Don't Panic! An Engineering Tutorial on Porting your Device Driver to Solaris 2.0

Deborah Gronke Bennett and Patrick Stirling

Sun MIcrosystems Computer Corporation

December 7, 1992

Solaris 2.0 is Sun Microsystems' new operating system release, which is based on AT&T System V Release 4. The SunOS 5.0 kernel included in Solaris 2.0 has many new features, one of which is the Solaris 2.0 SPARC DDI/DKI ("Device Driver Interface" and "Driver Kernel Interface"). This group of interfaces provides a new architecture-independent and regularized set of routines between the kernel and device drivers. This paper describes this new world, and addresses the task of porting a device driver to these new interfaces.

Throughout this paper, the operating system is referred to as SunOS 5.0. This is the operating system portion of the Solaris 2.0 release. The Solaris 2.0 SPARC DDI/DKI is a portion of SunOS 5.0, and is referred to elsewhere in this document as the "DDI/DKI". Also, the term SunOS 4.x refers to any version of SunOS since SunOS 4.1.

This paper is intended for the engineer who is familiar with UNIX device drivers, and has already written or maintained a driver. The examples will illustrate porting a driver from SunOS 4.1.1 to SunOS 5.0. Driver authors who are writing a Solaris 2.0 driver from scratch should consult the manual *Writing Device Drivers* (p/n 800-6502-10) in the Sun Documentation Set.

1.0 The New World of Solaris 2.0 for Device Driver Authors

1.1 Goals of the Solaris 2.0 SPARC DDI/DKI

When the DDI/DKI for Solaris 2.0 was being designed, the engineering team looked back at their experiences writing and porting device drivers in SunOS 4.x. They also remembered the experiences of Sun's customers and independent hardware vendors who had written and ported device drivers under SunOS 4.x. Several important design goals were identified which affected the scope and nature of the final product.

The DDI/DKI was designed to provide better compatibility guarantees for drivers from release to release, and also across multiple platform architectures. Many specifics of architecture were deliberately hidden to provide this compatibility. The

DDI/DKI was also designed to greatly simplify porting a device driver to a new kernel architecture. The ultimate goal is to allow device driver suppliers to provide a single driver executable which will run on all Solaris SPARC Sun DDI/DKI machines (a so-called “shrink-wrapped” driver).

From an engineering perspective, the autoconfiguration and device information model which was first developed for the SPARCstation 1 under SunOS 4.0.3c was refined, and extended to all Sun architectures. The concept of a dynamically configurable kernel with loadable device drivers was introduced. Of course, since the Solaris 2.0 kernel is multithreaded and supports multiprocessor machines, the DDI/DKI includes interfaces which support multiprocessing and multithreading. Finally, the SCSA (SCSI Common Command Set Architecture) was extended to all architectures, and made part of the Sun DDI/DKI.

Each of these goals had their affect on the new kernel you see in Solaris 2.0. The differences that affect a driver author are discussed in the next section, “Overview of the Solaris 2.0 SPARC DDI/DKI”

1.2 New Terms

Several terms are used in this paper which may be new to you.

Multithreaded (or MT for short) means a program can have multiple threads of control executing at the same time. Multithreadedness is a matter of degree, not an all-or-none proposition. In the context of the SunOS 5.0 kernel, multithreaded means that there can be many “threads of control” executing concurrently in the kernel. For example, any one or more of the routines in a driver may be running simultaneously on different processors.

Multiprocessor (or MP for short) relates to a system architecture. A multiprocessor system has more than one microprocessor which runs the system (or kernel) software. An example of a multiprocessor architecture is the SPARCserver 600 system.

A leaf device is a a device which has no child devices in the device information tree. Most “ordinary” peripherals are leaf devices, such as parallel or serial ports, framebuffers, etc. A leaf device always has a nexus device as its parent. A leaf device will have a leaf device driver. Only leaf device drivers can take advantage of all the DDI/DKI source and binary compatibility guarantees.

A nexus device is a device which has child devices in the device information tree. A nexus device provides some set of services for its child devices. The SBus is a nexus device, which has SBus cards as its child devices. The `esp` device is also a nexus device, which has various SCSI peripherals (disk drives, tape drives) as its children. A nexus device can have either nexus devices or leaf devices as its children. A nexus device will have a nexus device driver.

2.0 Overview of the Solaris 2.0 SPARC DDI/DKI

The Solaris 2.0 SPARC DDI/DKI is a source and binary interface specification for device drivers. It began with AT&T’s DDI/DKI for the 3B2 architecture, and has many extensions for Sun’s multiple kernel architectures, loadable driver modules, multithreading, etc. All Solaris 2.0 SPARC DDI/DKI routines are documented in chapter 9 of the *SunOS 5.0 Reference Manual* (the manpages), part number 800-6445-06. The Goals of Sun’s DDI/DKI are:

- better future compatibility; drivers will need less or no modification between SunOS versions;
- simplified driver porting to new architectures;
- move towards “shrink-wrapped” UNIX device drivers;
- subsume the Sun Common SCSI Architecture.

The price for achieving these goals is that all drivers must be rewritten for Solaris 2.0. There is no source or binary compatibility for device drivers from Solaris 1.0.

2.1 Platform Compatibility

One of the goals of the Solaris 2.0 SPARC DDI/DKI is multi-platform compatibility. Compliant drivers are binary compatible across all Sun SPARC Solaris 2.0 platforms, and are source compatible across platforms of differing architectures (e.g. SPARC and Intel).

A strategy for achieving architecture independence is to arrange device drivers in a hierarchy. Drivers that control a peripheral device are known as *leaf* drivers; drivers that control an intermediate device (e.g. a bus) are *nexus* drivers, and are parents to leaf drivers. The leaf drivers don't need to know any details of the bus, even whether it's an SBus or a VME bus. Bus services are requested via DDI routines in an architecture independent way. The same is true of cache and memory management (including DMA) services. This strategy makes it possible to write a single driver for a device that is implemented for different busses (e.g. an SBus and VME framebuffer).

2.2 Loadable Drivers

All device drivers are loadable in SunOS 5.0. Driver modules are loaded dynamically as references are made to them. Unlike SunOS 4.x, the system can also unload drivers if they are idle. Drivers must contain support for automatic loading, and are recommended to support unloading. In particular, a driver must release all resources when it is unloaded to avoid memory leaks.

2.3 Overview of Device Driver Configuration

This section presents an overview of what the system does to link a device to its driver and make it accessible. The other major aspect of device driver configuration is what the system administrator does to add a new device and driver to the system (discussed in "Installation and Debugging" on page 15).

During boot, the system sets up a kernel-resident device information tree. This is a hierarchical list of the devices available to the system. In this paper, we will call it the devinfo tree. Each node in the devinfo tree (devinfo node) contains device-specific information, and includes properties retrieved from a self-identifying device's Fcode PROM or a non self-identifying device's hardware configuration file. The structure of the devinfo tree follows the hierarchy of the physical devices; bus, or "nexus" drivers such as `esp` (the Sun's onboard SCSI controller) have pointers to their children (e.g. disk drives), and are pointed to by their parent (the sbus driver for onboard `esp`).

The SunOS 5.0 kernel is self-configuring. The only drivers present immediately after normal booting are those that were required to boot (e.g. the disk driver, the framebuffer driver and the terminal driver for the keyboard). Other drivers are loaded when the device is accessed by an `open(2)` call on its special file entries in the `/dev` or `/devices` directories. (These directories are explained in "/dev and /devices" on page 4.)

The loading sequence is as follows.

