Device Drivers 9

Introduction

A *device driver* is a kernel module responsible for managing low-level I/O operations for a particular hardware device or pseudo-device. (A *pseudo-device* is a driver that does not control any hardware.) A device driver contains all the device-specific code necessary to communicate with a device and provides a standard I/O interface to the rest of the system. Application programs and the rest of the kernel therefore need no device-specific code to address the device. In this way, device drivers make the system more portable and easier to maintain.

Driver Entry Points

Each device driver defines a standard set of functions called *entry points*. Pointers to entry points are placed in a data structure called `dev_ops(9S)`. The `dev_ops(9S)` structure provides the interface between the kernel and a device driver; it contains pointers to the driver's autoconfiguration entry points, as well as a pointer to a `cb_ops(9S)` data structure that in turn contains pointers to the rest of the driver entry points. Drivers for different types of devices have different sets of entry points according to the kinds of operations the devices perform. For example, a driver for a memory-mapped character-oriented device supports an `mmap(9E)` entry point.

There are however, certain operations common to all drivers. Most drivers have `open(9E)` and `close(9E)` entry points to control access to their devices, as well as the required autoconfiguration entry points `identify(9E)`, `attach(9E)`, and `probe(9E)` (for non-self-identifying devices). Drivers may also support the optional `detach(9E)` and `reset(9E)` entry points. In addition, all device drivers must support the utility entry points `prop_op(9E)` and `getinfo(9E)`.

The entry points listed in the `cb_ops(9S)` structure can run concurrently as separate kernel threads. These entry points must therefore protect shared data structures by using the synchronization primitives.

When a `cb_ops(9S)` entry point is in use, `attach(9E)` and `detach(9E)` are not called. In addition to the entry points listed in `dev_ops(9S)` and `cb_ops(9S)`, all drivers must implement the `_init(9E)`, `_fini(9E)`, and `_info(9E)` entry points to load, unload and report information about the driver module.

Types of Device Drivers

There are several kinds of device drivers and each handles a different kind of I/O. Block-oriented devices, such as disks, manage physically addressable storage media. All other devices are considered character-oriented. Drivers for character-oriented devices are divided into two groups: standard character device drivers and STREAMS device drivers.

Standard Character Device Drivers

In addition to the entry points discussed above, drivers for character-oriented devices, or *character device drivers*, may have entry points to perform byte-stream I/O, control I/O parameters, map memory, and poll the device for certain conditions.

Byte-Stream I/O

The main job of any device driver is to perform I/O, and many character device drivers do what is called *byte-stream* or *character* I/O. That is, the driver transfers data to and from the device without using a specific device address. This is unlike block device drivers that transfer data to or from a specific location on a storage medium such as a disk.

The `read(9E)` and `write(9E)`entry points handle byte-stream I/O for standard character drivers.

I/O Control

Many devices have characteristics and behaviors that can be configured or tuned. The `ioctl(2)`system call and the `ioctl(9E)` driver entry point provide a mechanism for application programs to change and determine the status of a driver's configurable characteristics. For example, the baud rate of a serial communications port is usually configurable in this way.

The `ioctl(2)` interface is open ended, allowing you to define special commands for your device. What the commands mean is entirely up to you and is restricted only by the requirements of the application programs using the device and the device itself.

Certain classes of devices such as frame buffers or disks must support standard sets of `ioctl(2)` requests. These standard `ioctl(9E)` interfaces are documented in the *SunOS 5.0 Reference Manual*. See the man page for `ioctl(9E)` for more detail.

Memory Mapping

For devices such as frame buffers, I/O is more efficient if application programs have direct access to device memory. Applications can map device memory into their address spaces using the `mmap(2)` system call. To support memory mapping, device drivers implement an `mmap(9E)` entry point and, optionally, a `segmap(9E)` entry point.

Drivers that define an `mmap(9E)` entry point usually do not define `read(9E)` and `write(9E)`, because application programs perform I/O directly to the devices after calling `mmap(2)`.

Currently, drivers supporting `mmap(9E)` should use `ddi_segmap(9F)` as the `segmap(9E)` entry point.

Device Polling

The `poll(2)` system call allows application programs to monitor or *poll* a set of file descriptors for certain conditions or *events*. Primarily, `poll(2)` is used to find out whether data are available to be read from the file descriptors or whether data may be written to the file descriptors without delay. Drivers referred to by these file descriptors must provide support for the `poll(2)` system call by implementing a `chpoll(9E)` entry point. See the `chpoll(9E)` man page.

Communications devices such as serial ports should support polling because they are used by applications that require synchronous notification of changes in read and write status. Many communications devices, however, are better implemented as STREAMS drivers.

STREAMS Device Drivers

STREAMS is a model and a set of interfaces for doing character I/O that provides modular, full-duplex byte-stream I/O using messages.

A *stream* consists of a group of interconnected *modules* including a *stream head* at one end that communicates with application programs and a *driver* at the other end that can address physical hardware. Messages flowing toward the stream head (*read-side* messages) are said to be flowing *upstream*. Messages flowing toward the driver (*write-side* messages) are said to be flowing *downstream*.

Communications devices such as serial ports and ethernet devices are suited to a STREAMS implementation.

