BY CHET DOUGLAS AND BOJI TONY KANNANTHANAM • INTEL

Common-driver-library architecture supports, maintains products

USING A CDL-BASED ARCHITECTURE FOR DEVICE DRIVERS CAN SIGNIFICANTLY REDUCE THE TIME TO DEVELOP, MAINTAIN, VALIDATE, AND SUPPORT DEVICE DRIVERS FOR MULTIPLE PLATFORMS.

With the plethora of operating systems available, a common problem for an independent hardware vendor is how to develop, maintain, and support device drivers for products. Although using a CDL (common driver library) can apply to any OS, the storage-device-driver example in this article focuses on Linux and Microsoft Windows. To make a CDL design work, a device-driver designer must understand and compartmentalize the typical driver functions into OS-specific and OS-agnostic functions. In this schema, the designer partitions a device driver into two logical sections based on its dependence on the operating system. Most of the driver code resides in the CDL, which essentially contains the hardware-specific function of the firmware interface, and it is OS-agnostic. The other section of the device-driver code acts as the glue that ties the common portion to an OS. The OS-dependent portion further divides into the CDHI (common-driver-host-interface) and CDHS (common-driver-host-services) layers (**Figure 1**).

The CDHI layer bolts the driver into the kernel. A typical device driver uses a kernel-specific API (application-programming interface) that all driver writers must use to register the driver with one or more kernel subsystems, such as the PCI, I/O, storage, or network. The API also provides the mechanism for the driver to learn of events such as an interrupt, I/O request, or I/O abort. The CDHI layer encapsulates this function and translates OS-specific routines to CDL routines; it comprises a well-defined API and mechanism to interface with the OS kernel.

The CDHS layer uses the services that the OS provides. A device driver also uses services the OS provides to manage the hardware it controls. These services include accessing system registers; communicating over I/O systems, such as PCI and PCI Express; and how to request, manipulate, and dispose of DMA memory, kernel virtual memory, and synchronization mechanisms, such as locks, "spin locks," and semaphores. (A spin lock is a busy-wait method of ensuring mutual exclusion for a resource. Tasks waiting on a spin lock sit in a busy loop until the spin lock becomes available.) A driver may also use special OS features such as "sysfs" support under Linux and event tracing under Windows. These functions, which the kernel architecture and API to a large extent dictate, tie closely to the capabilities of the OS. These OS-specific functions, on which the driver depends for its efficient operation, group together in the CDHS portion. The CDL uses the services that the CDHS interface provides and ensures that the CDL code remains OS-independent.

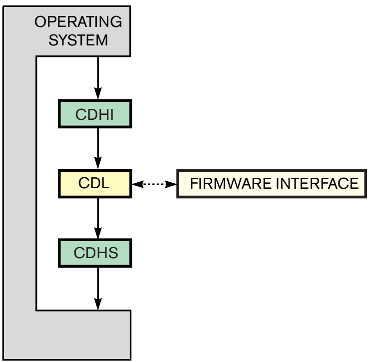

Figure 1 The OS-dependent portion of a CDL divides into the CDHI (common-driver-host-interface) and CDHS (common-driver-host-services) layers.

COMMON LIBRARY

The CDL resides between the CDHI and the CDHS layers. The CDL, the device-specific portion of the driver, is the core component that is aware of the hardware it controls. The intelligence of the device driver resides in the CDL. Typically, a driver "talks" to the device it controls; this communication can range from simple manipulation of certain registers on the device's hardware interface to sophisticated messaging protocols. Message-based protocols usually involve the translation of a standard OS-based protocol, such as SCSI or ATA, to an independent-hardware-vendor-supported messaging protocol that the device's firmware or hardware understands. An example for a storage driver would be translating a SCSI I/O request from the SCSI subsystem of the kernel to an independent-hardware-vendor-specific messaging mechanism.

