Appendix C

Device Driver Migration Notes

This appendix explains how to make an IRIX 4.x device driver compliant with the IRIX 5.3 and 6.0 environments.

This appendix contains the following sections:

Introduction

Intended Audience and Prerequisites

This appendix is intended to aid in the planning of device driver migration from previous versions of IRIX to the IRIX 5.3 and IRIX 6.0 environments. It assumes you are familiar with the existing material on developing IRIX device drivers.

Background

Before you upgrade a perfectly good device driver that runs on an older version of IRIX, compute the time it will take to rewrite your driver to run under the new operating systems against the actual performance improvement you hope to achieve. In some cases, it may be to your advantage to continue using an older driver by recompiling its source code under the new operating system. For those cases where it is to your advantage to modify an existing driver, the following notes are provided.

IRIX 5.x

IRIX 5.x is a binary-compatible upgrade to IRIX 4.0.x at the user program level. However, due to the extensive internal changes to the IRIX kernel, device drivers and other kernel components, such as STREAMS modules and file systems, must be revised and recompiled for the IRIX 5.x environment. It is possible that changes to IRIX 5.3 will become effective after the publication of this manual.

Most of the changes in the IRIX kernel result from supporting the SVR4 interfaces for both applications programs and for certain kernel internal interfaces. The IRIX 5.x kernel interfaces are:

- SVR4 Device Driver Interface/Driver Kernel Interface (SVR4 DDI/DKI) described in the *IRIX Device Driver Reference Pages.* The IRIX 5.x operating system implementation uses a multiprocessor version of DDI/DKI developed for Silicon Graphics platforms.
- SVR4 STREAMS Interface, which is documented in the *UNIX® System V Release 4 STREAMS Programming Guide*.
- IRIX 5.x Data Link Provider Interface (DLPI) (relevant to STREAMS protocol stacks and to network drivers). This standard interface, defined by IEEE Standard 802.3, permits flexible integration protocol stacks and their “peaceful coexistence” with the TCP/IP stack. Protocol stacks register themselves with DLPI, as do the network drivers present in the system. The new protocol stack is thus isolated from the specific network hardware support present in the system.

IRIX 6.0

IRIX 6.0 is the 64-bit operating system for all Silicon Graphics systems that use MIPS R8000 series microprocessors. Because the IRIX 6.0 kernel is itself a 64-bit object, device drivers must be ported to the 64-bit operating system.

Migration Overview

The migration of drivers from an IRIX 4.0.5 environment to an IRIX 5.x environment is straightforward. The migration of several typical IRIX drivers required changes to less than 10% of the source lines. Most of these changes were in declarations. Naturally, extremely complex drivers that reached into the IRIX kernel to access services not typically employed in device drivers will have to change, as many of these interfaces have been replaced or altered. In any case, *you* are responsible for getting your device drivers to work across operating system and platform upgrades.

The changes to drivers fall in a number of broad categories:

- Changes in drivers that are a result of changes in the SVR4 API (Application Programmatic Interface) as compared to the SVR3 API.
- Changes to the declaration of the DKI interface, such as additional arguments to the driver calling interfaces and, in some cases, different procedure typing (mostly to make them void).
- Changes to use the DDI interface instead of the now obsolete IRIX 4.0.x interfaces for kernel service invocation. This may include changes that result from the changes in the semantics of the DDI. There are also some SGI interfaces defined for services omitted by DDI, such as cache flushing and address map setup.
- Changes to accommodate the architecture of the CHALLENGE/Onyx family, especially in the richness of its bus structure.
- Changes to accommodate the architecture of the Indigo^2, notably the EISA bus. Since this is the first Silicon Graphics system to use this bus, there are no migration issues.
- Changes in network device drivers and protocol stacks to make use of the Data Link Provider Interface (DLPI). See the **dlpi**(7) man page.
- Changes to take advantage of the IRIX 5.x dynamically loadable kernel module support. This permits device drivers to be loaded into a running system without rebooting. This is discussed in the **mload**(4) man pages.
- Changes to the SCSI driver interface to unify all supported SCSI controllers. See Chapter 5, “Writing a SCSI Device Driver,” for details.

IRIX 4.x to 5.x Migration

This section serves as a preliminary guide to planning the migration of device drivers from the IRIX 4.0.x environment to IRIX 5.x. The discussion below assumes that you already have the device driver working in IRIX 4.0.x, and need only to migrate it to IRIX 5.x.

