Uniform Driver Interface

UDI Core Specification
Version 1.01

Volume I
(Chapters 1-18)

`http://www.project-UDI.org/specs.html`

UDI Core Specification

Abstract

The UDI Core Specification defines the core set of interfaces and semantics that are available to all UDI drivers and that are required to be provided in all UDI environment implementations. This book also defines the fundamental UDI architecture and interface requirements, and is the normative specification upon which all other UDI specifications depend. Additional UDI specification books are or will be defined as outlined in Chapter 2, *"Document Organization"*, as optional extensions to this specification.

UDI drivers and libraries must be written to conform to this specification, and can assume that all services described herein are available.

The intended audience for this book includes UDI driver writers, environment implementors, and metalanguage implementors, as well as developers of additional UDI definitions such as bus bindings and ABI bindings.

The UDI Core Specification is divided into two volumes for ease of handling. Volume I contains Chapters 1-19. Volume II contains Chapters 20-34 and the Appendices.

Status of This Document

This document has been reviewed by Project UDI Members and other interested parties and has been endorsed as a Final Specification. It is a stable document and may be used as reference material or cited as a normative reference from another document. This version of the specification is intended to be ready for use in product design and implementation. Every attempt has been made to ensure a consistent and implementable specification. Implementations should ensure compliance with this version.

Copyright Notice

Acknowledgements

The authors would like to thank everyone who reviewed working drafts of the specification and submitted suggestions and corrections.

The authors would especially like to thank their significant others for putting up with the many hours of overtime put into the development of this specification over long periods.

Thanks to the following folks who contributed significant amounts of time, ideas, or authoring in support of the development of this specification or in working on the prototype implementations which helped us validate the specification:

Richard Arndt (IBM)
Bob Barned (Lockheed Martin)
Mark Bradley (Adaptec)
Darren Busing (Adaptec)
Steve Bytnar (STG)
Thomas Clark (Sun)
Deven Corzine
Jack Craig (SCO)
Betty Dall (HP)
Tim Damron (IBM)
Burkhard Daniel (STG)
Don Dugger (Intel)
Mark Evenson (HP)
Barry Feild (SCO)
Scott Feldman (Intel)
Mike Firman (STG)
Kurt Gollhardt (SCO)
Bob Goudreau (Data General)
James Hall (SCO/Sun)
Jim Heidbrink (Lockheed Martin)
Chris Herzog (STG)
Chris Ilnicki (HP)
Bret Indrelee (SBS Technologies)
David Kahn (Sun)
Matt Kaufmann (SCO)
Andrew Knutsen (SCO)
Ahuva Kroizer (Intel)
Man Fai Lau (SCO)
John Lee (Sun)
Robert Lipe (SCO)
Mike Lyons (IBM)
Alex Malone (DEC)
Lynne McCue (IBM)
Bill Nicholls
Guru Pangal (Starcom)
Mark Parenti (DEC)
James Partridge (IBM)
Scott Popp (SCO)
Hiremane Radhakrishna (Intel)
John Ronciak (Intel)
Kevin Quick (Interphase)
Larry Robinson (Adaptec)
Andrew Schweig (STG)
Sam Shteingart (HP)
Ajmer Singh (SCO)
James Smart (Compaq)
Pete Smoot (HP)
David Stoft (HP)
Rob Tarte (Pacific Codeworks)
Wolfgang Thaler (Sun)
Ramaswamy Tummala (Starcom)
Linda Wang (Sun)
Kevin Van Maren (Unisys)
Mike Wenzel (HP)

Countless people have helped in one way or another and any omissions or errors on our part in the list above are just that: omissions or errors on our part.

Thanks to Kevin Quick and the folks at Interphase for hosting the Interoperability events which have provided a great venue for validating both prototype and production UDI products.

Finally, thanks to David Roberts (Certek Software Designs) for designing the Project UDI logo.

Table of Contents

Volume I

Section 2: Architecture

Section 3: Core Services

Volume II

Section 4: Core Utility Functions

Section 5: Core Metalanguages

Section 6: MEI Services

Section 7: Packaging and Distribution

Section 8: ABI Bindings

Section 9: Appendices

List of Reference Pages by Chapter

Volume I

Chapter 9 Fundamental Types

Chapter 10 Initialization

Chapter 11 Control Block Management

Chapter 12 Memory Management

Chapter 13 Buffer Management

Chapter 14 Time Management

Chapter 15 Instance Attribute Management

Chapter 16 Inter-Module Communication

Chapter 17 Tracing and Logging

Chapter 18 Debugging Services

Volume II

Chapter 20 String/Memory Utility Functions

Chapter 21 Queue Management Utility Functions

Chapter 22 Endianness Management Utility Functions

Chapter 24 Management Metalanguage

Chapter 25 Generic I/O Metalanguage

Chapter 26 Diagnostics Support

Chapter 28 Metalanguage-to-Environment Interface

Alphabetical List of Symbols

UDI Core Specification

Section 1: Overview

Introductory Material *1*

1.1 Introduction

The Uniform Driver Interface (UDI) specifications define a complete runtime environment for device drivers. This includes the complete set of services and other interfaces needed by a device driver to control its device or pseudo-device, and to interact properly with the rest of the system in which it operates. This runtime environment in which a UDI driver operates is referred to as the *UDI environment*.

The UDI interfaces allow UDI drivers to be completely portable from one OS or platform to another. All OS and platform specifics are contained in the UDI environment implementations for those OS's and platforms and are thus isolated from driver code.

