# UDI Driver Writer's Guide

# Table of Contents

# Table of Contents

# Introduction to programming a UDI driver

Programming a device driver to the UDI specifications has some significant differences from programming to traditional driver programming models. The differences lie in the layers of abstraction that encapsulate UDI components and shield your code from needing to know intimate details of the target operating system.

By using only UDI interfaces to talk to the OS (in the form of the GIO and External Mappers provided in the UDI environment on the target OS), a UDI driver is able to access OS services in a system− and platform−independent manner.

On the other side, where the driver interfaces with the target device and system hardware, a layer of abstraction provided by the target OS's UDI environment provides a similarly consistent interface to Direct Memory Access (DMA), interrupt processing, and the system bus, and insulates the driver from byte−ordering considerations.

## Differences between traditional and UDI models

In a traditional device driver programming model, device driver code uses facilities provided by the operating system to access intenal OS data structures and OS functions (sometimes called "system calls"). In this model, driver code will typically also use intimate knowledge of the internals of a specific device to implement common operations.

Drivers access operating system services and data structures and provide information back to the operating system through a set of interfaces documented as part of the operating system. Every operating system has a different set of interfaces (though some are similar).

These sets of interfaces taken as a whole across the industry vary widely enough that a new implementation of a particular driver is required for each OS vendor's product line and, within each product line, usually for each new version of the product.

UDI provides one set of interfaces per device type that applies across all operating systems that implement a UDI−compliant environment. See the UDI Environment Implementer's Guide for details.

In defining a model that accomplishes the goals of the UDI effort, it was necessary to make adjustments to the traditional operating system and device driver programming model.

The differences between the traditional and UDI programming models show up primarily in the definition of the UDI environment on a target operating system and in the design of the UDI metalanguage components. Among the most important differences in the UDI environment:

*execution model*
: Driver code is broken up into modules, each of which establishes regions of execution. A region is the basic unit of execution and operations within regions are serialized. UDI's non−blocking execution model allows for completely asynchronous processing in driver code. The UDI execution model provides no global entry points as in traditional driver programming models. Instead, a UDI driver module defines operations over communications channels between the module and the UDI environment, or between the module and another UDI driver module.

*data model*

Data is private to each driver region, with data shared only through communication channels; traditional drivers use much global data. Strong typing is used on all conforming platforms, and is common to all metalanguages; in the traditional model, typing varies widely across platforms and products.

*inter−module communication channels*

Rather than relying on varying inter−process communication techniques on target systems, UDI defines a metalanguage for communication channels between UDI modules. The channel operations defined by a driver are the entry points into a UDI driver module's code. Types of channels and channel operations are defined by UDI metalanguages. In contrast to traditional driver models where these interfaces are defined differently for every target operating system, UDI metalanguages are platform− and product−neutral. The channel also carries state information for the requested operation and is the only method for data sharing between regions.

*programmed I/O*

Programmed I/O (PIO) is the term used for data transfers initiated by a driver between the driver and registers or memory on a device. Some hardware platforms implement PIO via normal memory loads and stores (``memory−mapped I/O''); on others it requires special I/O instructions. In UDI, PIO operations are performed through service calls to the UDI environment coded as function calls rather than by direct memory references or I/O instructions in the drivers. This hides the details of PIO implementation on the host system from the driver.

The UDI Environment on a target system provides a standard interface to system services that might be used by any driver type. For each UDI driver type, UDI employs a metalanguage that provides a consistent interface for that type.

The UDI Core Specification defines basic services common to all driver types, such as driver instantiation, generic input/output, inter−module communication, buffer management, utility functions, and so on.

The UDI Physical I/O Specification defines the interface between the driver and the device itself, usually over an I/O bus or bridge. The manner in which registers on the physical device are accessed is defined, as well as interrupt servicing and direct memeory access. Any definitions that are specific to a particular type oif bus are enumerated in a bus−sepcific document, such as the UDI PCI Bus Binding Specification for the PCI bus.

Building on the foundation of the above documents, the Network Driver Specification and the UDI SCSI Specification define metalanguages and bindings for UDI network adapter drivers and SCSI host bus adapter drivers, respectively.

This modular approach allows new specifications to be added to UDI as they are accommodated in the UDI environment.

# UDI driver coding basics

In this chapter, we use two of the sample drivers, a CMOS RAM driver and a pseudo−device driver, to illustrate how to code the various logical parts of two simple UDI drivers.

The general flow of this description is intended as an example of a step−by−step approach to coding an elementary UDI driver. A different order is certainly likely, particularly for more complex drivers.

## Background information for understanding UDI drivers

As noted in the section ``Differences between traditional and UDI models", UDI drivers follow a fundamentally different programming paradigm from traditional driver programming models.

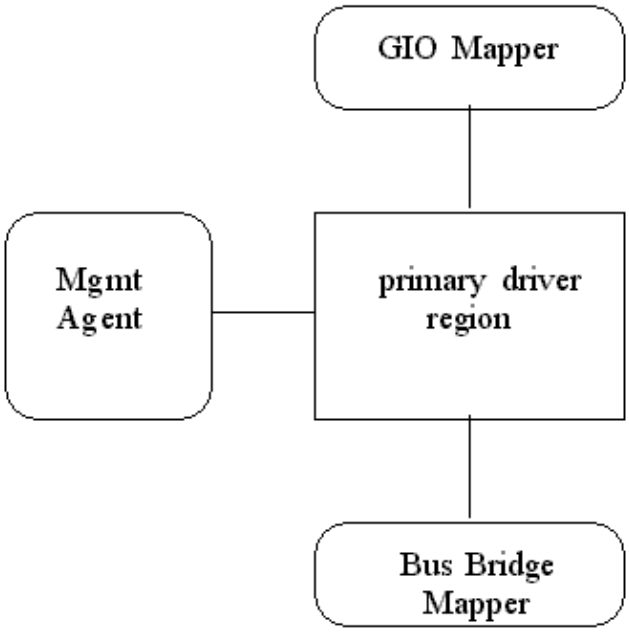
Before reading this chapter for the first time, please review the following sections of the *UDI Core Specification*:

- Terminology
- Execution Model
- Data Model
- Configuration Model
- Calling Sequence and Naming Conventions
- General Requirements

## Design of the CMOS and pseudo−device drivers

The two sample drivers used in this chapter are a CMOS RAM driver (**udi_cmos**) and a pseudo−device driver (**pseudod**).

The following diagram shows the basic module, region, and channel structure for the two drivers.




**udi_cmos and pseudod Driver Architecture**

These drivers exhibit a simple UDI driver architecture with one region and three channels.

As with any UDI driver, both drivers open a management communication channel with the Management Agent, which controls driver configuration and operating conditions. This channel is driven by the Management Agent.

Both drivers define a communciation chanel with the GIO Mapper. This channel is used primarily in these drivers to read from and write to the device, to service calls from the UDI environment, and for interrupt processing.

Finally, both drivers establish a communication channel with the Bus Bridge Mapper to bind to the device; this sets up the physical I/O mappings necessary for the driver to communicate with the device. Although the driver will use the GIO metalanguage to perform reads and writes (since no specialized metalangauge exists for these devices), the mappings are necessarily set up on the channel to the bridge.

For the **udi_cmos** driver, the device is the CMOS RAM on the motherboard; for **pseudod**, the device is a file that simulates a hardware device.

A driver's basic region, module, and channel architecture is reflected in its static property definitions in the *udiprops.txt* file.

# Static driver properties (udiprops.txt)

The source code of a UDI driver must be accompanied by a file that tells the system compiler and UDI packaging utilities about those properties of the driver that are known in advance and will remain the same in any instantiation of the driver.

This file (always named *udiprops.txt*) can be created at any time prior to the first compilation of the driver code; we treat it here first as it provides a good starting point for discussing the driver code itself.

While somewhat analogous to the traditional *Makefile* used by the UNIX System **make**(1) utility and other similar compilation methods, the information contained in *udiprops.txt* includes more than compilation directives.

Below are portions of the sample source for the *udiprops.txt* files for the CMOS RAM (*udi_cmos*) and pseudo−device (*pseudod*) sample UDI drivers. These samples show how a typical *udiprops.txt* file begins, with identification information for the driver and the UDI interface version it uses. A complete discussion of *udiprops.txt* declarations and syntax can be found under Static Driver Properties in the *UDI Core Specification*.

## Driver identification

The top of the *udiprops.txt* file contains the version of the specification to which the driver conforms, and some identification information for the driver source code.

| ***cmos_udi.c* sample code** |
|---|
| `properties_version 0x101`<br><br>## ## Identify ourselves. ##<br><br>supplier 1 message 1 Project UDI contact 2 message 2 http://www.project−UDI.org/participants.html name 3 message 3 CMOS RAM Sample UDI Driver<br><br>shortname udi_cmos release 2 alpha1.0rc3 |
| *pseudod.c* sample code |
| `properties_version 0x101`<br><br>message 1 Project UDI message 2 http://www.project−UDI.org/participants.html message 3 Pseudo−Driver message 4 Generic UDI Pseudo−Driver release 5 1.00_version,_compatible_with_UDI_Specification_1.01<br><br>supplier 1 contact 2 name 3 shortname pseudod |

The **properties_version** declaration is required as the first non−comment statement in the *udiprops.txt* file. The sample drivers conform to version 1.01 of the UDI Specifications.

Each **message** statement is associated with another statement (such as **supplier**) that identifies the content of the message, or is used elsewhere in the driver code.

These declaration are required, but are largely informational, with the exception of the shortname declaration, used by the **udimkpkg**(1) command to name the packaged driver file. For a full description, see Property Declaration Syntax and Common Property Declarations in the *Core Specification*.

## Module and region declarations

A driver's executable is composed of one or more UDI driver modules, each of which can be separately loaded and executed (e.g., into separate addressing domains or protection/privilege domains). Driver writers need to consider the partitioning of their driver into modules in conjunction with the partitioning of driver instances into regions.

Each driver module handles some (mutually−exclusive) subset of the driver's region types. Each driver has one module, called the ``primary module'', which handles the driver's primary region. Additional modules, called secondary modules, may be defined by the driver.

Each UDI driver module has a single well−known global variable, named **udi_init_info** that describes the module's entry points and size requirements (see ``Driver initialization structure''). There are no global entry points into UDI drivers; all entries are through function pointers defined in **udi_init_info**.

The information specified in this section of the *udiprops.txt* file is used in the region, operation, control block, and other definitions pointed to by the **udi_init_info** structure definitions for each driver module.

***cmos_udi.c* sample driver *udiprops.txt* (cont.)**

```
   ##
   ## Declare the driver's module(s) and region types.
   ## This driver has a single region in a single module.
   ##


module udi_cmos region 0

## ## Declare interface dependencies. ##

requires udi 0x101 requires udi_physio 0x101 requires udi_bridge 0x101 requires udi_gio 0x101

## ## Describe metalanguage usage. ##

meta 1 udi_gio # Generic I/O Metalanguage meta 2 udi_bridge # Bus Bridge Metalanguage

child_bind_ops 1 0 1 # GIO meta, primary region, ops_index 1 parent_bind_ops 2 0 2 4 # Bridge meta, primary region, ops_index 2

pio_serialization_limit 1 # We use only a single serialization domain.
```

**pseudod.c** sample driver *udiprops.txt* (cont.)

```
   ##
   ## Interface dependencies
   ##
   requires udi         0x101
   requires udi_physio  0x101
   requires udi_bridge  0x101
   requires udi_gio     0x101


## ## Build instructions. ##
```

```
module pseudod compile_options −DPSEUDO_GIO_META=1 −DPSEUDO_BUS_META=2 source_files pseudo.c pseudo.h region 0

## ## Metalanguage usage ##

meta 1 udi_gio # Generic I/O Metalanguage meta 2 udi_bridge # Bus Bridge Metalanguage

child_bind_ops 1 0 1 # GIO meta, primary region, ops_index 1 parent_bind_ops 2 0 2 2 # Bridge meta, primary region, ops_index 2 # bus bind cb idx 2
```

The region and module structure of the **cmos_udi** and **pseudod** drivers is apparent from the *udiprops.txt* declarations shown above. These declarations are used in the eventual definition of the required **udi_init_info** structure in the source code of each module of the driver.

All UDI drivers must first define the versions of each UDI interface used by the drivers, done in the requires declarations shown above, all of which point to version 1.0.0 interfaces defined in the Version 1.0.0 UDI Specifications.

Every UDI driver must declare at a minimum the versions of the four metalanguage interfaces shown used in the driver. These interfaces are all part of the UDI environment on the target system and are all present in a conforming UDI implementation. The driver code may or may not use all these interfaces, and in fact the two samples shown use only the udi_gio and udi_bridge interfaces.

Both of these drivers are single−module, single−region drivers as defined by the **module** and **region** declarations. Regions and modules are numbered (starting at zero) to provide an index to be used in the driver source code and subsequent declarations.

Similarly, the **meta** declarations show which defined metalanguages will be used by this driver and are also indexed; metalanguage indexes must be non−zero.

Both the region and metalanguage indexes are used in the subsequent operations declarations, which are used to construct the operations vector for each defined region.

Three types of operations declarations are permitted:

***child_bind_ops meta_idx region_idx ops_idx***

Defines a communication channel for the given region using the given metalanguage that will be used to receive operations requests from another region. There should be one of these declarations for each channel in which a driver region participates as a parent. The region on the other end of the channel may be a UDI environment service provider region or another UDI driver module region, as indicated by the metalanguage used.

***parent_bind_ops meta_idx region_idx ops_idx***

Defines a communication channel for the given region using the given metalanguage that will be used to send operations requests to another region. There should be one of these declarations for each channel in which a driver region participates as a child. The region on the other end of the channel may be a UDI environment service provider region or another UDI driver module region, as indicated by the metalanguage used.

***internal_bind_ops meta_idx region_idx primary_ops_idx secondary_ops_idx***

Defines a communication channel between the primary region and a secondary region of a driver module. Single−region drivers do not need any **internal_bind_ops** declarations. Multi−region drivers require exactly one **internal_bind_ops** declaration per secondary region.

Most driver modules will have at least one parent and one child relationship, though they are not strictly required.

Drivers without parent operations are called ``orphans'' and control no real devices. Orphans may receive operations requests from other drivers or the UDI environment, but do not issue any operations requests outside the driver's defined regions. [Note that although the **pseudod** sample driver does not control any real devices, it still has a **parent_bind_ops** declaration because it simulates real output to a device.]

It is likewise permitted to have a driver with no child operations defined, but such code could not be considered a device driver since it receives no operation requests, interrupts, etc., to which it can respond. It is really an application running as a UDI driver.

Each of these define a possible communication channel relationship between the driver module in which they appear and another UDI module. These definitions are used in defining communications channels either between two regions defined by the driver (**internal_bind_ops**) or between a driver region and a region of another UDI driver module or core service provider.

In the sample drivers' *udiprops.txt* files above, two channel operations are defined. The possible channels defined are a channel to the GIO mapper in the UDI environment in which the driver acts as the child, and a channel to the bus bridge mapper in which the driver acts as the parent. Each channel uses the indexed metalanguage appropriate to the operations needed on the channel.

We'll discuss how these indexes are used in driver code and what these operations declarations mean in more detail below, when we discuss the **udi_init_info** structure required for each driver (see ``Driver initialization structure'').

## Device types

A driver's *udiprops.txt* file must include one or more **device** declarations, describing the devices supported by the driver. The sample **udi_cmos** and **pseudod** drivers define these devices:

***cmos_udi.c* sample driver *udiprops.txt* (cont.)**

```
   # Since this is a "System"−bus device, which is not self−identifying,
   # we have to specify a default I/O address range with "config_choices".
   # For PCI devices, there'd be no "config_choices" and the "device"
   # declaration would include attributes like "pci_vendor_id".


# This device line is an example of what a real PCI device line # would look like. Currently the CMOS device is being supplied # as a fake PCI entry. See comments above. device 10 2 bus_type string pci pci_vendor_id ubit32 0x434d \ pci_device_id ubit32 0x4f44 #device 10 2 bus_type string system

message 10 Motherboard CMOS RAM #config_choices 10 ioaddr1 ubit32 0x70 any iolen1 ubit32 2 only
```

**pseudod.c** sample driver *udiprops.txt* (cont.)

```
   # This is a pseudo driver but to easily instantiate the device using the bus
   # bridge enumeration mechanism, we fake ourselves to be recognised from some
```

```
   # common PCI bridge and/or video controller IDs.
   # This list can be changed to add or remove other controller IDs depending
   # on the system being used.
   #
   # NOTE: At the moment only the first device line is referenced for matching
   # against the enumerated PCI devices. This means the order has to be
   # changed to match what PCI devices are physically present in the PC.
   # This will work properly when multiple device lines are handled by the
   # udi_MA code


# Toshiba Vendor ID = 0x1179 (These are probably pcmcia masters)

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x1179 \ pci_device_id ubit32 0x0701

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x1179 \ pci_device_id ubit32 0x0406

# Intel vendor ID = 0x8086 # Common Intel motherboard PCI masters

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x8086 \ pci_device_id ubit32 0x7180

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x8086 \ pci_device_id ubit32 0x7181

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x8086 \ pci_device_id ubit32 0x7111

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x8086 \ pci_device_id ubit32 0x7113

# Motorola Vendor ID = 0x1057 # Devices: # Motorola MPC 105 (Eagle) = 0x0001 # Motorola MPC 106
(Grackle) = 0x0002 # Both of these PCI bridges can be found in Old World Apple PowerMacs

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x1057 \ pci_device_id ubit32 0x0001

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x1057 \ pci_device_id ubit32 0x0002

# Apple Vendor ID = 0x106B # Devices: # PowerMac G4 sawtooth (New World/AGP) host bridge: 0x0020

device 4 2 \ bus_type string pci \ pci_vendor_id ubit32 0x106B \ pci_device_id ubit32 0x0020

# # Initialization, shutdown messages # message 1100 pseudod: devmgmt_req %d message 1500 pseudod:
final_cleanup_req
```

In general, a driver includes one device declaration for each device model supported by the driver:

```
device msgnum meta_idx { attr_name attr_type attr_value }...
```

The ***msgnum*** supplied must refer to a **message** declaration elsewhere in the file. These are meant to be human−readable descriptions of the device model.

The ***meta_idx*** must refer to a **meta** metalanguage declaration, as well as to the **meta_idx** used in a **parent_bind_ops** declaration, found elsewhere in the file.

The attribute name, type, and value triplets that follow must be valid enumeration attribute names, types, and

values for the metalanguage specified by **meta_idx**. These are found in the specification for the metalanguage.