1. The system copies the driver into the kernel virtual memory and calls its `_init(9e)` routine.

   All drivers must implement an `_init` routine, which must call the kernel support routine `mod_install(9f)` to link the driver to the devinfo node for the device.

2. For non-self-identifying devices, the system reads the driver's hardware configuration file (`xx.conf`) at this time and adds the information to the device's devinfo node.

3. The driver's `identify(9e)`, `probe(9e)` and `attach(9e)` routines are then called, each one being called only if the previous routine succeeds.

If all steps complete successfully, the device is now accessible, and the driver's `open(9e)` routine is called. The details of these routines are covered in the entry point sections in "Porting your Device Driver - Step by Step", below.

2.4 /dev and /devices

The `/devices` directory is new for SunOS 5.0. Devices are still accessed through entries in the `/dev` directory; however instead of special files, the entries in this directory are now symbolic links to the special files in `/devices`.

The entries in `/devices` are created by the `drvconfig(1m)` utility during a reconfiguration boot or when `add_drv(1m)` is run. `drvconfig` traverses the system's devinfo tree and creates an entry for each device found. The structure of `/devices` mirrors the devinfo tree, that is, nexus drivers are directories and leaf drivers are special files appearing in their parent's directory. The format of all entries is `<name>@<address>`, where `name` is the device's name in its devinfo node, and `address` is its address on its parent. For example, the full name of a SCSI disk might be:

```
/devices/sbus@1,f8000000/esp@0,800000/sd@0,0:a
```

The disk driver, `sd`, is at SCSI target 0, lun 0 on `esp`, the host adapter driver. The `esp` hardware is in sbus slot 0, at offset 0x800000, and the sbus itself is in "cpu address space" type 1 (iospace), at physical address f8000000. The `:a` appended to the disk device name identifies a particular minor device. See the section on `attach(9e)` below for more details.

The links in `/dev` are created by the `devlinks(1m)`, `disks(1m)`, `tapes(1m)` and `ports(1m)` utilities, based on the minor number information supplied in the `attach(9e)` routine (see the sections "xx_attach()" on page 10 and "Creating the /devices and /dev entries" on page 15 for more details).

The utilities to create `/devices` and `/dev` are run by the system during a reconfiguration boot (that is, boot with the `-r` flag), and also by the `add_drv(1m)` utility which is used to install a new driver into the system (see Section 4.4 on page 16). After adding a device for which the driver is already on the system (e.g. a new disk drive), you must do a reconfiguration boot to create the `/devices` and `/dev` entries for it. You must run `add_drv(1m)` to install a new device and driver to the system.

2.5 Kernel Differences which affect Drivers

The Solaris 2.0 kernel is multi-threaded. This is probably the single greatest difference from 4.x. The 4.x concepts of raising the priority level (`spl` and `splx`) no longer apply; neither do the 4.x `sleep()` and `wakeup()` calls. In Solaris 2.0, any one or more of the driver's routines may be running simultaneously on one or more processors. This includes the case where several copies of the same routine may be running at the same time. Solaris 2.0 has a number of DDI routines for thread synchronization. The extent to which multithreading affects your driver is quite driver-dependent. Drivers can vary from completely single threaded, to taking maximum advantage of the kernel's multithreading features. Multithreading is discussed in more detail in Section 5.0 on page 17.

In Solaris 2.0 several kernel data structures are now opaque. The devinfo pointer (type `dev_info_t`, passed to `identify`, `probe` and `attach`) is opaque. Drivers cannot access the elements of the devinfo structure. There are a number of DDI routines that take this pointer as a parameter and get information from your device's devinfo node, e.g. `ddi_get_name(9f)`, `ddi_get_parent(9f)`, `ddi_getprop(9f)`. The dev_t structure can also not be accessed directly, the routines `getmajor(9f)` and `getminor(9f)` are provided. The user and process data structures are also opaque and cannot be accessed from a driver.

Unlike in previous versions of SunOS, a driver can no longer access any information about the user process which has called the driver. This is because the user process may be on another thread, which could be running on another processor.

The 4.x concept of device *units* has changed. In Solaris 2.0, each device of a particular type is called an *instance*. The name and instance properties uniquely specify a devinfo node. The DDI routines `ddi_get_name(9f)` and `ddi_get_instance(9f)` are provided so that a driver can retrieve them. Device drivers should not count instance numbers (this was common in 4.x), there is no guarantee that the driver's `xx_identify()` or `xx_attach()` routines will be called in instance order. The only guarantee is that a given instance number will be an integer in the range of zero to the number of devices of that type attached to the machine. Instance number counting is now handled by the system framework. To get the instance number in any routine, call `ddi_get_instance(dip)`. Do not count instances or units anywhere, and do not make any assumptions about the values of instance numbers.

There is a group of routines that will manage per instance data structure allocation, `ddi_soft_state(9f)`. These provide dynamic structure allocation and deallocation. These routines are internally MT-safe.

The different subsystems which comprise the operating system kernel are located in the `/kernel` directory. The system device drivers are located in `/kernel/drv`. There is more information about locations for driver binaries in "Where to put the Driver's Binaries" on page 15. Because the kernel is auto-configuring, the `config(8)` utility is no longer used. Kernel configuration can be changed by modifying the `/etc/system` file and rebooting the system.

2.6 Device Driver entry points

All device driver entry points are specified in the driver's `dev_ops(9s)` and `cb_ops(9s)` structures. These replace the SunOS 4.0x `xx_ops`, `cdevsw`, and `bdevsw` structures. Loading and autoconfiguration entry points are discussed in section 3.3 and the other entry points are discussed in section 3.4.

3.0 Porting your Device Driver - Step by Step

Porting your device driver to SunOS 5.0 may seem like a daunting task. But it doesn't need to be. The effort can be divided into several relatively independent sections. The method presented here is one of several possible ways to divide the task The most important thing to remember is to proceed carefully, and understand each step. The steps are these:

1. Compile an initial "template" version of the driver, with only the loading routines implemented. All of the other entry points are stubbed out at this point
2. Implement and debug the autoconfiguration routines.
3. Port the remainder of the driver entry points.
4. Take advantage of desired new features available in SunOS 5.0.

A few more words of advice. Be sure that your testing setup is working reliably before you start the port. Trying to port a driver on a flaky peripheral will introduce too much aggravation. Don't start with that new prototype you just got. Use the old reliable board which has been working for months.

In the code examples, there are some structures and routines which normally include the name of the device or driver (such as `sd_ops` in the `sd` driver). The shorthand `xx` is used where the name of your driver goes. The generic version of the `sd_ops` structure would be called `xx_ops`.

3.1 Create an initial version of the driver

The very first step is to create an initial version of the driver. There are enough differences between 4.x and 5.0 that it's easier to start from scratch. The initial version should contain just the header information (`#include`, `#define`, and structure declaration lines) and the `_init(9e)`, `_info(9e)`, and `_fini(9e)` routines. All other routines should be empty at this point.

The easiest way to create the initial version is to get the *Preliminary Solaris 2.0 Device Driver Migration Kit* from Sun. It's available at no charge by calling 1-800-742-4484 (+415-336-0390 outside the US). The kit consists of a diskette of sample drivers and paper documentation. If one of the sample drivers is close to what you need, then most of the work has been done for you. All of the samples will compile and install cleanly. Otherwise, pick the closest sample and delete (or comment out) the contents of all routines *except* `_init()`, `_info()` and `_fini()`.

An alternative method of creating the initial driver is to copy the example code out of the *SunOS 5.0 Writing Device Drivers* Manual.

3.2 Compile the Driver under 5.0

The second step is to get the driver to compile. This step is as much to make sure that the C compiler is properly installed and licensed as to debug the driver. For the sample drivers in the Migration Kit, makefiles are supplied.