Drivers

See Chapter 9 of the *SunOS 5.0 STREAMS Programmer's Guide* to learn how to write STREAMS drivers.

Modules

See Chapter 8 of the *SunOS 5.0 STREAMS Programmer's Guide* to learn how to write STREAMS modules above the driver level.

Architecture-Related Issues

Each hardware device is made to attach to a specific bus. Device drivers have historically included bus-specific code to handle such things as DMA mappings, as well as vectored or polling interrupts. The DDI and DKI interfaces and the coding conventions described in this book and Section 9 of the *SunOS 5.0 Reference Manual* make much of this bus-specific code unnecessary.

Overview of the SunDDI

SunDDI is the Sun architecture-specific (DDI) part of the SunOS 5.x DDI/DKI. The SunOS 5.x DDI/DKI, like its AT&T counterpart, is intended to standardize and document all interfaces between device drivers and the UNIX kernel. In addition, the SunOS 5.x DDI/DKI is designed to allow source compatibility for drivers on any SunOS 5.x-based machine, regardless of the machine architecture or processor. Furthermore, it will provide binary compatibility for drivers running on any SunOS 5.x-based machine which has the same processor, regardless of the specific machine architecture.

The SunOS 5.x DDI/DKI will allow platform-independent device drivers to be written for SunOS-based machines. These "shrink-wrapped" drivers will allow third-party hardware and software to be integrated into SunOS-based machines much as they are on IBM-compatible personal computers today. In the IBM-compatible PC world this is accomplished by standardizing on a single machine architecture. The SunOS 5.x DDI/DKI is designed to be architecture independent and allows the same driver to work across a diverse set of machine architectures.

Architecture independence is accomplished in the design of DDI-only portions of the SunOS 5.x DDI/DKI, called SunDDI. The main areas that the SunDDI addresses are

- interrupt handling
- accessing the device space from the kernel or a user process (register mapping and memory mapping)
- accessing kernel or user process space from the device (DMA services)
- device properties (property management)

Architectural independence is achieved in the SunOS 5.x SunDDI through the use of a layered approach, implemented as a tree structure. Each node in the tree structure is described by a device-information structure. Standard device drivers and their devices are associated with leaf nodes, hence, these drivers are called *leaf drivers*. This chapter documents writing leaf drivers only.

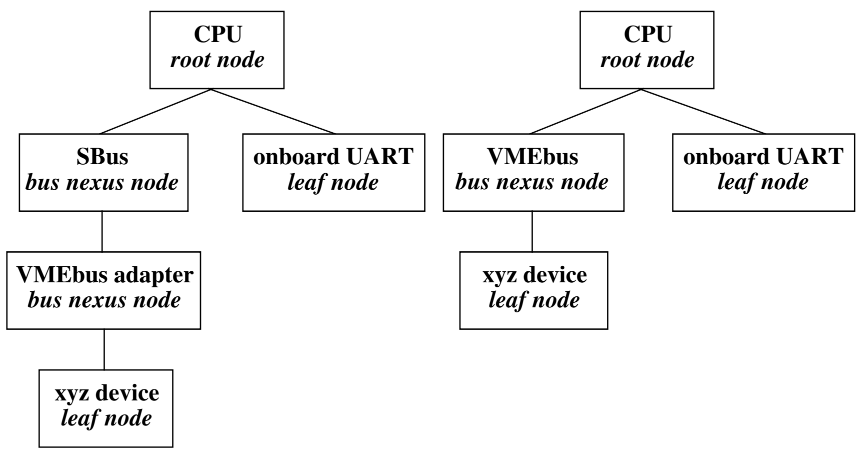

Figure 9-1 Possible Node Configurations

The intermediate nodes in the tree are generally associated with buses, such as SBus, SCSI, VMEbus. These nodes are called *bus nexus nodes* and the drivers associated with them are called *bus nexus drivers*. This book does not document writing bus nexus drivers. Bus nexuses are architectural entities and are intended to encapsulate the architectural dependencies associated with a particular architectural element.

The topmost node in the tree is called the *root node*. The tree structure creates a formal parent-child relationship between nodes. This parent-child relationship is the key to architectural independence. When a leaf or bus nexus driver requires a service that is architecturally dependent in nature, it requests its parent to provide the service.

This approach allows drivers to function regardless of the architecture of the machine or the processor. In all of the architectural configurations in Figure 9-1, the *xyz* driver can be source compatible; it can also be binary-compatible if the CPU modules use the same processor.

In Figure 9-1 the bus nexus driver associated with the SBus-to-VME-bus adapter card handles all of the architectural dependencies of the interface. The *xyz* driver only needs to know that it is connected to a VMEbus.

Device-Information Structure

The concept of a device-information structure originated in the architecture of the Sun4c (SPARCstation) machines. The system builds a tree structure that contains information about the self-identifying devices connected to the machine at boot time. The system uses this information to build the device-information structures for each device and to create a dependency tree. Therefore, there is one device information structure created for each device. Figure 9-2 illustrates a device information tree that might be created for a particular Sun4c machine.