Both the above drivers specify the PCI Bus Bridge Metalanguage for the devices they define. The enumeration attributes used are explained under Instance Attribute Bindings in the *PCI Bus Binding Specification*. This specification defines the implementation of Physical I/O on the PCI Bus.

For the set of allowable ***attr_type*** values, see the detailed explanation of the Device Declaration in the *UDI Core Specification*.

## Build instructions

Build instructions are statements in the *udiprops.txt* file that are evaluated at compile time of the driver modules. A discussion of all the compile time options can be found in the *UDI Core Specification* under Build−Only Properties.

In the sample drivers, the **compile_options** statement is used to define values used in the driver source files listed in the **source_files** statements.

| ***cmos_udi.c* sample driver *udiprops.txt* (cont.)** |
|---|
| `compile_options −DCMOS_GIO_META=1 −DCMOS_BRIDGE_META=2`<br>`source_files udi_cmos.c` |
| *pseudod.c* sample driver *udiprops.txt* (cont.) |
| `compile_options −DPSEUDO_GIO_META=1 −DPSEUDO_BUS_META=2`<br>`source_files pseudo.c pseudo.h` |

The values used for these constants correspond to the values given in the **meta** statements in *udiprops.txt*; see ``Module and region declarations''. They are also used in the source files; see ``Control block indexes and structures''.

# Versions and header files

This section begins the examination of the source code modules for the **udi_cmos** and the **pseudod** sample drivers.

Each source code module of a UDI driver must declare which versions of the UDI interfaces are used by the module code, before defining the header files on which the source depends.

| ***cmos_udi.c* sample code (cont.)** |
|---|
| `#define UDI_VERSION 0x101`<br>`#define UDI_PHYSIO_VERSION 0x101`<br><br>#include <udi.h> /* Standard UDI Definitions */ #include <udi_physio.h> /* Physical I/O Extensions */ |
| **pseudod.c** sample code (cont.) |
| `#define UDI_VERSION 0x101`<br>`#define UDI_PHYSIO_VERSION 0x101` |

```
#include <udi.h> /* Standard UDI definitions */ #include <udi_physio.h> /* Physical I/O Extensions */
#include "pseudo.h"
```

These declarations ensure that the UDI implementation on the target OS provides the correct compile−time and run−time environments for the driver.

As of the release of this document, the only valid values for UDI interfaces is 0x101, corresponding to the 1.01 verison of the UDI specifications.

Both the **udi_cmos** and the **pseudod** drivers uses interfaces defined in the UDI Core Specification and UDI Physical I/O Specification, and so includes the header files required by those specifications. The **pseudod** driver also includes its own header file, *pseudod.h*.

# Region data definitions

Each region of a driver is allocated a ``region data'' structure, a per−instance region of memory that is only accessible by the associated region and is used by that region to store information relevant to the operation of that region. This region−specific information often includes: context pointers, state variables, request queues, PIO handles for accessing the device, and information about the channels connected to that region.

---

**NOTE:** The only data truly global to a driver (that is, data that can be read from any module, region, or operation context) is called ``module−global data''; see ``Module−global data''.)

---

The size of the memory area for region data allocated by the UDI environment is determined by the `rdata_size` member of a region's **udi_primary_init_t**(3udi) or **udi_secondary_init_t**(3udi), ass appropriate, in the driver's **udi_init_info**(3udi) structure. The udi_cmos and pseudod drivers have only one region (see ``Primary region initialization'').

The beginning of the region data area is always a **udi_init_context_t**(3udi) structure, and is initialized by the Management Agent (MA) when a region is created. If the size of the memory area allocated is larger than a **udi_init_context_t** structure, the remainder of the memory area is initialized to zero.

A pointer to the initial region data area in memory is made available to the driver as the channel `context` for the region's initial channel, which the driver can access via `cb−>gcb.context` of any control block it receives over this channel.

Subsequently, when a channel operation or entry point is initiated by the UDI environment, the `context` pointer in the control block passed as part of the channel operation is set to point to the region data area in memory for the region associated with that channel. The channel context is the only information provided to the target region for a channel operation beyond the operation−specific data in the control block and associated parameters.

The region data definitions included in a driver are the driver's interpretation of the memory space allocated as region data by the MA.

A channel operation typically accesses the contents of the region data area by allocating a variable using the region data type declared by the driver, and initializing it to point to the region data area in memory. For example, a channel operation might use the following code to set a pointer to region data area in memory from the pointer passed as part of the channel operation control block:

```
my_region_data_t *rdata = gcb->context;
```

where `my_region_data_t` is a previously declared type in the driver cource. Assuming a data field in the rregion data structure, we might want to set that field in the region data area in memory to a new value, so that a future channel operation can access that value:

```
rdata->data = new_data;
```

This is how region data is typically used in the ``Channel operations'' for the sample drivers.

## udi_cmos region data structure

The **cmos_udi** driver defines a region data structure that includes the **udi_init_context_t** information automatically allocated in memory when the driver's primary region is intialized. This is followed by the necessary control block for bus binding, and two handles for PIO read and write for the CMOS RAM device. As region data, these definitions and values are available to all operations defined for the driver's primary region.

***cmos_udi.c* sample code (cont.)**

```
/*
 * Region data structure for this driver.
 */
typedef struct {
        /* init_context is filled in by the environment: */
        udi_init_context_t      init_context;
        udi_bus_bind_cb_t       *bus_bind_cb;
        /* PIO trans handles for reading and writing, respectively: */
        udi_pio_handle_t        trans_read;
        udi_pio_handle_t        trans_write;
} cmos_region_data_t;
```

**udi_cmos.c region data structure**

### init_context

When a region is created, the initial contents of the start of the region data area are initialized to be a **udi_init_context_t** structure (see **udi_init_context_t**(3udi)). Once initialized, region data is entirely under the control of the associated driver region code. The driver may choose to preserve this structure's content or overwrite it. This element is used by the subroutines in the sample drivers to pass context information back to the environment via channel operations.

A region's **udi_init_context_t** structure contains two pieces of data:

***region_idx***
: a region index of type **udi_index_t** (matching an index from a **region** statement in the driver's *udiprops.txt*) that identifies this as the primary or a secondary region

***limits***
: a structure of type **udi_limits_t** containing resource limits for the UDI implementation on the host system

Since the implementation of many system resource limits (for example, memory allocation and timer resolution limits) varies between systems on which a UDI driver might run, the **limits** structure provides a way for UDI drivers to query the region environment and adjust memory and other allocation algorithms to make best use of the UDI environment implementation on any host system. For more detail on this structure, see **udi_limits_t**(3udi).

### bus_bind_cb

**bus_bind_cb** is the control block used for bus bridge binding operations. The **udi_cmos** driver needs to bind to the bus in order to issue channel requests to CMOS RAM, and needs to occasionally save this control block to region data for access by subsequent channel operations. The control block used as the bus binding control block is the generic control block type, **udi_cb_t**, which contains these members:

***channel***
: is a handle to the channel currently associated with this control block.

***context***
: is a pointer to state information within the driver region. On entry to a channel operation, the environment sets context to the channel's current context. Drivers may change it if needed.

***scratch***
: is a pointer to the control block's scratch area. See ``Scratch space, offsets, and requirements" for a description of scratch areas.

***initiator_context***
: is a context pointer that the initiator of a request or indication operation can use to associate per−request context with this control block. If and when the control block is returned to the initiator via an acknowledgement, nak, or response operation, the initiator can use this context pointer to access any additional state data it needs to complete the operation.

See **udi_cb_t**(3udi) and **udi_bus_bind_cb_t**(3udi).

### trans_read/trans_write

The Programmed I/O (PIO) handle type, **udi_pio_handle_t**, holds an opaque handle that refers to an environment object that contains addressing, data translation and access constraint information, as well as a PIO transaction list. The transaction list specifies the PIO operations to be invoked when the handle is passed to **udi_pio_trans**. The udi_cmos driver requires two handles for read and write operations on CMOS RAM.

See ``Programmed I/O (PIO)" for more on how the PIO handles are used in the CMOS driver.

Note the **pseudod** driver does not use PIO, since there is no real device associated with the driver.

## pseudod.c region data structure

The pseudod driver's region data structure contains more data than the **udi_cmos** driver's region data structure, to manage simulating reads and writes on a device.

| ***pseudod.c* sample code (cont.)** |
|---|

**NOTE: pseudo_region_data_t** is defined in the *pseudod.h* header file.

```
/*
 * Region data
 */
#define PSEUDO_BUF_SZ 1024
typedef struct pseudo_region_data {
     udi_init_context_t init_context;
     test_state_t testmode;
     udi_ubit32_t testcounter;
     udi_ubit8_t rx_queue[PSEUDO_BUF_SZ];    /* A 'holding' place for
                                      * writes
                                      */
     udi_ubit8_t * tx_queue;         /* The test pattern we
                                      * generate to fulfill reads.
                                      * points to a movable mem blk.
                                      */
     udi_bus_bind_cb_t *bus_bind_cb;
     udi_xfer_constraints_t xfer_constraints;        /* xfer constraints */
} pseudo_region_data_t;
```

*init_context*
: See ``init_context".

*testcounter*
: This element is used to count the test data read in the **pseudo_gio_do_read** GIO channel operation (see ``pseudod child channel operations").

*rx_queue*

*tx_queue*
: These elements are used in various channel operations to simulate real data reads and writes.

*bus_bind_cb*
: See ``bus_bind_cb".

*xfer_constraints*
: The **udi_xfer_constraints_t**(3udi) structure is used to describe the ``transfer constraints" for a specific channel operation. These transfer constraints are passed to a child via a metalanguage−specific bind acknowledgement channel operation to communicate the transfer requirements to the child. The transfer constraints information determines how to divide requests into appropriate individual control blocks and buffers for handling by the parent driver. The **pseudod** driver uses defines and passes transfer constraints to the GIO Mapper via pseudo_gio_bind_req, the acknowledgemenmt routine for the GIO bind channel operation (see ``pseudod child channel operations").

# Scratch space, offsets, and requirements

A control block optionally defines additional space that may be used by the driver to store information related to a channel operation request. This space is referred to as the ``scratch space" of the control block and its

contents are determined by the driver. Scratch space is accessed via a scratch pointer in the control block.

Scratch space is transferred as part of the control block to another region during a channel operation. When this occurs, the region on the other end of the channel takes ownership of the scratch space, and so its contents may not be preserved when the control block is returned. (Note that the contents of the scratch space **are** preserved, however, on return from an asynchronous service call to the UDI environment. See Asynchronous Service Calls in the Core Specification.)

For the **udi_cmos** driver, this pointer is contained in the ``bus_bind_cb" structure. This segment of the driver module source defines what the scratch area for **bus_bind_cb** will look like.

***cmos_udi.c* sample code (cont.)**

```
/*
 * Scratch structures for various control block types.
 */
typedef struct {
     udi_ubit8_t     addr;           /* CMOS device address */
     udi_ubit8_t     count;          /* # bytes to transfer */
} cmos_gio_xfer_scratch_t;


/* * GIO transfer scratch offsets for PIO trans list processing. */ #define SCRATCH_ADDR
offsetof(cmos_gio_xfer_scratch_t, addr) #define SCRATCH_COUNT offsetof(cmos_gio_xfer_scratch_t,
count)

/* * Scratch sizes for each control block type. */ #define GIO_BIND_SCRATCH 0 #define
GIO_XFER_SCRATCH sizeof(cmos_gio_xfer_scratch_t) #define GIO_EVENT_SCRATCH 0 #define
BUS_BIND_SCRATCH 0 #if DO_INTERRUPTS #define INTR_ATTACH_SCRATCH 0 #define
INTR_EVENT_SCRATCH 0 #endif
```

*pseudod.c* sample code (cont.)

**NOTE:** The **pseudod** driver uses no scratch space.

The **udi_cmos** driver reads and writes CMOS RAM values. The scratch space for the control block used when communicating with the device is a structure containing two unsigned 8−bit values: a CMOS address and a byte count, (For a discussion of UDI data types, see in the Core Specification.)

After defining the **cmos_gio_xfer_scratch_t** structure type, two offsets are defined for the **addr** and **count** values contained in the scratch buffer.

What follows is a series of **#define** statements declaring the scratch space requirements for each control block type used in the driver code. These are used in the control block structure definitions that follow.

# Control block indexes and structures

Control block arrays specify the structure of the control block for each channel operation. There is a one−to−one correspondence between elements in the control block array and the elements of the channel operations vector.

Control blocks passed into a region from another region contain ``per−request data", whose scope is contained to the channel operation request. The ownership of these per−request data objects is transferred to the target region on the channel; the source region has no access to them once they are transferred. No assumptions about their content can be made on return from the operation.

Control blocks are referred to by an index, and are intialized as part of the **uid_init_info** per−driver intialization structure (see ``Driver initialization structure"). The indexes and structures for the sample drivers are shown below.

***cmos_udi.c* sample code (cont.)**

```
   /*
    * CB indexes for each of the types of control blocks used.
    */
   #define GIO_BIND_CB_IDX              1
   #define GIO_XFER_CB_IDX              2
   #define GIO_EVENT_CB_IDX     3
   #define BUS_BIND_CB_IDX              4
   #if DO_INTERRUPTS
   #define INTR_ATTACH_CB_IDX   5
   #define INTR_EVENT_CB_IDX    6
   #endif


/* * ------------------------------------------------------------------------ *
Control Block init section: *
------------------------------------------------------------------------ */ static
udi_cb_init_t udi_cmos_cb_init_list[] = { { GIO_BIND_CB_IDX, /* GIO Bind CB */ CMOS_GIO_META,
/* from udiprops.txt */ UDI_GIO_BIND_CB_NUM, /* meta cb_num */ GIO_BIND_SCRATCH, /* scratch
requirement */ 0, /* inline size */ NULL /* inline layout */ }, { GIO_XFER_CB_IDX, /* GIO Xfer CB */
CMOS_GIO_META, /* from udiprops.txt */ UDI_GIO_XFER_CB_NUM, /* meta cb_num */
GIO_XFER_SCRATCH, /* scratch requirement */ 0, /* inline size */ NULL /* inline layout */ }, {
GIO_EVENT_CB_IDX, /* GIO Event CB */ CMOS_GIO_META, /* from udiprops.txt */
UDI_GIO_EVENT_CB_NUM, /* meta cb_num */ GIO_EVENT_SCRATCH, /* scratch requirement */ 0, /*
inline size */ NULL /* inline layout */ }, { BUS_BIND_CB_IDX, /* Bridge Bind CB */
CMOS_BRIDGE_META, /* from udiprops.txt */ UDI_BUS_BIND_CB_NUM, /* meta cb_num */
BUS_BIND_SCRATCH, /* scratch requirement */ 0, /* inline size */ NULL /* inline layout */ }, #if
DO_INTERRUPTS { INTR_ATTACH_CB_IDX, /* Interrupt Attach CB */ CMOS_BRIDGE_META, /*
from udiprops.txt */ UDI_BUS_INTR_ATTACH_CB_NUM, /* meta cb_num */
INTR_ATTACH_SCRATCH, /* scratch requirement */ 0, /* inline size */ NULL /* inline layout */ }, {
INTR_EVENT_CB_IDX, /* Interrupt Event CB */ CMOS_BRIDGE_META, /* from udiprops.txt */
UDI_BUS_INTR_EVENT_CB_NUM, /* meta cb_num */ INTR_EVENT_SCRATCH, /* scratch
requirement */ 0, /* inline size */ NULL /* inline layout */ }, #endif /* DO_INTERRUPTS */ { 0 /*
Terminator */ } };
```

*pseudod.c* sample code (cont.)

```
   #define PSEUDO_GIO_XFER_CB_IDX       1       /* Generic xfer  blk for udi_gcb_init */
   #define      PSEUDO_BUS_BIND_CB_IDX  2


static udi_ops_init_t pseudo_ops_init_list[] = { { PSEUDO_GIO_INTERFACE, PSEUDO_GIO_META, /*
meta index */ UDI_GIO_PROVIDER_OPS_NUM, /* meta ops num */ 0, /* chan context size */
(udi_ops_vector_t *) & pseudo_gio_provider_ops, /* ops vector */ pseudo_default_op_flags }, {
PSEUDO_BUS_INTERFACE, PSEUDO_BUS_META, /* Bus meta index [from udiprops.txt] */
UDI_BUS_DEVICE_OPS_NUM, 0, /* channel context size */ (udi_ops_vector_t *) &
```

```
pseudo_bus_device_ops, pseudo_default_op_flags }, { 0} };
```

All control blocks are initialized using an array of **udi_cb_init_t** structures, using the control block indexes as indexes into the array.

```
typedef struct {
     udi_index_t            cb_idx ;
     udi_index_t            meta_idx ;
     udi_index_t            meta_cb_num ;
     udi_size_t             scratch_requirement ;
     udi_size_t             inline_size ;
     const udi_layout_t     *inline_layout ;
} udi_cb_init_t ;
```

For the sample drivers, the `cb_idx` definitions are shown above. The `meta_idx` members are populated using defined constants from the build instructions section of *udiprops.txt* (see ``Build instructions").

The type of the control block is specified by the `meta_cb_num` element, a ``control block group number" from the UDI Specifications. The **meta_idx** and **meta_cb_num**, taken together, specify a control block structure type defined for a particular metalanguage. For example, in the **udi_cmos** sample, a **meta_idx** of **CMOS_GIO_META** points to the GIO metalanguage defined in the Core Specification, and a **meta_cb_num** of **UDI_GIO_XFER_CB_NUM** specifies the control block for GIO transfer operations, **udi_gio_xfer_cb_t**(3udi). This is used later in the driver's child channel operations (see ``udi_cmos child channel operations").

Scratch space requirements for the **udi_cmos** driver are given using the constants defined earlier (see ``Scratch space, offsets, and requirements"). The **pseudod** driver uses no scratch space, so gives its requirements as zero.

The next two elements of the control block intialization structure, **inline_size** and **inline_layout**, may contain size and layout information for ``inline data" in the control block. This is data particular to a channel operation that is local to the operation (called ``per−request data" in ``Control block indexes and structures").

For example, in the **pseudod** sample driver, the first control block definition above specifies inline data size and layout information for the **udi_gio_xfer_cb_t** control block type specified in the control block initialization structure **udi_cb_init_t**. The layout is specified using the `udi_layout_t` type prior to the control block intialization structure definition in the source, The layout uses UDI fundamental types, and is specific to the channel operation. See ``Channel operation data layouts".

# Programmed I/O (PIO)

The **udi_cmos** driver uses PIO to read and write values on the CMOS RAM device. This is done by first specifying the I/O characteristics of the device.