A device driver may also need to support custom I/O control or similar management mechanisms that the end user employs to manage the hardware. For example, to update the firmware on a device, the device driver would need to provide a mechanism to pass custom commands to the device that are—and must be—OS-agnostic. Modern device

drivers do robust error checking and use sophisticated methods to detect and recover from errors; this feature is especially true of storage and RAID (redundant-array-of-independent-disk)-related products. In addition, to enhance usability, the device driver often translates cryptic error and status codes from the firmware to human-understandable formats.

All of these functions in a device driver are device-specific, so does a driver designer need to rewrite the driver for each new OS that it will support? CDL is a "write-once, run-anywhere" mechanism that enables a designer to cleanly, rapidly, and with the least overhead write the driver once and carry it to a new OS, even to an embedded environment.

The CDL layer implements device-specific functions in an OS-agnostic manner. It abstracts the device-specific layer, and it accesses the device via the CDHI and CDHS through a well-defined API. A design team can avoid reimplementing the entire driver or porting it over with significant costs when adopting a different OS by writing the CDHI and CDHS layers. This task is relatively easy for a reasonably seasoned device-driver developer with the aid of the OS-kernel documentation or driver-development kit.

The task of encapsulating the device-driver intelligence into an OS-independent layer is more complex than it sounds. To gain the benefits of using a CDL, the driver-design team should comprise a group of cross-platform driver experts.

When attempting to use a CDL approach to creating device drivers, it is important for the driver designer to understand how each OS that the CDL supports works at a kernel level. This knowledge will influence the CDL architecture and is necessary to effectively partition the CDHI and CDHS interfaces to bar any OS-specific functions from the CDL. For example, the CDHI- and CDHS-component interface for a Microsoft Windows storage-device driver (typically, a miniport) would have to interface with the Microsoft-port-driver interface so that the CDL does not directly interface with the Windows storage stack (**Figure 2**). The same example using a Linux storage subsystem represents no change for the CDL, but the CDHI and CDHS components work with different OS-driver interfaces from those in a Windows environment (**Figure 3**).

One of the core responsibilities of an OS is to provide synchronization and locking capabilities for a kernel-mode driver. This function allows portions of a driver to run at the same time as other portions or to ensure that only one portion of a driver can execute at any time. In Windows, a typical storage miniport driver has minimal control over the synchronization of starting an I/O versus handling I/O completion in the interrupt-service routine. In a SCSI port miniport model, these activities are mutually exclusive, whereas the newer Storport miniport model improves on this feature by allowing separate locking for building an I/O, queuing an I/O to the controller, or completing an I/O using a simplified spin lock and deferred procedure-call interface.

Linux, on the other hand, gives the driver writer complete control over the use of spin locks and semaphores to control driver synchronization. The widely differing approach these operating systems take to synchronization can complicate the design of a CDL if the CDL code itself enforces locking and synchronization. To simplify the design, a designer may choose not to implement locking and synchronization in the CDL itself but to rely on a set of constraints that force the CDL to hand over the locking and synchronization to the CDHI component.

A good rule of thumb is to make all of the CDL interfaces non-reentrant, to make them run to completion, and to force users of the CDL to enforce this constraint. Another possibility is to include certain classes of CDL interfaces that adhere to different constraints, such as using the routines add_request_to_queue() and remove_request_from_queue(), which might be mutually exclusive of each other but can execute at the same time as other CDL API.