These specifications also define requirements on UDI build environments used to build UDI drivers and packages from source. Some environments will be both runtime environments and build environments.

1.2 Scope

The UDI Core Specification defines the core set of UDI interfaces that are available to all UDI drivers and that are required to be provided by all UDI environment implementations. The UDI interfaces defined in this document represent the interfaces that are always provided to a UDI driver by the UDI environment and may safely be used by any UDI driver implementation.

The UDI specifications are defined in terms of the C language and establish a C language binding for the UDI interfaces. Thus the UDI specifications support device driver portability at the C source code level. When combined with a UDI ABI binding, the UDI specifications support device driver portability at the binary level.

Other language bindings could be created for UDI; some of the syntax would differ, but the principles and the UDI-defined names would be the same. In particular, UDI interfaces can be accessed from assembly language code, as long as the shape of data structures and calling conventions are made to match the C language conventions for the target platform.

1.3 Normative References

The UDI Core Specification references the following non-UDI standards, listed below. These standards contain provisions that, through reference in this document, constitute provisions of the UDI Core Specification.

1. ISO/IEC 9899-1990 (ISO C Programming Language Standard).
2. ISO 10646 (Unicode), Annex P (UTF-8 Character Encoding Standard).

3. ISO/IEC 9945-1 (POSIX locale specifier format).
4. ISO 639-2/T (Language Codes).
5. ISO 3166 (Country Codes).
6. IEEE Std. 1003.1-1988 (Archive/Interchange File Format)
7. ISO 9960 (CDROM filesystem specification).
8. IETF RFC 1071 *“Computing the Internet checksum”*
9. IETF RFC 1141 *“Incremental updating of the Internet checksum”*
10. IETF RFC 1624 *“Computation of the Internet Checksum via Incremental Update”*
11. IETF RFC 1936 *“Implementing the Internet Checksum in Hardware”*

Other UDI specification books rely on the UDI Core Specification, and may rely on additional non-UDI standards. For example, the UDI SCSI Driver Specification relies on the ANSI SCSI Standards, and the PCI Bus Binding depends on the PCI Local Bus Specification. The degree to which a UDI specification depends on these other standards, or specific versions of those standards, is indicated in the applicable UDI specification document.

1.4 Conformance

1.4.1 Environment Conformance

A conforming UDI environment implementation shall provide all of the interfaces defined in the UDI Core Specification, with their associated rules and semantics, including the architectural requirements defined in “Section 2: Architecture”. Environments that support related functionality that is covered by other UDI specifications shall also provide all of the interfaces and semantics defined in those specifications.

A conforming environment shall also provide the header file “`udi.h`” for the interfaces in the UDI Core Specification, and additional header files as required by other UDI specifications supported by the environment. These header files must be ISO C conforming programs.

To provide portability guarantees to UDI drivers, conforming UDI environment implementations must provide all the interfaces defined in the UDI Core Specification. However, static environments, in which it is not possible to load new drivers or otherwise modify the configuration of the system, may know *a priori* that certain interfaces are not needed by any of the applicable drivers. Such a static environment that doesn’t completely implement the relevant UDI specifications is not considered *fully conformant*; it is however considered *statically conformant* if it conforms to the requirements of the UDI interfaces that are applicable to it – i.e., if the applicable drivers are completely conformant UDI drivers. Note that in this case the applicable drivers would be portable to any *fully comformant* UDI environment, but not necessarily to another *statically conformant* environment.

Note – UDI environment implementations may vary in the way that they implement a particular UDI interface, the amount of internal debugging and interface consistency checking provided, the underlying address or protection or synchronization domain in which UDI drivers execute, etc.

However, as defined above, fully conformant UDI environments must implement the full set of interfaces defined in this Core Specification, and all UDI environments must adhere to the requirements of the UDI architectural model as defined in this Specification.

1.4.2 Device Driver Conformance

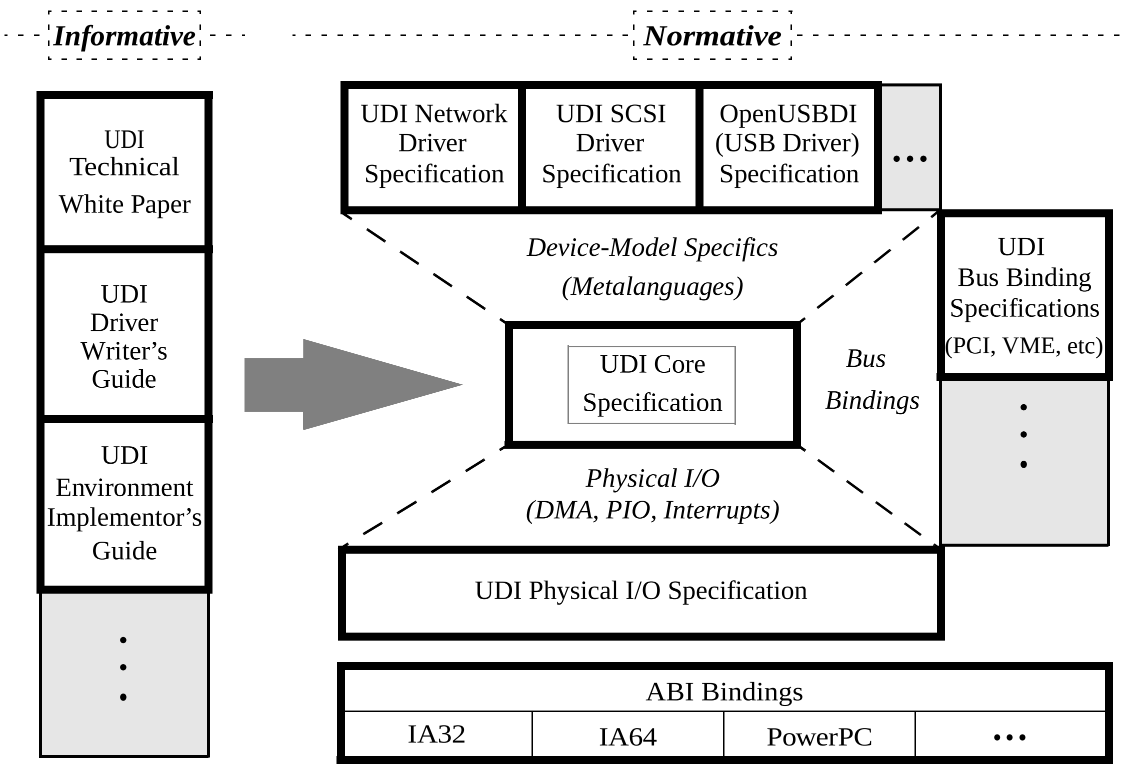
A conforming UDI device driver implementation shall not, at the source code level, reference any interfaces external to the driver except those defined in the UDI specifications or exported explicitly to drivers via UDI-defined mechanisms. A conforming UDI device driver shall also follow all the rules and semantics defined for the use of these UDI interfaces. In particular, conforming UDI drivers must adhere to the general requirements regarding UDI_VERSION, header files, and the use of ISO C, as defined in Chapter 8, *"General Requirements"*.