**"*cmos_udi.c* PIO device properties"**

```
/*
 * PIO properties of the physical I/O registers on the CMOS device.
 */
#define CMOS_REGSET  1       /* first (and only) register set */
#define CMOS_BASE    0       /* base address within this regset */
#define CMOS_LENGTH  2       /* two bytes worth of registers */
#define CMOS_PACE    5       /* wait 5 microseconds between accesses
                              * (not really need for this device,
```

```
                                * but illustrates use of pacing)
                                */
```

The above definitions are used in calls to **udi_pio_map** in the driver subroutines to establish device characteristics for the CMOS RAM device. This is called from a driver subroutine defined later. The set of definition needed for a given device are usually quite specific to that device.

**"*cmos_udi.c* PIO offsets"**

```
/*
 * PIO offsets for various device registers, relative to the base of
 * the device's register set.
 */
#define CMOS_ADDR    0       /* address register */
#define CMOS_DATA    1       /* data register */
```

The address and data register definitions are used in the transaction lists described below.

**"*cmos_udi.c* other device properties"**

```
/*
 * Other device properties.
 */
#define CMOS_DEVSIZE 0x40    /* # data bytes supported by device */
#define CMOS_RDONLY_SZ       0x0E    /* first 14 bytes are read-only */
```

The device size definitions are used in binding, transfer, and write subroutine calls defined later in the driver, to avoid overflowing the device or writing to read−only device memory.

**"*cmos_udi.c* sample code (cont.)"**

```
/*
 * PIO trans lists for access to the CMOS device.
 */
static udi_pio_trans_t cmos_trans_read[] = {
     /* R0 <- SCRATCH_ADDR {offset into scratch of address} */
     { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_ADDR },
     /* R1 <- address */
     { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
                            UDI_PIO_1BYTE, UDI_PIO_R1 },
     /* R0 <- SCRATCH_COUNT {offset into scratch of count} */
     { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_COUNT },
     /* R2 <- count */
     { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
                            UDI_PIO_1BYTE, UDI_PIO_R2 },
     /* R0 <- 0 {current buffer offset} */
     { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, 0 },
     /* begin main loop */
     { UDI_PIO_LABEL, 0, 1 },
     /* output address from R1 to address register */
     { UDI_PIO_OUT+UDI_PIO_DIRECT+UDI_PIO_R1,
                            UDI_PIO_1BYTE, CMOS_ADDR },
     /* input value from data register into next buffer location */
     { UDI_PIO_IN+UDI_PIO_BUF+UDI_PIO_R0,
                            UDI_PIO_1BYTE, CMOS_DATA },
     /* decrement count (in R2) */
     { UDI_PIO_ADD_IMM+UDI_PIO_R2, UDI_PIO_1BYTE, (udi_ubit8_t)-1 },
     /* if count is zero, we're done */
     { UDI_PIO_CSKIP+UDI_PIO_R2, UDI_PIO_1BYTE, UDI_PIO_NZ },
     { UDI_PIO_END_IMM, UDI_PIO_2BYTE, 0 },
```

```
        /* increment address and buffer offset */
        { UDI_PIO_ADD_IMM+UDI_PIO_R1, UDI_PIO_1BYTE, 1 },
        { UDI_PIO_ADD_IMM+UDI_PIO_R0, UDI_PIO_1BYTE, 1 },
        /* go back to main loop */
        { UDI_PIO_BRANCH, 0, 1 }
    };
    static udi_pio_trans_t cmos_trans_write[] = {
        /* R0 <- SCRATCH_ADDR {offset into scratch of address} */
        { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_ADDR },
        /* R1 <- address */
        { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
                                UDI_PIO_1BYTE, UDI_PIO_R1 },
        /* R0 <- SCRATCH_COUNT {offset into scratch of count} */
        { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_COUNT },
        /* R2 <- count */
        { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
                                UDI_PIO_1BYTE, UDI_PIO_R2 },
        /* R0 <- 0 {current buffer offset} */
        { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, 0 },
        /* begin main loop */
        { UDI_PIO_LABEL, 0, 1 },
        /* output address from R1 to address register */
        { UDI_PIO_OUT+UDI_PIO_DIRECT+UDI_PIO_R1,
                                UDI_PIO_1BYTE, CMOS_ADDR },
        /* output value from next buffer location to data register */
        { UDI_PIO_OUT+UDI_PIO_BUF+UDI_PIO_R0,
                                UDI_PIO_1BYTE, CMOS_DATA },
        /* decrement count (in R2) */
        { UDI_PIO_ADD_IMM+UDI_PIO_R2, UDI_PIO_1BYTE, (udi_ubit8_t)-1 },
        /* if count is zero, we're done */
        { UDI_PIO_CSKIP+UDI_PIO_R2, UDI_PIO_1BYTE, UDI_PIO_NZ },
        { UDI_PIO_END_IMM, UDI_PIO_2BYTE, 0 },
        /* increment address and buffer offset */
        { UDI_PIO_ADD_IMM+UDI_PIO_R1, UDI_PIO_1BYTE, 1 },
        { UDI_PIO_ADD_IMM+UDI_PIO_R0, UDI_PIO_1BYTE, 1 },
        /* go back to main loop */
        { UDI_PIO_BRANCH, 0, 1 }
    };
```

**PIO transaction lists for cmos_udi**

The figure above shows the transaction lists for the read and write PIO operations to the CMOS RAM device. Each of the array members of the **cmos_trans_read[]** array contains three values that specifies an atomic operation on the device. When the **trans_read** PIO transaction handle is specified with the **udi_pio_trans** environment call, the list of operations specified in the **cmos_trans_read[]** array is performed on the device. This is done in the driver subroutines described later.

Similar processing occurs with the **cmos_trans_write[]** array when the **trans_write** transaction handle is passed to **udi_pio_trans**.

As with device characteristics, the transaction lists necessary for any device are unique to that device.

The operation, register, and address mode constants used in the transaction lists ared defined in the Physical I/O Specification description of which documents transaction list syntax. The remainder are defined in the driver code.

Each **cmos_trans_read[]** and **cmos_trans_write[]** array element is a **udi_pio_trans_t** structure with three

members:

```
typedef struct {
udi_ubit8_t pio_op ;
udi_ubit8_t tran_size ;
udi_ubit16_t operand ;
} udi_pio_trans_t ;
```

Simply put, these specify a PIO operation (such as a logical OR of a register and an operand), a transaction size specifying the size of the result of the operation, and any operand required by the PIO operation.

The **cmos_trans_read[]** PIO list, described below in detail, reads a data value from CMOS RAM and copies it to the scratch space in the control block passed as part of the channel operation invoking the PIO read.

Note that the UDI environment provides eight special registers for use in constructing transaction operations. We'll refer to these internal UDI registers in the same manner as the UDI Specification does, using the notations R0 through R7. We'll refer to registers on the device by spelling out the word ``register'' before the number.

1. Load R0 with a two−byte address value, an offset into scratch space. **UDI_PIO_LOAD_IMM** is a PIO operation that sets one of R0 through R7 (in this case R0), to the value of the supplied operand. **UDI_PIO_2BYTE** gives the size of the transfer as two bytes. **SCRATCH_ADDR** is a constant set elsewhere in the driver module code (see ``Scratch space, offsets, and requirements''), and contains the offset of the `addr` element in the **cmos_gio_xfer_scratch_t** control block scratch space.
2. Load R1 with the **addr** value from scratch space. **UDI_PIO_LOAD** is a PIO operation that sets on of R0 through R7, named by the 3 low order bits of the provided operand, to the value found at a provided address, using a specified ``addressing mode''. **UDI_PIO_SCRATCH** is the addressing mode, specifying that the address used is a 32−bit offset into the scratch space of the control block passed as part of the channel operation that invoked the read transaction. **UDI_PIO_R0** is R0, which currently holds the offset into scratch space from transaction 1. **UDI_PIO_1BYTE** is the size of the data transferred from the address. **UDI_PIO_R1** is R1, which gets loaded with the value from scratch space.
3. Transaction 3 is similar to transaction 1, execpt this time R0 is loaded with the offset of the `count` element in the **cmos_gio_xfer_scratch_t** control block scratch space.
4. Transaction 4 is similar to transaction 2, except now R2 is loaded with the **count** value from scratch space.
5. Register 0 is now initialized to the current buffer offset (0). This operation uses the same format as transactions 1 and 3.
6. This transaction labels the beginning of a loop which reads **count** bytes (the value currently in R2) from the address on the CMOS RAM device (held in R1), into the buffer at the offset specified in R0 (which gets incremented as data is read byte by byte).

   **UDI_PIO_LABEL** sets a label to which a subsequent **UDI_PIO_BRANCH** can switch control. The **tran_size** element is unused and must be set to zero. The **operand** element is set to a unique integer to be used as the **operand** of a subsequent **UDI_PIO_BRANCH**.
7. Output the value in R1 directly to register 0 (the address register) of the CMOS RAM device (this sets the data register on the device to the value at the specified address in device memory). **UDI_PIO_OUT** is a PIO operation that sets the location on the PIO device specified by the operand to a value held in a UDI internal register, **UDI_PIO_DIRECT** is the addressing mode, which specifies that the content of **UDI_PIO_R1** (R1) is to be used as the data (not the address of the data). **UDI_PIO_1BYTE** specifies a one−byte transfer. **CMOS_ADDR** is the previously set constant containing the offset into device address space for the address register (see ``trans_read/trans_write'').

8. Read the data from the CMOS device data register into the buffer. **UDI_PIO_IN** is the reverse of **UDI_PIO_OUT** from the previous transaction. **UDI_PIO_IN** sets the location specified by the value in R0 (using the **UDI_PIO_BUF** addressing mode), with the byte of data located on the PIO device at the offset specified by the operand **CMOS_DATA**. **UDI_PIO_BUF** specifies that the value in R0 is a 32−bit offset into the data buffer.
9. The **count** held in R2 is decremented to indicate that one byte has been read from the device to the buffer. This is done using the previously described **UDI_PIO_ADD_IMM** operation, this time with an explicit cast of the operand (−1) to a one−byte UDI fundamental type. [No subtraction operation is supplied, so a negative value is added instead.]
10. Compare the count to zero to see if all bytes have been read from the device; if the count is not zero, then skip the next transaction and continue. UDI_PIO_CSKIP is the conditional skip PIO operation. The comparison specifed by the ``condition code" operand (**UDI_PIO_NZ**) is carried out on the one−byte (**UDI_PIO_1BYTE**) value in **UDI_PIO_R2** (R2). If the result of the comparison is **TRUE**, then the next transaction is skipped. **UDI_PIO_NZ** equates to the expression ***reg*!=0**, where ***reg*** is one of R0 through R7. In this case, as long as R2 does not equal zero, the next transaction is skipped.
11. This transaction gets executed only when the count in R2 reaches zero, meaning all bytes have been read from the CMOS RAM device. UDI_PIO_END terminates processing of the tranascation list and sets the result code (an arbitrary 8−bit code that is defined by the driver) in the **udi_pio_trans_call_t** callback to the low byte of the operand value. In this case, the result code is a two−byte zero.
12. If the count in R2 has not yet reached zero, then increment the address (R1) to read the address of the next byte in CMOS RAM. This is done with the previously described **UDI_PIO_ADD_IMM** operation to add 1 to the value in R1.
13. Increment the buffer offset (R0) to write the next byte from CMOS RAM. This is done with the previously described **UDI_PIO_ADD_IMM** operation to add 1 to the value in R0.
14. Branch bake to the **UDI_PIO_LABEL** transaction with the specified operand (in this case, 1). This starts the process of reading the next byte from CMOS RAM. Note that the last transaction in a transaction list array **must** be either a **UDI_PIO_END** or a **UDI_PIO_BRANCH** transaction.

The transaction list for the cmos_trans_write[] array ia exactly the same as the read array described above, with the exception of transaction 8. In the write operation, a byte is read from the buffer into the CMOS RAM memory at the specified address. **UDI_PIO_OUT** sets the location on the PIO device at the offset specified by the operand (**CMOS_DATA**, or the CMOS RAM data register) to the one−byte value in the buffer at the 32−bit buffer offset in R0.

### Further information on transaction lists

Transaction list usage is discussed further on, in the description of the CMOS RAM driver subroutines. More complex transaction lists will be discussed in the sections of this document on the SCSI HBA and NIC sample drivers.

In the UDI Specifications, see in the Physical I/O Specification for a complete list of operations, operation classes, addressing modes, and transaction list syntax.

## Channel operations indexes

An operations index (`ops_idx`) is a non−zero integer, assigned by the driver to uniquely identify a set of entry−point related properties defined in a **udi_ops_init_t** structure. An array of these structures is used to define all the channel operations for a source module, and the `ops_idx`) member of each is used as a unique identifier.

***cmos_udi.c* sample code (cont.)**

```
/*
 * Ops indexes for each of the entry point ops vectors used.
 */
#define GIO_OPS_IDX          1
#define BUS_DEVICE_OPS_IDX   2
#if DO_INTERRUPTS
#define INTR_HANDLER_OPS_IDX 3
#endif
```

*pseudod.c* sample code (cont.)

```
#define PSEUDO_GIO_INTERFACE 1       /* Ops index for GIO entry points */
#define      PSEUDO_BUS_INTERFACE    2       /* Ops index for bus entry points */
```

These indexes are used in the ``Channel operations initialization'' section.

# Management metalanguage operations vector

All device configuration, driver instantiation, and initial channel creation is driven and controlled by the UDI ``Management Agent'' (MA), which is always present in any UDI environment implementation. All UDI drivers must provide entry points (really, callback functions) for the MA into the driver, in a management metalanguage operations vector.

The Management Metalanguage defines the communication between a driver instance and the MA and is used for the ``management channel'', which is a channel that is always present between the MA and the driver's primary region.

The MA generates a Management Metalanguage request to a driver over the management channel when it needs that driver to perform some management−related function. The driver must define the required callback functions to send data back to the MA. This is a one−way realtionship, meaning that the driver never generates any requests over this channel, it only responds to requests from the MA.

The **udi_mgmt_ops_t** structure is the management operations vector.

```
typedef struct {
     udi_usage_ind_op_t              *usage_ind_op ;
     udi_enumerate_req_op_t          *enumerate_req_op ;
     udi_devmgmt_req_op_t            *devmgmt_req_op ;
     udi_final_cleanup_req_op_t      *final_cleanup_req_op ;
} udi_mgmt_ops_t ;
```

This structure contains pointers to routines required to answer queries from the MA as it sets up a primary region.

For an overview of the Management Metalanguage and the role of the MA in the UDI environment, see Management Metalanguage in the *UDI Core Specification*. The exact sequence of events that occurs when a region is intialized is documented in the Core Specification, under Driver Instantiation.

Here we will summarize what each required routine must do, and the **udi_mgmt_ops_t** structures provided by the sample drivers. Shown below are the management operations vectors for the **udi_cmos** and **pseudod** drivers.

***cmos_udi.c* sample code (cont.)**

```
   /*
    * Driver entry points for the Management Metalanguage.
    */
   static udi_devmgmt_req_op_t cmos_devmgmt_req;
   static udi_final_cleanup_req_op_t cmos_final_cleanup_req;


static udi_mgmt_ops_t cmos_mgmt_ops = { udi_static_usage, udi_enumerate_no_children,
cmos_devmgmt_req, cmos_final_cleanup_req };
```

*pseudod.c* sample code (cont.)

```
   /*
    * Management Metalanguage/Pseudo Driver entry points
    */


static udi_devmgmt_req_op_t pseudo_devmgmt_req; static udi_final_cleanup_req_op_t
pseudo_final_cleanup_req;

static udi_mgmt_ops_t pseudo_mgmt_ops = { udi_static_usage, udi_enumerate_no_children,
pseudo_devmgmt_req, pseudo_final_cleanup_req };
```

The `usage_ind_op` element either contains a driver−specific function name, or the proxy function **udi_static_usage**. The driver−specific function name starts with a unique identifier usually associated with the driver, and followed by **_usage_ind**. It's syntax follows the **udi_usage_ind**(3udi) management metalanguage function, and it specifies how the driver adjusts resource utilization levels and whether it supports tracing. Drivers like the **udi_cmos** and **psedudod** drivers that do not adjust their resource utilization levels and do not support tracing specify **udi_static_usage** as the entry point for this operation.

The `enumerate_req_op` element either contains a driver−specific function name, or the proxy function **udi_enumerate_no_children**. The driver−specific function name starts with a unique identifier usually associated with the driver, and followed by **_enumerate_req**. It's syntax follows the **udi_enumerate_req**(3udi) management metalanguage function, and it specifies the presence, number, and initial instance attributes of any child devices (actual or pseudo) associated with the current driver instance. The enumeration request and acknowledgement operation, along with associated data objects is described in the section Enumeration Operations in the Core Specification. Drivers like the **udi_cmos** and **psedudod** drivers that do not need to enumerate any children can use the UDI−environment proxy function **udi_enumerate_no_children** as the entry point for this operation.

The `devmgmt_req_op` and `final_cleanup_req_op` elements are required to contain the name of a driver−specific function. These functions will respond to MA requests to, respectively, manage I/O transfers within a driver instance during hot plug operations, and perform final cleanup operations when an instance is being unloaded by the MA. The **cmos_devmgmt_req** and **pseudo_devmgmt_req** functions perform the acknowlegdement operations that the driver will use to respond a **udi_devmgmt_req** call from the MA. Similarly, the **cmos_final_cleanup_req** and **pseudo_final_cleanup_req** functions respond to a **udi_final_cleanup_req** call from the MA.

These functions will be discussed further under ``udi_cmos child channel operations".

For an overview of UDI function naming conventions and calling sequences, see Calling Sequence and Naming Conventions in the Core Specification.

# Bus bridge metalanguage entry points

The Bus Bridge metalanguage allows a device driver to communicate with the driver for its parent bus bridge. The device may be a leaf device or another bridge.

A driver region may use the Bus Bridge metalanguage in any of four roles. For each role there's a corresponding channel ops vector, which is registered via an associated **udi_ops_init_t** structure. The Bus Bridge metalanguage roles are:

***bridge***
: supports binding/unbinding and interrupt registration operations invoked on a parent bridge object by a child device driver; uses the **udi_bus_bridge_ops_t** operations vector

***device***
: supports binding/unbinding and interrupt registration operations invoked on a child device object by a parent bridge driver; uses the **udi_bus_device_ops_t** operations vector

***interrupt dispatcher***
: supports interrupt acknowledgment operations invoked on a parent bridge object by a child device driver; uses the **udi_intr_dispatcher_ops_t** operations vector

***interrupt handler***
: supports interrupt event operations invoked on a child device object by a parent bridge driver; uses the **udi_intr_handler_ops_t** operations vector

Both the **udi_cmos** and **pseudod** sample drivers use the device role, and provide the **udi_bus_device_ops_t** vectors shown below.