SVR4 API Changes

The SVR4 API changes the size of a number of system structures to accommodate the changes in Expanded Fundamental Types. (Briefly, a number of fields, such as the user and group ID and device numbers are changed from 16 to 32 bits.) For driver work, the most important impact of these API changes is in the access to the major and minor numbers. The device number is now a 32-bit quantity, by changing the underlying type of the *typedef dev_t*. Access to the major and minor numbers can no longer be done with the 8-bit shift-and-mask technique common in prior versions of UNIX. (This is especially true as the split of the 32 bits between major and minor is a "hidden" parameter.) DDI defines a series of functions/macros to provide access to the major and minor numbers.

STREAMS Changes

For STREAMS, there are the standard SVR3 (4.x) to SVR4 (5.x) STREAMS changes, most notably in the arguments to the driver's **open** routine. For further details, please refer to *STREAMS Modules and Drivers*, UNIX SVR4.2. UNIX Press, 1992.

DDI Changes

The *IRIX Device Driver Reference Pages* documents the routines that DDI supports. Generally, these have analogs to routines supported in prior versions of IRIX, but the syntax and semantics may differ, and the driver may need to have minor logic updates to accommodate these changes. In some drivers, the changes to support DDI simplify the driver by invoking common service interfaces in DDI instead of performing the work in the driver.

As it stands, DDI is insufficient to write a driver using *only* the routines in DDI. The lacks are in areas specific to a particular architecture or family, such as cache flushing and device register addressability in complex bus architectures. SGI has defined a number of additional service interfaces to support these needs.

DKI Changes

The changes from the new DKI affect the driver routines called by the rest of the IRIX kernel, both the **devsw* entry points and the interrupt handlers. The details of these interfaces are described in the *IRIX Device Driver Reference Pages* . The primary changes are:

- Device numbers are passed as a *dev_t* rather than as an *int* in prior IRIX versions. As mentioned elsewhere, the DDI contains new functions to access the major and minor numbers. Note that *open* takes a *dev_t **, while **read** and **write** take a *dev_t* (no indirection).
- Many of the entry points (**open**, **close**, **read**, **write**) take a new argument that is an opaque credentials structure. (It is opaque in the sense that the device driver never examines the internal makeup of the structure.) This is used to call the DDI **drv_priv**() routine, which takes the place of checking *u.u_uid == 0*. This permits different models of privileges to be used, for example, in a trusted system.
- The **read** and **write** routines take a *uio_t **, a pointer to a structure that defines the addresses and lengths of the data in the user space. (The *IRIX Device Driver Reference Pages* describe this structure. It is defined in */usr/include/sys/uio.h*.) Previously, this information was computed in the driver from the *u* vector. The driver typically passes this structure to a DMA setup routine such as the DDI **uiophysio**() or the Silicon Graphics enhancement, **biophysio**(), which is nearly identical to **iophysio**(), except that it takes a block number instead of a file offset).
- Generally, drivers can no longer access the *u* vector directly. DDI defines access routines for fields in the *u* vector. Direct access to the *u* vector tends to make a driver more dependent on specific aspects of the system than is desirable. This also extends to access *u.u_error*. DKI expect to return the error value (0 indicates no error), and are not supposed to set *u.u_error*.
- Interrupt routines are now of type *void*.

Addressability

The driver needs to arrange for addressability of the device registers in the CHALLENGE/Onyx family and, in the interest of having common code for VME drivers between the POWER Series™ and these systems, drivers must set up the appropriate mappings. This is done by the new **pio_map**() routines, which have similar calling interfaces to the **dma_map**() routines from prior versions.

CHALLENGE/Onyx Family Support

The CHALLENGE/Onyx family architecture has added new features and restrictions on the drivers, configuration, and other tools that could not be dealt with by the existing software framework. Supporting these systems with the common source used for all Silicon Graphics products necessitated changes in the way kernels are configured and support in for the new hardware features.

Some of the CHALLENGE series architecture differences include:

- Multiple types of bus support (such as VME, GIO, SCSI).
- VME buses are not automatically mapped into known K2 segment addresses.
- VME-bus controllers cannot access physical memory directly.
- Only a portion of the VME-bus A32 space can be mapped into the kernel at any one time.

