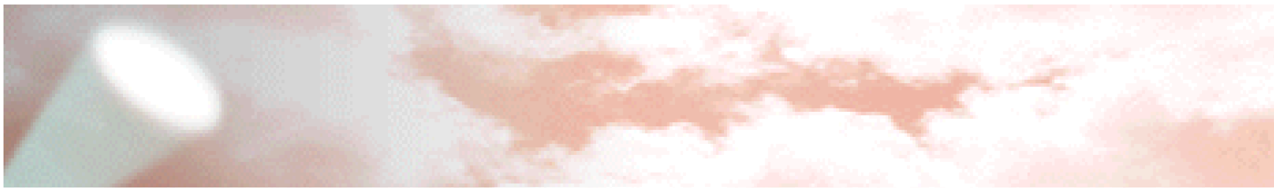

Device Drivers in Solaris 2

by Wolfgang Thaler, OS Development — DDI/DKI

Among the most important features in Solaris® 2 is the new device driver architecture. Device Drivers developed for the UNIX® environment have traditionally posed problems for developers. Drivers would often break when moved to different platforms, required reconfiguration of the kernel, and recompilation between successive operating system releases. For developers, this meant higher maintenance costs, and many unproductive hours spent maintaining drivers.

SunSoft considered these problems during the planning of the Solaris 2 release, and determined that a new approach was needed so developers could use a single “shrinkwrapped” driver on multiple platforms across subsequent releases. The solution for achieving this portability is the new driver interface for Solaris 2 called the Device Driver Interface/ Driver Kernel Interface (DDI/ DKI).

The Solaris Device Driver Interface and Driver Kernel Interfaces

In System V Release 4 (SVR4), the interface between device drivers and the rest of the UNIX kernel has been standardized and documented in the SunOS™ 5.3 Reference Manual. The reference manual documents driver entry points, driver-callable functions and kernel data structures used by device drivers. These interfaces, known collectively as the DDI/ DKI, are divided into three groups: DKI-only interfaces, which are specific to SVR4 and may not be supported in future releases of System V; DDI-only interfaces, which are architecture-specific; and DDI/DKI interfaces, which are architecture-independent and will be supported throughout System V.

The Solaris 2 DDI/DKI, like its SVR4 counterpart, is intended to standardize and document all interfaces between device drivers and the SunOS kernel. In addition, the Solaris DDI/ DKI is designed to allow source compatibility for drivers on any SunOS 5.x-based system, regardless of the system architecture. It is also intended to provide binary compatibility for drivers running on any SPARC® or x86 system that supports the Solaris 2 DDI/ DKI, regardless of the system architecture.

Platform independence is accomplished in the design of the SPARC DDI portions of the Solaris DDI/DKI. The main areas that the Solaris DDI/DKI addresses are interrupt handling, accessing the device space from the kernel or a user process, accessing kernel or user process space from the device Direct Memory Access (DMA) services, and management of device properties, mutual exclusion locks, and services.

The Device Tree, Device-Instances, and Configuration Files

Architectural independence is achieved in the Solaris 2 DDI/DKI through the use of a layered approach, implemented as a tree structure. Each node in the tree structure is described by a device information (`dev_info`) structure where the system stores all information it knows about a device. *Leaf nodes* in this tree generally represent devices that an application program might open and use. Intermediate nodes in the tree are generally associated with buses, such as SBus, EISA or SCSI, to which devices can be attached. These nodes are called *bus nexus nodes* and the drivers associated with them are called *bus nexus drivers*. Bus nexus drivers encapsulate the architectural dependencies associated with a particular bus.

The topmost node in the device 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.

Figure 1 illustrates a device tree, complete with leaf and nexus nodes. The SCSI Device Node appears twice: below a SCSI Host Adapter Nexus Node connected to an EISA Nexus Node, and (on the left side) connected to the SCSI Host Adapter Nexus Node attached to an SBus Nexus Node. By hiding all the information in higher layers, the same device driver can be ported across different architectures.