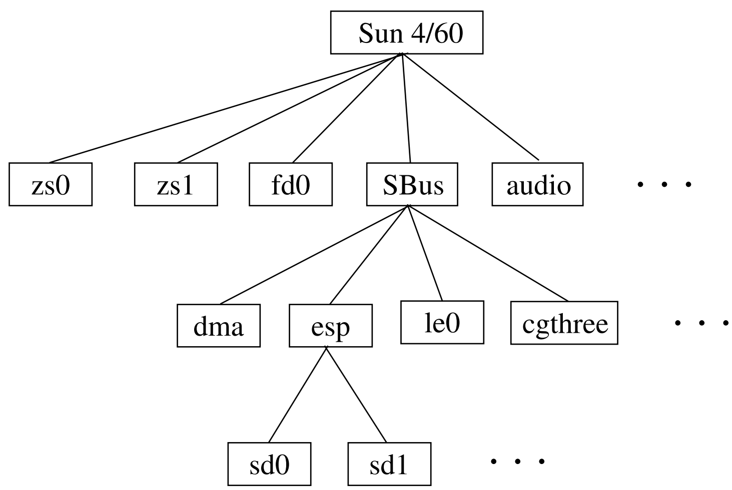

Figure 9-2 The Device Information Tree

Associated with each leaf or bus nexus node may be a driver. Each device driver has associated with it a device operations structure (`dev_ops(9S)`) that defines the operations that the device driver can perform. During configuration, each device operations structure may be pointed at by one or more device-information structures for the devices which this driver controls. The basic device operations structure contains function pointers for generic operations, such as `identify(9E)` and `attach(9E)`. The basic device operations structure also contains a pointer to operations specific to bus nexus drivers and a pointer to operations specific to leaf drivers.

Bus Nexus Drivers

Bus nexus drivers implement the functions associated with bus nexus nodes. In general, bus nexus functions provide leaf drivers with services. This chapter documents only how to write leaf drivers; it does not describe the implementation of bus nexus drivers.

Accessing Device Memory

Register Mapping

The function `ddi_map_regs(9F)` is provided to allow a driver to gain access to its registers, or any other portion of its hardware, such as a frame buffer. `ddi_map_regs` maps in a specified register set. A register number determines which register set will be mapped if more than one exists. `ddi_map_regs()` returns the kernel virtual address of the specified register set.

When addressing device registers, you should be aware that the way the system hardware buffers data written to device memory may affect the synchronization of your I/O operations. Writes to device registers may pass through several system I/O buffers before reaching the registers. To make sure writes to registers occur when you need them to, you'll have to take some explicit steps.

When acknowledging, or clearing an interrupt, you usually set or clear a bit in the device control register. To keep spurious interrupts from happening, you must make sure your write to the control register has reached the device before you continue with your interrupt handler. Similarly, if your device requires you to delay (busy wait) after writing a command to the control register, you must be sure the write has reached the device before you start the delay.

If your device registers can be read without undesirable side effects, you can verify a write to a register by reading the register immediately after writing to it.

If your device registers can't be read without undesirable side effects, you can use the `ddi_poke(9F)` routines to write to your registers. When these routines return, they guarantee that the write has reached the device.

Memory Mapping

To support the `mmap(2)` system call, drivers provide `mmap(9E)` and `segmap(9E)` entry points in the `cb_ops` structure. You must provide an `mmap(9E)` routine. Currently, you should supply `ddi_segmap(9F)` for the `segmap(9E)` entry point rather than explicitly defining a `segmap(9E)` routine. The SunDDI does not, at present, support segment drivers.

Interrupt Handling

The SunDDI addresses the following aspects of device interrupt handling:

- registering device interrupts with the system
- removing device interrupts from the system

Because the internal implementation of interrupts is an architectural detail, special *interrupt cookies* are used to allow drivers to perform interrupt-related tasks. The types of cookies for interrupts are:

- device interrupt cookies
- block interrupt cookies

Device Interrupt Cookies

Defined as type `ddi_idevice_cookie_t`, this cookie is used by a driver to program the interrupt-request level (or something equivalent) for a programmable device. This type of cookie is bus-architecture specific, and the driver must understand how to interpret it to program its device correctly.

Block-Interrupt Cookies

Defined as type `ddi_iblock_cookie_t`, this cookie is used by a driver to initialize the mutex locks it will use to protect data.

Property Management

A *property* is a name-value pair. Properties may be defined by the FCode of a self-identifying device or by the driver itself using `ddi_prop_create(9F)`. For a definition of FCode, please refer to the *SBus Developer's Kit*. Examples of properties are the height and width of a frame buffer or the number of blocks in a partition of a block device. The value of a property may be one of three types:

- a boolean property, which has no length, either exists or does not exist
- an integer property, which has length `sizeof(int)` and has an integer value
- a long property, which has an arbitrary length, and whose value is a series of bytes

A driver can request a property from its parent, who in turn may ask its parent. The driver can control whether the request can go higher than its parent. The `ddi_getprop(9F)` routine is used to get integer- or boolean-sized properties. The `ddi_getlongprop(9F)` and `ddi_getlongprop_buf(9F)` routines are used to get larger properties of any size.

Direct Memory Access (DMA) Services

Many devices are capable of being bus masters and accessing other devices on a bus. In particular, these devices are designed to access main system memory to perform data transfer. Setting up memory-mapping resources so that a device can access main system memory is the area of DMA service. For the most part, the details of setting up mappings and of ensuring I/O and CPU cache consistencies are hidden from the developer.

