Uniform Driver Interface

Introduction to UDI
Version 1.0

Technical White Paper

http://www.project-UDI.org

Introduction to UDI

Abstract

The Uniform Driver Interface (UDI) allows device drivers to be portable across both hardware platforms and operating systems without any changes to the driver source. With the participation of multiple operating system (OS), platform and device hardware vendors, UDI is the first interface that is likely to achieve such portability on a wide scale. UDI provides an encapsulating environment for drivers with well-defined interfaces which isolate drivers from OS policies and from platform and I/O bus dependencies. This allows driver development to be totally independent of OS development. In addition, the UDI architecture insulates drivers from platform specifics such as byte-ordering, DMA implications, multi-processing, interrupt implementations and I/O bus topologies. The formal UDI specifications are currently available from the Project UDI web page (`http://www.project-UDI.org`).

Copyright Notice

Table of Contents

Introduction to UDI 1

1.1 Introduction

Operating systems access I/O devices through the use of device-specific software modules called device drivers, or simply “drivers”. By limiting interaction between the device driver and the rest of the system to a well-defined interface, development of device drivers can be undertaken independently from core operating system development. This is particularly valuable when device drivers are written by independent companies such as Independent Hardware Vendors (IHVs).

Historically, drivers have needed to be modified for each hardware platform and operating system. This is becoming a considerable maintenance burden, requiring significant development resources, since the number of devices and the number of operating systems and platforms is growing dramatically.

The Uniform Driver Interface (UDI) allows drivers to be ported across hardware platforms and across operating systems without any changes to the driver source code. By providing a single interface which is identical across all supported platforms and operating systems, UDI allows a single driver to run unmodified on a wide range of systems. This will result in a significant reduction of the effort required to develop and maintain device drivers.

UDI provides a common framework for drivers of many different device types (e.g., SCSI HBAs, CD-ROMs, network interface cards, serial ports, etc.). This makes it easy to write new types of drivers, since all drivers share a common look and feel. It also allows great flexibility in hardware and software configurations, since any driver can potentially communicate with any other driver.

1.2 The UDI Advantage

Creating a device driver interface that can be efficiently supported on multiple operating systems and a wide variety of hardware platforms involves significant challenges. UDI is the first multi-vendor interface designed for meeting these challenges.

There are many differences among current operating systems that influence the environment for device drivers and other kernel modules. Some support kernel threads; others do not. Some support preemption; others do not. Some support dynamically loadable kernel modules; others do not. Variations in memory management and synchronization models also impinge upon the device driver environment.

Operating system differences will likely increase in the future, as vendors move to support distributed systems, fault tolerance/isolation, and other advanced features, using technologies such as microkernels, I/O processors, and user-mode drivers.

UDI is operating system neutral. It abstracts OS services and execution environments through a set of interfaces that are designed to hide differences like those listed above. All OS-specific policy and mechanisms are kept out of the device driver. This allows UDI to be supported on a wide range of systems such as traditional OS kernels, client/server LAN OSs, microkernel-based OSs, and distributed or networked OSs.

Variations in hardware platforms add additional challenges such as:

- Devices may be connected via different I/O buses, some proprietary, on different systems.
- Different systems have different types of caches and buffers in I/O data paths.
- Bus bridges in the path to an I/O device may introduce additional alignment constraints.
- The “endianess” (byte ordering) of an I/O card may be different from the endianess of the CPU on which the driver is running.
- Some systems access card registers via special I/O instructions; others use memory-mapped I/O.
- Interrupt notification and masking mechanisms differ greatly from system to system.

UDI is platform neutral. It abstracts all Programmed I/O (PIO), Direct Memory Access (DMA), and interrupt handling through a set of interfaces that hide the variations listed above.

UDI drivers are written in ISO standard C and do not use any compiler-specific extensions. Thus, a single driver source works regardless of compiler, operating system, or hardware platform.

UDI helps IHVs:

- Reduced number of driver variants means lower development and maintenance costs.
- Implicit synchronization and other techniques reduce driver complexity.
- High-performance design features such as resource recycling and parallelism are easy to achieve with UDI.

UDI also helps operating system vendors:

- OS vendors can utilize drivers not directly targeted for their OS.
- OS vendors can more easily take advantage of IHV-provided solutions.
- UDI allows a high degree of flexibility in OS implementation.
- UDI allows high-performance implementations (such as copy-avoidance and resource recycling) while retaining support for a large number of devices via standardized drivers.

UDI provides location independence for drivers. This allows drivers to be written without consideration for where the code must operate (e.g., kernel, application, intra-OS, interrupt stack, I/O front end). Code regions may even be divided among multiple nodes in a cluster, if desired.

UDI imposes restrictions on shared memory which, by design, prevent the driver from affecting other portions of the system. This allows the system to isolate and effectively “firewall” the driver code from the remainder of the OS, improving reliability and debuggability.

UDI scales well across all target platforms, from the low-end such as embedded systems and personal computers to high-end servers and multi-user MP platforms.

UDI provides strict versioning that allows evolution of the interfaces while preserving binary compatibility of existing drivers.

UDI facilitates rapid deployment of new I/O technologies across a broad range of systems and architectures.

1.3 UDI Functional Requirements

The UDI design is based on a set of functional requirements established by a multi-vendor working group, comprised of several systems and IHV vendors. These requirements are documented in Appendix *A* and define the design objectives for the UDI environment as described in this documentation.

1.4 UDI Driver Types

In UDI terminology, a *hardware device driver* is any software module that controls a piece of hardware in the I/O path (including bus-bridges, host bus adapters, and attached devices). Device drivers also include *software-only driver*s that do not control any hardware. One type of software-only driver is a *pseudo-device driver*, such as a RAM disk, which presents the appearance of a hardware device driver, but operates entirely in software.