These differences cause the following software changes:

1. The need for a general mechanism for probing for devices on multiple and different types of buses caused changes to the VECTOR line to specify which type of bus, which bus of that type, which address space on that particular bus, which address on that bus, and so on. Since there are no automatic K2 addresses, the addresses are abstracted and more meaningful than they were in the past. Different buses require different types of information, as well. VME has interrupts specifying both a

vector and IPL, whereas EISA does not. So, a bus-specific portion was added to the *edt* structure. This caused the need for new VECTOR line probing specifications, as well.

2. Since there are no known K2 addresses that correspond to VME-bus locations, user-level VME drivers can no longer be performed through */dev/mmem*, but instead, they now have their own */dev/vme* device interface. This also affects kernel drivers. Kernel drivers must map in the address space of their controllers before they can access them. This is done with special pio_mapping routines, similar to existing DMA mapping routines. This is further complicated by the fact that, for VME A32 space, pio_mapping registers are a limited resource. Only 13*8 MB can be mapped into the kernel at any one time; so 12 of these registers can be locked down and the thirteenth used as a floater to be shared by whoever needs it, from drivers to ***lboot***.

   Note: A32 pio_mappings start and end on 8 MB boundaries. The address you map plus the length of the mapping must not cross an 8 MB boundary.

3. Since VME-bus controllers cannot access physical memory directly, you can no longer pass the controller a K0 address and expect it to work. EVERYTHING has to be *DMA* mapped. For example, in the past, IOPB addresses were normally converted to K0 addresses and passed to the controller. Now the IOPBs must be *DMA* mapped.

4. Some new functionality has been added to the VME driver interface. Since most VME boards can have their IRQ vector and ipl programmed through software, it is now possible to allocate VME vectors dynamically, so they need not be specified on the VECTOR line. This frees you from worrying about finding one that is not already in use. Simply call **vme_ivec_alloc**() to allocate a free vector, then call **vme_ivec_set**() to register your interrupt routine.

5. The driver interface now uses the SVR4 MP DDI/DKI interface except for the Silicon Graphics-specific routines, such as **pio_map**() and **dma_map**(). For example, entry points such as **open**, **close**, **read**, and **write** all have slightly different arguments and, in some cases, different procedure types, in 5.x than they had in earlier versions.

6. The DDI/DKI *drvrflags* variable, not the flag in the *master.d* file, is used for the driver to indicate that it is MP-safe.

Kernel Configuration Issues

The highly flexible architecture of the CHALLENGE family requires extensions to the descriptions of devices in the IRIX kernel configuration process, specifically to the VECTOR lines in the system configuration files. The new VECTOR line for VME would look like the following:

```
VECTOR: bustype=VME module=jag ipl=1 ctlr=0 adapter=0
iospace=(A16S, 0, 0x800) probe_space=(A16S, 0, 1)
```

Note: The VECTOR line is still all one line. It is broken here to fit on the page.

New fields are *bustype*, which in this case is *VME*. The *ctlr* field takes the place of *unit* in 4.0.x. The *adapter* field specifies which VME bus. (CHALLENGE systems can be configured to have multiple VME buses).

The *iospace* triple is used to pass in the address of the controller. The first argument defines the address space. Valid values are *A16S*, *A16NP*, *A24S*, *A24NP*, *A32S*, *A32NP*, *A64S*, *A64NP*. The second argument is the address with the specified address space. The third argument is the length of the mapping. The *probe_space* line performs a **badaddr** like function (that is, it tries to read the specified address and catch any errors) on the specified address. In this case, the arguments are the address space, address within that space, and the size of the read.

There is also an *exprobe_space* extended probe space defined as follows:

```
exprobe_space=(r,A16S,0,2,0xfdd1,0xffff)
```

The difference between it and the 4.0.x version of *exprobe* is in the specification of the address to test.

Notice that no vector is specified. The old *vector*= primitive is still supported for boards that are jumpered. Generally, drivers would use the **vme_ivec_alloc**() and **vme_ivec_set**() routines to allocate and set the vector.

Equipped Device Table (EDT) Changes

The form of the structures for the Equipped Device Table (EDT) have changed for the same reasons as the VECTOR line in the configuration (the VECTOR information is used as initialization values for the EDT entries).