DMA Mapping

The basic DMA service provided to map a memory object for DMA is the `ddi_dma_setup(9F)` routine. It establishes a mapping to the object that the caller can use to transfer data. It returns an object called a *DMA handle*, which is then used by many other DMA service functions. The handle is used by the SunDDI implementation as a unique identifier for the mapping the driver requested.

Device Addressability of Memory Objects

When a device driver is ready to set a device's DMA controller in motion, it must derive the address the controller will use from the DMA handle. The service function `ddi_dma_htoc(9F)` interprets the DMA handle and returns a DMA *cookie*, a data structure that contains the address.

Memory Object Synchronization

At times during the lifetime of a DMA mapping, the driver may need to synchronize I/O with respect to various caches and system write buffers, including when the I/O mapping is removed. The function `ddi_dma_sync(9F)` is provided as the general service routine to perform this function.

For example, if you fill in a command data structure in device-addressable memory, and then write a bit to your device control register telling the device to read the command structure, the command data will not necessarily be available to the device yet, so it may read garbage. You must prevent this from happening by synchronizing the command structure for the benefit of the device by calling `ddi_dma_sync(9F)` with the `DDI_DMA_SYNC_FORDEV` flag.

Similarly, after a DMA transfer to system memory, you must assure that the system can read the data correctly by calling `ddi_dma_sync(9F)` with the `DDI_DMA_SYNC_FORCPU` flag.

Freeing DMA mappings

When an I/O transfer completes, the resources required to map the object for DMA should be completely released. This might require some data synchronization. The function `ddi_dma_free(9F)` provides this service by calling `ddi_dma_sync(9F)` before removing the mapping and freeing the associated resources.

Issues

A multi-threaded kernel requires developers to consider two additional issues: *data access control* and *thread synchronization*. See below for more information.

How SunOS Has Changed

Here is a simplistic view of how the earlier releases of the SunOS kernel ran on multiprocessors: only one processor could run kernel code at any one time, and this was enforced by using a *master lock* around the entire kernel. When a processor wanted to execute kernel code, it acquired the master lock, thus blocking other processors from accessing kernel code. It released the lock upon exiting the kernel.

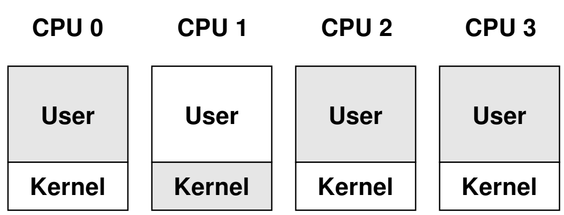

Figure 9-3 Earlier SunOS Kernels on a Multiprocessor

In the model shown in Figure 9-3, CPU 1 would execute the following instructions:

```
Acquire kernel_lock;
Execute kernel code;
Release kernel_lock;
```

All other processors would be locked out of the kernel, but they could run user code.

In the SunOS multithreaded kernel, instead of one master lock, there are many smaller locks that protect smaller regions of code or data. In the example shown in Figure 9-3, there is a kernel lock that controls access to data structure A, and another that controls access to data structure B. Because of the locks, only one processor can be running code dealing with data structure A at a time, but another could be accessing data structure B. This allows a greater degree of concurrency.

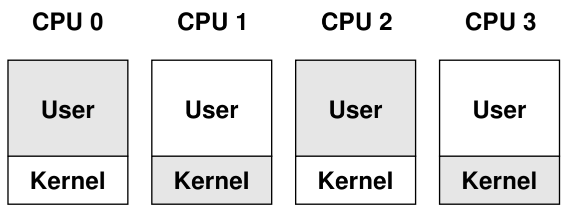

Figure 9-4 SunOS 5.x on a Multiprocessor

In the model shown in Figure 9-4, CPU 1 and CPU 3 could execute the following kernel codes simultaneously:

CPU1	***CPU3***
`Acquire lock_A;`	`Acquire lock_B;`
`  Modify A;`	`Modify B;`
`Release lock_A;`	`Release lock_B;`

Access to Shared Data

Before the SunOS 5.x operating system, any section of kernel code would run until it explicitly gave up the processor by calling `sleep()` or was preempted by a hardware interrupt. Now, however, a kernel thread can be preempted by the scheduler at any time to run another thread. Because all kernel threads share the kernel address space, and often may need to read and modify the same data, there are mechanisms to prevent threads from corrupting those data. These mechanisms include *mutual exclusion locks* and *readers/writer locks*.

Storage Classes of Driver Data

The storage class of data is a guide to whether the driver may need to take explicit steps to control access to them.

Automatic (Stack) Data

Because every thread has a private stack data, drivers never need to lock automatic variables.

Global Data

Global data can be shared by any number of threads in the driver; the driver may need to lock them at times.

Kernel Heap Data

Kernel heap data are shared by any number of threads in the driver; the driver may need to lock them at times.