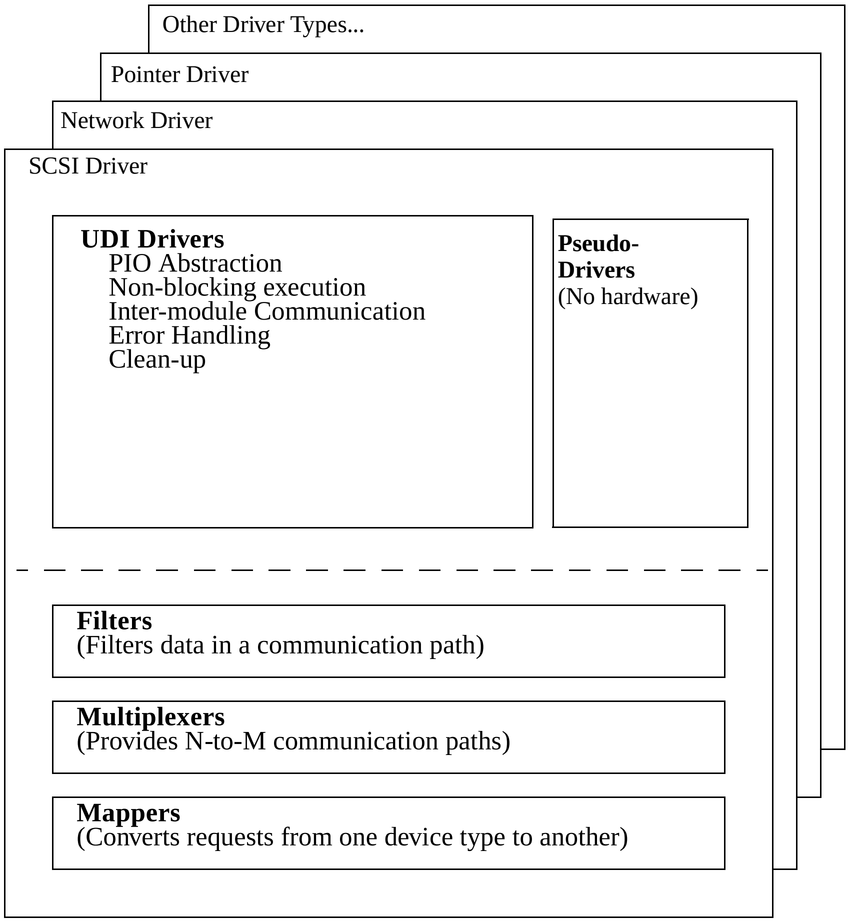


Figure 1-1 UDI Driver Classification

1.4.1 Device Types

The interface to a UDI driver is specific to the type of device controlled by the driver (or, in the case of software-only drivers, the device model presented by the driver). This allows the interface to be strongly typed and tuned to the needs of a particular device model. All UDI drivers, therefore, can be viewed as being associated with a type of device (see Figure 1-1). Some of the device types expected to be supported by UDI include:

- Storage (disk, tape, CD-ROM, etc.)
- SCSI

- Network Packet Transceiver
- USB
- Fibre Channel
- Parallel
- Serial
- Pointing Device
- Keyboard
- Bus Bridge (including interrupt controllers)

1.4.2 UDI Drivers

As seen in Figure 1-1, UDI supports a number of different types of drivers operating simultaneously. Some drivers will operate completely independently of other drivers and some drivers are “stacked” with other drivers to provide more versatile functionality. A UDI Environment can comprise an entire I/O subsystem rather than simply a shell or wrapper placed around an individual driver implementation. Within such an environment, there are several different classifications of these drivers:

Drivers:

Drivers are a general term typically used to refer to a software module that is installed into the system to extend the capabilities of that system, typically in the area of I/O activities. UDI drivers may take several different forms and may or may not control actual hardware, but each is essentially a service provider module that can be supplied by IHV’s or other third parties and loaded into the UDI environment.

Libraries:

Some IHV or third-party distributions do not supply a complete service module but instead supply a component that might be used by multiple service modules of the same or different types. These are supplied within UDI as Libraries, which may be combined with Drivers to provide the complete functionality. One example would be a library that provides PPP protocol functionality, which could be combined with a Driver for a Serial Adapter to provide Network Functionality.

Filters:

Filters are software-only drivers that can be “stacked” on top of other drivers, providing a filter on data passing to/from that driver. An example of a filter would be a disk compression driver.

Multiplexers:

Multiplexers, like filters, are software-only drivers. Unlike filters, however, they may be connected between more than two other drivers. Multiplexers take incoming data on one of their connections and route it to another one of their connections. Multiplexers are typically used in implementing network protocols.

Mappers:

Mappers are software-only drivers that map one device type to another. For example, a SCSI-to-FC mapper would map SCSI requests to FC (Fibre Channel) requests, allowing a low-level Fibre Channel driver to be used to access SCSI devices on the fibre.

1.5 UDI Documents

The UDI documents have been divided into a sequence of books by functional area. There are two main groups: *informative* books, such as this White Paper, which help you to understand the UDI technology but do not provide definitive interface specifications; and *normative* books, which are the set of UDI Specifications, that provide the official definitions of the UDI interfaces.

This book, which is the initial book in the sequence, describes the motivations behind the creation of a Uniform Driver Interface (UDI), introduces the basic concepts and terminology used in UDI, and gives an overview of the UDI architecture. It enumerates the benefits of UDI for both operating system providers and I/O hardware vendors. This book stands on its own, but may also be used as an introduction to the UDI Specifications.