Document Organization 2

2.1 Overview of UDI Documentation

The UDI documentation is organized into several related specifications. The UDI Core Specification is mandatory for all UDI implementations; all other UDI Specifications define optional sets of interfaces that may be available for supporting specific sets of functionality.

In addition to the UDI Specifications, there are several other documents referred to as Guides or White Papers. These Guides and White Papers provide additional information and descriptions for using UDI but are supplementary to the UDI Specifications and define no additional interfaces. All UDI Specifications shall be considered as normative material and all Guides and White Papers shall be considered to be informative material.



This picture is intended to show the types of books in the UDI document set: driver-type specific specifications, bus bindings, ABI bindings, physical I/O interfaces, etc., all of which are centrally supported by the UDI Core Specification. Not all of the books mentioned in this figure will be available coincident with the publishing of the UDI Core Specification.

2.2 Overview of the UDI Core Specification

2.2.1 Core Specification Sections

The UDI Core Specification is organized into the following main sections:

Overview	The current section, providing an overview of the UDI specifications.
Architecture	Defines the fundamental UDI architectural concepts, including the UDI execution model, data model, function call types and associated standard calling sequences.
Core Services	Defines the core environment services that all UDI environments are required to provide.
Core Utility Functions	Defines the core utility functions that all UDI environments are required to provide. Some utility functions that are very specific to particular environment services are defined instead in the appropriate chapter of the Core Services section, but are also required to be provided by all UDI environments.
Core Metalanguages	Provides an introduction to the concepts, requirements, and conventions applicable to all metalanguages; defines the interfaces that are common to all metalanguages; and defines the core metalanguages which all UDI environments must provide.
MEI Services	Defines the interfaces needed by portable metalanguage libraries.
Packaging and Distribution	Defines the methods by which UDI drivers are packaged and distributed through electronic or physical means for installation into target systems.
ABI Bindings	Describes the type of material that would need to be specified by an ABI specification for UDI.
Appendices	Contains the glossary and auxiliary details not covered in the main specification.

2.2.2 Core Specification Topics

Some of the topics covered in the UDI Core Specification include:

- Memory management
- Buffer management
- Timer functions
- Context and execution control (Control Blocks)
- Tracing and Logging functions
- Utility functions
- Configuration, Distribution, and Packaging

This Core Specification also defines the set of data types and objects used within a UDI environment and the execution model for UDI drivers running in a UDI environment.

Topics not found in this UDI Core Specification but covered in other optional UDI specifications include:

- Non-Core Metalanguages (e.g., SCSI, Networking)
- Physical Device Access Interfaces (e.g., PIO, DMA, and Interrupts)
- Bus Bindings (e.g., PCI, EISA, etc.)
- ABI Bindings (e.g., IA32, IA64, PowerPC, etc.)

These topics are not found in this UDI Core Specification because they are specific to the needs of a given I/O technology, device class, hardware or bus type, or processor type; the Core Specification provides interfaces that can apply to any type of driver or hardware. A typical UDI driver for a PCI adapter would make use of UDI Specifications for Physical Device Access and the PCI Bus Binding, while a compliant UDI driver for a USB device (an OpenUSBDI driver) would use the USB Metalanguage defined in the OpenUSBDI Specification, but no physical device access or physical I/O bus bindings.

Terminology 3

3.1 Introduction

This chapter defines common terminology used in the UDI Core Specification. There are two categories of terminology defined here: terms whose purpose is to provide directives on the behavior, features and semantics of the UDI Specification, called *directive terms*; and *common terms* that are not UDI-specific. This chapter clarifies how these terms are used in this Specification.

UDI architectural terms and other UDI-specific terms are defined in the Glossary in Appendix A.