Choosing a Locking Scheme

Choosing an appropriate locking scheme for a device driver depends on how the driver is going to be used. Using more locks allows more concurrency but costs more in overhead. Using fewer locks is cheaper but allows less concurrency. The locking scheme for most device drivers, though, should be fairly straightforward. Generally, use one mutex per data structure, a condition variable for each event or condition the driver must wait for, and a mutex for each major set of data global to the driver.

You may find you need to adjust your locking scheme once you start testing your driver. If you find contention for a particular lock is becoming an unnecessary bottleneck for example, you could try using more locks.

The scope of a mutex, condition variable, or readers/writers lock (that is, what data or code path it protects) is entirely up to the programmer. A mutex protects some particular data structure because you say so and use it accordingly.

Locking Guidelines

The mutex routines are approximately equivalent to the explicit interrupt blocking calls, such as `spln()`, used in previous releases.

Use A Lock Only When Needed

If some members of a data structure are initialized in attach, destroyed by detach, but only read in other entry points, there is no need to acquire a lock to read those members of the structure.

On SPARC architectures, integer load, store, and test are atomic operations, and do not require locking. However, integer addition, subtraction, increment, and so forth are not atomic, and therefore require locking.

Reduce the Amount of Shared Data

Change static variables to automatic, or add them to your state structure. Kernel thread stacks are currently 8Kbytes in size, so do not allocate large automatic variables and do not do deep recursions.

Acquire Multiple Mutexes in the Same Order on Each Path

For example, assume mutexes A and B are used to protect a resource in the following manner:

Path1	Path2
`mutex_enter(&A);`	`mutex_enter(&B);`
`>...`	
`mutex_enter(&B);`	`mutex_enter(&A);`
`>...`	
`mutex_exit(&B);`	`mutex_exit(&A);`
`>...`	
`mutex_exit(&A);`	`mutex_exit(&B);`

If a thread is executing each path, the following could occur:

1. Thread 1 acquires mutex A.
2. Thread 2 acquires mutex B.
3. Thread 1 needs mutex B, so it blocks and holds mutex A.
4. Thread 2 needs mutex A, so it blocks and holds mutex B.

These threads are now deadlocked. This is hard to track down because the code paths are rarely this straightforward. Also, it only occurs in the preceding situation, so it is an occasional (but severe) problem.

Hold Locks for a Short Time

In particular, drop locks before calling routines that may block. For example:

```
mutex_enter(&softc->lock);
...
softc->foo = bar;
softc->thingp = kmem_alloc(sizeof(thing_t), KM_SLEEP);
...
mutex_exit(&softc->lock);
```

can be better coded as:

```
bufferp = kmem_alloc(sizeof(thing_t), KM_SLEEP);
mutex_enter(&softc->lock);
...
softc->foo = bar;
softc->bufferp = bufferp;
mutex_exit(&softc->lock);
```

Acquiring and Releasing Locks in the Same Routine

Locks acquired and released in different routines, although legal, are difficult to maintain.

You may find you need to adjust your locking scheme once you start testing your driver. If, for example, you find contention for a particular lock is becoming an unneccessary bottleneck, you could try using more locks.

Mutex-Related Panics

The following error messages may occur:

```
panic: recursive mutex_enter. mutex %x caller %x
```

Mutexes are not reentrant by the same thread. If you already own the mutex, you cannot own it again. Doing this leads to this panic.

```
panic: mutex_adaptive_exit: mutex not held by thread
```

Releasing a mutex that the current thread does not hold causes this panic.

```
panic: lock_set: lock held and only one CPU
```

If you attempt to acquire a spin-lock mutex that is already held on a uniprocessor machine, something has gone horribly wrong because there is nobody there to ever wake you up again, and leads to this panic.

Note – In principle, this is only a problem for drivers that operate above lock level.

Levels of Driver Multithreading

Multithreading-unsafe, or *MT-unsafe* drivers only support one thread in the body of the driver code at a time.

Multithreading-safe, or *MT-safe* drivers support multiple threads.

Multithreading-hot, or *MT-hot* drivers are a subclass of MT-safe drivers. *Hot* refers to performance, and usually means that the locking is fine-grained and safely allows many threads in the driver simultaneously.

Note – An MT-hot driver is usually a goal. It is not currently possible to determine whether your driver is MT-hot aside from experimentation.

Changing Your Device Driver

When porting a device driver from SunOS 4.x to SunOS 5.x, the general strategy is to convert as much as possible under SunOS 4.x so your driver will compile under SunOS 5.x. Once the compile works, comment out all entry points that do not deal with loading and unloading. Once loading and unloading work, you can add and debug each entry point until the driver works.

Here are detailed steps for porting your device driver from SunOS 4.x to SunOS 5.x:

1. **Make the driver loadable.**
   If your current driver cannot be loaded, convert it to a loadable driver under SunOS 4.x before starting the port to the DDI. *Writing SBus Device Drivers* has more information on loadable device drivers.

2. **Compile the driver under SunOS 5.x**
   To get your driver to compile under SunOS 5.x, use the installed SPARC compilers.