The remaining books are described in the *Document Organization* chapter of the UDI Core Specification.

1.6 Packaging and Distribution

The UDI specification describes the source-level interfaces necessary to write a driver. In order to get the driver onto a target system, the driver source or compiled binary, combined with static configuration information, must be grouped into a single bundle, or *package*, for distribution. UDI defines a standard packaging format for UDI drivers and libraries, so they can be distributed to any UDI-compliant system and installed using a standard UDI command called “`udisetup`”.

The UDI package format supports:

- Multiple media types to allow for systems that do not support particular media types.
- Well-accepted filesystem formats on all media types.
- Multiple UDI drivers on a particular piece of media.
- Other files, in arbitrary directory structure, on the same media as UDI drivers.
- Encapsulation in a file, for network distribution.

Note – UDI packages do not contain any OS-specific files, such as UNIX shell scripts or DOS .BAT files.

UDI Concepts 2

2.1 Overview

The UDI environment is a relatively autonomous, low level, I/O subsystem. It surrounds conforming device driver modules, providing them with a consistent interface to and from the host operating system and among cooperating driver modules, as shown in Figure 2-1.

.

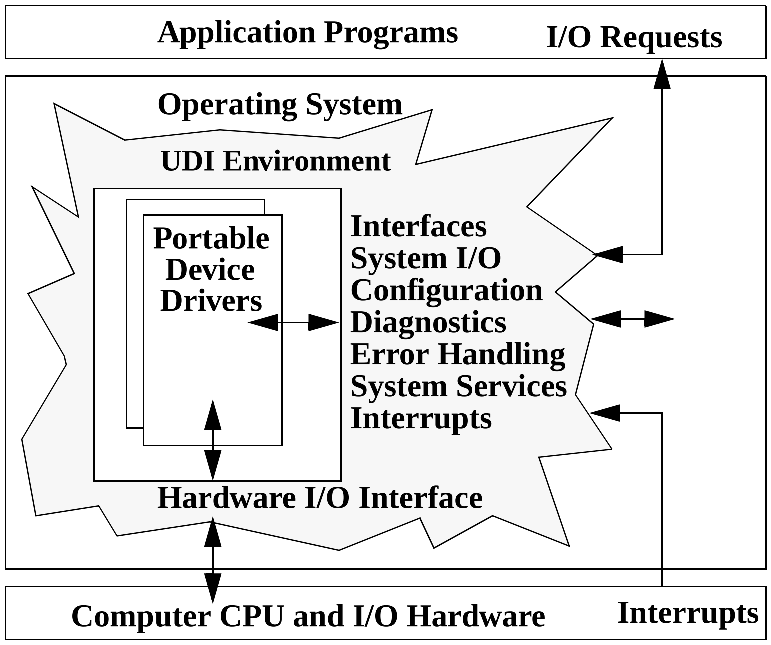


Figure 2-1 UDI Defines All Driver Interfaces

The environment includes interfaces for configuration, diagnostics, error handling, interrupts, system services and hardware access. UDI thus creates a completely specified and encapsulated environment in which UDI-compliant drivers live. Therefore, UDI drivers are not influenced by OS-specific factors; all those details are hidden within each UDI implementation on each individual OS. This is why UDI-compliant drivers are transparently portable: they are truly OS-neutral.

UDI device drivers exchange information among one another and with the environment through a non-blocking, strongly-typed, remoteable procedural interface. These interfaces are designed to span thread and/or domain boundaries and to provide driver synchronization.

The following discussion of UDI fundamentals is key to understanding why UDI-compliant drivers are completely source-code portable between operating systems.

Figure 2-2 shows an abstract view of how an application would access a hardware device through a UDI driver. Applications access I/O services through OS requests (such as system calls); these requests are mapped to standard UDI operation calls and passed to the UDI driver by an external mapper, or directly as part of the OS I/O subsystem. An external mapper acts as a UDI driver on one side and an OS-specific driver on the other side. The UDI driver then accesses the actual hardware device through UDI service calls. Interrupts, as well as I/O requests, are also delivered to the driver via UDI operations.