**_cmos_udi.c_ sample code (cont.)**

```
   /*
    * Driver "bottom-side" entry points for the Bus Bridge Metalanguage.
    */


static udi_channel_event_ind_op_t cmos_parent_channel_event; static udi_bus_bind_ack_op_t
cmos_bus_bind_ack; static udi_bus_unbind_ack_op_t cmos_bus_unbind_ack; #if DO_INTERRUPTS static
udi_intr_attach_ack_op_t cmos_intr_attach_ack; static udi_intr_detach_ack_op_t cmos_intr_detach_ack;
#endif

static udi_bus_device_ops_t cmos_bus_device_ops = { cmos_parent_channel_event, cmos_bus_bind_ack,
cmos_bus_unbind_ack, #if DO_INTERRUPTS cmos_intr_attach_ack, cmos_intr_detach_ack #else
udi_intr_attach_ack_unused, udi_intr_detach_ack_unused #endif };

#if DO_INTERRUPTS static udi_channel_event_ind_op_t cmos_intr_channel_event; static
udi_intr_event_ind_op_t cmos_intr_event_ind;

static udi_intr_handler_ops_t cmos_intr_handler_ops = { cmos_intr_channel_event, cmos_intr_event_ind };
#endif
```

*pseudod.c* sample code (cont.)

```
   /*
    * Driver "bottom-side" entry points for the Bus Bridge Metalanguage
    */
   static udi_channel_event_ind_op_t pseudo_bus_channel_event;
   static udi_bus_bind_ack_op_t pseudo_bus_bind_ack;
   static udi_bus_unbind_ack_op_t pseudo_bus_unbind_ack;


static udi_bus_device_ops_t pseudo_bus_device_ops = { pseudo_bus_channel_event, pseudo_bus_bind_ack,
pseudo_bus_unbind_ack, udi_intr_attach_ack_unused, udi_intr_detach_ack_unused };
```

As child devices of the parent bus bridge, the sample drivers must manage bus bindings with the bridge driver and handle interrupts coming from the bridge driver.

Each of the entries in the cmos_bus_device_ops and pseudo_bus_device_ops arrays is a function that handles one of these corresponding service calls from the UDI environment (in this case, the bus bridge driver):

| | |
|---|---|
| **udi_channel_event_ind** | Used by the environment to signal that a generic event has occurred on the other end of the channel. The type of event that has occurred, and additional parameters for the event, are contained in a **udi_channel_event_cb_t** control block passed as part of the service call. The sample drivers provide driver−side entry points **cmos_parent_channel_event** and **pseudo_bus_channel_event** for this call. |
| **udi_bus_bind_ack** | Used by a bridge driver to acknowledge binding with a child device driver (or failure to do so, as indicated by status), as originally requested by a **udi_bus_bind_req** operation from the driver. The sample drivers provide the **cmos_bus_bind_ack** and **pseudo_bus_bind_ack** entry points. |
| **udi_bus_unbind_ack** | Used by a bridge driver to acknowledge unbinding with a child device driver as originally requested by a **udi_bus_unbind_req** operation from the driver. The sample drivers provide the **cmos_bus_unbind_ack** and **pseudo_bus_unbind_ack** entry points. |
| **udi_intr_attach_ack** | Used by an interrupt dispatcher driver to acknowledge attachment of an interrupt handler (or failure to do so, as indicated by status), as requested by a **udi_intr_attach_req** operation. The sample drivers do not use the **udi_intr_attach_req** operation, and so never expect an acknowledgement. They use the environment−provided**udi_intr_attach_ack_unused** proxy function instead of defining an entry point. |
| **udi_intr_detach_ack** | Used by an interrupt dispatcher driver to acknowledge detachment of an interrupt handler, as requested by a **udi_intr_detach_req** operation. The sample drivers do not use the **udi_intr_detach_req** operation, and so never expect an acknowledgement. They use the environment−provided **udi_intr_detach_ack_unused** proxy function instead of defining an entry point. |

The specifics of the driver−side functions are discussed in the section ``Channel operations".

The Bus Bridge environment service calls are described in detail in the Bus Bridge Metalanguage section of the *UDI Physical I/O Specification*. Also see **udi_channel_event_ind**(3udi).

# GIO metalanguage entry points

The Generic I/O (GIO) metalanguage is a generic pass−through metalanguage and can be used as a "top−side" metalanguage for drivers when a more specific metalanguage does not exist. The GIO metalanguage can also be used as an internal metalanguage between a multi−region driver's primary and secondary regions or between secondary regions.

The GIO metalanguage provides the ability to:

- send and receive data buffers to/from the driver
- send control operations to the driver
- timeout outstanding data and control operations
- send event notifications from the driver "upward"

There are two roles to the GIO Metalanguage: the ``client" and the ``provider". Each region in a driver module can assume either of these roles independently of the other regions. When this metalanguage is used between drivers, the client is always a child of the provider. When used as an internal metalanguage, the client and provider are both in the same driver, but in different regions. Only clients can issue a bind request, and only providers can respond to a bind request.

GIO metalanguage entry points are defined in much the same way as Management metalanguage operations vectors (see ``Management metalanguage operations vector").

The structure used for client−side operations is **udi_gio_client_ops_t**:

```
typedef struct {
     udi_channel_event_ind_op_t      *channel_event_ind_op ;
     udi_gio_bind_ack_op_t           *gio_bind_ack_op ;
     udi_gio_unbind_ack_op_t         *gio_unbind_ack_op ;
     udi_gio_xfer_ack_op_t           *gio_xfer_ack_op ;
     udi_gio_xfer_nak_op_t           *gio_xfer_nak_op ;
     udi_gio_event_ind_op_t          *gio_event_ind_op ;
} udi_gio_client_ops_t ;
```

The structure used for provider−side operations is **udi_gio_provider_ops_t**:

```
typedef struct {
     udi_channel_event_ind_op_t      *channel_event_ind_op ;
     udi_gio_bind_req_op_t           *gio_bind_req_op ;
     udi_gio_unbind_req_op_t         *gio_unbind_req_op ;
     udi_gio_xfer_req_op_t           *gio_xfer_req_op ;
     udi_gio_event_res_op_t          *gio_event_res_op ;
} udi_gio_provider_ops_t ;
```

The **udi_cmos** and **pseudod** drivers both act solely as GIO providers. The structure names they define mirror the naming of the environment functions (as with the Management metalanguage definitions):

***cmos_udi.c* sample code (cont.)**

```
/*
 * Driver "top−side" entry points for the GIO Metalanguage.
 */


static udi_channel_event_ind_op_t cmos_child_channel_event; static udi_gio_bind_req_op_t
cmos_gio_bind_req; static udi_gio_unbind_req_op_t cmos_gio_unbind_req; static udi_gio_xfer_req_op_t
cmos_gio_xfer_req;
```

```
static udi_gio_provider_ops_t cmos_gio_provider_ops = { cmos_child_channel_event, cmos_gio_bind_req, cmos_gio_unbind_req, cmos_gio_xfer_req, udi_gio_event_res_unused };
```

*pseudod.c* sample code (cont.)

```
/*
 * GIO/Pseudo Driver entry points
 */


static udi_channel_event_ind_op_t pseudo_gio_channel_event; static udi_gio_bind_req_op_t pseudo_gio_bind_req; static udi_gio_unbind_req_op_t pseudo_gio_unbind_req; static udi_gio_xfer_req_op_t pseudo_gio_xfer_req;

static udi_gio_provider_ops_t pseudo_gio_provider_ops = { pseudo_gio_channel_event, pseudo_gio_bind_req, pseudo_gio_unbind_req, pseudo_gio_xfer_req, udi_gio_event_res_unused };
```

The **cmos_*** and **pseudo_*** functions defined in the provider operations arrays shown above provide the driver−side entry points for handling the corresponding service calls coming from the environment.

For example, the **cmos_child_channel_event** and **pseudo_gio_channel_event** functions (defined later in the source code), are the driver−side entry points for the **udi_gio_channel_event_ind** service call, used by the environment to signal that a generic event has occurred on the other end of the channel. The driver side functions respond to the event appropriate and perform other necessary operations as required by the particular event.

The table below lists the environment side service calls to which the driver−side functions defined in the provider operations array respond.

| | |
|---|---|
| **udi_channel_event_ind** | Used by the environment to signal that a generic event has occurred on the other end of the channel. The type of event that has occurred, and additional parameters for the event, are contained in a **udi_channel_event_cb_t** control block passed as part of the service call. Both of the sample drivers are GIO providers, and so provide driver−side entry points **cmos_child_channel_event** and **pseudo_gio_channel_event**. |
| **udi_gio_bind_req** | A GIO client uses this operation to request binding to a GIO provider and send a GIO bind control block (**udi_gio_bind_cb_t**) to the provider. The GIO provider function must send a corresponding **udi_gio_bind_ack** to the client before the client sends any further operations on the bind channel. The sample drivers provide **cmos_gio_bind_req** and **pseudo_gio_bind_req** entry points for this operation. |
| **udi_gio_unbind_req** | A GIO client uses this operation to request unbinding from a GIO provider. and send a GIO bind control block (**udi_gio_bind_cb_t**) to the provider. The GIO provider function must send a corresponding **udi_gio_unbind_req** back to the client to complete the operation. The sample drivers provide **cmos_gio_unbind_req** and **pseudo_gio_unbind_req** entry points for this operation. |
| **udi_gio_xfer_req** | A GIO client uses this operation to send a data transfer reuest to a GIO provider. A GIO transfer control block (**udi_gio_xfer_cb_t**) specifies the semantics and parameters for the request. The sample drivers are GIO providers, and so provide **cmos_gio_xfer_req** and **pseudo_gio_xfer_req** entry points for this operation. |
| **udi_gio_event_res** | A GIO client uses this operation to acknowledge an event indication from a GIO provider, as delivered by a **udi_gio_event_ind** operation. The control block used |

| | |
|---|---|
| | must be the same **udi_gio_event_cb_t** control block passed to the driver in the corresponding **udi_gio_event_ind** operation. A proxy function, **udi_gio_event_res_unused** may be used as a GIO provider's **udi_gio_event_res** entry point if the provider never sends any event indications (and therefore expects no responses); this is the case with the sample drivers. |

The specifics of the driver−side functions are discussed in the section ``Channel operations".

The GIO environment service calls are described in detail in the section of the Core Specification. Also see **udi_channel_event_ind**(3udi).

# Primary region initialization

Every driver must have exactly one primary region whose interaction with the UDI environment's Management Agent (MA) must be described in a **udi_primary_init_t** array. When the Management Agent creates a driver instance, it automatically creates a primary region according to the parameters provided in the **udi_primary_init_t** structure. ``Primary region initialization"). A management channel will also be created and anchored to this region. The channel context for this channel will be set to point to the region's region data area (see ``Region data definitions").

***cmos_udi.c* sample code (cont.)**

```
/*
 * ------------------------------------------------------------------------
 * Management operations init section:
 * ------------------------------------------------------------------------
 */
static udi_primary_init_t udi_cmos_primary_init_info = {
     &cmos_mgmt_ops,
     cmos_default_op_flags,  /* op_flags */
     0,                      /* mgmt_scratch_size */
     0,                      /* enumerate attr list length */
     sizeof(cmos_region_data_t), /* rdata size */
     0,                      /* no children, so no enumeration */
     0,              /* buf path*/
};
```

*pseudod.c* sample code (cont.)

```
static udi_primary_init_t pseudo_primary_init = {
     &pseudo_mgmt_ops,
     pseudo_default_op_flags,
     0,                              /* mgmt scratch */
     0,                              /* enumerate no children */
     sizeof (pseudo_region_data_t),
     0,                              /*child_data_size */
     0,              /*buf path */
};
```

The **cmos_mgmt_ops** and **pseudo_mgmt_ops** arrays were defined in the section ``Management metalanguage operations vector" and hold the Management metalanguage operations vectors. This tells the UDI environment about all the channel operations defined for the driver region. The `cmos_default_op_flags` and `pseudo_default_op_flags` elements are defined as follows in the driver code:

```
static udi_ubit8_t cmos_default_op_flags[] = {
     0, 0, 0, 0, 0
```

```
};
```

static udi_ubit8_t pseudo_default_op_flags[] = { 0, 0, 0, 0, 0 }; These are arrays of flag values with a one−for−one correspondence between entries in the array and entries in the `cmos_mgmt_ops` and `pseudo_mgmt_ops` arrays. This array may be used to indicate characteristics of the implementation of the corresponding operation to the environment. Neither of the sample drivers requires any special operation flags, so the arrays are all zeros. See **udi_primary_init_t**(3udi) for flag values and usage.

The fifth element of each array holds the size, in bytes, of the region data area to be allocated for the primary region of each driver instance. These were defined in ``Region data definitions".

See **udi_primary_init_t**(3udi) for an explanation of the other elements in this structure.

# Channel operations initialization

Communication channels between regions are initialized with an array of **udi_ops_init_t** structures, one array element per channel. The **udi_ops_init_t** structure contains information the environment needs to create channel endpoints for a particular type of operations vector and control block usage. Each channel is initialized according to the specifications in the array element.

Once this array is initialized in the driver code, it's used in an **init_info_t** structure, which the UDI environment uses to initialize a driver instance (see ``Driver initialization structure").

The **udi_ops_init_t** structure has these fields:

```
typedef void udi_op_t (void);
```

typedef udi_op_t const udi_ops_vector_t ;

typedef struct { udi_index_t ops_idx ; udi_index_t meta_idx ; udi_index_t meta_ops_num ; udi_size_t chan_context_size ; udi_ops_vector_t *ops_vector ; } udi_ops_init_t ;

*`ops_idx`*
: a non−zero channel ops index number, assigned by the driver to uniquely identify this set of entry−point related properties for use in other initialization structures and service calls, or zero to terminate the **ops_init_list** list to which this structure belongs. These are normally defined within the C source code of the driver module, as shown for the sample drivers in ``Channel operations indexes".

*`meta_idx`*
: a non−zero metalanguage index number, assigned by the driver to uniquely identify a set of metalanguage related properties for the channel. These are defined for the sample drivers in the build instructions section of each driver's *udiprops.txt* file (see ``Build instructions").

*`meta_ops_num`*
: a metalanguage−specific number, defined in the UDI specification for the metalanguage named by `meta_idx`, that uniquely identifies an operations vector type defined by that metalanguage.

*`chan_context_size`*

the size, in bytes, of a context area that will be automatically allocated, if non−zero, whenever the specified `ops_idx` is used to bind a child or parent instance to a driver instance or to bind a secondary region to the primary region for this driver. If non−zero, the value must be at least **sizeof(udi_chan_context_t)** (see **udi_chan_context_t**(3udi)) and must not exceed **UDI_MIN_ALLOC_LIMIT** (see the discussion of the **init_context** structure under ``Region data definitions").

*`ops_vector`*

is a pointer to the metalanguage−specific ***meta_role*__ops_t** channel operations vector structure (see Calling Sequence and Naming Conventions in the Core Specification for an overview of UDI function naming conventions). This structure contains pointers to the various entry point routines for this type of channel and must be a constant initialized variable. The address of this structure must be cast to **(udi_ops_vector_t *)** in order to be used as a value for `ops_vector`.

The **udi_cmos** and **pseudod** drivers define these channel operation initialization arrays:

***cmos_udi.c* sample code (cont.)**

```
/*
 * ------------------------------------------------------------------------
 * Meta operations init section:
 * ------------------------------------------------------------------------
 */
static udi_ops_init_t udi_cmos_ops_init_list[] = {
     {
             GIO_OPS_IDX,
             CMOS_GIO_META,          /* meta index
                                      * [from udiprops.txt] */
             UDI_GIO_PROVIDER_OPS_NUM,
             0,                      /* no channel context */
             (udi_ops_vector_t *)&cmos_gio_provider_ops,
             cmos_default_op_flags   /* op_flags */
     },
     {
             BUS_DEVICE_OPS_IDX,
             CMOS_BRIDGE_META,       /* meta index
                                      * [from udiprops.txt] */
             UDI_BUS_DEVICE_OPS_NUM,
             0,                      /* no channel context */
             (udi_ops_vector_t *)&cmos_bus_device_ops,
             cmos_default_op_flags   /* op_flags */
     },
#if DO_INTERRUPTS
     {
             INTR_HANDLER_OPS_IDX,
             CMOS_BRIDGE_META,       /* meta index
                                      * [from udiprops.txt] */
             UDI_INTR_HANDLER_OPS_NUM,
             0,                      /* no channel context */
             (udi_ops_vector_t *)&cmos_intr_handler_ops,
             cmos_default_op_flags   /* op_flags */
     },
#endif /* DO_INTERRUPTS */
     {
             0       /* Terminator */
     }
};
```

*pseudod.c* sample code (cont.)

```
static udi_ops_init_t pseudo_ops_init_list[] = {
     {
      PSEUDO_GIO_INTERFACE,
      PSEUDO_GIO_META,               /* meta index */
      UDI_GIO_PROVIDER_OPS_NUM,      /* meta ops num */
      0,                             /* chan context size */
      (udi_ops_vector_t *) & pseudo_gio_provider_ops,        /* ops vector */
      pseudo_default_op_flags
      },
     {
      PSEUDO_BUS_INTERFACE,
      PSEUDO_BUS_META,               /* Bus meta index [from udiprops.txt] */
      UDI_BUS_DEVICE_OPS_NUM,
      0,                             /* channel context size */
      (udi_ops_vector_t *) & pseudo_bus_device_ops,
      pseudo_default_op_flags
      },
     {
      0}
};
```

The two sample drivers define similar operation vectors for the GIO provider operations defined in ``GIO metalanguage entry points" and for the bus device role operations defined in ``Bus bridge metalanguage entry points". The **udi_cmos** code also provide an optional set of interrupt handling operations. See the element descriptions above.

The `chan_context_size` is ``0" for each vector, as required by the *UDI Core Specification*. A lone `ops_idx` of ``0" terminates each list.

For more on how these structures are used during driver initialization, see ``Driver initialization structure".

# Channel operation data layouts

Most channel operations use strongly typed functions to exchange data, and therefore the UDI environment on the target system knows the data format for the parameters used.

