UDI Architecture In-Depth

http://www.sco.com/forum1999/conference/developfast/f8

Robert Lipe
UDI Development Team Lead
E-mail: robertl@sco.com

This is the third in a series of eight UDI presentations for SCO Forum 1999.

This session expands upon the Introduction to UDI, describing the UDI architecture at the next level of detail. Learn about inter-module communication, implicit synchronization, driver configuration, metalanguages, and other key areas of the UDI infrastructure. This is a prerequisite for the UDI Tutorial (F9,F10).

Agenda

- **UDI Portability and Extensibility**
- UDI Execution Model
- UDI Data Model
- Intro to UDI Specifications
- Q & A

Driver Portability with UDI

- **100% driver source portability**
  - Binary portability for IA32 and IA64 and ...
- **Complete driver encapsulation**
  - All APIs defined
- **OS-neutral and platform-neutral**
  - OS policy removed from driver
  - Transparent endianness conversions

F8: UDI Architecture In-Depth

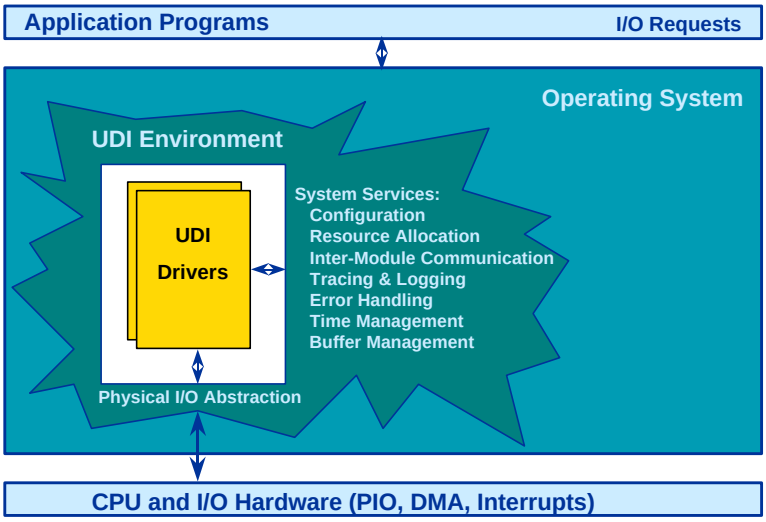
"API" = Architected Programming Interface

Can also refer to a specific function within an API.

"ABI" = Architected Binary Interface

The IA32 and IA64 ABI bindings for UDI are being specified by the UDIG (UNIX Developers Interface Guide), jointly developed by Intel, UNIX OSVs, and IHVs, with help from Project UDI.

UDI Fully Encapsulates Drivers



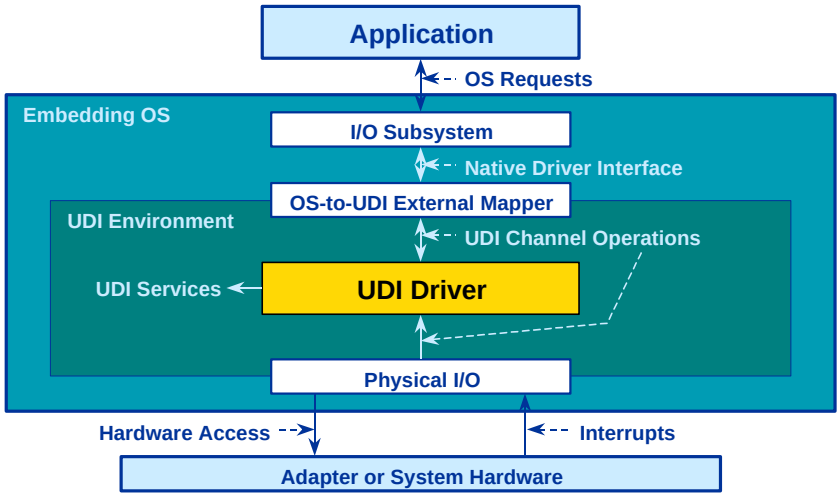
F8: UDI Architecture In-Depth

UDI drivers are completely surrounded by the UDI environment. All calls in and out of the driver go through UDI APIs. All system services are provided through UDI APIs. This is how complete portability is achieved.

The UDI environment may be integrated into the embedding OS to varying degrees--even for different parts of the same environment. Thus the jagged line between the UDI environment and the OS.

Path From Application to Driver

Layered Implementation



This is one way support for UDI drivers could be implemented, by layering on top of native driver interfaces.