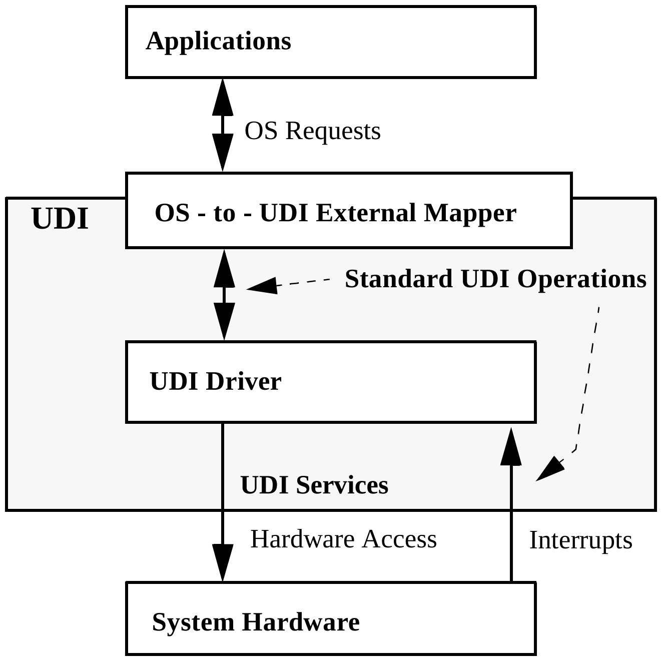


Figure 2-2 User view of UDI driver isolation

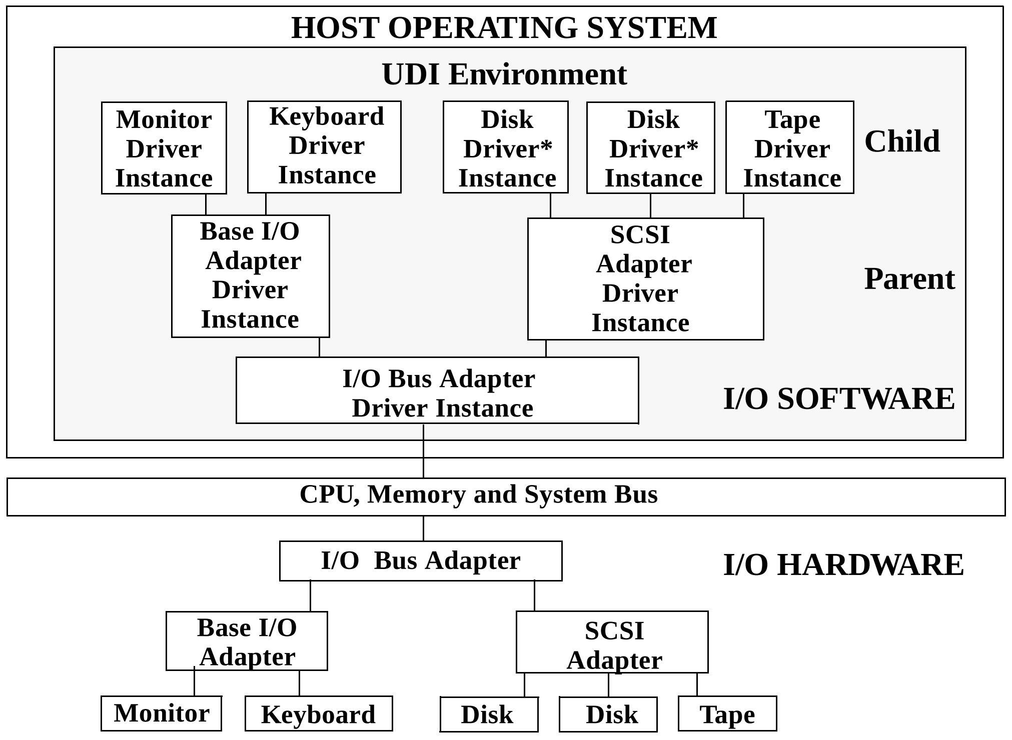
2.2 The UDI Model

2.2.1 UDI Driver Organization

UDI drivers can be connected together in any way appropriate to their semantics. This allows the UDI driver organization to mirror the hierarchical hardware organization. Each I/O hardware component has a corresponding software device driver instance.

When we describe the relationship between two communicating driver instances, we refer to one driver as the child and one as the parent. A child driver accesses its device through the parent driver. In a hierarchical organization this means the parent driver is the one closest to the root of the device tree.

Figure 2-3 shows a simplified example of a possible system configuration. It illustrates how a collection of UDI device drivers might be organized to reflect the I/O hardware subsystem they support. Here an I/O interconnect adapter is attached to the system’s CPU/Memory bus. It in turn supports a base I/O adapter with a monitor and a keyboard, and a SCSI bus adapter with two disks and a tape.



*** Multiple instances of the same driver may share the same driver code.**

Figure 2-3 Example of Hierarchical Driver Organization

Each UDI driver may be associated with multiple instances of its corresponding device, resulting in multiple instances of the driver. These instances have unique data but may share the same code.

2.2.1.1 Regions

Driver instances may be further divided into *regions*. Drivers can specify different attributes for different regions. These attributes allow the environment to determine appropriate priorities and contexts (e.g. user, kernel, or interrupt) for each region. For example, a driver's interrupt handler might run in kernel mode at interrupt priority while other parts of the driver run in user space.

Every driver instance starts with one region. From this region, drivers may optionally create additional regions. With multiple regions, drivers can obtain increased parallelism, as described in *Region Synchronization*.

2.2.2 Region Synchronization

When multiple operations are invoked for a particular driver instance, something needs to prevent them from interfering with each other. Since the operations may operate on the same set of driver data, this access must be serialized. UDI accomplishes this by serializing all operations that have access to the same data. That is, operation invocations that occur while an operation is already in progress are deferred until the first operation completes.

The unit of serialization in UDI is a region. Each region provides the execution context for operations invoked on a set of channels. All channel operations within a region are serialized with respect to each other. Operations in different regions may be executed concurrently. Drivers can control their degree of parallelism by creating multiple regions per instance.

All driver data is contained within a region, and may only be accessed from channel operations running within that region. Since these operations are serialized, access to all region data is implicitly synchronized. Driver code contains no explicit synchronization primitives. Data may not be directly shared between regions because the access to that data would not be synchronized or controlled in any manner; instead, inter-module communications techniques must be used to explicitly transfer ownership of data objects or values between regions.

The region serialization mechanism handles single-processor concurrency cases, such as interrupts and preemption, as well as multi-processor concurrency.

2.2.3 Asynchronous Services

To prevent the execution context of one operation within a region from excessively blocking the handling of other operations for that region or for the calling thread, the service requests that a region makes to the environment are *asynchronous* (also referred to as “non-blocking”). This means that when a region’s code calls a UDI environment service function, execution control is returned to the caller *without* necessarily having provided the requested service (e.g. memory allocation). This allows the region to continue executing for the current or other operations without needing to wait until the environment service completes. When the environment service is finally completed, the environment schedules a callback to run in the execution context of the region and the callback supplies the requested service. If the resource was available immediately, the service function can invoke the callback function immediately, before returning to the calling driver code.