Drivers that define channel operations that use weakly typed, structured, data parameters must declare the format of the data to the UDI environment through an array of **udi_layout_t** structures. For example, a function parameter might be a pointer to memory. If that memory chunk contains anything more complex than an array of bytes, then that structure must be declared in the driver code.

While the **udi_cmos** driver does not use any structured, weakly typed data, the **pseudod** code contains one such declaration:

```
static udi_layout_t xfer_layout[] = { UDI_DL_UBIT32_T, UDI_DL_UBIT32_T,
     UDI_DL_UBIT16_T, UDI_DL_END
};
```

***pseudod.c* sample code (cont.)**

This layout indicates that the memory chunk passed as part of the GIO transfer control block inline data will be divided into segments of four data values. The constants used in the layout specification are defined in the Core Specification (see **udi_layout_t**(3udi)).

This definition is used in the initialization of the driver's control blocks (see ``Control block indexes and structures").

For the functions that use this layout, a description of how the memory that holds the data is allocated, etc., see the section ``udi_cmos child channel operations".

# Driver initialization structure

The **udi_init_info**, or driver initialization, structure is used by the UDI environment to link a newly loaded driver into the kernel, to launch new instantiations of a loaded driver, and to manage the regions, operations data, and other aspects of already instantiated drivers.

The **udi_init_info** structure contains pointers to constant information the environment needs to initialize a driver. Each driver module must include a constant initialized structure of type **udi_init_t** named **udi_init_info**.

Exactly one module in a multi−module driver must be the primary module, identified in the driver's *udiprops.txt* file as the module with a **region** declaration for region index zero, which identifies the primary region.

The **udi_cmos** and **pseudod** sample drivers provide very similar **udi_init_info** structures, though the underlying data is quite different.

The following figure shows the **udi_init_info** structure as defined in the *Core Specification*, along with the sample driver definitions.

**udi_init_t from the *Core Specification*;**

```
   typedef struct {
        const udi_primary_init_t        *primary_init_info ;
        const udi_secondary_init_t      *secondary_init_list ;
        const udi_ops_init_t            *ops_init_list ;
        const udi_cb_init_t             *cb_init_list ;
        const udi_gcb_init_t            *gcb_init_list ;
        const udi_cb_select_t           *cb_select_list ;
   } udi_init_t ;
   T}
```

*cmos_udi.c* sample code (cont.); T{

```
   udi_init_t udi_init_info = {
        &udi_cmos_primary_init_info,
        NULL,                           /* secondary_init_list */
        udi_cmos_ops_init_list,
        udi_cmos_cb_init_list,
        NULL,                           /* gcb_init_list */
        NULL                            /* cb_select_list */
   };
```

*pseudod.c* sample code (cont.);

```
   udi_init_t udi_init_info = {
        &pseudo_primary_init,
        NULL,                           /* Secondary init list */
        pseudo_ops_init_list,
        pseudo_cb_init_list,
        NULL,                           /* gcb init list */
```

```
        NULL                            /* cb select list */
    };
```

The elements of the **udi_init_info** structure include by reference the region, operation, control block, and other data defined previously in the driver code, as shown in the descriptions below.

***primary_init_info***

The primary module of all UDI drivers must have a non−**NULL** **primary_init_info**. In secondary module code, which also must provide a **udi_init_info** structure, this element is set to **NULL**. The **primary_init_info** element is a pointer to a structure containing information about the driver's primary region. The primary region structures for the **udi_cmos** and **pseudod** drivers were defined earlier in the driver code; see ``Primary region initialization'' for a description.

***secondary_init_list***

If the primary module also manages some secondary regions, the module must also include a non−empty **secondary_init_list**, a list of structures containing information about each secondary region implemented in the module. The list is terminated with an entry containing a zero `region_idx`, and a **NULL** pointer is treated the same as a list with only one entry containing a zero `region_idx`. Both the **udi_cmos** and **pseudod** drivers define a single module with a single region, and so this element is **NULL** for both drivers. The subject of secondary regions is treated in the description of the sample NIC driver; see ``Network Driver Coding Specifics''.

***ops_init_list***

All UDI drivers must include a pointer to an **ops_init_list** array that must contain at least one member with a non−zero `ops_idx`. This list must include at least one entry for each metalanguage used in this module. **ops_init_list** is a pointer to a list of structures containing information about channel operations usage for each operations vector implemented in this module. The list is terminated with an entry containing a zero `ops_idx`. The channel operations initialization lists for the **udi_cmos** and **pseudod** drivers were defined earlier in the driver code; see ``Channel operations initialization'' for a description.

`cb_init_list` is a pointer to a list of structures containing information about each control block type used by this module. The list is terminated with an entry containing a zero `cb_idx`, and **NULL** pointer is treated the same as a list with only one entry containing a zero `cb_idx`. The control block initialization lists for the **udi_cmos** and **pseudod** drivers were defined earlier in the driver code; see ``Control block indexes and structures'' for a description.

`gcb_init_list` is a pointer to a list of structures containing information about generic control block usage in this module, if any. Generally, these are control blocks used only for asynchronous environment service calls, such as **udi_mem_alloc**(3udi). The list is terminated with an entry containing a zero `cb_idx`, and a **NULL** pointer is treated the same as a list with only one entry containing a zero `cb_idx`. The sample UDI drivers do not use `gcb_init_list`; see **udi_init_info**(3udi) for more information.

`cb_select_list` is a pointer to a list of structures containing information about special overrides for scratch requirements when using specific control blocks with specific operations vectors. The list is terminated with an entry containing a zero `cb_idx`, and a **NULL** pointer is treated the same as a list with only one entry containing a zero `cb_idx`. The sample UDI drivers do not use `cb_select_list`; see **udi_cb_init_t**(3udi) and **udi_cb_select_t**(3udi) for more information.

# Channel operations

At this point in the driver code, we've defined all the necessary elements needed by the various channel operations that will do the work of the driver and interface otherwise with the UDI environment. Now we define the channel operations.

Both the **udi_cmos** and **pseudod** sample drivers define three groups of channel operations:

- parent operations for the channel between the driver and the bus bridge; these operations are entry points into the driver code for the bus bridge, and their associated driver routines (see ``udi_cmos parent channel operations'' and ``pseudod parent channel operations'')
- child operations for the channel between the driver and the GIO external mapper; these operations (and their associated routines) are executed by the driver in response to UDI service calls from the mapper (which is a GIO client for this channel) (see ``udi_cmos child channel operations'' and ``pseudod child channel operations'')
- management channel operations for the UDI Management Agent (this channel is present in all UDI drivers) (see ``udi_cmos Management Agent channel operations'' and ``pseudod Management Agent channel operations'')

The remainder of the code for both the **udi_cmos** and **pseudod** sample drivers defines the operations for these channels.

See Function Call Classifications and Calling Sequence and Naming Conventions in the *UDI Core Specification* for a review of driver function types, channel operations, and callback function conventions.

## udi_cmos parent channel operations

First, the udi_cmos code defines the channel operations for the channel between the primary region and the bus bridge. This channel and its operations vectors were defined previously under ``Bus bridge metalanguage entry points''.

```
static void
cmos_parent_channel_event(udi_channel_event_cb_t *channel_event_cb)
{
     udi_bus_bind_cb_t *bus_bind_cb;
     switch (channel_event_cb->event) {
     case UDI_CHANNEL_BOUND:
```

bus_bind_cb = UDI_MCB(channel_event_cb−> params.parent_bound.bind_cb, udi_bus_bind_cb_t);

/* Keep a link back to the channel event CB for the ack. */ UDI_GCB(bus_bind_cb)−>initiator_context = channel_event_cb;

/* Bind to the parent bus bridge driver. */ udi_bus_bind_req(bus_bind_cb); break; default: udi_channel_event_complete(channel_event_cb, UDI_OK); } }

The **cmos_parent_channel_event** operation responds to an event at the other end of the bus bridge channel; the bus bridge mapper generates a call to the **cmos_parent_channel_event** entry point, with an appropriately filled **udi_channel_event_cb_t** structure defining the event.

If the `channel_event_cb->event` element is UDI_CHANNEL_BOUND, The routine first converts the **channel_event_cb->params.bound.bind_cb** structure to the type required for the **udi_bus_bind_req** call later in the routine. It does this using the **UDI_MCB** macro call defined in Control Block Management in the *Core Specification*.

A pointer to the **channel_event_cb** is saved so it can be accessed by the **cmos_bus_bind_ack** operation. Then, a binding back to the bus bridge is requested using **udi_bus_bind_req** and the converted **bus_bind_cb** control block (see above). See **udi_bus_bind_req**(3udi), and **udi_bus_bind_cb_t**(3udi).

Otherwise, if `channel_event_cb->event` is not **UDI_CHANNEL_BOUND**, then a call to **udi_channel_event_complete** indicates to the bridge mapper that the event was processed successfully (see Inter−module Communication in the *Core Specification* for the definition of **udi_channel_event_complete** and **udi_channel_event_cb_t**).

The bridge mapper responds to the **udi_bus_bind_req** call with a call to the **cmos_bus_bind_ack** channel operation, listed below.

```
static void cmos_bus_bind_ack_1(udi_cb_t *, udi_pio_handle_t);
```

static void cmos_bus_bind_ack( udi_bus_bind_cb_t *bus_bind_cb, udi_dma_constraints_t constraints, udi_ubit8_t preferred_endianness, udi_status_t status) { cmos_region_data_t *rdata = bus_bind_cb−>gcb.context; udi_channel_event_cb_t *channel_event_cb = bus_bind_cb−>gcb.initiator_context;

/* * Don't need to do anything with preferred_endianness, since * our device doesn't do DMA. Even if it did, * preferred_endianness is only used with bi−endianness devices * that can change endianness. */ udi_dma_constraints_free(constraints);

/* * Save the bus bind control block for unbinding later. */ rdata−>bus_bind_cb = bus_bind_cb;

if (status != UDI_OK) { udi_channel_event_complete( channel_event_cb, UDI_STAT_CANNOT_BIND); return; }

/* * Now we have access to our hardware. Set up the PIO mappings * we'll need later. */ udi_pio_map(cmos_bus_bind_ack_1, UDI_GCB(bus_bind_cb), CMOS_REGSET, CMOS_BASE, CMOS_LENGTH, cmos_trans_read, sizeof cmos_trans_read / sizeof(udi_pio_trans_t), UDI_PIO_NEVERSWAP|UDI_PIO_STRICTORDER, CMOS_PACE, 0); }

static void cmos_bus_bind_ack_2(udi_cb_t *, udi_pio_handle_t);

static void cmos_bus_bind_ack_1( udi_cb_t *gcb, udi_pio_handle_t new_pio_handle) { cmos_region_data_t *rdata = gcb−>context;

/* Save the PIO handle for later use. */ rdata−>trans_read = new_pio_handle;

udi_pio_map(cmos_bus_bind_ack_2, gcb, CMOS_REGSET, CMOS_BASE, CMOS_LENGTH, cmos_trans_write, sizeof cmos_trans_write / sizeof(udi_pio_trans_t), UDI_PIO_NEVERSWAP|UDI_PIO_STRICTORDER, CMOS_PACE, 0); }

```
static void cmos_bus_bind_ack_2( udi_cb_t *gcb, udi_pio_handle_t new_pio_handle) { cmos_region_data_t
*rdata = gcb−>context; udi_channel_event_cb_t *channel_event_cb = gcb−>initiator_context;

/* Save the PIO handle for later use. */ rdata−>trans_write = new_pio_handle;

#if DO_INTERRUPTS /* Attach interrupts here... */ #endif

/* Let the MA know we've completed binding. */ udi_channel_event_complete(channel_event_cb, UDI_OK);
}
```

The **cmos_bus_bind_ack** channel operation is called by the bridge mapper (in response to a **udi_bus_bind_req** from the driver) to send the status of the bind request back to the driver. It is called with The **cmos_bus_bind_ack** routine needs to take appropriate actions based on the status given by the mapper.

The bridge mapper calls **cmos_bus_bind_ack** with the same bus_bind_cb control block passed to it via **cmos_bus_bind_req** from the driver, a DMA constraints vector, an indication of the most effective endianness to use for the bus, and the status of the previously requested bind operation.

Since this driver does not do any DMA, the DMA constraints are freed via a call to **udi_dma_constrins_free**, and the preferred_endianness argument is simply ignored.

The **bus_bind_cb** control block passed to the driver is first used to update the beginning of the region data structure (**rdata**) to the `context` element passed as part of **bus_bind_cb** to the driver. This is the context from the driver's earlier call to **cmos_bus_bind_req**, and was passed back to the driver as part of the **cmos_bus_bind_ack**.

The **cmos_udi** driver's region data structure **cmos_region_data_t** was defined earlier in the driver code; see ``Region data definitions".

Similarly, the context of the channel when the event was generated by the bridge mapper, the `initiator_context`, is saved at the beginning of a **channel_event_cb** structure. This will be used later in the event the bind was unsuccessful.

The entire **bus_bind_cb** control block is then saved in the **rdata** region data structure for later use when unbinding the channel.

Next, the `status` argument passed in the call to **cmos_bus_bind_ack** is checked, and if the status is anythingother than **UDI_OK**, then a call to **udi_channel_event_complete** acknowledges the failureof the bind to the bridge mapper, using the **channel_event_cb** control block containing the `initiator_context` of the channel event. See **udi_channel_event_complete**(3udi).

If `status` is UDI_OK, then the channel to the bus bridge is bound and we can access the CMOS RAM device. The **udi_pio_map** routine is called to map the device memory and registers for access by the driver. Two PIO handles must be allocated: one for reading the device and one for writing to it.

Since **udi_pio_map** is an asynchronous service call to the environment, and may not return immediately, a ``callback" function is provided with the call. When the environment allocates the PIO handle, it returns it to the driver by invoking the callback routine.

In the **udi_cmos** driver, the PIO handle for the write operation is requested in the code for the **cmos_bus_bind_ack** routine. The **cmos_bus_bind_ack_1** routine is specified as the callback function; when

the environment allocates the handle, it calls **cmos_bus_bind_ack_1** to return it to the driver. The **cmos_bus_bind_ack_1** routine saves the context of the call and the write transaction's PIO handle, and then requests a handle for the read transaction using a callback function of **cmos_bus_bind_ack_2**.

The **cmos_bus_bind_ack_2** saves the PIO handle for the write operation in the region data structure. Finally, **cmos_bus_bind_ack_2** issues a **udi_channel_event_complete** call to tell the environment that the bind operation is completed.

The two **udi_pio_map** calls use similar arguments (in order of appearance):

*callback*
: The function to be called once the requested handle is allocated. These are different with each call to provide the mappings through successive calls to **udi_pio_map**.

*regset_idx*

*base_offset*

*length*
: The same three constants are specified for both calls; these were defined earlier in the driver code; see ``Programmed I/O (PIO)".

*trans_list*
: The two transaction lists for the read and write transactions were defined earlier in the driver code (see ``PIO transaction lists for cmos_udi"). These transaction lists specify the actions taken when **udi_pio_trans** is called with the associated PIO handle. [This is done in the child channel operations.]

*list_length*
: The number of elements in the `trans_list`.

*pio_attributes*
: The data translation and ordering requirements for accessing device memory, in and OR'd list of constants; see **udi_pio_map**(3udi) for the full list of attributes. This list is specific to a device, and so is the same for both read and write transactions in this driver. UDI_PIO_NEVERSWAP specifies access to data with no byte swapping. Transactions are limited to one byte or less in size if this flag is set. UDI_PIO_STRICTORDER specifies that the CPU must issue the transactions in order, as the programmer specified.

*pace*
: The PIO pacing time, in microseconds, for this handle. The environment guarantees that any device access that occurs using this PIO handle will be followed by at least `pace` microseconds before another access occurs to the same device register set (via any handle). (The UDI_PIO_STRICTORDER attribute is required for non−zero `pace` values.)

We also need a channel operation entry point for the environment to use to respond to a **udi_bus_unbind_req** from the driver, so the bridge mapper can send the status of the unbind request back to the driver.

The bridge mapper calls **cmos_bus_bind_ack** with the same **bus_bind_cb** control block passed to it via **cmos_bus_unbind_req** from the driver.

```
static void
cmos_bus_unbind_ack(udi_bus_bind_cb_t *bus_bind_cb)
{
     udi_mgmt_cb_t *cb = bus_bind_cb->gcb.initiator_context;
     cmos_region_data_t *rdata = UDI_GCB(bus_bind_cb)->context;
```

udi_pio_unmap(rdata−>trans_read); udi_pio_unmap(rdata−>trans_write);

udi_cb_free(UDI_GCB(bus_bind_cb));

udi_devmgmt_ack(cb, 0, UDI_OK); }

This routine first saves the context of the operation in a **udi_mgmt_cb_t**(3udi) control block for later use in the acknowlegdement operation, and gets the region data from the control block passed to it. It then frees the region data memory blocks (using **udi_pio_unmap**(3udi)) and the control block (using **udi_cb_free**(3udi)). The **udi_devmgmt_ack**(3udi) call acknowledges the completion of the unbind operation on the driver side.

## udi_cmos child channel operations

The **udi_cmos** driver acts as a GIO provider, meaning that it waits for requests to read from or write to CMOS RAM from the client side of the GIO communication channel. All the child channel operations defined here were declared to the environment earlier in the code; see ``GIO metalanguage entry points".

First, we provide the routines to bind and unbind a channel, and respond to general channel events.

```
static void
cmos_gio_bind_req(udi_gio_bind_cb_t *gio_bind_cb)
{
     udi_gio_bind_ack(gio_bind_cb, CMOS_DEVSIZE, 0, UDI_OK);
}
```

static void cmos_gio_unbind_req(udi_gio_bind_cb_t *gio_bind_cb) { udi_gio_unbind_ack(gio_bind_cb); }

static void cmos_child_channel_event(udi_channel_event_cb_t *channel_event_cb) {
udi_channel_event_complete(channel_event_cb, UDI_OK); }

The above routines are the driver−side counterparts of the **udi_gio_bind_req**(3udi) **udi_gio_unbind_req**(3udi) and **udi_channel_event_ind**(3udi) environment−side channel operations; **cmos_gio_bind_req**, for example, is executed in response to a **udi_gio_bind_req** call from the UDI environment. The routines used by the **udi_cmos** driver, it should be noted, are quite simple; complex drivers may need to do more (such as saving state information or freeing control blocks) in response to channel bind, unbind, and event indication operations from the UDI environment.

The **cmos_gio_bind_req** call responds to the environment with a **gio_bind_cb** control block (used to hold the same structure passed to the driver via **udi_gio_bind_req**), two arguments indicating the device size, and a status indicator. See **udi_gio_bind_ack**(3udi).