   To compile a driver, type

```
# cc -D_KERNEL -I. -c example_one
# cc -D_KERNEL -I. -c example_two as -P -D_ASM \
     -D_KERNEL -I. -o examle_asm.o example_asm.s
# ld -r -o example exapmle_one.o example_two.o \
    example.asm.o
```

   All the kernel headers are located in the `sys` directory. The following information regarding headers is pertinent to device driver conversion:

- You must include the files <sys/devops.h> and <sys/ddi.h> at the end of your include list.
- Include <sys/sunddi.h> in the kernel header include list.
- For full DDI compliance, you may only include kernel header files listed in the synopsis sections of Section 9 of the man pages.

The following structure changes are also necessary:

- Add or change your *xx*_ops structure to the new form; the dev_ops(9S) man page has more information on this. If you have no devo_reset routine, use nulldev.
- Define an *xx*_cb_ops structure. The cb_ops(9S) man page has information on how to do this. If you have no devmap routine, use nodev; if you have no detach routine, use nulldev. Set the cb_flag field to (D_NEW | D_MTSAFE) if your driver is MT-safe.
- The structure vdldrv has been renamed modldrv and its form has been changed; see the man page for modldrv(9S) for more information. Also declare either mod_driverops or mod_pseudodrvops as external structures.
- Add the new structure modlinkage; the man page for modlinkage(9S) has more details on this structure.
- The MB structures from SunOS 4.x have been eliminated. Instead of using mb_driver, use dev_ops. The elements from the mb_ctlr and mb_device structures are now opaque data in the dev_info structure. The information in your *xx*.conf file will cause some properties to be created for your driver when it is loaded.

The following compiler notes are also applicable:

- If your driver used an #ident construct with a trailing semicolon, remove the semicolon.
- If your driver used a static char with a trailing semicolon to document revision level and module name, change it to use the #ident form with no semicolon.
- The type addr_t has been replaced by caddr_t.
- To get the driver to compile, you can write "dummy routines" to hold places for new routines, and prototype the new routines needed in the dev_ops and cb_ops structures, such as *xx*_getinfo. However, you will need to write these routines before the driver has any chance of loading.

The following are good practices when porting code:

- Prototype all your entry points in the ANSI C style as noted in *SunOS 5.0 Writing Device Drivers*.
- Use the volatile type specifier to declare software structures which overlay actual hardware register or for specifying any other location which could be updated by an alternate thread of control.
- Use the const type specifier for an object or array which is initialized, but then never changed.

3. **Load and unload the driver under SunOS 5.x.**
Under SunOS 4.x, the only way to load a loadable driver was with the `modload` utility. Under SunOS 5.x, drivers can be automatically loaded when their corresponding node in `/devices` is referenced. However, during early driver porting, you may want to use `modload` and `modunload` until you are sure your driver loads and unloads properly, and then use the `add_drv` utility to add your driver to the system. Note that the `modload` utility now requires a full pathname; for details, see the man page for `modload(1M)`.

To support a loadable driver, the following routines must be loaded:

`_init`	`_fini`	`_info`	*xx*`_identify`	*xx*`_probe`[1]
xx`_attach`	*xx*`_detach`[2]	*xx*`_getinfo`		

You should be able to debug the loading and unloading by writing "dummy routines" to hold places for the rest of your driver routines. Once your driver loads and unloads, most of the work is done.

Note – The kernel `log()` function and kernel `printf()` have been replaced by `cmn_err()`. The `cmn_err(9F)` man page has more information. `CE_PANIC` will panic the system after printing your message, and some levels of `cmn_err()` generate automatic labels before your message and newlines after.

You can also vary the value of the system global variable `moddebug` to see what is happening as the module loads and unloads. The possible values of `moddebug` are in `<sys/modctl.h>`.

Under SunOS 5.x, you can now symbolically debug with `kadb` and loadable drivers. Remember that your kernel symbols will only be mapped into the kernel when your driver is loaded.

Some of the routines and structures have changed as follows:

- *xx*`_init` and *xx*`_vdcmd`
  The *xx*`_init()` and *xx*`_vdcmd()` routines, which had three sections labeled `VDLOAD`, `VDUNLOAD`, and `VDSTAT`, have now been replaced with the three routines `_init()`, `_fini()`, and `_info()`. Further information on these three routines can be found on the `_fini(9E)` man page.

1. For NSID devices only

2. If your driver will be unloadable

- _init and *xx*_init
  The _init() routine does any per-driver initialization, such as allocating memory for each driver. This routine is called before *xx*_identify(), *xx*_probe(), and *xx*_attach(), therefore it is not safe to assume that any unit numbers or unit state structures are present. Anything which was in the *xx*_init structure needs to be here. You no longer need to call vd_installmod(); instead, call ddi_soft_state_init(). The _init() routine must call mod_install(9F) with the address of your modlinkage structure.
- _fini
  The _fini(9E) routine does any per-driver shutdown. It quiesces all the units controlled by the driver, then frees any memory allocated in _init(). Call this routine just before the driver executable image is removed from memory. Anything which used to be in the VDUNLOAD case of init_module() belongs here. You no longer need to call vd_removemod(). If the decommissioning is successful, call mod_remove() with your modlinkage structure address; otherwise, return an appropriate error code.
- _info
  The _info(9E) routine is passed a modinfop pointer address. Return the value from mod_info() by using the code line:

```
return(mod_info(&modlinkage, modinfop));
```

You can also print out information about your driver. The _TIME_ and _DATE_ preprocessor directives are useful for recording the compile date of your driver.