The following information is from */usr/include/sys/edt.h*:

```
#define NBASE 3
typedef unsigned long iopaddr_t;
typedef struct iospace {
   unchar    ios_type;     /* io space type on the adapter */
   iopaddr_t ios_iopaddr;  /* io space base address */
   ulong     ios_size;
   caddr_t   ios_vaddr;    /* kernel virtual address */
} iospace_t;

typedef struct edt {
   uint_t    e_bus_type;   /* vme, scsi, eisa... */
   unchar    v_cpuintr;    /* cpu to send intr to */
   unchar    v_setcpuintr; /* cpu field is valid */
   uint_t    e_adap;       /* adapter */
   uint_t    e_ctlr;       /* controller identifier */
   void*     e_bus_info;   /* bus-dependent info */
   int       (*e_init)();  /* device init/run-time probe */
   iospace_t e_space[NBASE];
} edt_t;

#define    e_base e_space[0].ios_vaddr
#define    e_base2 e_space[1].ios_vaddr
#define    e_base3 e_space[2].ios_vaddr
#define    e_iobase e_space[0].ios_iopaddr
#define    e_iobase2 e_space[1].ios_iopaddr
#define    e_iobase3 e_space[2].ios_iopaddr

The e_bus_info field points to the following structure:

typedef struct vme_intrs {
   int (*v_vintr)();        /* interrupt routine */
   unsigned    char v_vec;  /* vme vector */
   unsigned    char v_brl;  /* interrupt priority level */
   unsigned    char v_unit; /* software identifier */
} vme_intrs_t;
```

The following fragment illustrates an *edt***init** routine using the new structures and the new **pio_*** routines.

```
mydrvredtinit(struct edt *e)
{
   piomap_t *piomap;
```

```
   vme_intrs_t *intrs = e->e_bus_info;
[...]
 piomap = pio_mapalloc(e->e_bus_type, e->e_adap,
   &e->e_space[0], PIOMAP_FIXED, ”mydrvr”);

 if (piomap == 0) {
      /* This could fail because the adapter isn’t valid
       * or invalid addresses or there are no more
       * fixed mappings available in the case of A32.
       */
      printf(“mydrvr not installed\n”);
      return;
   }

 e->e_base = pio_mapaddr(piomap, e->e_iobase);

   /* You can now use e->e_base as a normal address
    * to access your controller.
    */
[...]
/* Now allocate a VME IRQ vector and register
    * the interrupt routine.
    */
 ipl = intrs->v_brl;


 vec = vme_ivec_alloc(e->e_adap);

 if (vec == -1) {
 cmn_err(CE_WARN,”mydrvredtinit: no interrupt vector\n”);
 pio_mapfree(piomap);
 return;
 }
 vme_ivec_set(e->e_adap, vec, mydrvrintr, e->e_ctlr);
[...]
}
```

IRIX 5.2 to 5.3 Migration

Note: This information is preliminary and subject to change. While likely to be correct, it is based on extracted **diff** listings of actual drivers migrating between 5.2 and 5.3, so there may be errors or omissions. Please use the information with caution.

All IRIX 5.2 kernel components, including drivers and STREAMS modules, require some amount of work to migrate to IRIX 5.3. In the simplest cases, no source changes are required, but changes in the size of certain kernel structures make it necessary to recompile the same source in a 5.3 build environment.

In certain device drivers, three types of changes from the IRIX 5.2 kernel to the IRIX 5.3 kernel require some work:

1. Support for a multi-threaded version of TCP/IP.

   This requires change in network drivers to use the new blocking scheme instead of the older **splnet**() blocking scheme.

2. Incorporation of hooks for Trusted IRIX/B to allow the use of a least privilege model.

   The changes in drivers are to replace any calls to **suser**() or explicit checks of **u.u_uid == 0** with a corresponding call to one of the capability check routines.

3. **mbuf** management scheme.

   The **mbuf** pool is now initialized early in the system life, so drivers do not need to call **mbinit**().