**cmos_gio_unbind_req** similarly acknowledges an unbind request by returning **udi_gio_unbind_ack**(3udi) with an unmodified control block.

The general event−response routine, **cmos_child_channel_event**, simply acknowledges receipt of an event notification and passes **UDI_OK** back to the environment.

Next, we define the operation to respond to a transfer request from the UDI environment, which is the way a CMOS read or write is invoked.

```
static void cmos_do_read(udi_gio_xfer_cb_t *, udi_ubit8_t);
static void cmos_do_write(udi_gio_xfer_cb_t *, udi_ubit8_t);
```

static void cmos_gio_xfer_req(udi_gio_xfer_cb_t *gio_xfer_cb) { udi_gio_rw_params_t *rw_params = gio_xfer_cb−>tr_params;

/* * We can ignore the timeout parameter since all of our * operations are synchronous. */

switch (gio_xfer_cb−>op) { case UDI_GIO_OP_READ: udi_assert(rw_params−>offset_hi == 0 && rw_params−>offset_lo <= CMOS_DEVSIZE && gio_xfer_cb−>data_buf−>buf_size <= CMOS_DEVSIZE − rw_params−>offset_lo); cmos_do_read(gio_xfer_cb, rw_params−>offset_lo); break; case UDI_GIO_OP_WRITE: udi_assert(rw_params−>offset_hi == 0 && rw_params−>offset_lo <= CMOS_DEVSIZE && gio_xfer_cb−>data_buf−>buf_size <= CMOS_DEVSIZE − rw_params−>offset_lo); cmos_do_write(gio_xfer_cb, rw_params−>offset_lo); break; default: udi_gio_xfer_nak(gio_xfer_cb, UDI_STAT_NOT_UNDERSTOOD); } }

This routine is the driver−side counterpart to the ,LR udi_gio_xfer_req 3udi environment−side channel operation. Essentially, once the **udi_cmos** driver is loaded and bound, it waits for a **udi_gio_xfer_req** call from the environment.

The control block type used here is a standard UDI type (see **udi_gio_xfer_cb_t**(3udi)), and was declared previously in the control block initialization vector of the driver code (see ``Control block indexes and structures"). The **gio_xfer_cb** structure used by **cmos_gio_xfer_req** is filled in by the UDI environment prior to invocation of **udi_gio_xfer_req**.

First, **cmos_gio_xfer_req** sets a **udi_gio_rw_params_t**(3udi) structure to the contents of the `tr_params` element of the **gio_xfer_cb** structure (set by the environment). The format of the `tr_params` element was set indirectly by the driver code's **cmos_gio_bind_req** call (see above), by specifying a non−zero `device_size_lo` to **udi_gio_bind_ack**(3udi). See **udi_gio_rw_params_t**(3udi).

Next, **cmos_gio_xfer_req** issues a call to the appropriate subroutine (**cmos_do_read** or **cmos_do_write**), based on the value of `gio_xfer_cb->op` passed to it from the environment. The driver only expects two generic operation types (UDI_GIO_OP_READ and UDI_GIO_OP_WRITE) from the environment (see udi_gio_op_t 3udi ).

Any other operation type code casues a **udi_gio_xfer_nak**(3udi) to be sent indicating the operation was not understood (see Common Status Codes for a list of status codes and their meanings).

In both the read and write cases, the driver code provides an example of using **udi_assert**(3udi) to check the consistency of the data being passed to the read or write operation. The **udi_assert** calls shown check to see that the `offset_hi` and `offset_lo` parameters being passed match what was set previously in the driver's call to **udi_gio_bind_ack**(3udi) (see **cmos_gio_bind_ack**, defined above). If this assertion returns ``false", then the UDI environment is expected to abort the execution of the driver instance.

Now, we define the read routine:

```
static void cmos_do_read_2(udi_cb_t *, udi_buf_t *, udi_status_t, udi_ubit16_t);
```

static void cmos_do_read(udi_gio_xfer_cb_t *gio_xfer_cb, udi_ubit8_t addr) { cmos_gio_xfer_scratch_t *gio_xfer_scratch = gio_xfer_cb−>gcb.scratch; cmos_region_data_t *rdata = UDI_GCB(gio_xfer_cb)−>context;

/* * Store address into first byte of scratch space, * so the trans list can get at it. */ gio_xfer_scratch−>addr = addr; gio_xfer_scratch−>count = gio_xfer_cb−>data_buf−>buf_size;

udi_pio_trans(cmos_do_read_2, UDI_GCB(gio_xfer_cb), rdata−>trans_read, 0, gio_xfer_cb−>data_buf, NULL); }

Invoked when the environment executes a **udi_gio_xfer_req** operation with `gio_xfer_cb->op` set to **UDI_GIO_OP_READ** (see above), **cmos_do_read** sets up a **gio_xfer_scratch** structure and ititializes it to the scratch structure in the **gio_xfer_cb** control block passed from the UDI environment (via **cmos_gio_xfer_req**). The **cmos_gio_xfer_scratch_t** type was declared earlier in the code (see ``Scratch space, offsets, and requirements'').

Region data (`rdata`) is set to the `context` found in the control block passed by **cmos_do_read**.

**cmos_do_read** then sets the `addr` and `count` elements in the scratch space to the address passed from **cmos_gio_xfer_req** (`rw_params->offset_lo`) and the buffer size passed from the UDI environment. The declarations from ``Scratch space, offsets, and requirements'' and ``Programmed I/O (PIO)'' let the environment know to use this scratch space in the PIO transactions executed by the **udi_pio_trans** operation that follows.

Finally, **udi_pio_trans**(3udi) is called to do the read from CMOS RAM. **cmos_do_read_2** is the callback routine (explained below) that is executed by the environment once it's done servicing the call to **udi_pio_trans**. The **gio_xfer_cb** control block originating from the environment's call to **udi_gio_xfer_req** is passed via a call to **UDI_GCB**(3udi) to convert it to the generic control block **udi_cb_t**(3udi) type required. The PIO handle is passed via the region data structure (`rdata->trans_read`; see ``udi_cmos region data structure''). The **trans_read** PIO transaction handle causes the list of operations specified in the **cmos_trans_read[]** array (see ``Programmed I/O (PIO)'') to be performed on the device. The 0 value that follows the PIO handle indicates to start the PIO transation list from the beginning. `gio_xfer_cb->data_buf` is the data buffer originally passed to the driver from the UDI environment, to be used by UDI_PIO_BUF transactions in the **cmos_trans_read[]** array. (The final NULL argument indicates that there are no UDI_PIO_MEM transactions in the PIO transaction list.)

Once **udi_pio_trans** finishes the transaction list, it executes the callback function, **cmos_do_read_2**. (The format of the callback function is defined on **udi_pio_trans**(3udi).)

```
static void
cmos_do_read_2(
     udi_cb_t *gcb,
     udi_buf_t *new_buf,
     udi_status_t status,
     udi_ubit16_t result)
{
     udi_gio_xfer_cb_t *gio_xfer_cb =
                              UDI_MCB(gcb, udi_gio_xfer_cb_t);
```

/* * The pio trans trans may (and probably will) have handed us a * new buffer. Use that buffer in the udi_gio_xfer_ack. */ gio_xfer_cb−>data_buf = new_buf;

udi_gio_xfer_ack(gio_xfer_cb); }

This function simply passes on the control block passed to it (**gcb**, which contains the initiator's context), after converting it to the appropriate type required by **udi_gio_xfer_ack**(3udi) and setting the `data_buf` element to point to `new_buf`.

Note that any service call or channel operation that uses a callback must return a new buffer pointer in the corresponding callback, as does **cmos_do_read_2**, since the original buffer passed to the call (in this case, **udi_pio_trans**) may have been reallocated by the environment. See **udi_buf_t**(3udi).

The **udi_gio_xfer_ack** call requires that the `data_buf` element of the control block passed to it must either be the same as in the original request (in this case, the **udi_gio_xfer_req** from the environment side of the GIO channel), or a direct ``descendant'' of the original buffer (i.e. a buffer resulting from a chain of one or more service calls that replace the original buffer with a modified version), or NULL. The `data_buf` element is set to `new_buf` to meet this requirement.

The `status` and `result` parameters are ignored in the callback, which assumes success of the read operation.

The write operation is defined similarly to the read operation, as shown below:

```
static void
cmos_do_write_1(udi_cb_t *, udi_buf_t *, udi_status_t, udi_ubit16_t);
```

static void cmos_do_write(udi_gio_xfer_cb_t *gio_xfer_cb, udi_ubit8_t addr) { cmos_region_data_t *rdata = UDI_GCB(gio_xfer_cb)−>context; cmos_gio_xfer_scratch_t *gio_xfer_scratch = gio_xfer_cb−>gcb.scratch;

/* * The first CMOS_RDONLY_SZ bytes of this device are not * allowed to be written through this driver. Fail any attempt * to write to these bytes. */ if (addr < CMOS_RDONLY_SZ) { udi_gio_xfer_nak(gio_xfer_cb, UDI_STAT_MISTAKEN_IDENTITY); return; }

/* * Store address into first byte of scratch space, * so the trans list can get at it. The data to write * will be accessed directly from the buffer. */ gio_xfer_scratch−>addr = addr; gio_xfer_scratch−>count = gio_xfer_cb−>data_buf−>buf_size;

udi_pio_trans(cmos_do_write_1, UDI_GCB(gio_xfer_cb), rdata−>trans_write, 0, gio_xfer_cb−>data_buf, NULL); }

Just as the **cmos_do_read** routine did, **cmos_do_write** first sets a region data structure to the `context` passed to it, and prepares a scratch structure by setting it to the contents of the scratch area passed to it (see ``Scratch space, offsets, and requirements'').

The **addr** argument is checked to make sure it addresses the writeable portion of the device as earlier defined (see ``Programmed I/O (PIO)''). If it addresses the read−only portion of the device, a **udi_gio_xfer_nak**(3udi) is returned using the gio_xfer_cb control block and a status code of UDI_STAT_MISTAKEN_IDENTITY

(see Common Status Codes in the *UDI Core Specification* for a list of status codes and their meanings).

The scratch data structure's elements are then set to the `addr` argument and the `buf_size` element from the **gio_xfer_cb** argument. As with the read operation, these are used in the PIO transactions executed as part of the **udi_pio_trans** that follows.

**udi_pio_trans**(3udi) is called to do the write to CMOS RAM. **cmos_do_write_1** is the callback routine (explained below) that is executed by the environment once it's done servicing the call to **udi_pio_trans**. The **gio_xfer_cb** control block originating from the environment's call to **udi_gio_xfer_req** is passed via a call to **UDI_GCB**(3udi) to convert it to the generic control block **udi_cb_t**(3udi) type required. The PIO handle is passed via the region data structure (`rdata->trans_write`; see ``udi_cmos region data structure''). The **trans_write** PIO transaction handle causes the list of operations specified in the **cmos_trans_write[]** array (see ``Programmed I/O (PIO)'') to be performed on the device. The 0 value that follows the PIO handle indicates to start the PIO transation list from the beginning. `gio_xfer_cb->data_buf` is the data buffer originally passed to the driver from the UDI environment, to be used by UDI_PIO_BUF transactions in the **cmos_trans_write[]** array. Presumably, it contains the data to write to the device. (The final NULL argument indicates that there are no UDI_PIO_MEM transactions in the PIO transaction list.)

Once **udi_pio_trans** finishes the transaction list, it executes the callback function, **cmos_do_write_1**. (The format of the callback function is defined on **udi_pio_trans**(3udi).)

```
static void
cmos_do_write_1(
     udi_cb_t *gcb,
     udi_buf_t *new_buf,
     udi_status_t status,
     udi_ubit16_t result)
{
     udi_gio_xfer_cb_t *gio_xfer_cb =
                              UDI_MCB(gcb, udi_gio_xfer_cb_t);
```

/* We're done with the buffer. Free it. */ udi_buf_free(gio_xfer_cb−>data_buf); gio_xfer_cb−>data_buf = NULL;

udi_gio_xfer_ack(gio_xfer_cb); }

This function passes on the control block passed to it (**gcb**, which contains the initiator's context), after converting it to the appropriate type required by **udi_gio_xfer_ack**(3udi) and freeing the `data_buf` element in the control block.

The **udi_gio_xfer_ack** call requires that the `data_buf` element of the control block passed to it must either be the same as in the original request (in this case, the **udi_gio_xfer_req** from the environment side of the GIO channel), a direct ``descendant'' of the original buffer (i.e. a buffer resulting from a chain of one or more service calls that replace the original buffer with a modified version), or NULL. The `data_buf` element is set to **NULL** to meet this requirement.

The `status` and `result` parameters are ignored in the callback, which assumes success of the write operation.

## udi_cmos Management Agent channel operations

The **cmos_devmgt_req** and **cmos_final_cleanup_req** entry points are required to interact with the Management Agent, as defined earlier in the code under ``Management metalanguage operations vector''. They are the driver−side counterpart functions of **udi_devmgmt_req**(3udi) and **udi_final_cleanup_req**(3udi), respectively. The UDI environment executes the associated driver−side function after calling one of the above, passing appropriate data to the driver for it to respond to the request.

```
static void
cmos_devmgmt_req(
     udi_mgmt_cb_t *cb,
     udi_ubit8_t mgmt_op,
     udi_ubit8_t parent_id)
{
     cmos_region_data_t *rdata = cb−>gcb.context;
```

switch (mgmt_op) { case UDI_DMGMT_UNBIND: #if DO_INTERRUPTS /* Detach interrupts here... */ #endif /* Keep a link back to this CB for use in the ack. */ rdata−>bus_bind_cb−>gcb.initiator_context = cb;

/* Do the metalanguage−specific unbind. */ udi_bus_unbind_req(rdata−>bus_bind_cb); break; default: udi_devmgmt_ack(cb, 0, UDI_OK); } }

After setting the region data structure to the context element of the control block passed from the environment, the driver switches on the `mgmt_op` argument. If the operation requested is UDI_DMGMT_UNBIND, then the driver needs to send a **udi_bus_unbind_req**(3udi) back to the environment to begin the unbind operation, as well as free any remaining resources (some resources may be left open by some drivers until the **udi_final_cleanup_req** is executed by the environment).

**udi_bus_unbind_req** requires a **udi_bus_bind_cb_t** structure. The driver sets the `initiator_context` field of **bus_bind_cb** in the region data structure to the control block passed to the driver, to indicate the parent context of the call. It then passes `rdata->bus_bind_cb` back to the environment via **udi_bus_unbind_req**.

Then, the driver sends a UDI_OK status back to the environment via **udi_devmgmt_ack**(3udi).

Once the environment completes the unbinding of the driver, it calls **udi_final_cleanup_req**(3udi) and the corresponding channel operation in the driver, **cmos_final_cleanup_req**:

```
static void
cmos_final_cleanup_req(udi_mgmt_cb_t *cb)
{
     /*
      * We have nothing to free that wasn't already freed by
      * unbinding children and parents.
      */
     udi_final_cleanup_ack(cb);
}
```

The purpose of the **cmos_final_cleanup_req** routine is to provide an entry point for the UDI environment to call when unbinding the device, and the management channel is the only channel left to unbind. This allows the driver the oppportunity to free any resources still held and perform any other cleanup necessary before the driver is finally unbound from the environment.

In the case of the **udi_cmos** driver, all resources have already been freed when the other channels (the parent GIO channel and the child bus channel) were unbound, so the driver just passes the control block it gets from the environment back via **udi_final_cleanup_ack**(3udi). This is the last operation the driver performs before it is removed from the system.

See also: Device Management Operations in the *UDI Core Specification*.

## pseudod parent channel operations

The pseudod driver defines parent channel operations that are almost identical to those defined by the udi_cmos driver. The difference is that when binding the channel, pseudo_bus_channel_event (the analog of cmos_bus_channel_event in the udi_cmos driver), allocates some memory to use as a transaction buffer.

```
/*
 *------------------------------------------------------------------------
 * Bus Bridge Interface Ops
 *------------------------------------------------------------------------
 */
```

static void got_tx_mem(udi_cb_t *gcb, void *new_mem) { pseudo_region_data_t *rdata = gcb−>context; rdata−>tx_queue = new_mem;

udi_bus_bind_req(UDI_MCB(gcb, udi_bus_bind_cb_t));

}

static void pseudo_bus_channel_event(udi_channel_event_cb_t * cb) { udi_bus_bind_cb_t *bcb = UDI_MCB(cb−>params.parent_bound.bind_cb, udi_bus_bind_cb_t);

switch (cb−>event) { case UDI_CHANNEL_BOUND: /* * Get our tx buffers. Let it allocate a bus bind cb */ UDI_GCB(bcb)−>initiator_context = cb; udi_mem_alloc(got_tx_mem, UDI_GCB(bcb), PSEUDO_BUF_SZ, UDI_MEM_MOVABLE); break; default: udi_channel_event_complete(cb, UDI_OK); } }

static void pseudo_bus_bind_ack(udi_bus_bind_cb_t * bus_bind_cb, udi_dma_constraints_t dma_constraints, udi_ubit8_t preferred_endianness, udi_status_t status) { udi_channel_event_cb_t *channel_event_cb =

UDI_GCB(bus_bind_cb)−>initiator_context; pseudo_region_data_t *rdata = UDI_GCB(bus_bind_cb)−>context;

/* * As we don't have any hardware we can simply complete the * UDI_CHANNEL_BOUND event after stashing the bus_bind_cb for the * subsequent unbind() */ udi_dma_constraints_free(dma_constraints);

rdata−>bus_bind_cb = bus_bind_cb;

udi_channel_event_complete(channel_event_cb, status); }

static void pseudo_bus_unbind_ack(udi_bus_bind_cb_t * bus_bind_cb) { udi_mgmt_cb_t *cb = bus_bind_cb−>gcb.initiator_context; pseudo_region_data_t *rdata = UDI_GCB(bus_bind_cb)−>context;

udi_cb_free(UDI_GCB(bus_bind_cb)); udi_mem_free(rdata−>tx_queue);

udi_devmgmt_ack(cb, 0, UDI_OK); }

The difference in the code is that when **pseudo_parent_channel_event** switches on UDI_CHANNEL_BOUND, instead of executing **udi_channel_event_complete**, it issues a call to **udi_mem_alloc**(3udi) instead, with **got_tx_mem** as a callback:

```
udi_mem_alloc(got_tx_mem, UDI_GCB(bcb),
              PSEUDO_BUF_SZ, UDI_MEM_MOVABLE);
```

**got_tx_mem** issues the **udi_channel_event_complete**, after it stores the pointer to the new memory allocated by the environment in region data; the new memory pointer was passed back to the driver via a call to **got_tx_mem**, after the environment finished servicing the **udi_mem_alloc**.