2.2.4 Inter-Module Communication

To pass data object ownership and request operations between UDI driver modules UDI defines bidirectional communication channels. These *channels* are established by a configuration process according to the required driver organization. Channels may be used to interconnect any pair of modules in any topology appropriate to the semantics of the modules involved. There may be multiple communication channels active for each module, each of which may communicate with a different target module.

For example, a module that can support multiple channels to several child modules may redirect those communications to one or more parent modules. This type of functionality is typical of a multiplexer.

Each channel supports a set of *channel operations* (i.e. function calls) that a region may issue to its local end of the channel. These operations result in scheduling the execution of registered operations handlers in the region at the other end of the channel. Although the environment may implement a channel operation as a very thin veil between the two regions, the specification requires that the two regions remain unaware of the implementation details of the other region. Because of this separation, it is possible for a channel to cross a domain boundary, thereby allowing two regions to operate in a relatively asynchronous manner as needed. To support this, channel operations are non-blocking, in that any information or data that is requested via the channel operation is provided by a separate operation in the other direction rather than by waiting for completion of the initial operation to return the information.

Domain boundaries may represent the division between kernel space and application space, or they may represent multiple machines in a distributed cluster environment. The driver itself and its implementation remains unware of the level of domain distribution (if any).

In many current operating systems, device drivers export well-known entry points to the OS in order to establish communication. In UDI drivers, the only global information provided is a set of configuration structures that register the functions used to handle channel operations (called an *channel ops vectors*) and other structures that provide general configuration information about the driver. All execution of driver code is done in the context of a specific region as a result of a channel operation and therefore there are no “global entry points” or other implementation requirements for UDI drivers.

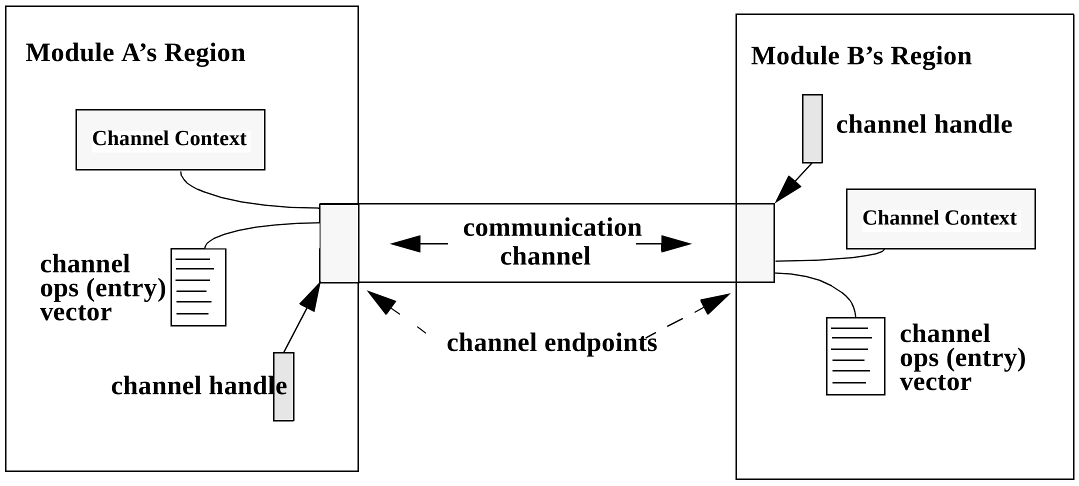


Figure 2-4 Inter-Module Communication

2.2.4.1 Channel Context

The channel context associated with each endpoint allows the driver entry points to find data and state that is specific to the particular channel on which an operation was invoked. Multiple channels may share the same ops vector, but they may each have a different channel context. The environment automatically inserts the appropriate channel context pointer as part of a parameter to each channel operation entry point invoked.

2.2.5 Metalanguages

Each UDI communication channel is associated with a specific type of activity and supports a specific set of strongly typed operations. For example, a SCSI HBA driver would communicate with its child via a channel that supports the standard UDI SCSI operations.

Specific channel operations are defined in the context of a set of operations that work together to achieve some common goal, such as to communicate SCSI information among various SCSI drivers. This set of channel operations, along with its rules of operation and related data structures (e.g., ops vector types) is called a *metalanguage*.

A metalanguage defines the communication paradigm used by two or more cooperating modules, including any required channels and their associated channel operations. Metalanguages form the basis for supporting specific device types. UDI provides metalanguages that are analogous to SCSI CAM, DLPI, ODI, and other similar existing interfaces. A single driver may speak multiple metalanguages as appropriate to the driver's functionality.

The UDI specification defines several standard metalanguages and can be easily extended by Project UDI, IHVs, or other third parties to implement new metalanguages. For example, the Univeral Serial Bus Driver Interface (OpenUSBDI) Working Group has taken this approach to defining how USB devices should be supported in an OS-neutral fashion.

2.2.6 Handles

UDI drivers access hardware and software components through opaque handles provided by the environment. Opaque handles prevent the driver from directly accessing the internals of an object and therefore allow the environment flexibility in the implementation and location of the actual objects. The use of handles allows the driver to perform generic, portable actions which rely on the environment to provide specific hardware or software functionality.

The handle also provides a mechanism for implementation-specific performance optimizations in the environment. Since the handle can be allocated during early initialization processes, it can be pre-loaded with information regarding desired optimizations, removing run-time decisions from the performance path of the driver. UDI uses handles or related constructs for many objects or operations, including: PIO, DMA, and buffer handling.

