char` or `short` to `va_arg` makes your code non-portable, because arguments seen by the called function are not `char` or `short`. C converts `char` and `short` arguments to `int` before passing them to a function.

**Name** vsprintf – format characters in memory

**Synopsis** #include <sys/varargs.h>  
#include <sys/ddi.h>  
#include <sys/sunddi.h>

```
char *vsprintf(char *buf, const char *fmt, va_list ap);
```

**Interface Level** Solaris DDI specific (Solaris DDI).

**Parameters** *buf* Pointer to a character string.  
*fmt* Pointer to a character string.  
*ap* Pointer to a variable argument list.

**Description** `vsprintf()` builds a string in *buf* under the control of the format *fmt*. The format is a character string with either plain characters, which are simply copied into *buf*, or conversion specifications, each of which converts zero or more arguments, again copied into *buf*. The results are unpredictable if there are insufficient arguments for the format; excess arguments are simply ignored. It is the user's responsibility to ensure that enough storage is available for *buf*.

*ap* contains the list of arguments used by the conversion specifications in *fmt*. *ap* is a variable argument list and must be initialized by calling `va_start(9F)`. `va_end(9F)` is used to clean up and must be called after each traversal of the list. Multiple traversals of the argument list, each bracketed by `va_start(9F)` and `va_end(9F)`, are possible.

Each conversion specification is introduced by the % character, after which the following appear in sequence:

An optional decimal digit specifying a minimum field width for numeric conversion. The converted value will be right-justified and padded with leading zeroes if it has fewer characters than the minimum.

An optional `l` (`ll`) specifying that a following `d`, `D`, `o`, `O`, `x`, `X`, or `u` conversion character applies to a long (`long long`) integer argument. An `l` (`ll`) before any other conversion character is ignored.

A character indicating the type of conversion to be applied:

`d,D,o,O,x,X,u` The integer argument is converted to signed decimal (`d`, `D`), unsigned octal (`o`, `O`), unsigned hexadecimal (`x`, `X`) or unsigned decimal (`u`), respectively, and copied. The letters `abcdef` are used for `x` conversion. The letters `ABCDEF` are used for `X` conversion.

`c` The character value of the argument is copied.



- b** This conversion uses two additional arguments. The first is an integer, and is converted according to the base specified in the second argument. The second argument is a character string in the form `<base>[<arg> . . . ]`. The base supplies the conversion base for the first argument as a binary value; `\10` gives octal, `\20` gives hexadecimal. Each subsequent `<arg>` is a sequence of characters, the first of which is the bit number to be tested, and subsequent characters, up to the next bit number or terminating null, supply the name of the bit.
- A bit number is a binary-valued character in the range 1-32. For each bit set in the first argument, and named in the second argument, the bit names are copied, separated by commas, and bracketed by `<` and `>`. Thus, the following function call would generate `reg=3<BitTwo, BitOne>\n` in *buf*.
- ```
vsprintf(buf, "reg=%b\n", 3, "\10\2BitTwo\1BitOne")
```
- s** The argument is taken to be a string (character pointer), and characters from the string are copied until a null character is encountered. If the character pointer is NULL on SPARC, the string `<nullstring>` is used in its place; on x86, it is undefined.
- %** Copy a %; no argument is converted.

**Return Values** `vsprintf()` returns its first parameter, *buf*.

**Context** `vsprintf()` can be called from user, kernel, or interrupt context.

**Examples** EXAMPLE 1 Using `vsprintf()`

In this example, `xxerror()` accepts a pointer to a `dev_info_t` structure *dip*, an error level *level*, a format *fmt*, and a variable number of arguments. The routine uses `vsprintf()` to format the error message in *buf*. Note that `va_start(9F)` and `va_end(9F)` bracket the call to `vsprintf()`. *instance*, *level*, *name*, and *buf* are then passed to `cmn_err(9F)`.

```
#include <sys/varargs.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#define MAX_MSG 256

void
xxerror(dev_info_t *dip, int level, const char *fmt, . . . )
{
    va_list      ap;
    int          instance;
    char         buf[MAX_MSG],
                *name;

    instance = ddi_get_instance(dip);
```

**EXAMPLE 1** Using `vsprintf()` *(Continued)*

```
name = ddi_binding_name(dip);

/* format buf using fmt and arguments contained in ap */
va_start(ap, fmt);
vsprintf(buf, fmt, ap);
va_end(ap);

/* pass formatted string to cmn_err(9F) */
cmn_err(level, "%s%d: %s", name, instance, buf);
}
```

**See Also** [cmn\\_err\(9F\)](#), [ddi\\_binding\\_name\(9F\)](#), [ddi\\_get\\_instance\(9F\)](#), [va\\_arg\(9F\)](#)

*Writing Device Drivers*

- 
- Name** WR, wr – get pointer to the write queue for this module or driver
- Synopsis** `#include <sys/stream.h>`  
`#include <sys/ddi.h>`
- ```
queue_t *WR(queue_t *q);
```
- Interface Level** Architecture independent level 1 (DDI/DKI).
- Parameters** *q* Pointer to the *read* queue whose *write* queue is to be returned.
- Description** The WR() function accepts a *read* queue pointer as an argument and returns a pointer to the *write* queue of the same module.
- CAUTION: Make sure the argument to this function is a pointer to a *read* queue. WR() will not check for queue type, and a system panic could result if the pointer is not to a *read* queue.
- Return Values** The pointer to the *write* queue.
- Context** The WR() function can be called from user, interrupt, or kernel context.
- Examples** EXAMPLE 1 UsingWR()
- In a STREAMS [close\(9E\)](#) routine, the driver or module is passed a pointer to the *read* queue. These usually are set to the address of the module-specific data structure for the minor device.
- ```
1 xxxclose(q, flag)
2     queue_t *q;
3     int flag;
4     {
5         q->q_ptr = NULL;
6         WR(q)->q_ptr = NULL;
7     }
```
- See Also** [close\(9E\)](#), [OTHERQ\(9F\)](#), [RD\(9F\)](#)
- Writing Device Drivers*
- STREAMS Programming Guide*