3.2.1 Install C compiler

Since Solaris 2.0 does not include a bundled C-compiler, you should verify that the C compiler is properly installed on the SunOS 5.0 system where you will be compiling the driver. Compiling a simple "hello,world" program using Make should identify any problems with compiler installation or your PATH environment variable. All the examples in this paper assume that you are using the SunPro C compiler from Sun Technologies.

Many of the system `#include` files are not ANSI-C compliant. For this reason, you cannot compile drivers with the C++ compiler. Also, compiling with the ANSI-C conformance flags set (`-Xs` or `-Xc`) will not work.

A warning. It's easy to run `/usr/ucb/cc` inadvertently instead of `/opt/SUNWspro/bin/cc`, because `/usr/ucb` is in the default path. The spro cc must be first in path. Run "`which cc`" to verify that you are running the right compiler.

3.2.2 Modify Makefiles

The flags which you pass to `cc` will need to change in your Makefile. Also, you must do an `ld` step, even if your driver has only one `.o` file. You can no longer load a `.o` file for a loadable device driver.

So, for a driver called `example`, which is produced from two `.c` files, `example_one.c` and `example_two.c`, and one assembly-code file, `example_asm.s`, the following compile lines would be used to produce the loadable driver module:

```
cc -D_KERNEL -I. -c example_one.c
cc -D_KERNEL -I. -c example_two.c
as -P -D_ASM -D_KERNEL -I.-o example_asm.o example_asm.s
ld -r -o example example_one.o example_two.o example_asm.o
```

(This example assumes that the header files for `example` are in the same directory as the `.c` and `.s` files.)

Using architecture-dependent compiler directives (e.g. `-Dsun4c`) is not DDI-compliant. Leaf device drivers don't need to know about architecture specifics. An advantage of this policy is that you need not compile the driver on any specific kernel architecture. A driver compiled on any SPARC machine should run on any other SPARC machine running the same major revision of SunOS.

The module loading code in the kernel is picky about the name of your module. You should not link it as one name, then move it to another name before loading the module into the kernel.

3.2.3 Include Files

All of the kernel include files are in the directory `/usr/include/sys`. The header files have been reorganized, so your `#include` directives may change. Three header files you will certainly need are:

```
<sys/devops.h>
<sys/ddi.h>      /* these two   */
<sys/sunddi.h>   /* must be last */
```

The files `ddi.h` and `sunddi.h` *must* appear last in your list of kernel include files.

The file `<sys/vddrv.h>` has changed name to `<sys/modctl.h>`.

For full DDI compliance, you are only allowed to include kernel header files which are listed in the Synopsis sections of the section 9 manual pages. No other kernel header files may be included.

3.2.4 Structure Changes

The `xx_ops` structure has changed form. See the `dev_ops(9s)` manpage. If you have no `devo_reset` routine, use `nodev`. Use `nulldev` if you have no `probe(9e)` routine. Do not use `nodev` for `xx_probe()`! This always fails, which will cause the entire driver installation to fail as well.

Define an `xx_cb_ops` structure following the `cb_ops(9s)` manpage. This structure replaces the 4.x `cdevsw` and `bdevsw` structures. Use `nodev` for the `xx_devmap` entry. Use `nodev` for the `devo_detach` if your driver will not have an `xx_detach()` routine. Set the `cb_flag` field to `(D_NEW | D_MP)` -- your driver should be MT-safe from the start.

The struct `vdldrv` name has changed name to `modldrv`, and its form has also changed. See the `modldrv(9s)` manpage. Be sure to declare `mod_driverops` as an extern structure as well.

Add the new struct `modlinkage`. See the `modlinkage(9s)` manpage.

If your 4.x driver used the `mb` structures, you must restructure it. The elements of the `mb_driver` struct are (more or less) contained in the `dev_ops` struct. The elements which used to be in the `mb_ctlr` and `mb_device` structures are now opaque data in the `dev_ops struct.` The information in your `xx.conf` file will cause some properties to be created for your driver when it is loaded.

3.2.5 Other Compile Notes

If under 4.1 you used to use either of these constructs:

```
#ident "<some SCCS stuff>";
static char SCCSid[] = "<some SCCS stuff>";
```

you should change it to the #pragma form, with no semicolon:

```
#pragma ident "<some SCCS stuff"
```

The type `addr_t` is gone. It has been replaced with the type `caddr_t`.

Just to get the driver to compile, you can stub out and just prototype the new routines needed in the `dev_ops` and `cb_ops` structures, such as `xx_getinfo`. You will need to write these routines before the driver has any chance of loading, though.

3.2.6 Good Practices

Since you are now using an ANSI C compiler, there are several new features that are useful.

All entry points should be prototyped. These prototypes will cause you to be informed at compile time of parameter type and number mismatches, rather than mysterious error messages and/or panics at run time.

Use the new `volatile` type specifier to declare software structs which overlay actual hardware registers, or for specifying any other location which could be updated by an alternate thread of control. Also, use the new `const` type specifier an object or array which is initialized, but then never changed. See any good ANSI-C manual for further details.

3.2.7 Try the Compile

Go through the driver source code, and comment out the interior of all the entry points, so that, for instance, the `xx_open()` routine looks like this:

```
static int
xx_open(dev_t *dev, int openflags, int otyp, cred_t *credp)
{
#if 0
  . . . rest of open code here
#endif
  return(ENXIO);
}
```

Also, completely comment out all the utility routines in your driver which are only used internally. The driver source code should now compile. Any compile errors at this point are structural rather than functional. When the code compiles at this point, you know your Makefile works, your `#include` directives are correct, you have all the structures defined which the kernel uses, and all the necessary entry points are present. The next step is to get the driver to load into the kernel without error.

3.3 Autoconfiguration and Installation

First a note about `modload` versus first-reference loading. In 4.x, the only way to load a loadable driver was with the `modload` utility. In 5.0, drivers are automatically loaded when their corresponding node in `/devices` is referenced. However, for the automatic load to work, the driver must have been installed with `add_drv(1m)`. So the next step after compiling the initial version of the driver is to install it with `add_drv`.

It's important to understand the difference between `modload(1)` and `add_drv(1m)`. `modload` just copies the driver text into the kernel and resolves symbols. It does not do any driver configuration -- only the `_init()`, `_info()`, and `_fini()` driver routines are called. To add a new driver to the system, you must run `add_drv(1m)`, which configures the driver into the system in addition to loading it into the kernel. See "Running add_drv(1m)" on page 16 for details.

The following driver entry points must be working for `add_drv` to work properly:

```
_init(9e)
_fini(9e)
_info(9e)
xx_getinfo()
xx_identify()
xx_probe()
xx_attach()
xx_detach()
```
`xx.conf` (hardware configuration file, non self-identifying devices only)

You should stub out the rest of your driver routines while debugging the loading/unloading. Once you can install the driver successfully with `add_drv`, much of the work will be done for simple drivers. When you stub out a routine, be sure to make it FAIL, not succeed. This will avoid frustrating system panics.

A preliminary note: the kernel `log()` function and kernel `printf()` have been replaced by `cmn_err()`. See the `cmn_err(9f)` manpage. Note that `CE_PANIC` will panic the system after printing your message, and that some levels of `cmn_err` generate automatic labels before your message, and newlines after.

The routine you might have had in 4.x called `xx_init`, or `xx_vdcmd` which had three sections labelled `VDLOAD`, `VDUNLOAD` and `VDSTAT`, has now been (approximately only) replaced with the three entry points, `_init`, `_fini`, and `_info`.