For example, a driver wishing to access device memory would obtain a PIO (Programmed I/O) handle to that memory. It would then issue PIO requests to the environment using that handle to read from or write to that memory. This allows the environment to handle platform-related issues such as byte-ordering, intermediary caches, requirements for bus bridges, etc.

2.3 Interrupts

In most architectures, mechanisms are provided for I/O devices to send asynchronous notification of events to their corresponding device driver software. Typically these are called interrupts.

Classic interrupts carry essentially one bit of information (with respect to a particular device): the fact that an event occurred. The device driver software is then responsible for reading status information provided by the device to find out more details about the event. Other architectures allow devices to send additional information along with the interrupt. This can range from a small fixed number of bits to a full-blown variable-length message. UDI interrupt handling is designed to work across this range of architectures.

I/O devices commonly generate interrupts for the following reasons:

- An I/O operation completes and the driver needs to be notified.
- An asynchronous event occurs that requires the attention of the processor.
- An error condition is detected that requires corrective action by the processor.

UDI provides for interrupt handling as part of its Bus Bridge Metalanguage. This metalanguage covers registration and de-registration of interrupt handlers, as well as delivery of the events themselves.

Interrupts are initially fielded, of necessity, by the operating system's first-level interrupt handler, which hands appropriate I/O interrupts over to the UDI environment. The environment maps interrupts into procedure calls to the appropriate driver channel operations for the driver closest to the processor-memory interconnect (PMI). The interrupt definition is hierarchical, in that interrupt notification may be forwarded through one or more bridge drivers before being handled by the driver for the ultimate interrupting device.

The interrupt handling is then scheduled in the execution context of a driver's region as a result of the invocation of the channel operation for that driver. This initial region context is responsible for acknowledging the hardware device's interrupt to terminate the interrupt cycle. This interrupt region then typically issues a channel operation to the driver's primary region to further process the event that caused the device to interrupt. Optionally the driver may provide a list of PIO operations that are to be performed when an interrupt is fielded by the first-level interrupt handler, removing the need for the interrupt region.

2.3.1 Interrupt Roles

The UDI interrupt handling model is based on a pair of agents: an interrupt handler, which is prepared to receive notification of an interrupt event and provide the device-specific response; and an interrupt dispatcher, which is responsible for reading hardware state (typically from an interrupt controller) to determine which of a set of events (interrupt requests) occurred and then sending an event notification to the appropriate interrupt handler(s).

This model allows for cascaded interrupt dispatching, in which an interrupt handler is also an interrupt dispatcher. This allows UDI to support hardware with interrupt delivery cascaded through multiple interrupt controllers, bus bridges, and multi-function devices. Each level of interrupt handler understands a particular (interrupt delivery) device and knows the set of sub-devices that can deliver interrupts to that device. It reads out information from the device it controls and uses it to dispatch the handler for the appropriate sub-device, which may in turn read out information from its device and dispatch a sub-device handler.

Typically, the first level interrupt dispatcher will be part of the environment rather than strictly a UDI driver, but it will look just like a UDI interrupt dispatcher driver to the interrupt handler, via the channel operations for its channel, in much the same fashion as with other metalanguage mappers.

2.4 The UDI Non-Blocking Model

One of the key aspects of the UDI design is the capability of supporting many different OS and platform architectures while providing a high-performance I/O subsystem. One of the ways UDI achieves this is with its non-blocking execution model. Under this model, no driver code or service calls made by the driver may sleep or otherwise block. Instead, operations that need to wait (e.g., to obtain resources) are queued as described in Section 2.2.3, "Asynchronous Services"; when the needed condition is satisfied, the driver is re-entered via a callback function to continue the operation.

The non-blocking model allows the environment to efficiently support multiple I/O operations while preventing resource availability from affecting the thread of execution. It also minimizes dependencies on underlying OS mechanisms (and differences therein) that would be required to support blocking.

Applications that directly interface to device drivers using the traditional blocking system-call model can be supported by using an external mapper that provides this blocking effect for the application while interacting with UDI via non-blocking operations.

2.5 Conclusion

UDI provides a portable, flexible, fully functional environment for device driver implementation, through a uniform set of platform- and operating system-neutral interfaces. These interfaces define paths for operating system access to device drivers for configuration, diagnostics, I/O requests and interrupt handling. They define paths for device driver access to system services, related device drivers, and underlying I/O hardware.

The implicit synchronization resulting from UDI's region-based implementation facilitates the design of drivers that are safe in multi-processor environments without sacrificing performance or resource use in non-preemptive, uni-processor systems.

This paper has shown how UDI allows the flexible organization of its simple building blocks (channels and regions) to build both simple and sophisticated device drivers. UDI allows the configuration process to combine driver instances into a hierarchical representation of the underlying hardware.

The UDI architecture allows developers to support a device with a single driver, applicable across the family of systems supporting the UDI environment. This will, in turn, greatly reduce the engineering cost and accelerate the availability of I/O solutions for those systems.

UDI Functional Requirements — *A*

1. OS-Neutral: The interfaces exposed to drivers must allow the drivers to be OS-neutral. That is, there should be no dependence on OS-specific features, behaviors, or execution models.

1.1. The architecture must keep policy decisions out of the driver, since policies may vary from OS to OS and may change over time. For example, drivers should communicate errors to the OS, and let the OS determine the appropriate action to take, rather than having the driver directly abend (panic) the server.

1.2. Driver interfaces must be well defined such that there is no ambiguity in method of implementation.

1.3. The architecture must not preclude coexistence with non-conflicting native drivers on each OS.

2. Simple Drivers: The IHV driver interface must be lightweight making it easy for developers to create and maintain code and to port code to other hardware platforms.