- *xx*_info and *xx*_getinfo
  The *xx*_info() and *xx*_getinfo() routines return a dev_info pointer for the dev_t (DDI_INFO_DEVT2DEVINFO) or the instance (unit) corresponding to the dev_t (DDI_INFO_DEV2INSTANCE).
- *xx*_identify
  You cannot depend on the *xx*_identify() routine to be called for all units before *xx*_attach() is called. The unit name is no longer passed to you; you must use ddi_get_name() to get it. The return value of *xx*_identify() is either DDI_IDENTIFIED or DDI_NOT_IDENTIFIED. The system framework handles the unit number counting.
- *xx*_probe
  Use the *xx*_probe() routine to determine whether a non-self-identifying device (NSID) is present, and if so, to gather information about the NSID such as its version and configuration. If the device you are porting statically allocates an array of controller state structures and uses them in a manner similar to:

```
register struct xx_ctlr *c = &xx_ctlrs[ctlr];
```

you can now use the following manner of allocating state structures:

```
c = (struct xx_ctlr *)
kmem_zalloc (sizeof (*c), KM_SLEEP);
ddi_set_driver-private(dev, (caddr_t) c);
```

- `ddi_get_driver_private`
  You then use `ddi_get_driver_private(9F)` to get the `ctrl` pointer elsewhere in the driver. You should also store you *xx*`_units` structures in the `dev_info` corresponding to each unit in an equivalent fashion.
- `ddi_prop_create` and `ddi_map_regs`
  If your driver supports multiple types of devices, use `ddi_prop_create(9F)` to create descriptive properties which the other driver routines can use to determine what to do. You may create as many properties as are appropriate for your device. Note that the module loading process turns the information in your *xx*`.conf` file into properties in your `dev_info` structure. Next, use `ddi_map_regs(9F)` to map in your registers.
- *xx*`_attach`
  The arguments to the *xx*`_attach` have changed. Be careful to undo any attach, if a unit fails to attach.

  The *xx*`_attach()` routine undoes any unit allocations upon failure to attach; you should watch for recursive `mutex_enter` calls when a unit fails to attach.
- `ddi_get_instance` and `ddi_soft_state_*`
  To get the unit instance number, call `ddi_get_instance(dip)`. Use the `ddi_soft_state_*` routines to manage your state structures and keep them MT-safe.
- `kmem_alloc and kmem_zalloc`
  `kmem_alloc()` requires a second (`flag`) argument. See `kmem_alloc(9F)` and `kmem_zalloc(9F)`.
- `ddi_slaveonly`
  The `slaveslot()` routine has been replaced by `ddi_slaveonly(9F)`.
- `ddi_map_regs`
  Use `ddi_map_regs()` to map in your registers instead of `map_regs()` or `mapin()`.
- `dii_iopb_alloc`, `ddi_dma_addr_setup`, and `ddi_dma_devalign`
  If you need `iopb` memory, use `ddi_iopb_alloc(9F)` and then `ddi_dma_addr_setup(9F)` instead of using `rmalloc(iopbmap,size)`. You may also need to call `ddi_dma_devalign(9F)`.

- ddi_prop_create and ddi_prop_op
  You can no longer view the contents of the dev_info structure; dev_info_t is now an opaque pointer. There are various DDI routines to acquire the property information formerly viewed in dev_info. See the ddi_prop_create(9F) and ddi_prop_op(9F) man pages for more details on property management.
- ddi_add_intr
  Use the ddi_add_intr(9F) routine where the addintr() routine is used. The ddi_add_intr(9F) man page has more details on this routine.
- adddma
  You no longer need to notify the system that you are using DVMA with a call such as adddma().
- splN and splr
  The multithreading properties of the SunOS 5.x kernel make these functions obsolete. Use mutexes to block out threads of control. To do this, call mutex_init(9F) on your mutexes in *xx*_attach().
- *xx*_poll
  You no longer need to call *xx*_poll() to determine which unit is interrupting an SBus and then followed by *xx*_intr(). Instead, call ddi_add_intr(9F) for each instance, use *xx*_intr() as the int_handler argument, and pass the unit number or pointer to the unit structure in the int_handler_arg argument. This is a more general approach, and should work without modification if your device ever exists on another bus, such as an autovectoring bus like VME.
- ddi_create_minor_mode and ddi_remove_minor_node
  Call ddi_create_minor_mode(9F) for each attach. This routine creates the minor node structure used by the devfs to make the nodes in /devices at boot -r time. Be sure to call ddi_remove_minor_node(9F) if the call fails.
- ddi_peek and ddi_poke
  The peek and poke calls have been replaced with ddi_peek(9F) and ddi_poke(9F). There are ddi_peek* calls for all sizes.
- ddi_getprop
  Because the mb structures have been eliminated, you may have to redesign your driver. The information in your *xx*.conf file gets turned into properties in your dev_info structure by the module loading process. Use ddi_getprop(9F) to fetch these properties.

- *xx*_detach

  The *xx*_detach(9E) routine allows your driver to be automatically unloaded to save system resources. Note that it is up to the driver to allow unloading; if it does not, the system cannot forcibly unload your driver.