3.3.1 _init(9e)

This routine should do any per-driver (rather than per-instance) initialization, such as allocating per-driver memory. This routine is called before `xx_identify()`, `xx_probe()`, and `xx_attach()`, so don't assume anything about instance numbers or instance state structures being present.

If you are using the `ddi_soft_state(9f)` routines to manage per-instance data structures, your should call `ddi_soft_state_init(9f)` here.

Anything which used to be in the `VDLOAD` case of `xx_init` belongs here, except that you no longer need to call `vd_installmod`.

`_init` must call `mod_install(9f)` with the address of your `modlinkage` struct.

3.3.2 _fini(9e)

This routine prepares a driver module for unloading. It should deallocate any per-driver resources and call `mod_remove(9f)` and (if appropriate) `ddi_soft_state_fini(9f)`. It's typically only a few lines long.

`_fini()` should return the return code from `mod_remove()`.

3.3.3 _info(9e)

This routine is passed a `modinfop *`. You should return the return value from `mod_info(9f)`:

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

You can also print out information about your driver. You may find the `__TIME__` and `__DATE__` preprocessor directives handy to record the compile date of the driver.

3.3.4 xx_getinfo()

This new routine (see `getinfo(9e)`) returns a `dev_info` pointer for the `dev_t` (`DDI_INFO_DEVT2DEVINFO` case) or the instance corresponding to the `dev_t` (`DDI_INFO_DEVT2INSTANCE`) case.

This routine may be called at any time, including before the device has been probed or attached, so it should not depend on the presence of the hardware.

3.3.5 xx_identify()

Unlike 4.x, 5.0 does not call your identify routine with the name of every node in the devinfo tree. Instead, the `xx_identify()` routine is called only with the name of the devinfo node that the system believes is the correct one. SunOS 5.0 performs a form of *lazy evaluation*; drivers are not attached to their devices until necessary. This means that if for example there is more than one instance of your device present on the system, each instance will be attached separately. That is, the `xx_identify()`, `xx_probe()`, and `xx_attach()` routines may be called multiple times, and the calls for different instances may be interleaved. The only guarantee is that for a given instance, the routines will be called in order, and each is called only if the preceding routine succeeded.

The `name` is no longer passed to you, you must use `ddi_get_name(9f)` to get it.

Consider this another good reason to use the new `ddi_soft_state_*` routines to manage your state structures. (They're internally MT-safe, too.) You can no longer count on `xx_identify()` being called for all instances before `xx_attach()` is ever called. The return value should be `DDI_IDENTIFIED` or `DDI_NOT_IDENTIFIED`.

3.3.6 xx_probe()

Drivers for non self-identifying devices must implement a `probe(9e)` routine. For self-identifying devices, the system performs the probe functions, and the driver should specify `nulldev(9f)` for the probe routine in its structure. Do not use `nodev(9f)`; this always fails and will prevent the driver from installing into the system.

The probe routine is called if the driver's `identify(9e)` routine succeeded. It tests for the presence of the expected device, and can also verify that the device is working correctly. The routine should cause no side-effects or aftereffects; that is, any space allocated should be freed, and no external data structures should be modified.

If the probe routine succeeds, it should return the symbol `DDI_PROBE_SUCCESS`, or `DDI_PROBE_FAILURE` on failure. The routine may also return `DDI_PROBE_DONTCARE` if the probe failed but the driver's `xx_attach()` routine should be called anyway. A fourth value, `DDI_PROBE_PARTIAL`, will be implemented in a future release, so it should not be used. In Solaris 2.0, if the probe fails (returns failure or partial), the device's devinfo node is discarded.

You must now use `ddi_map_regs` to map in your registers. don't forget to unmap them before returning.

If your device has SBus (self-identifying, probe not needed) and VME (non self-identifying) versions that are otherwise the same, you can use the routine `ddi_dev_is_sid(9f)` to detect the device type and decide whether further probing is necessary.

3.3.7 xx.conf file(s)

Hardware configuration files are required for all non self-identifying devices. The hardware configuration file and the `xx_probe()` routine together perform the same functions as a self-identifying device's FCode PROM and the system boot PROM. That is, they ascertain whether the expected device is present.

The usual form of the file is:

```
name="xx" parent="string" reg=bus-type,phys-addr,map-size
        interrupts=bus-level,vector
        property=value;
```

The `parent` string is the name of the nexus to which the `xx` device is attached, e.g. `"vme"`. The `reg` property is parent-device specific. It tells the device's parent where the device is in the parent's address space. For SBus devices, *bus-type* is the slot number, *phys-addr* is the offset of the start of the device's memory (e.g. registers) in the slot, and *map-size* is the length of the device's memory. For VME devices, *bus-type* is the VME address space and the other two fields are the same as for SBus. The VME space is one of the `#defines` in the header file `/usr/include/sys/bustypes.h`. In the `interrupts` property, *bus-level* is the device's interrupt level on the bus, and *vector* (for busses with vectored interrupts) is the device's interrupt vector. You can add arbitrary properties for your device by specifying *property*=*value* pairs in the hardware configuration file. These properties will be in the devinfo tree (and available to your driver) at the time its `xx_probe()` or `xx_attach()` routine is called.

3.3.8 xx_attach()

A driver's `attach(9e)` entry point is called if its `probe(9e)` routine succeeded. Note that the arguments to `xx_attach` have changed from 4.x.

As noted previously, don't count instances anywhere. Use `ddi_get_instance(9f)` to get your instance number. This can then be used with the routine `ddi_soft_state_zalloc(9f)` to allocate a per-instance data structure (you must have called `ddi_soft_state_init(9f)` in the `_init(9e)` routine first). Thereafter, call `ddi_get_soft_state(9f)` with the instance number to retrieve the per-instance data structure. For example:

```
if (ddi_soft_state_zalloc(statep, instance) != DDI_SUCCESS)
        return(DDI_FAILURE);
instp = ddi_get_soft_state(statep, instance);
```

The `ddi_soft_state` routines are for your convenience only. You can still use the `kmem_*alloc()` routines to allocate space; note that they now require a second (`flags`) argument. See the `kmem_alloc(9f)` and `kmem_zalloc(9f)` manpages.

A warning for SCSI drivers! The SCSI host adapter driver (`esp`) calls `ddi_set_driver_private(9f)` on the target drivers' devinfo node, so target drivers must not call `ddi_set_driver_private` themselves. Instead, call `ddi_get_driver_private` to get the preallocated `scsi_device(9s)` structure, and use the `sd_private` field in it for private data.

The `slaveslot()` routine has been replaced with `ddi_slaveonly(9f)`.

Use `ddi_map_regs(9f)` to map in your registers where you used to use `map_regs()` or `mapin()`.

If you need iopb memory, the call

```
rmalloc(iopbmap, size)
```

can be replaced with

```
ddi_iopb_alloc(9f)
```

then

```
ddi_dma_addr_setup(9f).
```

You may also need to call `ddi_dma_devalign(9f)`.

SunOS 4.x SBus drivers which followed the example of the sample driver in the *Writing Device Drivers for SBus* manual probably have an `xx_poll()` routine (registered with `addintr()`) which determines which unit is interrupting, then calls `xx_intr()` with the unit number. This is no longer necessary. Your SunOS 5.0 driver's `xx_attach()` should call `ddi_add_intr(9f)` for each instance, passing `xx_intr` (your interrupt routine) as the `int_handler` argument and the instance number or a pointer to the instance's data structure in the `int_handler_arg` argument. This is a more general approach, and should work without modification if your device ever exists on a bus with vectored interrupts, such as VME. A warning. It is possible that your interrupt routine may be called after the attach routine has called `ddi_add_intr()` but before calling `mutex_init(9f)` (if another device interrupts on the same interrupt level). Your interrupt routine must therefore be able to spot this and return `DDI_INTR_UNCLAIMED`.