3.2 Definitions

3.2.1 Directive Terms

These terms provide directives on the behavior, features, and semantics of the UDI Core Specification. Other UDI specifications are encouraged to reference and use these directive terms for consistency with the Core Specification, but may choose to define their own set of directive terms (e.g., for consistency with a related hardware standard).

can indicates that the existence of a particular feature or behavior of a UDI driver is optional; UDI drivers may choose whether or not to use the feature or behavior.

ignored indicates that the contents of a particular field cannot be usefully examined; UDI drivers must not examine such fields.

illegal indicates a violation of the Specification. The consequences of illegal actions on the part of a UDI driver are implementation dependent, and may include abrupt termination of the driver or catastrophic system failure.

implementation-dependent indicates that a particular feature or behavior is not consistent across all environment implementations and must not be relied upon by UDI drivers.

invalid indicates a condition which is not valid within a given context.

may indicates that the existence of a particular feature or behavior of the UDI environment is optional; UDI drivers must not rely on the existence of the feature or behavior. To avoid ambiguity, the antonym of may is expressed as ***need not***, instead of *may not*.

must indicates a requirement on a UDI driver.

shall indicates that the feature or behavior described is a requirement on the UDI environment; UDI drivers can rely on the existence of the feature or behavior.

should indicates that a feature or behavior is strongly recommended but not mandatory.

unspecified indicates that the contents of a particular field are not consistent across all environment implementations and must not be relied upon by UDI drivers.

will Same as ***shall***.

3.2.2 Common Terms

The following are terms which are commonly used in the industry in a manner similar to the usage in this Specification; these definitions clarify how these terms are used in this Specification. Other UDI specifications are encouraged to reference and use these definitions for consistency with the UDI Core Specification.

adapter I/O hardware which provides a specific function or connectivity/bridging capability and which is accessed via a system bus. Also called a ***card***, a ***controller,*** a ***NIC,*** or an ***HBA***. The adapter is typically accessed via programmed I/O and may be capable of generating interrupts or DMA activity (or be capable of being a DMA target).

adapter driver device driver software responsible for managing an adapter.

address domain an address space wherein the same address always refers to the same memory object. Two software modules are in different address domains if the same address does not refer to the same memory when used in each module, or if an address that is accessible to one is not accessible to the other. Thus, it is useless to pass an address across domains. Whenever information is passed across a domain boundary, all pointers must be converted, either by copying the information to which they point or by remapping the same physical memory to a new virtual address.

ABI Architected Binary Interface. This is a set of binary bindings for a programming interface specification such as the UDI Core Specification. (When applied to applications rather than system programming interfaces, ABI is usually interpreted as Application Binary Interface.)

API Architected Programming Interface. This is a programming interface defined in a UDI specification; e.g., a function call interface or structure definition with associated status or function codes, as well as associated semantics and rules on the use of the interfaces. (When applied to applications rather than system programming interfaces, API is usually interpreted as Application Programming Interface.)

ANSI American National Standards Institute. ANSI is a United States national standards body, and is the sole U.S. representative and dues-paying member of the two major non-treaty international standards organizations, the International Organization for Standardization (ISO), and, via the U.S. National Committee (USNC), the International Electrotechnical Commission (IEC).

big endian data storage format in which a multi-byte data value is stored with the most-significant data byte through least-significant data byte in the lowest through highest byte addresses, respectively. This is the storage format traditionally used by the Motorola 680x0, HP PA-RISC, Sun SPARC, and AMD 29000-series processors.

blocking the process of suspending a thread of execution until an event occurs, possibly switching to other threads in the meanwhile. To the programmer, this appears to be a procedure call that may not return for a long and indeterminate amount of time. Also known as *sleeping*. Can also be used as an adjective describing OS service calls that can cause such a suspension.

UDI uses *asynchronous service calls* instead of blocking service calls, so there is no way for a UDI-compliant driver to block. Note that this does *not* affect the embedding OS and its users' applications, since native synchronous or asynchronous operations are supported by external mapper implementations independently of the UDI drivers.

byte A unit of data storage made up of eight binary digits (bits). An *octet*. UDI does not use the archaic meaning of "byte" to refer to anything other than 8-bit data units.

capability domain see ***protection domain***.

card see ***adapter***.

ccNUMA Cache Coherent Non-Uniform Memory Access, an architecture for highly parallel systems with shared memory of varying latencies.

controller see ***adapter***.

data encapsulation a method of maintaining the functional independence of separately designed and/or compiled code modules by hiding all the data relevant to a module in an abstract object that may only be manipulated by calls to the module itself.

device physical hardware, under software control, which is typically attached either directly to an I/O bus or to an auxiliary bus (e.g. SCSI) attached to a directly-connected adapter. The device typically combines a hardware controller with the raw mechanism (disk controller with disk, display controller with frame buffer, etc.).

device driver a software module that turns I/O requests into control of a specific physical device or a hardware or protocol interface. A device driver contains all the device-specific code necessary to control and communicate with its hardware or logical function and provides a standard interface to the rest of the system. A driver may or may not control "raw" hardware.

device endianness the endianness of the device's accesses to memory (typically either its own memory or system memory).

device ID a numeric or string value with a device-interconnect specified format used to provide device identification. Usually stored in I/O card ROM.

device instance an instance of a physical device, such as an adapter, or a pseudo-device. A single UDI driver may manage multiple device instances, however, UDI implements instance independence which makes these multiple device instances invisible to each other.

device model a semantic model for accessing and controlling a particular class of I/O device, such as SCSI or Network.

device node a node in the *device tree*.

device tree an abstract data structure that represents the physical and logical topology of an I/O system. This data structure is usually thought of as an n-ary tree structure, but can occasionally have multiple parents for the same node, so is really an acyclic directed graph. Even with multiple parents, however, the graph ultimately has a single root. Each node represents a device instance.