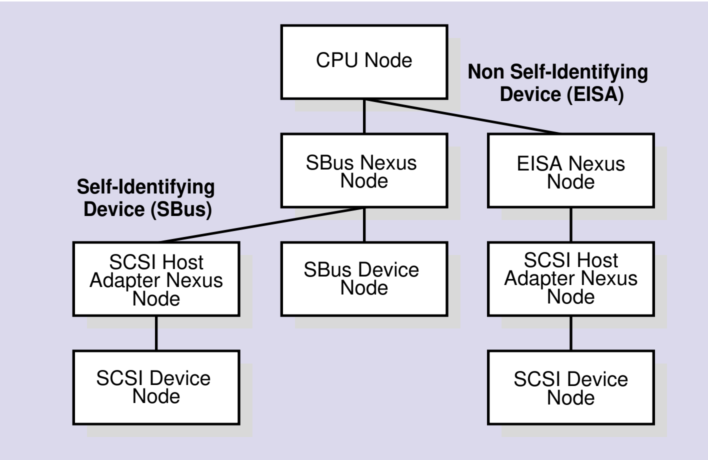

Figure 1. Device Tree

The system uses information from the PROM (for self-identifying devices) and a driver associated configuration file, `driver.conf`, to know which bus nexus driver should be parent in the `dev_info` tree of the driver being loaded. PROM or the configuration file describe properties of the device or devices controlled by that driver. Properties include the physical address of the device, its interrupt specification, and any other information the driver writer would like to communicate to the driver.

Dynamic Loading of Kernel Modules

Solaris supports dynamic loading of kernel modules. The modules are automatically loaded when references are made to them. Many kernel modules are also dynamically loadable. Device drivers must be dynamically loadable, and should be unloadable.

Each device driver has associated with it a device operations (`dev_ops`) structure that defines the operations that the driver can perform. Once a device is loaded, it may then attach one or more *instances* of a device to the system. There are four driver entry points related to attaching and detaching device instances. The `identify()` entry point determines whether that device is supported by the driver, `probe()` tries to access the device to determine whether it exists on the system, `attach()` prepares the driver to control a device instance, and `detach()` deallocates resources associated with the device (including interrupts, locks, device mappings, and data structures for bookkeeping).

A device instance may have one or more minor device numbers associated with it. While it is common to have one minor device number per instance, for some drivers it is convenient to have a set of minor devices for each instance. For example, a tape driver may associate several minor nodes with a particular drive to allow selection of density or rewind behavior. The mapping from minor numbers to instance numbers is defined by the driver. The system can query the driver about this mapping by use of the `getinfo()` entry point. The minor device file system node name are specified by the driver when it attaches the device.

Device special file nodes in Solaris 2 are stored in a directory tree starting with `/devices`. The `/devices` tree is a file system representation of the `dev_info` tree, with device nodes for each minor device specified by various drivers having called `ddi_create_minor_node()`. Entries in the traditional `/dev` directory are symbolic links to the actual device nodes in the `/devices` tree.

Multithreading & Mutual Exclusion Locks

A thread of control, or thread, is a sequence of instructions in a program. The Solaris 2 operating system supports multithreading: a program may have more than one thread of control. Threads can share data and code with other threads, and on a multiprocessor (MP) system, multiple threads can execute simultaneously. There are two kinds of threads: application threads, which run in application programs, and kernel threads, which run in the kernel. Drivers are concerned with kernel threads.

Solaris 2 has a multithreaded kernel. There are kernel threads associated with users' processes, and there are system kernel threads such as those created to handle interrupts. Device driver routines run in kernel threads. On a multiprocessor system, several kernel threads can run simultaneously.

The system may preempt a thread at any time to run another thread. This is a departure from the behavior of earlier versions of UNIX. In those earlier versions, kernel code could be written under the assumption that a process would continue to run until it voluntarily gave up control by calling a `sleep()` or `swtch()`. Only interrupt routines could temporarily gain control from kernel code, and that could be prevented by using calls to the `spl()` routines.

Sometimes it is necessary to perform atomic operations in the kernel such as updating link fields in a queue. Because of thread preemption and multithreading, the kernel code must be written so that a thread that updates some data does so in a way that won't confuse other threads that might be simultaneously trying to use that same data.

Solaris 2 provides mutual exclusion (mutex) lock functions that drivers can use to prevent more than one thread at a time from executing critical sections of code. A driver may allocate a mutex to protect some data structure. Only one thread at a time may hold a particular mutex. When a section of code requires atomic access to a data structure, it acquires the mutex, performs the operations, and then releases the mutex. If another thread tries to acquire that same mutex while the first thread is holding it, the second thread will block (suspend execution) until the first thread releases the mutex. At that time, the second thread will unblock (be made eligible for execution again).

Interrupt Handling

Most devices are capable of interrupting the processor when they complete an operation previously initiated by software. The Solaris DDI/DKI provides a mechanism for a device driver to register an interrupt with the system so that the driver's interrupt routine will be called when the device later generates an interrupt. A driver normally adds interrupts for a device instance in its `attach()` routine and removes them in its `detach()` routine.