If your device interrupts at a level above that of the system scheduler, you cannot use the normal interrupt registration mechanism, or any of the normal kernel synchronization mechanisms (e.g. conditional variables). The routine `ddi_intr_hilevel(9f)` determines if your device is in this category. If it is, you must write a fast high level interrupt routine that triggers a lower priority software interrupt to actually handle the device. Your high-level interrupt routine is very restricted - it can only call `mutex_enter(9f)`, `mutex_exit(9f)`, `ddi_trigger_softintr(9f)` and local private routines. You can still use `mutex(9f)` in the high level routine; `mutex()` will spot the special case and spinlock. Note that you cannot attempt to acquire a non-high-level mutex after having acquired a high-level mutex. Your xx_attach() routine should call ddi_add_softintr(9f) (and xx_detach() should call ddi_remove_-softintr(9f)), and the high-level interrupt routine should call ddi_trigger_softintr(9f) to trigger the software interrupt.

It is no longer necessary to notify the system that you are using DVMA with a call to something like adddma.

Any per-instance mutexes should be initialized here by calling mutex_init(9f). See "Multithreading your driver" on page 17 for more details on mutexes.

The attach routine must inform the system about the device instance's minor number(s) by calling ddi_create_minor_node(9f). Minor numbers are used only by the driver; the instance number is commonly encoded into it. If the device instance is subdivided (e.g. a disk is partitioned, or a serial multiplexer has multiple ports), the minor number is used to distinguish which subdevice is being addressed. ddi_create_minor_node must be called once per minor device. The system uses the information to create the device's entries in /devices and /dev. The syntax for ddi_create_minor_node is

```
ddi_create_minor_node(devi, name, spec-type, minor, node-type, 0);
```

When drvconfig(1m) creates the entry for this minor device in /devices, it will append a colon and the value of the name field to the entry. If this instance of the device has multiple minor nodes, each call to ddi_create_minor_-node must specify a different name. The spec-type parameter specifies a character or block device, minor is the actual minor number (18 bits long), and node-type is a #define from <sys/sunddi.h>.

When devlinks(1m), disks(1m), tapes(1m), or ports(1m) run, they use the node-type and name parameters to match the device, and create a link to it in /dev. If your device is a disk, tape or serial driver, you can use the appropriate DDI_TYPE #define value (e.g. DDI_NT_BLOCK_CHAN for a SCSI disk driver), and the utility will create an appropriate entry in /dev. However if your device does not fit into one of these categories, you should use the DDI_PSEUDO #define and add an entry to the file /dev/devlinks.tab to get devlinks(1m) to create the link in /dev. See "Creating the /devices and /dev entries" on page 15 for more details.

xx_attach should be careful to undo any per-instance allocations and call ddi_remove_minor_node(9f) as needed if an instance fails to attach. (It may do this by calling xx_detach(). Be very careful to watch for recursive mutex_enter calls here.

The peek and poke calls have been replace with ddi_peek(9f) and ddi_poke(9f). There are ddi_peek* calls for all sizes.

You may have to re-architect your driver since the mb structures are gone. The info in your xx.conf file will be turned into properties in your dev_info structure by the module loading process. Use ddi_get*prop(9f) to fetch these properties.

If your driver supports multiple types of devices, you should use ddi_prop_create(9f) to create descriptive properties which the other driver routines may use to determine what to do. You may create as many properties as appropriate for your device. Note that the information in your xx.conf file will be turned into properties in your dev_info struct by the module loading process automagically.

3.3.9 xx_detach()

The detach(9e) routine allows your driver to be automatically unloaded to save system resources. It must release any resources allocated in your xx_attach() routine. It is up to the driver to *allow* unloading. The system won't unload a driver unless its xx_detach() routine succeeds (i.e. returns DDI_SUCCESS). The driver can refuse to unload either by returning DDI_FAILURE or by specifying nodev(9f) in its dev_ops(9s) structure. However this is somewhat antisocial, you should implement a `xx_detach()` routine. If you don't have a detach routine, you'll have to reboot the workstation to change the driver, for debugging or to upgrade it.

If you previously had a decommissioning subroutine which you called in the `VDUNLOAD` step of your `init_module` routine, you can probably rework it and make it your `xx_detach` routine.

If any of your `xx_identify()`, `xx_probe()`, or `xx_attach()` routines fail, the system will call `xx_detach()` and then `_fini()`. It is a common error for code in `xx_detach()` to depend on resources which might have been allocated in `xx_attach()`. Your `xx_detach()` routine should always check to see if a resource has been allocated before deallocating it. Or, to put it another way, a call to `xx_detach()` at the very beginning of your `xx_attach()` routine should not panic the system.

3.4 Get the rest of the driver entry points to work under 5.0

At this point, your driver should load and unload cleanly, with no system errors. Of course, you cannot open the device or move any data yet. Now you can proceed to port or implement the remainder of the device driver entry points.

3.4.1 xx_open()

The first arg to `open(9e)` ONLY is a `dev_t *`, not a `dev_t`.

It is good practice to 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 defines are in `<sys/open.h>`.

You are guaranteed that the system will not call the `open` routine for a particular instance unless the `identify`, `probe` and `attach` routines for that instance have been called and have returned. But this is per-instance, not per-driver.

3.4.2 xx_close()

Note that `close(9e)` now has two more arguments.

3.4.3 xx_read() and xx_write()

If you used to check user privileges using `suser()`, you should now use `driv_prv(9f)` instead.

You can no longer access any information about the user process which called the driver, since the user process and the kernel process are not on the same thread, and might be running on different processors.

You should not allocate `buf(9s)` structures statically anywhere. Instead, use `getrbuf(9f)` to allocate a buffer and `freerbuf(9f)` to release it. Also, `physio(9f)` can be passed `(struct buf *)0`, in which case it will provide a `buf` struct for the strategy routine.

If you are a DVMA device, and your dma engine has limited bits of resolution (such as the top 8 bits being registered rather than a counter), then you should set up a `ddi_dma_lim_t` structure describing these limits. Pass the address of this struct as an argument to `ddi_dma_buf_setup(9f)`. See `ddi_dma_lim(9s)`. Store the handle returned by `ddi_dma_buf_setup(9f)` in the per-instance data structure so you can free it later with `ddi_dma_free(9f)`.If you used to allocate your buffers uncached, you can now use `ddi_dma_sync(9f)` whenever you must require cache consistency. You need not allocate your buffers uncached.

If you need to DVMA data larger than `minphys(9f)` allows, see the `ddi_dma_movwin(9f)` manpage for details on the new sliding DVMA window feature.

The `sleep()` and `wakeup()` mechanism has been replaced with the `condvar(9f)` group of support routines (e.g. `cv_wait(9f)` and `cv_signal(9f)`). See "Multithreading your driver" on page 17 for more details.

3.4.4 xx_ioctl()

Note that `ioctl(9f)` now has two more arguments, a credentials pointer (the same as for the other entry points), and a return value pointer which may be set in addition to the normal return value.

If you used to define your ioctl cmd args this way:

```
#define     MYIOCTL   _IOR      (m, 1, u_int)       /* m is not in quotes */
```

you will have to surround the letter m with single quotes to get the `#define` to work. Or you can use the new definition method: (left-shifted letter or-ed with number):

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

Theoretically, the magic number you use (`DIOC` above) should be allocated by AT&T.