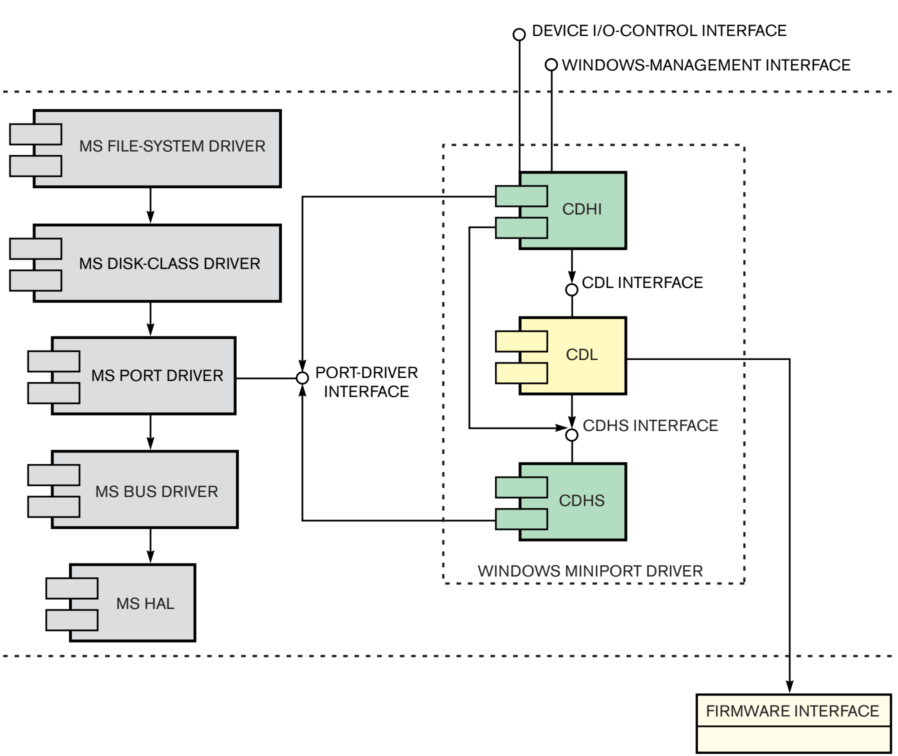

Figure 2 The CDHI- and CDHS-component interface for a miniport in a Windows storage subsystem would have to work with the Microsoft-port-driver interface so that the CDL does not directly interface with the Windows storage stack.

Another area in which operating systems vary considerably is memory management. Whereas most operating systems provide kernel facilities to allocate and free types of memory, such as kernel virtual, cache, and DMA memory, operating systems vary considerably in how and when they allocate the memory and how much memory is available.

In a typical Windows miniport environment, the OS must allocate most memory at driver-initialization time and not at runtime. The port driver allocates the DMA memory for building the I/O request and scatter-gather lists for DMA operations. The driver does this operation on behalf of the miniport and attaches the memory to each I/O it sends to the miniport for processing. Linux, on the other hand, allows the driver writer to allocate and free memory at any time during the driver execution.

To allow the CDL code to use these memory models, the code may leave memory allocation for I/O to the CDHI component, so that the CDL need not understand what OS is running and try to manage memory internal to the common code. This approach allows Windows and Linux to get the I/O memory in their unique way but to use of CDL to link the memory and the request. This approach also affects the CDHS component because the CDL code may need to allocate memory resources during runtime; in a Windows environment, this requirement may force the driver writer to allocate a single contiguous block of memory at initialization time and implement a page- or a slab-allocation/deallocation scheme to break that single block into more usable block sizes for the CDL.

Understanding the hardware the device needs to support is an obvious step for any device-driver designer. In some cases, the hardware may support a PCI-memory interface that a host-OS device driver typically uses; in an embedded-system application, the hardware may expose a local-bus memory interface. Because one goal of a CDL is to hide the underlying hardware details, if the driver must support multiple hardware platforms, the CDL designer may want to use separate components to manage this situation with a common interface that the rest of the CDL code can use. This approach allows the use of different hardware-specific modules without having to change other CDL components, and it allows the driver writer to create different flavors of the drivers with compilation-time build switches.

Understanding the protocols the device driver needs to support is another obvious step for a driver designer. If the hardware requires the use of one protocol for the SCSI controller and a different hardware protocol for the SATA controller, it might be advantageous to implement them as separate modules that have a common interface for use by other CDL components. Again, using compilation-time build switches allows a designer to change protocol-specific portions of the CDL without rewriting other portions of the CDL.

Once a designer understands these and a host of other, less significant constraints, the architecture of the CDL can become a modular design that potentially allows the designer to selectively compile multiple hardware-bus interfaces and protocols into or from the library, allowing for flexibility and extensibility.