domain A physical or logical area that shares some common characteristic. See ***address domain*** and ***protection domain***.

driver see ***device driver***.

driver endianness the endianness of the driver's accesses to its data. This is sometimes referred to as the endianness of the driver's region.

driver instance a set of one or more regions, all belonging to the same driver, that are associated with a particular instance of the driver's device. There may be multiple instances of a given driver, one for each physical device controlled or (in the case of software-only drivers) one for each logically-separate replication of a function. Each active device node has exactly one corresponding driver instance.

embedding system the surrounding Operating System in which the UDI environment is contained.

endianness see ***driver endianness***, ***device endianness***, ***protocol_endianness***

entry point a function within a driver that is called from outside that driver.

environment the UDI Environment: a description of all interfaces surrounding the driver and the implementation thereof. Includes system services, scheduling and synchronization, as well as inter-module communication mechanisms.

FIFO First In, First Out

handle an opaque reference to an environment object that must not be directly referenced by drivers. See the UDI architectural definition of ***handle***, ***transferable handle*** and ***nontransferable handle*** in the Glossary.

HBA Host Bus Adapter. Another name for an ***adapter***, but most commonly used for SCSI adapters.

informative provides information, guidance, instruction. Informative documentation describes, instructs, and provides guidance on the use of required interfaces, but does not define those requirements; normative documentation defines the requirements. See also **normative**.

instance a single, logically separate replication (associated with a thread of execution, not a physical copy of code) of a module along with its associated data, methods and services (see "driver instance").

IEC International Electrotechnical Commission.

IETF Internet Engineering Task Force.

ISA 1) Instruction Set Architecture. Defines the binary machine language syntax and semantics for a particular type of processor or processor family.

2) Industry Standard Architecture. An I/O bus type originally designed for the IBM AT and used in many PCs. Also known as the ATA bus.

ISO International Organization for Standardization. ISO is a worldwide federation of national standards bodies from some 130 countries, one from each country. Note that ISO is not an acronym, but is an international term used to refer to the International Organization for Standardization independent of national language, and is derived from the Greek word meaning "equal".

little endian data storage format in which a multi-byte data value is stored with the least-significant data byte through most-significant data byte in the lowest through highest byte addresses, respectively. This is the storage format used by Intel and Digital processors.

natural alignment alignment of a field in a structure on a boundary (offset) within the structure which is a multiple of the size of the field's data type. Thus, a naturally aligned 1 byte field begins on a byte boundary; a naturally aligned 2 byte field on a 2 byte boundary, etc.

non-blocking an interface or execution model which does not require blocking (see ***blocking***).

normative establishes a standard or norm. Normative documentation defines required interfaces and semantics. Often the term **normative** is used in juxtaposition to the term **informative**. E.g., the UDI Specifications are normative; the UDI white papers and implementation guides are informative.

object an instance of a data structure, encapsulating a logical instance of a software function, that is operated on with specific, defined function calls.

opaque type a type of data object whose fields are not visible to drivers, used in defining UDI data structures and handles.

Operating System the primary code executing on a hardware platform which is responsible for managing that platform and providing the environment under which applications may be run on that platform.

OS Operating System.

platform the overall system that embeds UDI, consisting of all the hardware, together with the native operating system.

protection domain *(*also called "capability domain"): a collection of software which shares the same memory access protection level (e.g. kernel v.s. user). When it is necessary for software running in one protection domain to invoke an operation in another domain, special provisions must be made in the environment for checking permissions and passing parameters across the domain boundary.

protocol endianness the endianness of hardware protocol data such as SCSI commands or networking protocol headers.

pseudo device a logical "device" which has no associated hardware. Pseudo-device drivers present the view of a device to their children even though they do not control an actual device. Pseudo-device instances are roots of their own device trees, separate from the hardware device tree.

RFC Request For Comment.

SCSI Small Computer Systems Interface, a standard storage architecture and protocol.

sleeping see ***blocking***.

thread an instance of execution consisting of a procedure stack and OS scheduling structures. A thread, together with an address space and permissions, is equivalent to a traditional “process”. On multiprocessor systems, multiple threads execute simultaneously.

trusted code code that the operating system is minimally suspicious of. Drivers are commonly trusted in that there are fewer run-time error checks included in the system interfaces in exchange for higher performance. UDI, however, allows for environments with low trust in drivers, and gives such environments the opportunity to do any error checking they might wish.

UDI Uniform Driver Interface. In some contexts, this is a short-hand term for the UDI environment and the entities in the embedding system that the UDI environment supports.

UDI Core Specification

Section 2: Architecture

Execution Model 4

4.1 Introduction

UDI drivers are prepared for execution in a target environment by compiling driver source code for a target system, either directly on the target system or by separate compilation into relocatable object files. The result is one or more independent executable modules, called *driver modules* where each module is comprised of a set of object files (as defined by the driver's static driver properties file, see Chapter 30).

4.2 Driver Object Modules

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, as described in "Multi-Module Drivers" on page 4-2. 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. The driver specifies its modules via the "module" property declaration in the driver's `udiprops.txt` configuration file (see Chapter 30).

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 **udi_init_info** on page 10-3.) There are no global entry points into UDI drivers; all entries are through function pointers in `udi_init_info`.

4.3 Driver Instances

In general, an *instance* refers to a specific occurrence of a generic item. An instance of a device, or *device instance*, refers to a specific occurrence of that device in a system. An instance of a driver, or *driver instance*, refers to the driver code attached to a particular device or pseudo-device combined with a set of driver state (control values, queues, control memory, etc.) that serve that device. The driver state associated with a particular device is often referred to as *per-instance state*. Even though driver instance are logically separate, environment implementations may use a single copy of the driver code for multiple instances of the same driver.