The `cred_p` can be used to check credentials on the call, and the `rval_p` can be used to return a pointer to a return value which means something (as opposed to the old method of always getting zero back for success). See the `ioctl(9e)` manpage.

You can no longer dereference `arg` directly. You must use `copyin(9f)` and `copyout(9f)` to move the data around. Note that the buffer pointer argument to `copyin()` and `copyout()` must be cast to `caddr_t`. The `cmds` which were defined as `_IOW` must use `copyin()`, `_IOR` must use `copyout()`, and `_IOWR` must use both.

3.4.5 xx_strategy()

Note that xx_strategy should now return an `int`. The value is ignored, so return whatever you like. Zero is safe.

If you used to find your unit number from the buf struct this way

```
unit_no = getminor(bp->b_dev);      /* uses b_dev field */
```

it must change to

```
instance_no = getminor(bp->b_edev); /* uses b_edev field */
```

If you were using `mb_mapalloc()` before, you should change that to using `ddi_dma_buf_setup(9f)`. If no `arg` is necessary, use `(caddr_t) 0`. You will need to use `ddi_dma_htoc(9f)` 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 you're programming a DVMA engine from this routine, the note about setting up a `ddi_dma_lim_t` in the section "xx_read() and xx_write()" on page 13 applies here.

The semantics of `timeout(9f)` and `untimeout(9f)` have changed. `timeout()` returns an identifier, which must be saved to pass to `untimeout()`. This may require changes to your per-instance structure, if you stored pending time-out information in any other form than an int for each timeout there could be.

Save your `buf(9s)` address passed to `xx_strategy()` so you can call `biodone(9f)` after the I/O completes, to wake up `physio(9f)`.

3.4.6 xx_chpoll()

This routine is not related to the SunOS 4.x `poll()` driver entry point. That routine is not needed in SunOS 5.0.

The SunOS 5.0 `chpoll(9e)` routine performs the same function as the 4.x `select()` driver routine: it provides a polling entry point for non-STREAMS character drivers. That is, it supports the `poll(2)` system call (which itself replaces the 4.x `select(2)` system call).

The manpage for `chpoll(9e)` describes how to implement this routine. Basically, it must create a `pollhead` structure and call `pollwakeup(9f)` when an event occurs.

4.0 Installation and Debugging

Installing a new driver for the first time is straightforward. Simply copy the driver module and hardware configuration files into the appropriate directory and run `add_drv(1m)`. Do not use `modload(1)` to install a new driver! All `modload` does is copy the driver into the kernel and resolve its symbols; it does not do any of the other configuration processing (such as calling `identify`, `probe` or `attach`).

4.1 Where to put the Driver's Binaries

You can install the driver module and hardware configuration file anywhere you like; however there are some rules. The system will look for driver modules in the directories `/kernel` and `/usr/kernel` by default. The directory `/kernel` is for drivers that may be needed before `/usr` is mounted - e.g. for all bootable devices. All other drivers can go into `/usr/kernel`. If you want to keep your driver separate from others in a different directory, e.g. `/opt/drivers-R-us`, you must add the directory to the `moddir` entry in the file `/etc/system` (see the `system(4)` manpage):

```
moddir:     /kernel:/usr/kernel:/opt/drivers-R-us
```

This entry overrides the system default (`/kernel` and `/usr/kernel`), so the default directories must be in it if it is defined. You should use this only for developing a driver; changing the setting of `moddir` on a production system is risky. It's best to install drivers into `/usr/kernel`. Putting the driver in `/usr/kernel` also allows it to be shared between machines of the same architecture.

4.2 Setting System Parameters

If your driver has variables in it that can be changed to set its behavior (e.g. a timeout value), you can set them in `/etc/system` (see the `system(4)` manpage) by adding an entry of the form:

```
set xx:xx_param=<value>
```

Where `xx` is the driver module name, `xx_param` is the variable to be set, and `<value>` is what it's to be set to. The `/etc/system` file is read once during boot, so if you change it you must reboot for the changes to take effect. Thereafter, when the driver is installed into the kernel the variable will be set.

4.3 Creating the /devices and /dev entries

The special file entries in /devices are created by the `drvconfig(1m)` utility. `drvconfig` is run when you do a reconfiguration boot (`boot -r`) or by the `add_drv(1m)` utility. You should not use `mknod(1m)` to do this.

If your device is a disk, tape, or serial port, use the appropriate setting for the `node-type` parameter in `ddi_create_minor_node(9f)`, see `<sys/sunddi.h>` to make the `disks`, `tapes`, or `ports` utility create links in `/dev`. Otherwise, you can either create the links manually, or add an entry to the file `/etc/devlink.tab` to cause `devlinks(1m)` to create the link. If you want devlinks to create the link, you must use `DDI_PSEUDO` as the node type parameter in ddi_create_minor_node(9f). If you use any other #define value, one of the other utilities will see it and create a wrong entry. The format of the entries in /etc/devlink.tab is described in the file itself.A typical entry for a generic SCSI character driver might be:

```
type=ddi_pseudo;name=sst;minor=character      rsst\A1
```

Here, the type field matches the DDI_PSEUDO specified as the node-type, and the minor field matches the name field in the call to ddi_create_minor_node. The name field matches the device's name in its devinfo node. The rest of the entry, after the tab separator, specifies the name of the link in /dev. Here, it is the literal string rsst with the first part of the device's address field from its /devices entry appended. In this case (a SCSI device), the value is the SCSI target number.

4.4 Running add_drv(1m)

This command (in /usr/sbin) will install a new driver into the system. You must run this command when adding a new driver, the system will not be able to link the driver to the device until you have done so. Here is its syntax:

```
add_drv [-m 'permission','...'] [-i 'identify_name,'...'] driver_module
```

The -m option allows you to set the access permissions and ownership of the /devices entries. The format is:

```
<minor> <perm> <user> <group>
```

minor is the symbolic minor device name (as specified in ddi_create_minor_node), and may be wildcarded with an asterisk. perm is the normal SunOS permissions (e.g. 0666) user and group are the user and group names for the file's owner. If you run add_drv without specifying minor permissions, they will default to 0600, owned by user root, group sys. You can set different permissions for different minor numbers by specifying multiple permissions. The add_drv manpage has a good example of this.

The -i option allows you to specify a different name for the driver. The most likely time you'd use this is when the device name (in its FCode PROM or hardware configuration file) is not the same as the driver module name. When you run add_drv, it will look for a device with the same name as the driver module. If the names are different add_drv will fail. For add_drv to work in this case, you must give it the device's PROM name as an alias (-i <PROM name>). For example, Sun's parallel port SBus card is called SUNW,bpp in its Fcode PROM, but the driver is called bpp. To install this driver into the system, you would run add_drv this way:

```
add_drv -i "SUNW,bpp" bpp
```

There is a symmetric driver removal script called rem_drv.

4.5 Loading a New Version of the Driver

Once you've successfully installed a new driver into the system, you may need to update it, i.e. replace the module with a new version. There are two "levels" at which you can do this. You can completely remove and reinstall the driver by running rem_drv(1m) and then add_drv(1m); or you can simply unload the module and reload the new version. Both methods will exercise the driver's probe() and attach() routines,. The difference is that add_drv informs the system about a new driver and builds /devices entries for it, so if you change the device's properties (e.g. a SCSI device's target number), you should run rem_drv and add_drv.

To reload a driver module, first run the modinfo(1) utility to find the module's id number and then run modunload(1) with the id to unload the module:

```
# modinfo
Id   Loadaddr   Size  Info  Rev   Module   Name
115  ff50a000   3bf8  85    1     sst      (SCSI Simple Target Driver)
# modunload -i 115
```

After unloading the module, replace the driver binary with the new version. The next access of the driver will cause the new version to be loaded automatically. Alternatively, you can use `modload(1)` to forcibly load the new version.