Figure 3 The CDHI and CDHS components for a Linux environment work with different OS-driver interfaces from those in a Windows environment.

ARCHITECTURE DETAILS

Figure 4 illustrates a possible CDL architecture that embodies some of these features. The resulting CDL code divides into a number of managers that compartmentalize groupings of functions. Starting at the bottom of the diagram, a local-memory bus and PCI-bus module allow the CDL to support an embedded version that a RAID-on-chip or storage appliance might use or to support a traditional host-OS device driver through an exposed PCI memory-mapped interface. To hide the potential difference between hardware interfaces, a bus manager resides atop the local- and PCI-bus components. This approach provides a consistent interface for the protocol components and any other higher level components that will use the hardware interface.

The transport-interface-manager module would use the bus-manager interfaces to implement the transport-firmware interface that a ven-

dor's product uses. This module does the "heavy lifting" in the CDL and provides an OS-independent transport interface that high-level components can use. In this example, a protocol manager sits on the transport-interface manager; this protocol manager handles SAS-, SATA-, and Fibre Channel-protocol-specific tasks, allowing them to use the transport-interface manager. The functions might include target-device discovery and rediscovery, protocol-specific asynchronous messages, and error handling.

Other high-level components, such as the configuration-manager component, sit on top of the protocol manager. The configuration manager serves as a data-storage area for configuration parameters that all CDL components use. Also above the protocol component, a target manager provides a common set of interfaces for discovering target devices, consistently mapping the target devices, and to potentially handle target-level resets and I/O aborts. An I/O-control manger provides a consistent set of high-level interfaces for management applications. A CDL component ties together all of the individual components and provides interfaces for other functions.

The proposed architecture suggests that the driver must traverse many layers of interfaces in the important I/O-performance path, but the designer can minimize the CDL-code overhead by providing an interface directly into the transport-interface manager to bypass most of the CDL components. Likewise, a designer can minimize the I/O-completion path by using a CDHS callback directly from the transport-interface manager to bypass other components and to improve performance. A designer can also minimize the layers below the transport-interface-manager module. In most cases, a designer can use a compilation-time macro substitution for the bus-manager layer whether using the PCI-manager or local-bus-manager components. The CDHS interfaces could also use a macro substitution that would place the calls inline and avoid the extra overhead of C-function calls for these interfaces.

MORE AT EDN.COM

We encourage your comments! Go to www.edn.com/ms4142 and click on Feedback Loop to post a comment on this article.

To facilitate the use of the CDL on many operating systems, the CDL, CDHI, and CDHS components should use a common types.h header file that defines all of the data types in an OS-independent manner. If the designer has correctly built the CDL code, this header file should be the only CDL file that will need OS-specific additions for each OS.

PROS AND CONS

Independent hardware vendors face the challenge of supporting many operating systems; this requirement was the primary motivation for developing a CDL. The need to develop, maintain, validate, and support device drivers for the various host- and embedded-system environments makes these tasks increasingly complex. With a CDL approach, the vendor can experience significant savings, leading to a shorter time to market.

The CDL is portable and can move to different environments with minimal changes. Intel has successfully ported a CDL not only to different hosts and embedded environments, but also to user applications. With the CDL architecture, it is relatively easy to support a new operating environment. This process involves developing the CDHI and CDHS portions, which the appropriate operating-environment specialist can easily do. If the driver designer can build enough flexibility into the CDL design, it is relatively painless to reconfigure the code base into a CDL "lite" version. An embedded-system with size limitations and high performance requirements could then use this version.

A related benefit is CDL extensibility, which depends on the internal design of the CDL itself. Adding a new function translates into adding a new functional manager that accomplishes the task and interfacing it with the other managers. The complexity of this task depends on how pervasive the new feature is. For example, adding a new I/O bus would involve adding a subcomponent to the bus manager in the CDL. This task is easier than adding a new configuration parameter that would impact all the managers.