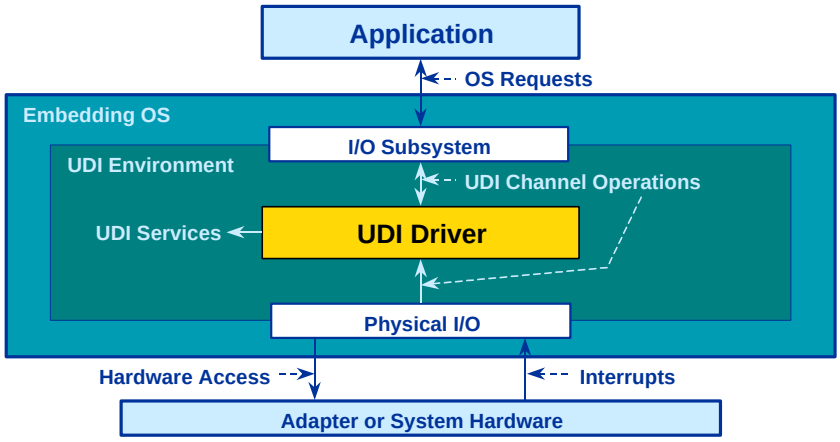
External mappers look like native drivers to the embedding OS and look like UDI drivers to the UDI environment and UDI drivers they talk to.

External mappers convert native I/O requests to UDI channel operations.

UDI channel operations are the way UDI drivers communicate with each other and the environment for processing I/O and control requests. Channel operations are described in more detail in later slides.

Path From Application to Driver

Integrated Implementation



F8: UDI Architecture In-Depth

An alternative environment implementation integrates UDI more tightly into the I/O subsystem of the OS.

This may be side-by-side with legacy native interfaces or the legacy interfaces may be re-implemented to be layered on top of UDI.

Uniformity Across Device Types

- **Basic model common for all drivers**
  - Execution and Data Models
    - » Common buffer model
  - Configuration Model
  - Inter-Module Communication
    - » Between drivers and/or environment modules
  - System Services and Utility Functions

UDI Metalanguages

- **Device-type specific communication**
- **Defines communication paradigm between cooperating modules**
  - Operations and sequences to implement technology-specific functionality
- **Analogous to SCSI CAM, DLPI, etc.**

More on metalanguages after we describe channels...

Agenda

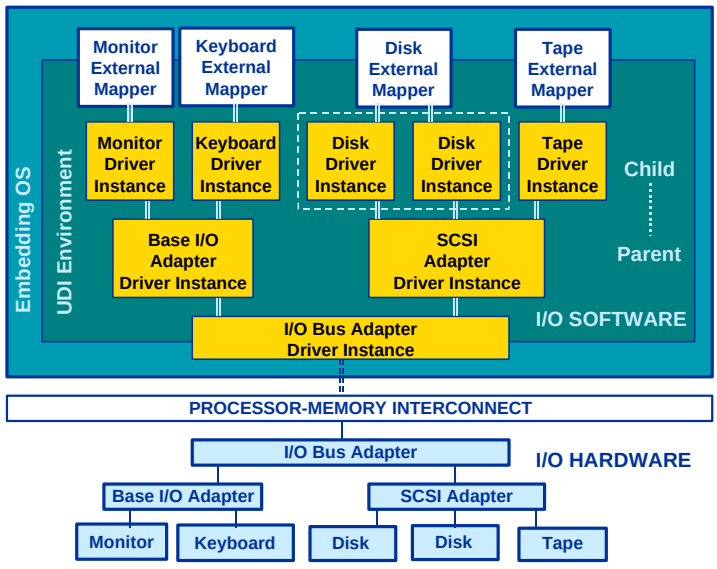
- UDI Portability and Extensibility
- **UDI Execution Model**
- UDI Data Model
- Intro to UDI Specifications
- Q & A

F8: UDI Architecture In-Depth

UDI Execution Model

- **No global entry points**
- **Driver's udi_init_info structure contains entry-point pointers, size requirements...**
- **All driver code executed in the context of a *region***
  - Regions are associated with driver instances
    - » One for each adapter/device controlled