4.6 Verifying the hardware

There are two ways to verify that the system has seen your hardware:

- From SunOS, the `prtconf(1)` command will print the kernel's devinfo tree. With the `-v` option, the FCode properties are also shown.
- From the boot PROM (rev 2.0 or later) `ok` prompt, you can use the `cd` and `ls` commands to navigate the PROMs devinfo tree. The `.attributes` command lists the current node's properties. The `show-devs` command will list all the devices in the devinfo tree.

4.7 Debugging

Unlike SunOS 4.x, a driver's symbols are available for use with adb. Clearly though, the symbols will not be available until the driver has been loaded. Since the kernel file, `/kernel/unix`, is just a core kernel, you cannot use it to debug a driver. Instead you should use the special device node `/dev/ksyms`:

```
adb -kw /dev/ksyms /dev/mem
```

You can also use `kadb` in the same way as in 4.x, although you can't put breakpoints into the loading or autoconfiguration routines of the driver (since the symbols won't be present beforehand).

You can vary the value of the system global `moddebug` to see what's happening at module loading/unloading time. The possible values are in `<sys/modctl.h>`.

The `trace(1)` system utility has changed name to `truss(1)`.

5.0 Multithreading your driver

As mentioned elsewhere in this paper, Multithreading (or "MT" for short) is one of the major differences between SunOS 4.x and 5.0. all drivers are affected by MT. The SunOS 5.0 kernel is multithreaded and fully preemptible. This means that your driver routine can be preempted at any time in favor of another thread. For example, if two applications make read requests through your driver, both requests can run simultaneously, on the same or different processors.

Multithreading implies that you must implement your driver in a way that shared data is protected from corruption by different threads of control in the driver, and also that different threads are correctly synchronized. For example, if both your strategy and interrupt routines access the device, you must implement safeguards so that they cannot both access the hardware at the same time. The same applies to data that is shared between threads, e.g. the per-instance data structure. Solaris 2.0 provides several DDI routines for thread control: `mutex(9f)` and `condvar(9f)` are the most commonly used. There are also semaphores and readers-writer locks. The next few sections introduce these routines; examples of their use are in later sections.

There is one exception to the statement that multiple instances of any driver routine can be executing simultaneously. The 2.0 kernel will not dispatch an interrupt thread into a particular interrupt level until the previous interrupt thread at that level has returned. Rephrasing, Interrupts at a given level L are single-threaded per system. This may change in future releases of Solaris but is true for Solaris 2.0.

5.1 Mutexes

The `mutex(9f)` mutual exclusion locks allow access to data to be controllable. When a mutex lock is held, other threads are prevented from accessing the locked data. This locking is entirely advisory -- when a thread wants to access shared data it should first acquire the lock. If the lock has already been acquired by another thread, the second thread will block until the first thread releases the lock.

To use a mutex, you must first initialize it, typically in the driver's `xx_attach()` routine:

```
mutex_init(kmutex_t *mp, char *name, kmutex_type_t type, void *arg);
```

The `type` parameter must be set to `MUTEX_DRIVER`, and `arg` is a pointer to the `iblock_cookie_t` filled in by `ddi_add_intr(9f)`. The mutex must also be destroyed in the driver's `xx_detach()` routine:

```
mutex_destroy(kmutex_t *mp);
```

A warning. It is possible that your interrupt routine may be called after the attach routine has called `ddi_add_intr()` but before the mutex has been initialized (if another device interrupts on the same interrupt level). Your interrupt routine must therefore be able to spot this and return `DDI_INTR_UNCLAIMED`.

To lock data, surround accesses to it with calls to the `mutex_enter()` and `mutex_exit()` routines:

```
mutex_enter(kmutex_t *mp);
/* access to shared data */
mutex_exit(kmutex_t *mp);
```

There are also some other mutex routines:

```
mutex_owned(kmutex_t *mp);
```

returns nonzero if the mutex is owned by the calling thread. It is most commonly used in an `ASSERT(9f)` statement. Remember that `ASSERT` is only active if you compile your driver with the `-DDEBUG` flag.

```
mutex_tryenter(kmutex_t *mp);
```

is a non-blocking `mutex_enter()`. If the mutex is held, it will return zero, otherwise it acquires the mutex and returns nonzero.

5.2 Condition Variables

Condition variables allow threads to synchronize with each other. The `cv_wait(9f)` routine will block the calling thread until another thread calls `cv_signal(9f)`. These routines provide similar functionality to the SunOS 4.x `sleep()` and `wakeup()` mechanism.

Before using a condition variable, initialize it, usually in the driver's `xx_attach()` routine:

```
cv_init(kcondvar_t *cvp, char *name, kcv_type_t type, void *arg);
```

The name parameter is a string used for statistics and debugging, type must be set to CV_DRIVER, and arg must be set to NULL. When the driver is finished with the condition variable, it should be destroyed, usually in the xx_detach() routine:

```
cv_destroy(kcondvar_t *cvp);
```

Condition variables are used to check a condition while holding a mutex to prevent other threads from changing the condition. If the condition is such that the thread should block (waiting for the condition to change), it calls cv_wait():

```
cv_wait(kcondvar_t *cvp, kmutex_t *mp);
```

cv_wait() suspends the calling thread and releases the mutex atomically, so that another thread holding the mutex cannot signal on the condition variable until the first thread is suspended. cv_wait() reacquires the mutex before returning.

A condition is signalled and a single blocked thread is woken by calling cv_signal():

```
cv_signal(kcondvar_t * cvp);
```

Alternatively, all threads blocked on the condition variable can be woken by calling cv_broadcast(9f):

```
cv_broadcast(kcondvar_t *cvp);
```

The routine cv_wait_sig(9f) is similar to cv_wait(), but is interruptible by a signal being sent to the thread (e.g. by kill(1)). If this occurs, cv_wait_sig() returns zero; it returns nonzero if another thread called cv_signal() or cv_broadcast().

cv_timedwait(9f) is also similar to cv_wait(), except that it returned zero if the specified time elapses before the condition is signalled:

```
cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, long timeout);
```

The timeout value is specified in absolute clock ticks since the system was last rebooted. The current time may be found by calling drv_getparm(9f) with the argument LBOLT.

5.3 Other Synchronization and Locking Mechanisms

There are two other multithreading control mechanisms: semaphores and readers-writer locks.

A semaphore has a value that is atomically decremented by sema_p(9f) and atomically incremented by sema_v(9f). If sema_p() is called and the value is already zero, the calling thread will block until another thread calls sema_v(). Semaphores must be initialized and destroyed like mutexes and condition variables. See the semaphore(9f) man page.

Readers/Writer locks are similar to mutexes in that they protect access to data, but they distinguish between read and write accesses. Any number of threads can obtain a read-only lock (multiple readers), but only one thread may hold a write lock (single writer). That is, if you try to acquire a writer lock while any reader locks are held, the thread will be blocked until all reader locks are released. After acquiring the writer lock, any threads attempting to acquire a reader or writer lock will block until the thread holding the writer lock releases it. This type of lock is useful if your driver has shared data that is read more often than it is written. However, Readers/Writer locks are much more expensive than mutexes. See the rwlock(9f) manpage for details.

5.4 Multi-threading a Driver

Multi-threading a driver is not an either-or proposition; there is a gradient of “multi-threadedness”, the degree of concurrency within the driver. The more code there is between calls to `mutex_enter()` and `mutex_exit()`, the greater the potential for contention, and the less concurrency there is.