Standard Device Drivers

The simplest migration is for standard (non-network and non-STREAMS) device drivers. Generally, these drivers require only recompilation, and do not require source changes. If these drivers check for root to protect privileged operations, however, the Trusted IRIX capabilities mechanism requires changes similar to the following fragment:

Note: If Trusted IRIX is not installed, these map to a stub that does a simple **u.u_uid == 0** check.

```
*** 30,35 ****
--- 30,36 ----
  #include "sys/pio.h"
  #include "sys/strsubr.h"
  #include "sys/ddi.h"
+ #include "sys/capability.h"

  extern time_t lbolt;

*** 1447,1453 ****
     * reasons.*/
    if (((cdp->cd_cflag & CLOCAL) ^ (cflag & CLOCAL)) &&
!       !suser()) {
            u.u_error = 0;  /* XXXrs */
            cflag &= ~CLOCAL;
            cflag |= cdp->cd_cflag & CLOCAL;
--- 1448,1454 ----
     * reasons.*/
    if (((cdp->cd_cflag & CLOCAL) ^ (cflag & CLOCAL)) &&
!       !_CAP_ABLE(CAP_DEVICE_MGT)) {
            u.u_error = 0;  /* XXXrs */
            cflag &= ~CLOCAL;
            cflag |= cdp->cd_cflag & CLOCAL;
```

Refer to the file *sys/capability.h* for a list of all capabilities.

STREAMS Modules

STREAMS modules and drivers resemble standard drivers in their modification requirements. In general, these need only to be recompiled. A STREAMS module should not, however, contain privilege checks because it does not have a valid user context in which to make them.

ifnet Drivers

ifnet-based networking device drivers that queue or dequeue packets on the **ipintrq** or **if_snd** queue must be modified to use the appropriate **IFNET_LOCK** macro. Definitions of the new locking macros are in *net/if.h* in an IRIX 5.3 system. Refer to Chapter 9, "Writing Network Device Drivers," for more information.

Context diff of Token Ring ifnet Driver

The following fragment illustrates the changes made in one such driver (this is for the token ring, a driver that was otherwise unchanged during these modifications).

```
>>> add include file for capabilities.
*** 105,110 ****
--- 105,111 ----
  #endif
  #include "sys/dlsap_register.h"
  #endif    /* _IRIX4 */
+ #include "sys/capability.h"

  #ifdef QDBG
  struct tr_qs fvqs;
```

Board Initialization

This is in the board initialization routine, where the interface lock has been set by the caller.

```
*** 547,552 ****
--- 548,554 ----
  {
      struct fv_info *fv = &fv_info[unit];
      struct ifnet *ifp = &fv->fv_if;
+     ASSERT(IFNET_ISLOCKED(ifp));

      if ((ifp->if_flags & IFF_RUNNING) != 0) {
          DP(("if_fvinit%d: already running\n", fv-
>fv_unit));

>>> also in the init routine.  Release the interface lock
>>> before sleeping, reacquire it after the sleep returns.
*** 553,560 ****
          return(0);
      }

!     if (fv->fv_state < FV_STATE_OK)
          sleep((caddr_t)&fv->fv_state, PZERO|PCATCH);
      if (fv->fv_state != FV_STATE_OK) {
          return(EIO);
      }
```

```
--- 555,565 ----
        return(0);
    }

!   if (fv->fv_state < FV_STATE_OK) {
!       IFNET_UNLOCKNOSPL(ifp);
        sleep((caddr_t)&fv->fv_state, PZERO|PCATCH);
+       IFNET_LOCKNOSPL(ifp);
+   }
    if (fv->fv_state != FV_STATE_OK) {
        return(EIO);
    }

>>> this is in the driver's ioctl routine. ASSERT that the
>>> ioctl routine was called with interface lock held.
*** 894,899 ****
--- 899,905 ----
    struct fv_info *fv = &fv_info[ifp->if_unit];

    ASSERT(&fv->fv_ac == (struct arpcom*)ifp);
+   ASSERT(IFNET_ISLOCKED(ifp));
    switch (cmd) {
    case SIOCSIFADDR: {
        struct ifaddr *ifa = (struct ifaddr *)data;

>>> TrIRIX change for privilege check.
*** 1167,1173 ****
    case SIOC_TR_ARM: {
        TR_SIOC *sioc = (TR_SIOC*)data;

!       if (!suser()) {
            error = EPERM;
            break;
        }
--- 1173,1179 ----
    case SIOC_TR_ARM: {
        TR_SIOC *sioc = (TR_SIOC*)data;

!       if (!_CAP_ABLE(CAP_NETWORK_MGT)) {
            error = EPERM;
            break;
        }
```