Example Driver Hierarchy



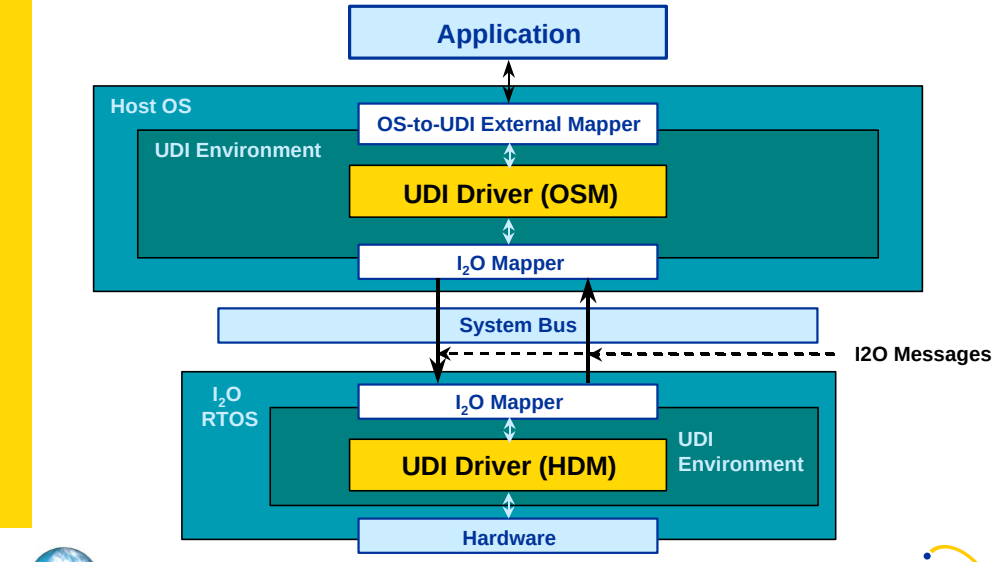
F8: UDI Architecture In-Depth

This picture illustrates that the driver software hierarchy is a mirror image of the hardware hierarchy.

It also shows that two different instances of the same type of device (in this case, disks) may be handled by the same (or different) device driver(s). The dashed box surrounds the two disk driver instances that may be (per the environment's choice) handled by the same driver.

Driver instances of the same driver may (per the environment's choice) share read-only memory for code and/or initialized data.

Example UDI & I_2O Combination



F8: UDI Architecture In-Depth

This picture illustrates an example of layered drivers managed by two independent (but cooperating) UDI environments. In this case, for I2O* (the Intelligent I/O architecture), one part of the host OS running on the host processor(s), and the other part of the Real-Time Operating System running on an I/O Processor (IOP).

In I2O terms, the two drivers are known as the OSM (Operating System-dependent Module) and the HDM (Hardware-Dependent Module).

Unlike with conventional I2O, the UDI OSM would not be operating system-specific, and the HDM would use UDI APIs rather than I2O APIs.

From an I2O standpoint, the two parts are independent. You could “UDI-ify” either one or both. The common factor is the I2O Messaging Protocol between the two operating systems.

UDI Regions

- **Basic unit for execution and scheduling**
  - Each call into the driver region is serialized
- **No direct data sharing between regions**
  - *You have to go through channels*
- **Region attributes (e.g. priority hints) specified at build time**

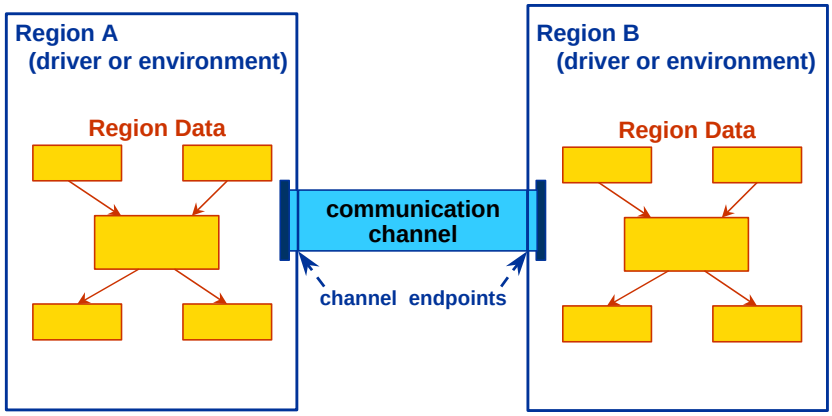
UDI Regions
(continued)

- **One driver instance per device instance**
- **One or more regions per driver instance**
  - Multi-region drivers may have higher parallelism
- **Enables *instance-independence***
  - Driver state separate for each device instance
- **Enables *location-independence***
  - Each region may operate in a different domain
    - » e.g. address space, NUMA or network node

Simple drivers have one region per driver instance. This still allows parallelism with other drivers, other instances of the same driver, and all the rest of the OS and application activity in the system.

More complex drivers that are computationally-intensive may be structured as multiple regions per instance (“multi-region drivers”), to take advantage of increased parallelism if available from the OS and platform.