For example, a driver with a single global mutex (initialized in the `_init(9e)` routine), a `mutex_enter()` at the start of every routine and a `mutex_exit()` before returning from each routine would be single-threaded, with no concurrency.

The best way to approach multi-threading is incrementally. Begin with a minimal level and gradually increase concurrency in the driver. It’s pretty straightforward to achieve an initial level of multi-threading by implementing a single mutex lock in the per-instance data structure:

```
struct instance_data {
      kmutex_t    my_mutex;
      :
};
```

In the driver’s `cb_ops(9s)` structure, set the `cb_flag` field to:

```
D_NEW | D_MP,
```

In the `xx_attach()` routine, after allocating the per-instance data structure, initialize the mutex:

```
mutex_init(&ptr->my_mutex, "My mutex", MUTEX_DRIVER,
            (void *)ptr->iblock_cookie);
```

Where `ptr->iblock_cookie` was filled in by a previous call to `ddi_add_intr(9f)`.

Then at the beginning of each routine in the driver, get a pointer to your private data structure and acquire the lock. Before returning, release the lock:

```
ptr = ddi_get_soft_state(statep, instance);
mutex_enter(&ptr->my_mutex);
:      /* code for the routine */
mutex_exit(&ptr->my_mutex);
return;
```

Remember that mutexes protect data, not code; they are only needed where you’re accessing data that needs to be protected. Clearly, data that’s not shared between threads (e.g. local automatic variables) doesn’t need to be protected. As you insert the pairs of mutex calls, you’ll see places where they can be closer together than at the beginning and end of the routine.

Once you’ve compiled and tested the driver with this level of concurrence, you can begin to increase its concurrency. The basic approach should be to decrease the time that the mutex lock is held, by reducing the number of instructions between acquiring and releasing the lock. In some cases, the mutex may not be needed at all, for example if the shared data is accessed read-only and is not changed later based on the value read.

Increasing the concurrency of a driver is still a “black art”, and is best approached with caution. You should carefully think through the possible flows of control in the driver, looking for places where contention may occur. A guideline is that mutexes are useful to make compound operations atomic, e.g. testing and then setting a variable, and to prevent the current thread from changing data that a different thread is relying on. For example:

```
mutex_enter(&ptr->my_mutex);
while (ptr->device_busy) {
   cv_wait(&ptr->my_cv, &ptr->my_mutex);
}
ptr->device_busy = 1;
mutex_exit(&ptr->my_mutex);
```

This thread tests to see if the device is in use. If it is, it calls `cv_wait()` to block until it's signalled that the device is no longer busy, at which point it marks the device busy to lock out other threads. The mutex makes the test and set of the `device_busy` flag atomic. If the thread is pre-empted between the test and set, the lock is held and other threads will not access the flag (assuming that they also acquire the mutex). The `cv_wait(9f)` call will drop the mutex before blocking the thread, and will re-acquire it before returning. This allows the signalling thread to clear the `device_busy` flag safely. It's usually safe to drop the mutex after setting the busy flag because any more threads entering this code will block in the `cv_wait()`. This may not be true though if this thread accesses shared data that another thread might depend on.

The "other half" of the above example might be this:

```
mutex_enter(&ptr->my_mutex)
ptr->device_busy = 0;
cv_signal(&ptr->my_cv);
mutex_exit(&ptr->my_mutex);
```

Here, the mutex is required for two reasons. It's acquired before clearing the `device_busy` flag to prevent this thread from corrupting it while the other thread is testing it (this thread will block until the other thread releases the lock). Also, the lock must be held across the call to `cv_signal()`, to force the signal to occur after the other thread's `cv_wait()`. if this thread runs concurrently with the first thread (e.g. it's the interrupt routine and the device interrupts very quickly), it will block in the `mutex_enter()` until the other thread calls `cv_wait()` and releases the lock.

Condition variables must be initialized and destroyed in a similar manner to mutexes:

```
cv_init(&ptr->my_cv, "My CV", CV_DRIVER, NULL);
cv_destroy(&ptr->my_cv);
```

5.5 Porting Notes

5.5.1 Mutexes vs. Interrupt Blocking

In SunOS 4.x, the `spl()` routines were used to synchronize different parts of a driver. These calls are no longer useful in Solaris 2.0. Changing the priority on one processor will not affect others, so an interrupt at that level can still be serviced by another processor. You should not use any of the `spl()` calls. For this reason, it's important to use mutexes in the interrupt routine, since threads on other processors can still run, and the interrupt thread could be pre-empted.

5.5.2 Mutex Recursion

A given thread cannot acquire a mutex that it already holds, nor can it release a mutex it does not own. Attempting to do either one will result in a system panic, with the message "recursive mutex_enter", or "mutex not held by thread".

5.5.3 High level Mutexes

As mentioned in "xx_attach()" on page 10, mutexes are adaptive. `Mutex_enter(9f)` takes the iblock cookie (from `ddi_add_intr()`) as a parameter, and if the device interrupts above the level of the scheduler, it will spin-lock instead of blocking. On a MP machine this is OK, but if a mutex spin-locks on a uniprocessor the machine will hang forever. Because of this, if `mutex_enter()` finds the lock already held on a uniprocessor it will panic the system with the message "Lock Held and only one CPU". Two high-level threads should never contend for the lock: if one holds it, the other cannot be scheduled because the scheduler cannot run while the lock is held. You should therefore be very careful in the high-level code. Do not call any routines that may block, and do not lower the system priority. Basically, you should call only `mutex_enter()`, `mutex_exit()` (on the high-level lock only), and `ddi_trigger_softintr()`.

5.5.4 Deadlocks

If you have more than one mutex, be careful to avoid deadlocking. Wherever both locks must be held, acquire them in the same order. Otherwise, if thread A acquires lock X then lock Y, and thread B acquires lock Y and then lock X, there's a race condition and a window where each thread may block waiting for the other.

Similarly, locks should be released in reverse order. That is, locks should be acquired and released in a FIFO manner.

5.5.5 Lock Granularity

Locks are not free! Since mutexes perform nontrivial work, the calls incur some overhead. There is therefore a trade-off between the locking granularity and performance. If the lock is acquired rarely and held for a long time (as is the case for the single-threaded example above, at the beginning of section 5.4), the driver takes less advantage of the kernel's multi-threading. On the other hand, if the locks are held for a very short time, the driver allows more multi- threading, but also spends more time acquiring and releasing the locks.

There are also other factors that affect this. For example, if the device is inherently single-threaded (e.g. a printer), there's no point in spending a lot of time increasing the degree of concurrency in the driver.

6.0 References

SunOS 5.0 Writing Device Drivers, Sun part # 800-6502-10

SunOS 5.0 STREAMS Programmers Guide, Chapter 13, "Multi-Threaded STREAMS", Sun part # 800-6538-10

There are also some MP/MT white papers available through your local Sun field office.

7.0 About the Authors

Deborah Gronke Bennett is a software engineer in the Operating System Ports group at Sun Microsystems Computer Corporation. She was one of the first users of the Sun DDI/DKI during the development of SunOS 5.0, porting the bpp device driver to internal development versions. She also contributed to the Writing Device Drivers manual and the DDI/DKI manual pages. Deborah has worked at Sun since 1987. Previously, she was employed at Ampex Corporation.

Patrick Stirling is an independent contractor. He wrote several of the sample drivers in the SVR4 Migration Kit available from Sun. He was a consultant in Sun's HQ Consulting Group from December 87 through October 90. Prior to that he was employed by Fortune Systems Corporation. He can be reached via email at "patrick.stirling@corp.sun.com".