2.1. The driver for a given device should be the same whether the device is a core I/O device, a standalone card, or part of a multi-function card.

2.2. A single driver must be able to function with reasonable performance in both uniprocessor and multiprocessor systems without modification.

2.3. The architecture must hide complexities of multi-processor systems and asynchronous event synchronization from the driver.

2.4. The architecture must allow for driver implementations which focus primarily on the hardware specifics of the device with minimal driver code devoted for infrastructure functions such as configuration.

3. Support for Multiple Device Types:
The architecture must support drivers for multiple device types, such as network adapters, mass storage adapters and attached devices (disk, tape, CD-ROM, etc.), and serial ports.

3.1. As much as possible, services whose semantics are not device specific should be exposed to drivers as generic interfaces available to drivers for all device types. The framework should appear as a single unified framework.

3.2. Drivers must be able to be connected together in any way appropriate to their semantics. This allows for stacked drivers, filters, and multiplexors.

3.3. The architecture must be extensible to dynamically support new devices (HBA and peripherals) and future storage requirements.

3.4. The architecture must allow exploitation of the capabilities of new hardware technology, such as Fibre Channel, SSA, Universal Serial Bus (USB), Plug-and-Play, SCI or MCS based distributed systems, and I/O processors.

4. Platform and I/O Bus Abstraction:

The architecture must provide an abstraction layer for access to hardware and bus interfaces which allows drivers to be portable across a wide range of platforms.

4.1. The environment must be able to compensate for any system architecture differences, including byte ordering, memory management, and memory alignment.

4.2. The environment must be able to simultaneously support multiple I/O buses, potentially of different types, including hierarchical bus organizations.

4.3. The interfaces exposed to drivers must be supportable on platforms on which some sizes of programmed I/O transactions may be non-atomic.

4.3. The programmed I/O interfaces exposed to drivers must be supportable on all platforms, regardless of the characteristics of I/O bus transactions, such as atomicity.

4.4. The interfaces exposed to drivers must be supportable on platforms which require explicit cache flushing or memory ordering for DMA or programmed I/O transactions.

4.5. The interfaces exposed to drivers must be supportable on platforms which have different physical address spaces for I/O cards than for CPU access to main memory. It must support platforms which map these spaces via map registers and those which use static mappings.

4.6. The DMA interfaces exposed to drivers allow for a variety of hardware implementations, including shared DMA channels, physical DMA bus mastering, and virtual DMA.

4.7. The architecture must enable ease of portability between processors of different word size.

4.8. The architecture must allow for support of hot docking and power management.

5. Location-Independence:

Drivers must be location-independent.

5.1. If supported by a particular environment implementation, drivers must be able to be run on host CPUs, I/O Processors, or remote nodes in a network with no change to the driver.

5.2. The architecture must allow for drivers to be designed such that different pieces of a driver can be split between different locations and domains; e.g., a driver implementation in which its interrupt handler is separated from the rest of the driver.

6. Flexible OS Implementation:

The architecture must allow for flexibility of OS implementation.

6.1. The architecture must not preclude (though specific implementations may not support) running drivers in user mode.

6.2. The architecture must not preclude (though specific implementations may not support) having driver code and/or data be pageable, even while the driver is running.

6.3. The environment must be implementable on embedded systems.

6.4. The environment must be implementable with minimal memory usage. In particular, the architecture must not require the use of multiple execution contexts (i.e. threads) with separate stacks.

6.5. The architecture must allow for implementations with varying degrees of trust in drivers, from completely trusted in-kernel drivers to fully-isolated user-mode drivers. The drivers themselves must not be required to change as a function of trust level.

7. Maximize I/O Performance:

Within its framework, the architecture must do all it can to maximize I/O performance.

7.1. The architecture must perform efficiently in a local (host CPU) environment.

7.2. The architecture must scale well in both directions, from low-end (preferably including embedded) systems to large MP and/or clustered servers, as well as large-scale distributed systems.

7.3. The architecture must enable optimal performance through mechanisms such as copy-avoidance and recycling of memory (allocation avoidance).

8. Source and Binary Compatibility:

The interfaces exported to drivers must allow for source and binary compatibility between systems, where appropriate.

8.1. Driver source must not require any change in order to be compiled for any target OS or platform.

8.2. Drivers must not be required to be recompiled to run on different versions of an OS on the same machine.

8.3. Drivers must not be required to be recompiled to run on different OSes which support the same ISA (Instruction Set Architecture), binary file format, and calling conventions.

9. Versioning Control:

The architecture must facilitate driver maintenance and compliance.

9.1. The interfaces exported to drivers must be strongly typed.

9.2. Drivers must indicate the version of the specification to which they conform, so that the environment can enforce conformance to the specific set of interfaces documented in the appropriate specification.

9.3. The environment must be able to simultaneously support drivers which conform to multiple versions of the specification.

10. Resource Allocation:

The architecture must provide a mechanism for dynamic allocation and reallocation of hardware and system resources (interrupts, I/O ports, DMA channels, memory, etc.)

10.1. The environment must be able to prevent resource allocation deadlocks.

11. Diagnostics, Monitoring, Tuning:

The architecture must provide hooks for standardized diagnostics, monitoring, and tuning capabilities of a channel.

12. Inter-Driver Communication:

The architecture must provide a mechanism for asynchronous communication between drivers and between drivers and the environment.

13. Timer Services:

The architecture must provide a mechanism for time-based driver callbacks.