Regions and Channels



F8: UDI Architecture In-Depth

Each region can be part of a driver or an environment entity such as the Management Agent. The same communication mechanism is used whether between MA and driver, between two layered drivers, or between two regions in one instance of a multi-region driver.

Any memory allocated by or on behalf of a driver is considered part of the region with respect to which it was allocated, and may only be accessed from that region.

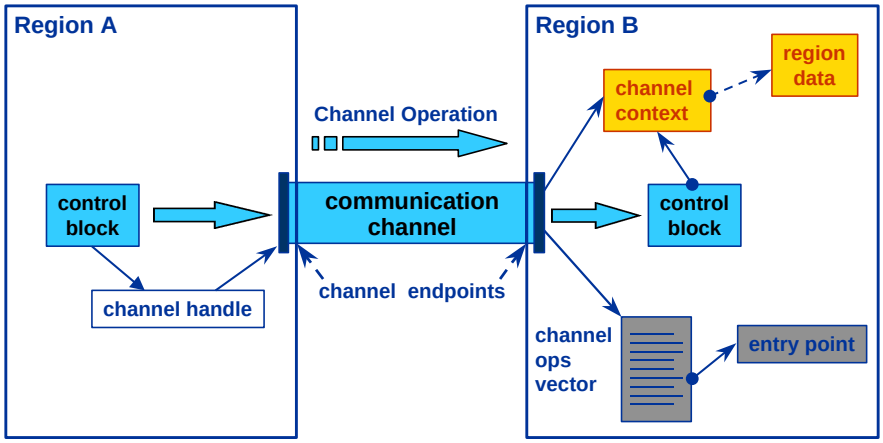
UDI Channels

Basis for Inter-Module Communication (IMC)

- **Bi-directional channels connect regions**
- **Communication via *channel operations***
  - Strongly typed function-call interface
  - Asynchronous one-way operations
    - » Each request has a corresponding response
    - » Context managed via *control blocks*

All channels are bi-directional, but each channel operation flows in one direction, with no results except those returned via a separate response operation.

UDI Channel Communications



F8: UDI Architecture In-Depth

Communications between regions is by channel operations over channels. There may be multiple channels between the two regions.

This picture shows an operation flowing in one direction, but all channels are bi-directional and symmetric.

The control block (in region A) contains an (opaque) channel handle, which identifies the channel over which to send the operation.

When the control block reaches region B, its context pointer will have been copied from the channel context associated with B's end of the channel. Thus, per-channel context data can be accessed from the control block, and region-global data may be accessed from there. Drivers can change their channel contexts at any time.

Also associated with B's end of the channel is a pointer to a channel ops vector, which is a (read-only) list of function pointers initialized by the driver. The ops vector contains one entry for each type of channel operation defined by the metalanguage for this type of channel.

The driver's entry point is determined by indexing into the ops vector. The selected function is called, passing it the channel operation parameters, including a pointer to the control block. From there it can find the context it needs to distinguish this channel from others of the same type.

The CB's channel handle will have been set to B's end of the channel.

Metalanguages and Channels

- **Metalanguages define:**
  - Number and types of channels
  - Channel operation types on each channel
    - » Control block plus meta-specific parameters
  - Control block types for each operation
    - » Struct includes meta-specific fields
    - » Generic control block header common to all

UDI System Services

- **System interface & resource management**
  - Implemented for all UDI environments
  - Abstract OS services
- **Calls from driver to environment services are called *service calls***

UDI Service Calls

Two Styles

- **Synchronous service calls**
  - Complete without blocking
  - Results returned “immediately”
- **Asynchronous service calls**
  - Return without blocking
  - Delayed completion
  - Results returned via callback function

Non-Blocking Execution Model

- **All service calls and channel operations return without blocking**
- **Drivers usually return after making one service call or channel operation call**
- **Gives environment complete control over thread usage and driver scheduling**
- **“Pseudo-threads” interleaved between callbacks**

F8: UDI Architecture In-Depth

The term “pseudo-thread” describes the sequence of calls into the driver starting from a channel op entry point, through zero or more callbacks and then ending with a channel op invocation. The pseudo-thread is a logical concept representing the progress of a particular request, whose state is maintained in the control block.

As the control block is transferred to the next driver, the pseudo-thread continues in that driver.

This concept is particularly useful in mapping from a blocking model. The activities on a "real" thread in a blocking model exactly correspond to the activities on a pseudo-thread in UDI, so you can just take your blocking code and insert a callback at each potentially blocking point.