4.4 Regions

A driver instance is composed of one or more well-defined sub-divisions called *regions*. A region is implicitly serialized by the UDI environment, and thus defines the unit of concurrent execution. There is no shared memory between regions. This allows driver regions to be separately replaceable and locatable (e.g., in different address or protection domains), supporting the *instance-independence* and *location-independence* of UDI drivers. (See Chapter 5, *"Data Model"* for a discussion of *instance-independence*, and Section 4.9 for a discussion of *location-independence*).

4.4.1 Driver Partitioning

Driver writers need to consider the partitioning of their driver into regions when designing the driver. A simple driver may be composed of a single region; more complex drivers may be composed of multiple regions. The former is called a *single-region driver*; the latter a *multi-region driver*; but in either case it must be emphasized that regions are sub-divisions of a driver instance.

Many driver instances are composed of multiple state machines, roles, or cooperating functions. For example, a driver may have a somewhat distinct state machine that handles outbound packets, another to handle inbound packets, a third to handle timeouts, etc. When a driver is designed, such separable pieces of the driver may be defined to run in separate regions.

When a driver is instantiated, an initial region is created by the environment; this is called the driver's *primary region*. If requested by the driver, additional regions, called *secondary regions*, are also created.

4.5 Multi-Module Drivers

Each module in a multi-module driver contains the code and static data for one or more mutually-exclusive region types. Each module contains its own `udi_init_info` structure.

The primary module must contain all of the code and static data for the primary region and may contain the code and static data for one or more secondary regions as well. Other modules ("secondary modules") must not specify a `primary_init_info` structure in their `udi_init_info` initialization structures.

4.6 Channels

A *channel* is a point-to-point communication and connection mechanism between two regions. This point-to-point design allows for simplicity of connection build-up and tear-down and low-overhead of the communication path. Attached to each end of a channel (a *channel endpoint*) is a set of driver entry points called a channel operations vector or *ops vector.* The definition of the channel operations implemented by these entry points (along with associated service routines, attribute bindings, etc.) for a particular type of channel is referred to as a *metalanguage*.

Channels form the basis of all communication between regions, since regions cannot share data directly. Channels are also used to communicate with the Management Agent and other logical entities within the environment that act as though they were executing in UDI driver regions. (The Management Agent manages driver and device instance configuration, and is described in more detail in Chapter 24, *"Management Metalanguage"*.)

4.7 Driver Execution Environments

All execution of driver code within UDI is done on a *per-instance* basis. Each instance is said to execute in the *context of a region* or in *region context*, and has access to driver and environment state associated with the particular region for which it was invoked.

Note – The UDI Physical I/O Specification defines a special type of region, called an *interrupt region*, that has additional restrictions in its execution environment.

4.7.1 Non-Blocking Model

While executing in the context of a region, UDI drivers are non-blocking; i.e. any service call that may require access to external resources or delayed completion is defined with a callback function so that the UDI environment does not have to block the thread from which the driver was called while waiting for resources. See the discussion of asynchronous service calls below for additional details.

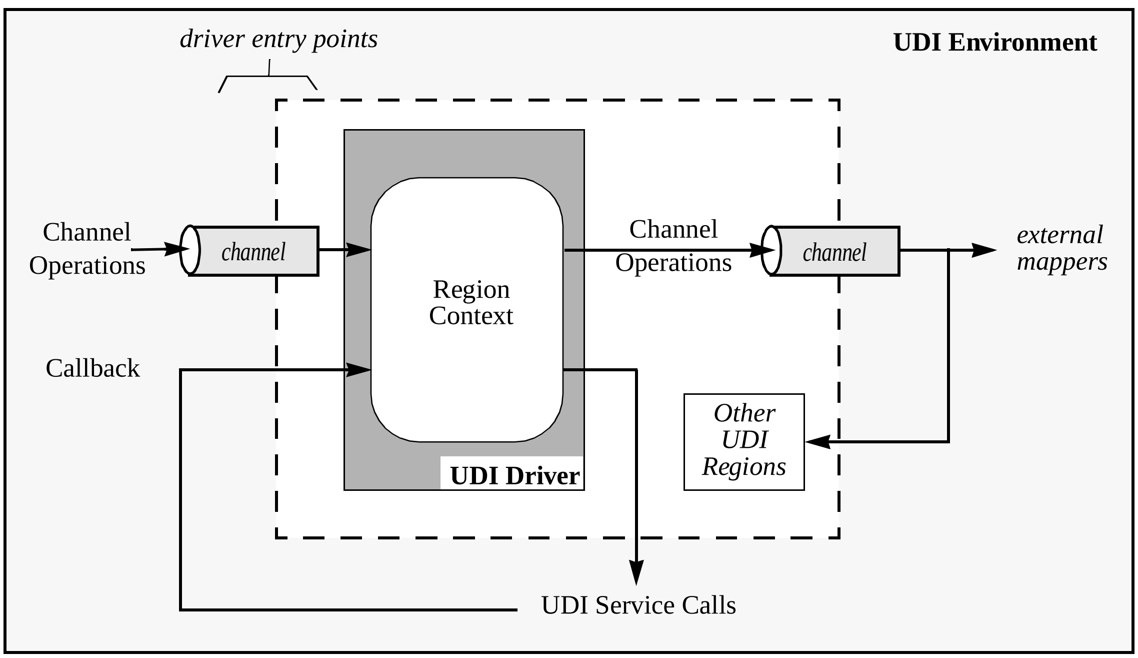
4.8 Function Call Classifications

Function calls used by UDI drivers can be categorized as either calls into the driver (*driver entry points*), calls from the driver to the environment to request environment services (environment *service calls*), or calls over channels between driver or environment regions (metalanguage-specific *channel operations*). Channel operations are specified in terms of both the caller side (*channel operation invocation*) and callee side (*channel operation entry point*). Some service calls complete asynchronously and re-enter the driver via *callback* functions.