**udi_mem_alloc** allocates a memory region solely for the use of the driver (it is not addressable by the device, as in direct memory access). This is used by the **pseudod** driver to imitate real I/O to and from a physical device; it is written and read by the driver subroutines for the child channel (see ``pseudod child channel operations''). It's passed a **udi_bus_bind_cb_t**(3udi) control block converted to a udi_cb_t control block by **UDI_GCB**(3udi), to preserve the call's context.

PSEUDO_BUF_SZ is the size of the buffer necessary (in bytes) to hold the test pattern; it was defined earlier in the code as 1024 (see ``pseudod.c region data structure'').

The UDI_MEM_MOVABLE flag allocates ``movable'' memory, which is required by the UDI specification for reference by control block fields or channel operation parameters.

See Memory Management and Memory Objects in the *UDI Core Specification*.

Finally, **got_tx_mem** copies the memory pointer allocated by **udi_mem_alloc** to a similarly−sized element in the region data structure, and requests the bind operation.

The acknowledgement of the bind operation, **pseudo_bus_bind_ack**, is identical to the routine used in the **udi_cmos** driver, **cmos_bus_bind_ack**.

The unbind operation, **pseudo_bus_unbind_ack**, has one difference from the **cmos_bus_unbind_ack** routine in the **udi_cmos** driver: it must free the memory pointed to by the region data structure before closing the channel. It does this with a call to **udi_mem_free**(3udi), using the pointer to the memory block (`rdata->tx_queue`) as the argument.

The remainder of these routines are identical to the similarly named routines for the **udi_cmos** driver, defined under ``udi_cmos parent channel operations''; refer back to that section for descriptions.

## pseudod child channel operations

The routines for the GIO child channel provide entry points for binding and unbinding a channel to the GIO mapper, responding to GIO channel events, and performing the read and write operations,

```
/*
 *------------------------------------------------------------------------
 * GIO operations
 *------------------------------------------------------------------------
 */
static void
```

```
pseudo_gio_channel_event(udi_channel_event_cb_t * channel_event_cb)
{
     udi_channel_event_complete(channel_event_cb, UDI_OK);
}
```

static void pseudo_gio_bind_req(udi_gio_bind_cb_t * gio_bind_cb) { /* * Fake up some constraints so that the GIO mapper can call * UDI_BUF_ALLOC safely */ gio_bind_cb−>xfer_constraints.udi_xfer_max = 0; gio_bind_cb−>xfer_constraints.udi_xfer_typical = 0; gio_bind_cb−>xfer_constraints.udi_xfer_granularity = 1; gio_bind_cb−>xfer_constraints.udi_xfer_one_piece = FALSE; gio_bind_cb−>xfer_constraints.udi_xfer_exact_size = FALSE; gio_bind_cb−>xfer_constraints.udi_xfer_no_reorder = TRUE; udi_gio_bind_ack(gio_bind_cb, 0, 0, UDI_OK); }

static void pseudo_gio_unbind_req(udi_gio_bind_cb_t * gio_bind_cb) { udi_gio_unbind_ack(gio_bind_cb); }

The **pseudo_gio_channel_event** and **pseudo_gio_unbind_req** routines are identical to the **cmos_child_channel_event** and **cmos_gio_unbind_req** routines used by the **uci_cmos** driver (see ``udi_cmos child channel operations").

The **pseudo_gio_bind_req** establishes some transfer constraints for the GIO channel; this routine is called by the environment to enter the driver code after a **udi_gio_bind_req**(3udi).

The driver uses the `xfer_constraints` element of the **udi_gio_bind_cb_t**(3udi) structure passed to the driver by the environment to establish the data transfer constraints for the buffer that will be used for reads and writes. These constraints are then sent back to the environment in the same **gio_bind_cb**, via a call to **udi_gio_bind_ack**(3udi).

Transfer constraints are device−dependent, and are provided so that the UDI environment can allocate buffers appropriately for the data to be transferred. Even though the **pseudod** driver does not control a real device, it uses buffers to simulate real data transfer.

See **udi_xfer_constraints_t**(3udi).

Once the GIO channel is bound, the driver essentially waits for the environment to call **udi_gio_xfer_req**(3udi) and the corresponding **pseudod** driver entry point, **pseudo_gio_xfer_req**:

```
static void
pseudo_gio_xfer_req(udi_gio_xfer_cb_t * gio_xfer_cb)
{
     pseudo_region_data_t *rdata = UDI_GCB(gio_xfer_cb)−>context;
```

switch (gio_xfer_cb−>op) { case UDI_GIO_DIR_WRITE:{ /* * From client (application) to provider (this driver) */ udi_size_t nread = 0; udi_buf_t *buf = (udi_buf_t *)gio_xfer_cb−>data_buf; udi_ubit8_t *cp; udi_ubit32_t src_offset = 0;

#if PSEUDO_VERBOSE udi_size_t n; udi_ubit32_t iter = 0;

udi_debug_printf("Size of complete write buffer 0x%x\n", buf−>buf_size); #endif while (src_offset < buf−>buf_size) { nread = sizeof (rdata−>rx_queue); /* Sz of src buffer */ if (buf−>buf_size − src_offset < nread) nread = buf−>buf_size − src_offset; udi_buf_read(buf, src_offset, /* src offset */ nread, /* Size data to read */ &rdata−>rx_queue); /* Destination */ cp = &rdata−>rx_queue[0]; #if PSEUDO_VERBOSE

udi_debug_printf("\nIteration %d\n" "Size of partial write buffer 0x%x\n", iter++, nread); #endif src_offset += nread;

/* * At this point, * rdata−>rx_queue[0] − rdata−>rx_queue[nread] * holds 'nread' bytes of data passed down from the * user to us. This would be where we could pass * it to hardware or something. We'll just print * it in hexdump−ish format. FIXME: assumes an ASCII * bytestream and encoding. */ #if PSEUDO_VERBOSE for (n = 0; n < nread; n++, cp++) { udi_ubit8_t c = *cp;

if (!(n & 15)) { udi_debug_printf("\n %04x: ", n); } if (((c >= ' ') && (c <= '~')) || (c == '\t') || (c == '\r') || (c == '\n')) { udi_debug_printf("%c", c); } else { udi_debug_printf("{%2x}", c); } } #endif } /* * Free the buffer locally so the GIO client (mapper) * doesn't have to. */ udi_buf_free(gio_xfer_cb−>data_buf); gio_xfer_cb−>data_buf = NULL; udi_gio_xfer_ack(gio_xfer_cb); } break;

/* * From provider (us) to client (application) */ case UDI_GIO_DIR_READ: pseudo_gio_do_read(gio_xfer_cb); break;

/* * If the user specified anything else, (including a * UDI_GIO_DIR_READ|UDI_GIO_DIR_WRITE), surrender */ default: udi_gio_xfer_nak(gio_xfer_cb, UDI_STAT_NOT_UNDERSTOOD); break; } }

---

**NOTE:** The PSEUDO_VERBOSE sections of the above code are discussed under ``Debugging code in the pseudod driver".

---

After setting a pointer to the channel operation context, **pseudo_gio_xfer_req** branches based on the value of `gio_xfer_cb−>op` passed from the environment. The routine then performs either the operations for UDI_GIO_DIR_READ or UDI_GIO_DIR_WRITE; the driver responds to any other operation from the environment with **udi_gio_xfer_nak**(3udi) with a status of UDI_STAT_NOT_UNDERSTOOD.

The routine for the UDI_GIO_DIR_WRITE case (that is, writing data from user−level to the pseudo−device), is in the **pseudo_gio_xfer_req** code, while the read case (reading data from the pseudo device and writing it to user−level) executes another routine, **pseudo_gio_do_read**, to perform the UDI_GIO_DIR_READ operation. **pseudo_gio_do_read** is explained further on in this section.

The UDI_GIO_DIR_WRITE section of **pseudo_gio_xfer_req** begins by setting some local data variables:

***nread***
: counter for the number of bytes read out of the data buffer passed to the driver

***buf***
: pointer to the data buffer

***cp***
: pointer to a region data structure to hold the data read

***src_offset***
: the current offset into the buffer for the udi_buf_read operation

The main **while** loop of the routine uses **src_offset** as a counter to make sure we don't try to read past the end of the data buffer we get from the environment. The variable **nread** is set to the size of the data buffer; it is

reset to the amount of data actually in the buffer in the following two lines of code if the amount of data in the buffer is less than the size of the buffer.

A call to **udi_buf_read**(3udi) then reads **nread** bytes of the data buffer strating at **src_offset** into **rdata−>rx_queue**. The variable cp is set to point to the first byte of the data in **rdata−>rx_queue**, and will be used to write the data to debug output (see ``Debugging code in the pseudod driver''). **src_offset** is incremented by the size of the data buffer to prepare for the next (possible) read.

In a driver that accessed a real device, this part of the code would use PIO calls to write to the device, as the CMOS RAM driver does via **udi_pio_trans**(3udi) (see ``udi_cmos child channel operations''). Instead, the data is written to user level via a call to **udi_debug_printf**(3udi) (see ``Debugging code in the pseudod driver'').

Once the while loop reads and writes the entire buffer, the driver frees the data buffer with **udi_buf_free**(3udi), sets the pointer to the data buffer to NULL, and lets the environment know it's completed the environment's call to **pseudo_gio_xfer_req**, using **udi_gio_xfer_ack**(3udi) and the original **udi_xfer_cb** passed to the driver.

The UDI_GIO_DIR_READ operation is carried out by **pseudo_gio_do_read**:

```
static void
pseudo_gio_do_read(udi_gio_xfer_cb_t * gio_xfer_cb)
{
     udi_cb_t *gcb = UDI_GCB(gio_xfer_cb);
     pseudo_region_data_t *rdata = gcb->context;
     udi_size_t len =
             (PSEUDO_BUF_SZ − 1 ) <
             gio_xfer_cb->data_buf->buf_size ? (PSEUDO_BUF_SZ −
                                                1) : gio_xfer_cb->data_buf->
```

buf_size; udi_size_t i = 0, n;

/* * Write a simple test pattern into the tx_queue. */ while (i < len) { n = udi_snprintf((char *)&rdata−>tx_queue[i], (len − i), "%d ", rdata−>testcounter);

/* * If the entire sprintf buffer wouldn't fit, lop it off * and issue a partial read. * If the string ends in a space, we know that a complete * number was generated, and thus should not be lopped off. * In other words, the driver will only work correctly all * the time if you read NUMBYTES each time where * NUMBYTES=sprintf(buf,"%d ",−1). * If the driver returns 0 bytes for some read, it is likely * that you are not requesting enough data. Why return 0 * instead of a partial buffer? Because it would get * really messy if we stored the state of the * partially written string in the driver for subsequent * reads. */ i += n; if ((n == 0) || (i == len)) { if (rdata−>tx_queue[i − (i == 0 ? 0 : 1)] != ' ') i −= n; else rdata−>testcounter++; break; } rdata−>testcounter++; }

/* * Tell the mapper how much data is present. */ gio_xfer_cb−>data_buf−>buf_size = i;

/* * Let the mapper have the data. */ #ifdef GIO_BUF_BUG udi_buf_write(pseudo_buf_written, gcb, rdata−>tx_queue, i, gio_xfer_cb−>data_buf, 0, i, UDI_NULL_BUF_PATH); #else udi_buf_write(pseudo_buf_written, gcb, rdata−>tx_queue, i, gio_xfer_cb−>data_buf, 0, 0, UDI_NULL_BUF_PATH); #endif /* GIO_BUF_BUG */ }

This routine essentially generates a test pattern to simulate a read operation from a real device, and passes this back to the GIO mapper (for return to user level).

The driver first sets a **udi_cb_t**(3udi) control block containing a copy of the **udi_gio_xfer_cb_t**(3udi) control block passed to it and a local copy of the pointer to region data passed as part of the control block context. **len** is set to either the length of the actual data in the control block's buffer or the size of the null−terminated holding place in region data (PSEUDO_BUF_SZ − 1).

The following **while** loop uses the previously initialized variable ``i'' as a counter, with an upper bound of **len**, to write the amount of data requested to the buffer.

Test data is written to the region data area using **udi_snprintf**(3udi) (modeled after, but not identical to, **snprintf**(3S)).

```
udi_snprintf((char *)&rdata->tx_queue[i], (len − i),
"%d ", rdata−>testcounter);
```

[In a driver that accessed a real device, this part of the code would use PIO calls to read the device, as the CMOS RAM driver does via **udi_pio_trans**(3udi) (see ``udi_cmos child channel operations'')].

After the test data is written to the holding place in region data, the size of the data written is placed into `gio_xfer_cb->data_buf->buf_size`, which will be passed to the GIO mapper via **udi_buf_write**:

```
udi_buf_write(pseudo_buf_written, gcb, rdata->tx_queue, i,
           gio_xfer_cb->data_buf, 0, 0, UDI_NULL_BUF_PATH);
```

**pseudo_buf_written** is a callback routine that the environment will execute once the **udi_buf_write** completes; it is explained below. **gcb** is a pointer to a copy of the **gio_xfer_cb** passed to the driver as part of the environment's call to **pseudo_gio_do_read**. **rdata−>tx_queue** is the region data element holding the data to be written to the buffer. The next argument, **i**, holds the size of the data to be written.

**gio_xfer_cb−>data_buf** is a pointer to the target data buffer. This will have been allocated by the environment before its call to **pseudo_gio_do_read**. The following two arguments, both zeros, tell the environment to write the source data from rdata−>tx_queue to the beginning of the buffer and to use the fourth argument, i, as the length of the data to write. Since we specified a non−NULL data buffer, there's no need for **udi_buf_write** to allocate a new one, so the final argument is UDI_NULL_BUF_PATH.

See **udi_buf_write**(3udi) and **udi_buf_copy**(3udi) for detailed explanations of these arguments and their usage.

Once the environment completes the **udi_buf_write**, it returns the new buffer back to the driver via the callback routine **pseudo_buf_written**:

```
static void
pseudo_buf_written(udi_cb_t *gcb,
                udi_buf_t *new_buf)
{
     udi_gio_xfer_cb_t *xfer_cb = UDI_MCB(gcb, udi_gio_xfer_cb_t);
     udi_size_t bufsize;


xfer_cb−>data_buf = new_buf; bufsize = new_buf−>buf_size;
```

#if PSEUDO_VERBOSE udi_debug_printf("pseudo_buf_written: data_len = %d\n", bufsize); #endif udi_gio_xfer_ack(xfer_cb); }

The purpose of thsi call is to issue an appropriate **udi_gio_xfer_ack** back to the environment to acknowledge completion of the **pseudo_gio_xfer_req** call that began the UDI_GIO_DIR_READ operation.

This routine first converts the generic control block it gets from the environment to the type it needs ( **udi_gio_xfer_cb_t**(3udi)) for the **udi_gio_xfer_ack**.

The **bufsize** variable is used, if PSEUDO_VERBOSE is set when the driver is compiled, to print the buffer size to debug output (see ``Debugging code in the pseudod driver'').

The routine then passes the new buffer with its contents back to the environment by setting **xfer_cb−>data_buf = new_buf** and using **xfer_cb** as the argument to **udi_gio_xfer_ack**(3udi).

The UDI environment deals appropriately with the data buffer, such as returning its contents to a user−level application that requested a **pseudod** device read.

### Debugging code in the pseudod driver

To summarize the operation of the **pseudod** driver as invoked from user−level (see UNRESOLVED XREF−0), a UDI_GIO_DIR_READ operation is generated by the UDI environment, to which the driver responds by writing a test pattern to the user level.

If the driver is compiled with PSEUDO_VERBOSE set, then the UDI_GIO_DIR_WRITE routine will write the same test pattern back to user level as debug output (simulating a write to a real device). Other information is also written to user−level by debugging statements in the code of the **pseudo_gio_do_read** and **pseudo_buf_written** routines discussed under ``pseudod child channel operations''.

The PSEUDO_VERBOSE sections of code all make use of the **udi_debug_printf**(3udi) routine to print to user level the current values of various variables. See Debugging Services in the *UDI Core Specification* for an explanation of how to use this and the other debugging service routines provided by the UDI environment.

---

**NOTE:** Note that calls to debugging service routines such as **udi_debug_printf** are not expected to be found in compiled, production UDI drivers. In particular, be aware that the *UDI Core Specification* permits a conforming UDI implementation to deal with calls to **udi_debug_printf** in an implementation−defined manner, including ignoring them.

---

In the case of the UDI_GIO_DIR_WRITE code in **pseudo_gio_do_read**, debugging output is used to simulate writing to a real device:

```
for (n = 0; n < nread; n++, cp++) {
     udi_ubit8_t c = *cp;
```

if (!(n & 15)) { udi_debug_printf("\n %04x: ", n); } if (((c >= ' ') && (c <= '~')) || (c == '\t') || (c == '\r') || (c == '\n')) { udi_debug_printf("%c", c); } else { udi_debug_printf("{%2x}", c); } }

**cp** is the pointer to region data holding the buffer data to be ``written'' to the ``device''. After doing some range and value checking on the buffer data, it's written in an appropriate format via **udi_debug_printf**.

## pseudod Management Agent channel operations

The **pseudo_devmgmt_req** routine performs the same operations as the **cmos_devmgmt_req** routine defined for the **udi_cmos** driver.

```
static void
pseudo_devmgmt_req(udi_mgmt_cb_t * cb,
                udi_ubit8_t mgmt_op,
                udi_ubit8_t parent_id)
{
     pseudo_region_data_t *rdata = UDI_GCB(cb)->context;
```

switch (mgmt_op) { case UDI_DMGMT_UNBIND: /* * Keep a link back to this CB for use in the ack. */ rdata−>bus_bind_cb−>gcb.initiator_context = cb;

/* * Do the metalanguage−specific unbind */ udi_bus_unbind_req(rdata−>bus_bind_cb); break; default: udi_devmgmt_ack(cb, 0, UDI_OK); } }

See ``udi_cmos Management Agent channel operations'' for a description.