The driver calls ddi_add_intr() to register a device's interrupt with the system. In this call, the driver must specify which interrupt is being registered, what routine to call when that interrupt might have occurred, and an argument to pass to that interrupt routine at interrupt time. The argument allows the interrupt routine to distinguish among multiple interrupts that could have been registered to call that same interrupt routine.

The driver specifies which interrupt is to be registered by passing a pointer to the device instance's dev_info node and an interrupt specification number. The bus nexus driver maintains an array of specifications for interrupts the device generates. The contents of that array originate as a property in the PROM or in the driver.conf file. The interrupt number specifies which of possibly multiple interrupt specifications ddi_add_intr() should register. Most devices can only generate one interrupt; if a device can generate more than one distinct interrupt, then it should register each of them in separate calls to ddi_add_intr().

Accessing Device Memory

The Solaris 2 operating environment provides functions to map device memory into kernel space. Solaris provides a "device register" model that hides the details of physical memory addresses beneath a layer of abstraction. The main routine for mapping device memory is ddi_map_regs().

Under Solaris 2, a device driver does not know the specific physical memory addresses of its device. Instead, that information is kept as a special property associated with the device instance. This information originates as the reg property in the PROM or the driver.conf file, where multiple sets of "device registers" may be defined. What this really amounts to is that several ranges of physical addresses may be specified for a device. When the driver needs to access the device, it may use ddi_map_regs() to map a specified one of these address ranges into kernel virtual space.

DMA Services

Solaris 2 provides a high-level, architecture-independent model for Direct Memory Access (DMA). This allows the framework to hide architecture-specific details such as: setting up DMA mappings, building scatter-gather lists and ensuring I/O and CPU cache consistency.

The basic DMA service provided to allocate DMA resources for a memory object is ddi_ma_setup(). It takes the memory object, a generic description of the device's DMA engine, and flags, indicating the direction of the transfers, and returns a DMA handle. This DMA handle is a unique identifier for the allocated DMA resources and is used by many other DMA functions.

A number of operations are defined on a DMA handle that provide the following services: Retrieving a DMA cookie that contains the DMA address for the controller, synchronizing caches with respect to a CPU or device (ddi_dma_sync()), and freeing DMA resources after the transfer is completed (ddi_dma_free()).

Sun Common SCSI Architecture (SCSA)

The SCSA divides the interface to SCSI devices into two major functional parts: target drivers and Host Bus Adapter (HBA) drivers. A "target" refers to the actual device on a SCSI bus, such as a disk or tape. "Host bus adapter" refers to the SCSI controller card that plugs into the system bus and allows a driver to transmit commands to a target device on the SCSI bus.

The SCSA interface allows drivers for SCSI target devices to be written to control those devices independent of the particular HBA to which the SCSI bus is connected. In the dev_info tree, the SCSI target is a child node to the SCSI HBA node. When the SCSI target driver, for example a SCSI tape driver, wants to issue a SCSI command to the SCSI target device, it makes a generic request to the appropriate SCSI HBA nexus driver to transport the SCSI command to the target. A SCSI target driver is not responsible for knowing anything about low-level SCSI-bus protocol; the SCSI HBA driver takes care of that.

As a result, it is possible to move the driver between different systems without changing or dealing with bus-specific details. The goal is to have one target driver portable across different host adapters.

Summary

Many of the motivating factors driving the development of Solaris 2 were associated with device drivers. We wanted to create a shrinkwrapped environment for hardware developers, that could continue to provide the kind of performance that UNIX affords. We also wanted to help developers leverage their work in writing device drivers, so that they could address multiple bus and system architectures with minimal effort. Finally, we wanted to make it very easy to deploy hardware products, by providing a loadable driver architecture that simplified installation.

We believe that careful design of device drivers will enable you to take the best possible advantage of the Solaris 2 operating environment, including its new and powerful multiprocessing and multithreading features.

For More Information

The following references are available on AnswerBook® or through the SunSoft TeleServices Center at 310-348-6090.

Porting to Solaris 2
Solaris 2.x Transition Guide

Driver Configuration File
Writing Device Drivers manual

Writing SCSI Device Drivers
SunOS 5.3 Reference manual — Section 9, DDI/DKI Interfaces

Sample Device Drivers
Writing Device Drivers manual
Device Driver Developer's Kit (DDK)

Code Examples
SunOS 5.3 Writing Device Drivers manual
Solaris Software Development Kit (SDK)

Packaging Device Drivers - Hints
SunOS 5.3 Application Packaging and Installation Guide