A UDI driver executes within the context of a region at all times and all driver entry points (channel operation entry points and callbacks from service calls) are called with region context. All channel operation invocations and service calls are called within the context of a region.

UDI environments also provide a set of *utility functions*, which are convenience functions defined for the driver. These convenience functions do not perform any operations that the driver could not do directly via its code and therefore do not set or test any environment state. These utilities may be implemented as environment function calls or as macros that result in inline code in the driver itself.

The following figure illustrates these function call categories.



4.8.1 Service Calls

UDI drivers request services of the UDI environment by calling environment service calls. These are functions provided by the environment and exported to all UDI drivers.

Service calls are common to all types of drivers. They are not metalanguage-specific.

4.8.1.1 Synchronous Service Calls

Service calls that can reasonably be expected to complete "immediately" (or at least in a small, finite amount of time) on all environment implementations are specified in the form of *synchronous service calls*. Synchronous service calls run entirely in the context of the calling region and complete all processing required to satisfy the request before returning to the calling driver.

Synchronous service calls are identifiable by the lack of a ***callback*** argument.

4.8.1.2 Asynchronous Service Calls

Service calls that might not complete "immediately" are specified in the form of *asynchronous service calls*. These need to operate asynchronously, returning results and/or completion status via *callbacks* (calls "back" into the driver) rather than as output parameters or return values from the service call itself. Any service call that might require allocation, access to external resources, or delayed completion shall be asynchronous. (This is required by UDI's non-blocking model—drivers executing in a region context cannot block the calling thread in order to wait for resources or I/O events.)

When an asynchronous service call returns to the calling driver, the service may or may not be complete. The associated callback may happen immediately, before the service call returns to the driver, or later, after the driver itself returns to its caller and the resource subsequently becomes available. The only restriction on callbacks is that they must not violate the region execution model and therefore delayed callbacks (ie. callbacks that occur after the service call has returned to the driver) must wait for any current execution in the region to complete before being scheduled to execute in that region (also see Section 5.5, "Implicit MP Synchronization"). Immediate callbacks may execute immediately before returning from the service call without violating region synchronization because only one thread of execution remains active in the region.

In the case of a delayed callback, the environment needs to be able to queue the pending request or callback and the driver must be able to continue the operational context that indicated the asynchronous call. In order to provide space for queuing the request (if needed) and maintaining the driver context a control block is passed to each asynchronous service call.

Control blocks are a finite resource that are provided to or allocated by the driver. Between the time that the service call is made and the time that the callback is called, the control block is under the control of the environment and must not be used in any way by the driver, except to cancel the service call (see **udi_cancel** on page 11-13). Only one allocation request must be pending on a given control block at a time. Any attempt to start another request using a control block that is already in use will produce indeterminate results.

Any type of control block can be used for this purpose. Because of this, the service calls are defined in terms of a least-common-denominator control block, which is itself part of every actual control block. This *generic control block* is denoted with the data type, `udi_cb_t` (see page 11-3).

4.8.2 Channel Operations

Channel operations are invoked by a driver or by the environment on one end of a channel and result in a procedure call to an operation entry point in another region at the other end of the channel.

Like asynchronous service calls, channel operations require control blocks, so the environment can queue them if the target region is busy. Control blocks used with channel operations can also be used by the environment to marshal and unmarshal[1] other parameters and data passed to the operations if they need to be queued or transferred between separate domains.

Any channel operations queued by the environment will be delivered in FIFO order to the region; the region is unaware of the queueing. The environment must ensure that operations queued to a region are delivered in FIFO order relative to the channel on which they are queued, but there is no ordering between channels, therefore requests arriving to a region on different channels may be presented to the region in any order as long as they order preserves the FIFO ordering of the individual channels. There are also no ordering requirements imposed on callbacks which enter the region as a result of asynchronous call completions. If a metalanguage requires that the driver process operations in the order received, the metalanguage specification will indicate this requirement and the driver must insure that ordering is maintained with regards to forwarding or responding to those operations.

4.9 Location Independence

The UDI execution model also provides for location independence, which is the ability to instantiate and execute a driver's code without requiring that code to be run in a particular domain (e.g. kernel, user, etc) or even on the same node in a cluster. Within UDI the fundamental unit of execution is defined as the region (see Section 4.4) and this is therefore the level at which location independence can be applied. Two regions within the same driver may be located independently since there is no shared memory or other external access by either region outside of the UDI specified interfaces (which are expressly designed to allow for location independence).

The location of a region instantiation is determined by the environment implementation and may be affected by a number of considerations, including environment architecture, resource availability or utilization, and driver-specified region attributes (see Section 30.6.8, "Region Declaration").

4.10 Driver Faults/Recovery

Any improper usage of the UDI service calls or illegal usage of UDI control blocks or other illegal actions on the part of a UDI driver (see the definition of "illegal" in Section 3.2.1, "Directive Terms") will lead to implementation-dependent and, in some environments, indeterminate results. Since the driver cannot in general be expected to recover from its own misbehavior, it's left to the environment to determine appropriate actions in such cases. One such action, which has been defined and enabled in various parts of the UDI Specifications, is to abruptly terminate the corresponding driver instance. Such abrupt termination is sometimes referred to as being "region-killed" or "instance-killed", the latter referring to region-kills of all the regions in an instance.