Agenda

- UDI Portability and Extensibility
- UDI Execution Model
- **UDI Data Model**
- Intro to UDI Specifications
- Q & A

F8: UDI Architecture In-Depth

UDI Data Model

- **Context managed via *control blocks***
  - Used with channel ops & async service calls
  - Environment uses CB to hold service call state
  - Driver uses context pointer in CB to find its data
- **No memory shared between regions**
  - Memory allocated in region private to that region
  - Regions share data with channel operations

UDI Control Blocks

- **CB contains *scratch* and *context* pointers (preserved across service calls, not ops)**
  - » Scratch space in CB holds per-request state
  - » Context pointer lets driver find the context of a channel op or callback
    - Initially set to channel context
    - Channel context struct points to global data
- **All CBs begin with generic udi_cb_t**

Implicit Synchronization

- **No locking primitives required in UDI**
  - All data accesses implicitly synchronized
    - » Region data accessible only from that region
    - » Only one thread per region active at a time
      - Other calls deferred until active call returns
        - Typically by adding CB to a region queue
  - Driver controls its parallelism by picking number and type of regions

Callbacks for async service calls can also be called “immediately”; I.e. before returning to the driver from the service call function. This still maintains the single-threaded nature ot driver execution.

If not called immediately, the callback must be deferred until the driver returns from its last invocation.

Region Kill

- **Different environments have different levels of trust in drivers**
- **UDI environments can:**
  - detect misbehaved drivers (e.g. bad pointers)
  - track resource ownership and transfers
  - abruptly terminate (“region-kill”) driver instances
    - » Frees all resources and shuts down device

Region kill acts like a hot plug abrupt removal event.

Environment chooses whether to region-kill all instances of a driver or just the misbehaving one, and whether to restart with a new instance.

Environment can shut down PIO devices using a pre-registered PIO transaction list handed to it by the driver’s initialization code.

Agenda

- UDI Portability and Extensibility
- UDI Execution Model
- UDI Data Model
- **Intro to UDI Specifications**
- Q & A

UDI Specifications

http://www.sco.com/UDI/specs.html

- **UDI Core Specification**
  - UDI Architecture
    - » Execution, Data and Configuration Models
  - Fundamental Data Types
  - Core Services and Utility Functions
  - Core Metalanguages
  - Packaging & Distribution

UDI Specifications
(continued)

- **UDI Physical I/O Specification**
  - DMA, Programmed I/O (PIO), Interrupts
  - Bus Bridge Metalanguage
- **UDI PCI Bus Binding Specification**
  - PCI enumeration attributes, etc.

UDI Specifications
(continued)

- **UDI Network Driver Specification**
  - Network Interface Card Metalanguage
  - See session F13: UDI Network Drivers
- **UDI SCSI Driver Specification**
  - SCSI Metalanguage
  - See session F12: UDI SCSI Drivers

Fundamental Data Types

- **Specific-length types**
  - udi_ubit8_t, udi_sbit8_t, udi_ubit16_t, udi_sbit16_t, udi_ubit32_t, udi_sbit32_t
  - udi_boolean_t (udi_ubit8_t)
- **Abstract types**
  - udi_size_t, udi_index_t

Fundamental Data Types
(continued)

- **Opaque types**
  - Contain environment-private fields and structure
  - Must be allocated using UDI service calls
  - Opaque handles
    - » udi_channel_t, udi_constraints_t
  - Semi-opaque types
    - » udi_cb_t *, udi_buf_t *

Core Services

- Inter-Module Communication (IMC)
- Memory Management
- Buffer Management
- Time Management
- Tracing and Logging

Core Utility Functions

- **String/Memory Utilities**
  - udi_strcpy, udi_strlen, udi_memcmp et al
  - udi_snprintf, udi_strtou32
- **Queue Management Utilities**
- **Endianness Management Utilities**

Core Metalanguages

- **Management Metalanguage**
  - Environment-initiated control operations
- **Generic I/O Metalanguage**
  - Generic read/write plus custom ops
  - Useful for prototyping and “one-off” extensions
  - Used to access driver diagnostics

Agenda

- UDI Portability and Extensibility
- UDI Execution Model
- UDI Data Model
- Intro to UDI Specifications
- **Q & A**

UDI Information

Web page

http://www.sco.com/UDI

Project UDI contacts

Chair: Kevin Quick, +1 214 654 5173, kquick@iphase.com

Vice Chair: Mark Evenson, +1 408 447 5601, mevenson@cup.hp.com

Secretary: John Lee, +1 650 786 5323, john.lee@eng.sun.com

Editor: Kurt Gollhardt, +1 908 790 2277, kdg@sco.com

Advisor: Mark Bradley, +1 303 684 4753, markb@btc.adaptec.com

Presenter's Notes

F8: UDI Architecture In-Depth

August 19, 1999