A Brief History of UDI B

UDI began in 1993 as an offshoot of an industry standard system I/O bus design (a system-level bus ala PCI but driven by the needs of workstation and server vendors). The motivation was simple: having an industry standard bus with industry standard I/O cards, chips, etc. increases the volume and lowers the costs for getting the associated hardware into systems, but in order to use those cards and devices you also need corresponding device drivers. So in the process of designing an industry standard bus, the group also considered the software portion of the equation and looked at what it would take to provide industry standard drivers.

The offshoot driver effort was originally called the Common Device Driver Environment (CDDE). The bus effort didn't go forward but several people involved in the CDDE effort continued pursuing the software issues and developed a strawman document of how the device driver APIs and many of the basic driver services would be designed. Three people were primarily involved with this initial strawman development: Mike Wenzel at HP, Richard Arndt at IBM, and Larry Robinson at DEC. By late 1993 the strawman had been developed and the CDDE working group went inactive, lacking resources or commitment to take it further at that time.

In 1994 a group within X/Open began looking at what would be required for portable device driver interfaces, with particular focus on resolving the differences among the various flavors of "DDI/DKI" driver interfaces in SVR4-based UNIX systems. As a result of this investigation it became apparent that the differences among the flavors were greater than the similarities (only about 30% of the interfaces were common), and that resolving some of these differences without redesign or rethinking fundamental assumptions would lead to additional problems. In the process of doing this, David Kahn at Sun recommended that the folks involved in the X/Open group consider the CDDE approach to the problem, which had already considered many of these issues in its basic design.

This led to restarting CDDE in mid-1994 with a larger group of companies and participants including Adaptec, Apple, DEC, HP, IBM, Interphase, Novell, Sun, and Taligent, along with many other less actively involved players. The primary focus during the rest of 1994 was to develop functional requirements and to choose between two proposals as the basis for the CDDE design: the Strawman from the previous CDDE effort and a competing proposal from Taligent that was based on their own object-oriented I/O technology using C++. At the end of 1994 the group chose to go with the Strawman and began specification development based on that in early 1995.

In mid-1995, for various marketing, aesthetic, and perceptual reasons, the name of the driver interface was changed from CDDE to UDI (Uniform Driver Interface), and the working group became known as Project UDI. 1995 also saw rapid development of the UDI Specification: proceeding to version 0.50 in March, 0.60 in July and 0.70 in December. In late 1995, Project UDI hooked up with POSIX, giving a day and a half presentation to the October quarterly POSIX Conference and forming a POSIX Study Group. Participation in POSIX also generated additional interest and participation in UDI from companies such as Lockheed-Martin and from the OSJTF (Open Systems Joint Task Force) of the US Department Of Defense. However, in the spring of 1996 the Study Group voted to disband for the time being and allow the Project UDI members to instead concentrate resources on constructing prototypes of the environment interfaces. While the standardization of UDI within a recognized standards body was still a goal, it was felt important to complete these implementation-related activities first.

There were three main goals for the prototyping effort: (1) proof-of-concept, (2) feedback into the UDI specification to make it more robust and real-world-applicable, and (3) providing the foundation for what could become a reference implementation of the UDI environment, allowing the UDI technology to be more easily spread to additional OSes and platforms. By mid-1996, Project UDI completed version 0.75 of the UDI Specification, which was designed to be the basis for the prototyping effort. While some individual companies began the prototyping work at this point, the effort could not begin in earnest until a coordinated project could be put in place.

The signing of a landmark co-development agreement in December 1996 between eight Project UDI member companies set the stage for full-scale joint development to begin. 1997 was the year of the prototype, culminating in December with a successful multi-platform demonstration, with a follow-up public demonstration at SCO Forum in mid-1998. These demonstrations of the UDI technology showed a single UDI driver source running on both 32-bit and 64-bit processors and OSes, big and little endian systems, single and multi-processor systems, x86, Sparc, Alpha, and PA-RISC processors, and systems with and without software-visible caches, all with no changes to the UDI drivers or special cases to accommodate the various systems and OSes. OSes demonstrated included Digital Unix, NCR MP-RAS, HP-UX, SCO UnixWare, and Solaris. UDI drivers included a SCSI Adapter driver from Adaptec and an Ethernet driver from Interphase.

In 1998 Project UDI focused on getting ready to fast-track through a standards body and updating the Specification based on prototype implementation experience and getting it ready to be released as a reference implementation. A lot of good experience and insights were gained from the prototype, which resulted in significant changes and improvements being done to some of the core architecture. Languishing somewhat in early 1998 due to outside demands on the Project UDI representatives, UDI was revitalized in July and August of 1998 with the goal of completing quality 1.0 specifications as soon as possible. At this point, Intel also joined in support of the effort in recognition of the need to facilitate the support and adoption of hardware and software standards.

Over the next 12 months, a high-level of intensity was maintained by the core UDI members with monthly face-to-face meetings for 3-4 days each and significant advancement of the specification. At this time, key players were Kevin Quick (Interphase), Mark Evenson (Hewlett-Packard), Kurt Gollhardt (SCO), Mark Bradley (Adaptec), James Smart (Compaq), Jim Partridge (IBM), and Linda Wang (Sun). UDI was presented at the February'99 Intel IDF conference and forms the basis of the device support for Intel's UDIG (UNIX Developers Interface Guide) specification efforts.

As 1999 draws to a close, the development of a 1.0 reference implementation—primarily driven by SCO but heavily supported by the other members—is expected to result in several UDI implementations in the upcoming millenial transition, the final 1.0 versions of the initial set of specifications—ready for implementation—were published on September 1st, and negotiation is under way to submit these specifications for de-jure standardization. UDI is on its way.

- September 1999