Implementing a test strategy that employs the modularity of the CDL architecture can significantly reduce the testing cycles for device drivers for multiple operating environments. Designers can simultaneously test the CDL across multiple environments and distribute test coverage among them. Bugs in the CDL should appear in any environment using the CDL-based driver. Regression testing also becomes easier if the designer updates only the OS-dependent portion of the driver code base.

In addition to rapid development of drivers, the CDL architecture enhances the maintainability of the driver code because most of it is now common. It is easy to maintain this code base rather than manage diverse driver-code bases. With the shorter testing cycle, the

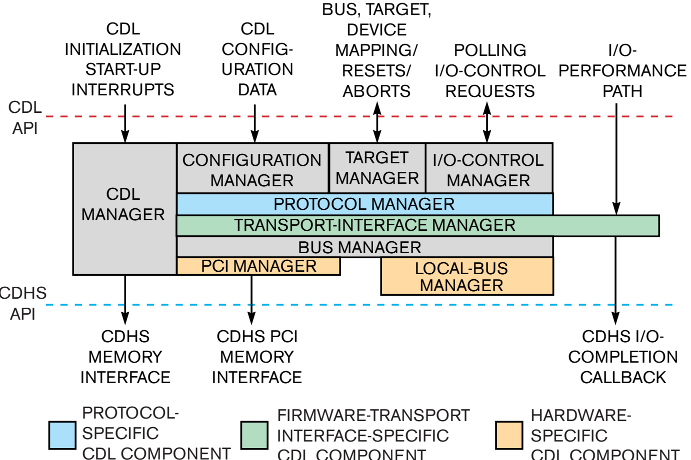

Figure 4 A possible CDL architecture embodies CDL code that divides into a number of managers that compartmentalize groupings of functions.

CDL architecture also provides the advantage of having to fix bugs only once in the common portion even if the bug existed across multiple operating systems. A company can save a significant amount of time in testing and validation.

A typical implementation vertically partitions device-driver development teams, according to the operating environment for which they develop. This increase in developer and tester resources, in the form of OS experts and domain specialists, grows exponentially with the addition of each operating environment. Little interaction or cooperation occurs among the teams, leading to inefficient use of resources, because each team duplicates the effort necessary for going through a product's life cycle. The requirements, design, and development for each driver are usually distinct and varied. This disparity becomes more visible during the maintenance phase of the driver. Teams duplicate efforts to find, characterize, and resolve bugs within each vertical segment. With CDL, an organization can have just one device-driver team comprising at least one hardware/firmware-domain specialist and at least one operating-environment expert, providing for a cross-functional driver-development team.

Despite the benefits of the CDL architecture, it is not without pitfalls. An obvious one is the concern regarding the performance penalty designers incur due to the level of abstraction CDL introduces into the driver. The driver-development team can minimize this drawback by taking extreme care when designing and implementing the CDL components. Another potential issue with a CDL is the kind of licensing the developer chooses for the CDL component. If you have one or more operating environments with conflicting license requirements, a problem may arise. Most organizations face misconceptions and fears regarding mixing open- and closed-source code. One possible approach is to license the CDL under both a commercial and a compatible open-source licensing scheme. But each organization needs to determine its own needs and work out the details with its legal department.EDN

AUTHORS' BIOGRAPHIES

Chet Douglas is a principal software engineer at Intel Corp (Chandler, AZ), where he is a lead storage-software-driver architect and designer. He has a bachelor's degree in electrical engineering from Clarkson University (Potsdam, NY). His leisure pursuits include windsurfing, sailing, travel, and music.

Boji Tony Kannanthanam is a senior software engineer at Intel (Chandler, AZ), where he designs and develops device drivers for the Storage Components Division. He has been involved with Linux, FreeBSD, and other open-source development for the last six years. He has a bachelor's degree in technology from the College of Engineering (Trivandrum, India) and a master's in computer science from Arizona State University (Tempe, AZ). He is working toward a master's degree in business at the Garvin School of International Management (Glendale, AZ).