Interrupt Handler

This is in the interrupt handler. Acquire the interface lock. *fv* is a unit info structure pointer, one element of which is the pointer to **struct ifnet**, the interface structure shared with IP.

```
*** 1231,1236 ****
--- 1237,1243 ----
          printf("fv%d: early interrupt\n", unit);
          goto fvintr_ret;
      }
+     IFNET_LOCKNOSPL(&fv->fv_if);
      QDBGUP(fvints.tot,1);
      mem = fv->fv_mem;
  iloop:

 More of the interrupt handler.  Free the lock as we exit.
*** 1237,1242 ****
--- 1244,1250 ----
      /* get the type of interrupt */
      cmdsts = io->sifcmd_stat;
      if ((cmdsts&TR_STAT_INTR) == 0) {
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          goto fvintr_ret;
      }
      found++;

>>> still more of the interrupt handler.
*** 1300,1305 ****
--- 1308,1314 ----
          fv->fv_state = FV_STATE_SICK;
          }
          QDBGUP(fvints.buf,1);
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          goto fvintr_ret;

      case TR_INT_SCB_CLEAR:

>>> frame receive handler.  Lock the IP input queue to add a
packet to it.
*** 2031,2044 ****

      switch (port) {
      case ETHERTYPE_IP:
-         schednetisr(NETISR_IP);
```

```
          ifq = &ipintrq;
                 if (IF_QFULL(ifq)) {
              IF_DROP(ifq);
              fv->fv_if.if_iqdrops++;
              goto drop;
          }
!                IF_ENQUEUE(ifq, m0);
          goto read_ret;
      case ETHERTYPE_ARP:
          if (sri) {
--- 2041,2057 ----

      switch (port) {
      case ETHERTYPE_IP:
          ifq = &ipintrq;
+         IFQ_LOCK(ifq);
                 if (IF_QFULL(ifq)) {
              IF_DROP(ifq);
              fv->fv_if.if_iqdrops++;
+             IFQ_UNLOCK(ifq);
              goto drop;
          }
!         IF_ENQUEUE_NOLOCK(ifq, m0);
!         IFQ_UNLOCK(ifq);
!         schednetisr(NETISR_IP);
          goto read_ret;
      case ETHERTYPE_ARP:
          if (sri) {

>>> Output routine, assert caller has if structure
>>> locked for us.
*** 2269,2274 ****
--- 2282,2288 ----
      ASSERT((ifp->if_unit >= 0) && (ifp->if_unit <
FV_MAXBD));
      fv = &fv_info[ifp->if_unit];
      ASSERT(0 != fv->FVIO && ifp == &fv->fv_if);
+     ASSERT(IFNET_ISLOCKED(ifp));

      /* 2: make sure board has been initialized properly */
      if (fv->fv_state != FV_STATE_OK || iff_dead(ifp-
>if_flags)) {

>>> close routine.  ASSERT caller locked interface,
>>> release and reacquire lock around sleep.
```

```
*** 3750,3761 ****
--- 3765,3780 ----
      FVMEM *mem = fv->fv_mem;

      DP((“fv%d: close%\n”, fv->fv_unit));
+     ASSERT(IFNET_ISLOCKED(&fv->fv_if));
+
      if (fv->fv_state != FV_STATE_OK)
          goto close_ret;

      while ((fv->cmd_Flags[TR_CMD_CLOSE]&CMD_BUSY) != 0) {
          fv->cmd_Flags[TR_CMD_CLOSE] |= CMD_PENDING;
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          sleep((caddr_t)&fv-
>cmd_Flags[TR_CMD_CLOSE],PZERO|PCATCH);
+         IFNET_LOCKNOSPL(&fv->fv_if);
      }
      fv->cmd_Flags[TR_CMD_CLOSE] |= CMD_BUSY;


>>> later in the close routine, another release/reacquire
    around sleep.
*** 3776,3782 ****
--- 3795,3803 ----
      io->sifcmd_stat = TR_CMD_INT_ADAPT | TR_CMD_EXECUTE |
                  TR_CMD_SCB_REQUEST | TR_STAT_INTR;

+ IFNET_UNLOCKNOSPL(&fv->fv_if);
  sleep((caddr_t)&fv->cmd_Status[TR_CMD_CLOSE],
PZERO|PCATCH);
+ IFNET_LOCKNOSPL(&fv->fv_if);
  if (fv->cmd_Status[TR_CMD_CLOSE] == 0) {
      fv->fv_state = FV_STATE_CLOSE;
      fv->fv_if.if_flags &= ~(IFF_UP|IFF_RUNNING);
```