  The *xx*_detach() routine undoes and deallocates all the actions and allocations done by the *xx*_attach() routine. As with *xx*_attach(), watch for recursive mutex_enter() calls. If your driver has a decommissioning subroutine which is called in the VDUNLOAD step of your init_module() routine, you can probably convert it into your *xx*_detach() routine.

4. Configure the driver entry points for SunOS 5.x

The following routines and structures have changed under SunOS 5.x:

- *xx*_open

  The first argument to open(9E) is a dev_t *, not a dev_t. Verify that the otyp you are passed is OTYP_CHR for character devices or OTYP_BLK for block devices. This avoids having your driver opened in a layered open unexpectedly. These definitions are in <sys/open.h>. Note that the system will not call the open routine for a particular unit unless the identify(), probe() and attach() routines for that unit have been called and returned per-instance.

- *xx*_close

  The *xx*_close() routine has changed; the close(9E) man page has more details. Note that read() has two more arguments.

- *xx*_read

  Use driv_priv(credp) instead of suser() to have your driver check user privileges. Note that you cannot call physio passing the address of a buffer you have statically allocated. You may pass (struct buf *)0 as the second argument, allowing physio to allocate a buffer from the pool. If you must change this, remember to store the address of the allocated buffer in the *xx*_strategy(); you need it to call biodone(9F) in your interrupt routine. Another strategy is to use getrbuf(9F) to allocate the buf structure and freerbuf(9F) to free it.

- *xx*_write

  The *xx*_write() function has been changed; see the *xx*_read() notes and the man page for *xx*_write(9E) for more details.

- *xx*_ioctl
  The ioctl(9E) routine now has two more arguments. If your ioctl() arguments are defined in a manner similar to:

```
#define MYIOCTL _IOR(m,1,u_int)<-m
```

you must now surround the letter m with single quotes (') to get the #define to work.

Alternatively, you can use this definition method:

```
#define DIOC ('d'<<8)
#define DKIOCGGEOM (DIOC|2)
```

Theoretically, the magic number you use (DIOC) should be allocated by AT&T. Use the cred_p() routine to check credentials on a call, and use the rval_p() routine to return a pointer to a meaningful return value (as opposed to always having zero returned to indicate success). See the ioctl(9E) man page for more information.

- arg
  You can no longer reference arg directly; instead, use copyin() and copyout() to move data. Note that the buffer pointer argument to copyin() and copyout() must be cast to caddr_t. The commands which had been defined as _IOW use copyin(); those defined as _IOR use copyout(), and, and those that use _IOWR use both.

- *xx*_strategy
  The *xx*_strategy(9E) routine should now return an integer value that is ignored.

  If your driver uses the b_dev field to obtain its unit number from the buf structure using a line similar to:

```
unit_no = getminor(bp->b_dev);
```

it must now use the b_edev field, as in

```
unit_no = getminor(bp->b_edev);
```

- ddi_dma_buf_setup
  If your driver uses mb_mapalloc(), change it to use ddi_dma_buf_setup(). If no argument is necessary, use (caddr_t)0. Use ddi_dma_htoc() to convert the DMA handle into a DMA cookie, from which you can get the kernel virtual DVMA address to plug into your DMA engine address register.

  If your device is a DMA device, and your DMA engine has limited bits of resolution, such as the top eight bits being registered rather than a counter, you should set up a ddi_dma_lim_t structure describing these limits. Pass the address of this structure as the fifth argument to ddi_dma_buf_setup(). The man page for ddi_dma_lim(9S) has more details.

  Define a new element of your unit structure to store the DMA handle; you will need it later to free the memory after the DMA has completed.

- ddi_dma_sync
  If your device allocates uncached buffers, you can now use ddi_dma_sync() whenever your driver requires cache consistency.

- timeout and untimeout
  The semantics of timeout() and untimeout() have changed. The timeout() routine returns an identifier which must be saved to pass to untimeout(). This may require changes to your unit structure if your driver store pending timeout information in any form other than an integer for each possible timeout. Remember to save your buf address.

- *xx*_poll and *xx*_intr
  Use ddi_dma_free() instead of mb_mapfree(). The information about eliminating *xx*_poll() iscussed previously also applies here. Remember to call biodone() with the new buffer address if your driver allocates its own buf structure.

- spl and splx
  Calls to spl() and splx() no longer do anything. All critical sections should be surrounded by mutex_enter()/mutex_exit() pairs.

5. Make the driver MT-safe
To make your driver MT-safe, execute the following steps:

- In your cb_ops structure, change your driver compatibility flag (the last struct element) to

```
D_NEW | D_MTSAFE/* driver compatibility flag */
```

- Define a mutex in your driver state (softc) structure similar to the following:

```
mutex_t    my_mutex    /* mutex for any op */
```

Bibliography

SunOS 5.0 Writing Device Drivers, Part Number 800-6502-10, SunSoft, Inc., Mountain View, CA.

SBus Developer's Kit II Plus, Part Number SBUS-DEC_KIT-II+, Sun Microsystems, Inc., Mountain View, CA. (Order through SunExpress).

Writing SBus Device Drivers Part Number 800-5323-05, SunSoft, Inc., Mountain View, CA.