Final cleanup for the **pseudod** driver is different from the final cleanup for the **udi_cmos** driver, in that the **pseudod** driver writes a log entry before final cleanup. Although logging is not required (the **udi_cmos** driver, for example, does no logging), it is an important means by which driver errors can be traced.

```
static udi_log_write_call_t cleanup_req_logged_cb;
```

static void pseudo_final_cleanup_req(udi_mgmt_cb_t * cb) { udi_log_write(cleanup_req_logged_cb, UDI_GCB(cb), UDI_TREVENT_LOCAL_PROC_ENTRY, UDI_LOG_INFORMATION, 0, UDI_OK, 1500); }

static void cleanup_req_logged_cb(udi_cb_t *gcb, udi_status_t correlated_status) { udi_mgmt_cb_t *cb = UDI_MCB(gcb, udi_mgmt_cb_t);

/* * TODO: Release any data which was allocated and not freed by other * tear−down processing */ udi_final_cleanup_ack(cb); }

When the UDI environment issues the **udi_final_cleanup_req**(3udi) call, the driver responds with **pseudo_final_cleanup_req**, which (instead of issuing the call to **udi_final_cleanup_ack**(3udi) as the **cmos_final_cleanup_req** routine does) calls **udi_log_write**(3udi). The callback routine for **udi_log_write**, **cleanup_req_logged_cb**, then issues the **udi_final_cleanup_ack** (after **udi_log_write** completes).

The **udi_log_write** records the shutdown of the driver. The first argument is the callback routine. The second argument is the saem control block passed to the driver from the environment, which includes the context of the operation. UDI_TREVENT_LOCAL_PROC_ENTRY and UDI_LOG_INFORMATION indicate that the log is recording entry into a local driver routine, and that this event is an expected driver operation (see **udi_trevent_t**(3udi) and **udi_log_write**(3udi)).

The ``0'' argument indicates theat this event applies to the Management Metalanguage; other events might use other metalanguage numbers as defined by the driver in a **meta** declaration in the driver's *udiprops.txt* (see ``Module and region declarations''.

Also see Tracing and Logging in the UDI Core Specification for other event codes and information on tracing events.

# Advanced UDI programming topics

This section lists major programming facilities and topics from the UDI Specifications that were not covered or mentioned only tangentially by the descriptions of the **udi_cmos** and **pseudod** drivers in this chapter.

*Interrupt handling*
: The **udi_cmos** and **pseudod** drivers provide interrupt service routines, but these are mostly stubs used as examples.

  See Interrupt Registration Operations and Interrupt Event Operations in the *UDI Physical I/O Specification*.

*Buffer management*
: See Buffer Management in the *UDI Core Specification*.

*Utility functions*
: See String/Memory Utility Functions, Queue Management Utility Functions, and Endianness Management Utility Functions in the *UDI Core Specification*.

*Timers*
: See Time Management

# Tips

## Module−global data

All non−automatic variables (that is, variables not automatically allocated in channel control blocks as inline variables) in the driver are considered ``module−global data", regardless of whether the data is local to particular functions or compilation units, or truly global to the driver code.

Module−global data is read−only throughout the execution of a UDI driver and is visible to all instances of a driver within a given domain (for that reason, it is sometimes called ``domain−global data" in the *UDI Specifications*).

C global variables (i.e., C variables defined outside of any function) may only be used for module−global data, and therefore **must** be declared read−only. It is recommended that all such variables be declared as static constants using the C language **const** and **static** keywords. Note that the UDI environment will always treat such data as read−only, even if you omit the recommended keywords.

# Network Driver Coding Specifics

The UDI NIC driver interfaces allow you to write NIC drivers that work with existing and future networking protocol stacks regardless of the OS and protocol stack characteristics.

This is accomplished through the Network Service Requester (NSR), which is that part of the UDI framework through which all network data and control requests pass to and from the Network Driver (ND) code. Your driver code (the ND) and the NSR communicate exclusively using the Network Interface Metalanguage, and so the ND is completely portable between implementations of the NSR that conform to the UDI NIC Driver Specification Version 1.0.

This means that the same driver code can be distributed with the driver hardware for all conforming UDI environment implementations on any operating system platform.

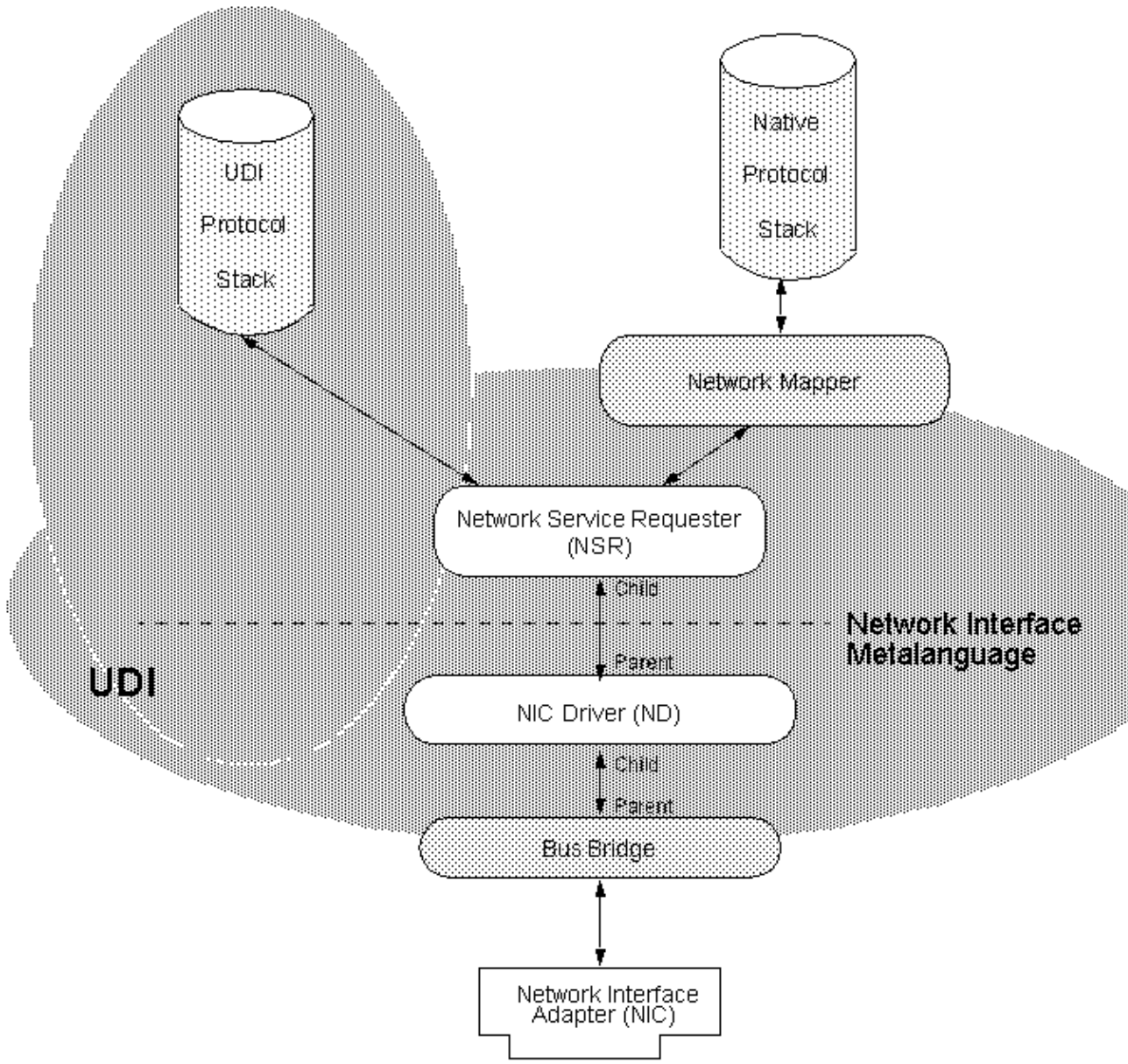
The next section of this chapter provides an overview of the ``Network Interface Metalanguage Architecture". The remaining sections of this chapter discuss in detail how the sample UDI NIC driver is coded, beginning with the section ``Sample Osicom 2300 NIC Driver (shrkudi)".

## Network Interface Metalanguage Architecture

The UDI Network Interface Metalanguage (NIM) defines a set of functions that provide all the services required by a Network Driver (ND), in a protocol− and transport−independent manner.

The ND primarily sends and receives network packets, and manages the hardware and the state of the network link, Details such as management of the network address space, topology discovery, and packet composition/decomposition operations are not the responsibility of the NIC driver; they are negotiated between the UDI networking environment and the network protocol stacks.

As shown in the following figure, the role of the NIM is to provide a consistent, portable interface between the driver and the UDI networking environment.




**UDI Networking Environment**

The driver (ND) communicates with the environment exclusively throught the Network Service Requester (NSR). The ND does this by populating the appropriate control blocks to establish three communication channels (management control and status, transmit data, and receive data) between itself and the NSR. Only one NSR/ND interface is active at any given time.

Transmit flow control is exercised by the ND. Transmit control blocks are owned by the ND and loaned to the NSR. The NSR sends transmit packets to the ND. The ND returns control blocks to the NSR on transmit complete. The NSR never allocates transmit blocks.

Receive flow control is exercised by the NSR. Recieve control blocks are owned by the NSR, filled by the ND, and send to the NSR by the ND. The ND never allocates receive buffers.

The ND also needs to establish a channel with the Bus Bridge Mapper by populating the appropriate control blocks and issuing approproate Bus Bridge Metalanguage calls over a separate bus bridge control channel.

The driver will also use Physical I/O metalanguage calls to communicate with the NIC over the bus.

Other channels not specific to the networking environment, such as Management Agent channels and inter−region communciation channels, will also be required in a UDI NIC driver. See the ``Sample Osicom 2300 NIC Driver (shrkudi)" for more on the other channels required by the sample NIC driver.

## Network driver instantiation

Once a UDI NIC driver is compiled, packaged, installed on a taret system, and loaded into the running kernel, the UDI environment follows this process to initialize a Network Driver (ND) instance for a newly installed NIC:

1. The Bus Bridge, which is the parent of all NDs, is instantiated, and a mgmt channel between the Bus Bridge and the MA is established. The MA sends a parent enumeration request to the Bus Bridge.
2. THe MA creates an ND instance (primary and secondary regions, management channel, internal bind channels) that corresponds to the parent enumeration response information.
3. The MA issues udi_usage_ind() on mgmt channel − first operation on newly instantiated driver instance
4. The MA begins the child/parent bind sequence by issuing UDI_CHANNEL_BOUND on the child (ND) end of ND−Bus bind channel. This channel may be in the primary region, or a static or dynamic secondary region.
5. ND performs internal initialization (specific to NIC), reads instance attributes, and binds to parent (Bus) with **udi_bus_bind_req**.
6. The parent (Bus) processes the ND's bind request, and replies to ND with **udi_bus_bind_ack**.
7. The ND completes initialization and issues **udi_channel_event_complete**; this completes the bind operation to the MA. On this channel, the MA is the parent and the ND is the child.
8. Finally, a channel is established between the ND (parent) and NSR (child) using NIM bind operations.

# Sample Osicom 2300 NIC Driver (shrkudi)

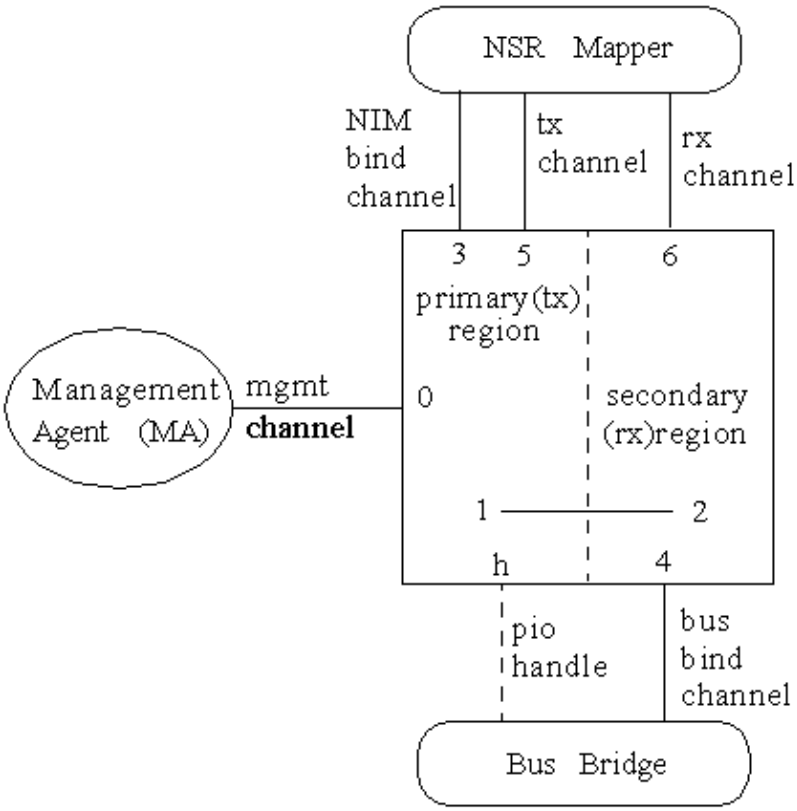
The sample UDI NIC driver for the Osicom 2300 uses the Network Interface Metalanguage to transmit and receive data through the network card, and is a dual−region UDI driver.

The source code for the driver is installed as part of the UDI Development Kit under */usr/src/hdk/udi/driver/shrkudi*. It is recommended that you open the above link in a new window, and follow along in the source code as you read this text.

The remainder of this chapter follows the sample NIC driver code to explain how to program a UDI NIC driver, and builds upon the basic UDI driver concepts introduced in ``UDI driver coding basics". It concentrates on presenting the concepts unique to programming a UDI NIC driver, as well as other aspects of the UDI programming environment (such as multi−region programming) not presented in earlier chapters.

## shrkudi Driver Architecture

The following figure shows how the UDI NIC sample driver, **shrkudi**, interacts with the UDI environment on the target system.




**shrkudi Driver Architecture**

*0*

Management Metalanguage channel − controls driver configuration and operating conditions; driven by Management Agent.

*1−2*

Inter−region channel − uses Generic I/O Metalanguage to send status, etc. across regions.

*3*

Network Metalanguage control channel − initial bind channel between the ND and the NSR.

*4*

Bus Bridge control channel − initial bind channel between the ND and the Bus Bridge Mapper.

*5*

Network Metalanguage transmit channel − controls transmit data flow.

*6*

Network Metalanguage receive channel − controls receive data flow.

*7*

Bus Bridge Interrupt handler channel − handles interrupt events.

*h*
: PIO handle − used for transmit/setup I/O (not a channel)

The **shrkudi** driver uses two regions: a primary region that communicates with the Management Agent (MA) and establishes an initial bind channel with the NSR that uses NIM functions to manage the transmission of data over the NIC; and, a secondary region that manages reception of data from the NIC as well as interrupts.

These two regions execute independently, and coordinate their activities over the GIO channel set up between them.

The primary region uses Physical I/O calls as defined in the Physical I/O Specification. to write data from its internal buffers to the NIC. To do this, it requests a ``PIO handle'' from the UDI environment. The PIO handle, while not a channel itself, provides the driver a convenient means of referring to control blocks that tell the environment about the NIC and how to communciate with it over the bus.

The secondary region handles the reception of data from the NIC, and so defines a channel between itself and the NSR. The channel between the bus bridge mapper and the secondary region, while initiated by the driver, is used by the bus bridge mapper when there is an interrupt from the NIC.

# NIC driver source code

The UDI NIC sample driver source code is installed under */usr/src/hdk/udi/driver/shrkudi*, and includes these files:

- */usr/src/hdk/udi/driver/shrkudi/README*
- */usr/src/hdk/udi/driver/shrkudi/udiprops.txt*
- */usr/src/hdk/udi/driver/shrkudi/shrkudi.c*
- */usr/src/hdk/udi/driver/shrkudi/shrk.h*

A future version of this document will explain the NIC source code in detail. For more information, see the UDI NIC Specification for a description of the Network Interface Metalanguage (NIM) used in this driver. Also see the Project UDI Web Site for a slide presentation on *UDI NIC Drivers*,

# SCSI HBA Driver Coding Specifics

The UDI SCSI HBA sample driver source code is installed under */usr/src/hdk/udi/driver/udi_dpt*, and includes these files:

- */usr/src/hdk/udi/driver/udi_dpt/README*
- */usr/src/hdk/udi/driver/udi_dpt/udiprops.txt*
- */usr/src/hdk/udi/driver/udi_dpt/udi_dpt.c*
- */usr/src/hdk/udi/driver/udi_dpt/udi_dpt.h*

A future version of this document will explain the SCSI HBA source code in detail. For more information, see the UDI SCSI Specification for a description of the SCSI Metalanguage used in this driver, which implements a UDI HBA Driver for DPT SmartCACHE and SmartRAID Generation III and IV controllers (see the *README* file above for supported models). Also see the Project UDI Web Site for a slide presentation on *UDI SCSI Drivers*,

# Building a UDI driver from source

UDI drivers are built from source files using the **udibuild**(1) command. Once you have coded the *udiprops.txt* file, header files, and C language source for your driver, use **udibuild** to build a driver binary.

The way in which your driver is built is largely under the control of the compiler directives in the *udiprops.txt* file. See ``Build instructions" and Build−Only Properties in the *Core Specification*.

Once you have built a driver binary, see ``Packaging, Installing, and Configuring a UDI Driver" for how to get the UDI driver into the running kernel on the target system. Also see UNRESOLVED XREF−0.

# Packaging, Installing, and Configuring a UDI Driver

Once you have built your driver code using **udibuild**(1), you need to create a UDI driver package directory hierarchy for installation on the target.

UDI drivers are packaged for installation using the **udimkpkg**(1) command,

Once packaged, the driver is installed and configured into the UDI environment using **udisetup**(1M). Additional commands on the target system may also be required after **udisetup**(1M) to complete the configuration. (such as **modadmin**(1M), **dcu**(1M), or **idbuild**(1M) on UnixWare 7).

The resulting driver package directory hierarchy can be further packaged for distribution using the native packaging tools on the target system. The installation of such a driver package then becomes a two−step process: installing the native package using the native packaging tools (for UnixWare 7, **pkgadd**(1M)), and then installing and configuring the driver into the kernel using **udisetup**(1M) (and other necessary commands).

The running of such configuration commands as well as **udisetup** could be done from an installation script in the native package to automate installation of a driver. For UnixWare 7, these commands might appear in the **postinstall**, **preremove**, or **postremove** scripts.

When testing a new driver, you will usually omit the native packaging step, and configure the driver into the kernel manually.

See ``Packaging your software applications" for a description of the UnixWare 7 packaging tools, and *UDI Overview and Configuration*. for a description of how to use **udisetup**(1M).

See UNRESOLVED XREF−0 to see how the sample UDI drivers are installed on UnixWare 7.