mbuf Manager Changes

```
>>> delete the mbinit() call, no longer needed.
*** 3174,3183 ****
      IDP((“fv%u: IVEC set!\n”, unit));
  #endif /* !_IRIX4 */

-     /* 3: start the mbufs */
-     mbinit();
```

```
!     /* 4: setup PRIVATE data.*/
      /* TBD: allocate mcast filter table.
       *    Probably, ALLMULTI(via ffffffff) should be
used        *    and locally calculate correct filter.
--- 3174,3181 ----
      IDP((“fv%u: IVEC set!\n”, unit));
  #endif /* !_IRIX4 */


!     /* setup PRIVATE data.*/
      /* TBD: allocate mcast filter table.
       *    Probably, ALLMULTI(via ffffffff) should be used
       *    and locally calculate correct filter.
```

Migration to IRIX 6.0

The following issues are important when attempting to convert a device driver for a 32-bit kernel to a 64-bit kernel driver. For details on drivers for POWER Indigo^2 or POWER CHALLENGE M, see “POWER Indigo2 and POWER CHALLENGE M Drivers” in Appendix A.

Virtual Page Size

The virtual page size for 64-bit kernels is currently 16 KB, while it is 4 KB for 32-bit systems. However, various I/O hardware items (such as DMA map registers on CHALLENGE/Onyx platforms) still deal with 4 KB pages for I/O, requiring that the driver use a different set of constants or procedures when dealing with I/O pages.

Most of the following new I/O-related items already exist without the *IO_* prefix and refer to virtual memory page size rather than I/O page size.

Constants

IO_NBPP number of bytes in an I/O page

IO_PNUMSHFT number of bits to shift I/O address to page number

IO_POFFMASK mask for offset into I/O page

Macros

io_pnum(x) I/O page number from address

io_poff(x) I/O page offset from address

io_numpages(addr, len)
number of I/O pages to span address range starting at *addr* for *len* bytes

io_ctob(x) convert I/O pages to bytes

io_btoc(x) convert byte count to number of pages (rounded up)

io_btoct(x) convert byte count to number of pages (rounded down)

To convert an existing driver, a good starting point would be to examine all uses of **NBPP**, **PNUMSHFT**, **POFFMASK**, **pnum**, **poff**, **ctob**, **btoc**, and **btoct** and consider:

- whether they refer to virtual memory pages and should be left alone

  or

- whether they refer to I/O pages and need to be converted.

ioctl Support

In addition to the usual 64-bit conversion worries (*longs* and pointers becoming 64-bits, *ints* remaining 32-bits), drivers may need to support ioctls from user programs. Since user programs may be 32-bit or 64-bit, data items from those programs may need to be converted into the appropriate internal form for the driver. (The driver needs to know whether a pointer or a *long* from a user program should be treated as a 32-bit quantity or a 64-bit quantity.) To ease the driver conversion, a new system intrinsic that informs the driver of the size of various types for the currently executing user has been supplied. See the ioctl(D2) man page.

The following definitions are available in *sys/types.h* and *sys/ddi.h*:

```
/* Since device drivers may need to know which ABI the
/* current user process is running under in order to use the
/* correct types, we provide the following structure. See
/* ddi.h for the definition of userabi().
 * All sizes are in bytes. */

typedef struct __userabi {
    short uabi_szint;
    short uabi_szlong;
    short uabi_szptr;
    short uabi_szlonglong;
} __userabi_t;

/* function: userabi(__userabi_t *)
 * purpose: determine the size int bytes of various C types
 * in the ABI under which the current user process is
 * running. 0 indicates success,nonzero indicates failure.
 * This function must only be called from a user's
 * context, and the values copied into __userabi_t are only
 * valid for the process executing when userabi is called.
 */
int
userabi(__userabi_t *currentabi)
```

Pointers

Pointers in memory buffers need to be double-word aligned to avoid address errors when the pointers are loaded. Both pointers and longs must be aligned on 8-byte boundaries; neither should be cast to int because the int structure is only 32-bits wide.

Hardware Data Copying

When copying data to or from a hardware device, drivers should use the functions **hwcpin** and **hwcpout** rather than **bcopy**. The reason is that the kernel **bcopy** routine is optimized for memory access and may use double-word **loads** and **stores**, which may not be supported by the hardware device. The routines **hwcpin** and **hwcpout** perform only word (or byte and half-word) operations.