4.10.1 Overview of Region-Kill

When an illegal action is detected by the UDI environment that results in an abrupt termination (region-kill) of the offending region, the region will typically be exited immediately and will be marked to disallow any further entries into the region. If the offending region is part of a Physical I/O Driver that has registered a "PIO Abort Sequence" handle via a `udi_pio_abort_sequence` then the associated PIO sequence will be executed to shut down the corresponding device. All channels attached to the

1. *Marshalling* is the activity of identifying and possibly collecting all of the information related to the request so that the information may be moved to a different domain (via an unspecified mechanism) where it will be *unmarshalled* back into operational form to deliver to the recipient.

region will then be closed, notifying any neighbors that the region has abruptly terminated. Any control blocks marked "recoverable" (via `UDI_MEI_OP_RECOVERABLE`) will be returned to the initiating region via the corresponding response operation, where applicable, along with associated transferable objects. Lastly, all other data objects owned by that region, including control blocks, allocated memory, and buffers, will be freed and the region destroyed. As a result of the channel closes, any neighboring regions must cease operation on channels to that region.

4.10.2 Improper Channel Operation Usage

UDI channel operations involve the source region, the associated metalanguage library, environment support code, and the target region. Illegal actions detected by the environment while between regions doing a channel operation may result in a region-kill of the source region. Illegal actions detected by the metalanguage library (considered to be a portable metalanguage library for purposes of this discussion, since a non-portable metalanguage library can be considered to be part of the environment) should result in a call to `udi_mei_driver_error` by the metalanguage library code. This may result in either an immediate error response operation back to the source region with status `UDI_STAT_NOT_UNDERSTOOD` or a region-kill operation by the environment.

Illegal actions by the source driver that end up being detected in the target driver must result in an appropriate status such as `UDI_STAT_NOT_UNDERSTOOD` being sent back in the corresponding response operation, when applicable, to the offending source region, and as defined in UDI Tracing and Logging, a call to `udi_log_write`.

4.11 Metalanguage Model

4.11.1 Metalanguage Roles

Each metalanguage defined in the UDI environment is typically bilaterally asymetric. In other words, the region at one end of the channel will typically initiate operations and the region at the other end of the channel will typically respond to operations. This is exemplified by the metalanguage-specific bind operation where one side initiates the metalanguage-specific binding and the other side responds to that binding.

In this context, each end of the channel is referred to as playing a *role* in the overall metalanguage design. This role is often referred to as either the “parent” role or the “child” role based on the typical device node tree representation of the device drivers, but this sense of orientation does not always apply.

Each metalanguage will therefore define the role for each end of the channel and will typically define the metalanguage operations and states in terms of those roles.

Example metalanguage roles:

- Bridge Metalanguage: interrupt dispatcher and interrupt handler
- SCSI Metalanguage: HD (HBA Driver) and PD (Peripheral Driver)
- Network Interface Metalanguage: ND (NIC Driver) and NSR (Network Service Requester)

4.11.1.1 Management Metalanguage Roles

The Management Metalanguage is somewhat unusual in that there are three parties rather than the usual two: the Management Agent (MA), the Parent, and the Child. There are also three channels involved:

1. MA--Parent channel (a.k.a. the parent's management channel)
2. MA--Child channel (a.k.a. the child's management channel)
3. Parent--Child channel (a.k.a. the child's bind channel)

In this configuration the Management Metalanguage accordingly defines three roles: the "MA", the "parent", and the "child". For more information, see Chapter 24, *"Management Metalanguage"*.

Data Model 5

5.1 Overview

Data available to a UDI driver can be categorized as (1) module-global data, (2) per-instance data, (3) per-request data, or (4) function-local variables.

Module-global data has *driver-scope*; i.e., it is global to all instances of a driver within a given domain, and for that reason is sometimes called *domain-global data*. All non-automatic variables in the driver, whether local to particular functions or compilation units, or truly global to the driver, are considered module-global data. Module-global data is read-only throughout the execution of a UDI driver, regardless of whether or not it is declared with the ISO C "`const`" keyword. Environment implementations may choose to share a single copy of a driver's module-global constant data between multiple instances of that driver within a particular domain.

Per-instance data has *region-scope* and is often referred to as *region-local data* or simply *region data*. A driver instance is composed of one or more *regions*, each of which has its own private data which isn't visible to or shareable with other regions. The *region* data model allows drivers to be *instance independent,* meaning that the driver state for each device instance is independent of all other instances so that a new instance can be added at any time or an instance can be removed and the remaining instances will continue independently. It is also critical that, when a driver is entered on behalf of a particular device instance, it does not access any hardware of another device instance; this allows the driver instances to be independently bound to different CPU's (or, on ccNUMA configurations, CPU groups) or otherwise constrained to specific locations.

Any data objects allocated by the driver when executing within a driver region are attached to the region and are region-local.

Data that is not specific to a particular channel or individual operation is sometimes referred to as *region-global* data, since it is global to the region.

Data objects such as control blocks passed into the region from another region contain *per-request* data. The ownership of these objects is transferred to the target region and they therefore also become region-local to the target region and no longer accessible from the source region. Objects which can be transferred from one region to another are called *transferable.*

Function-local variables are C variables of function or block scope. C global variables (i.e., C variables defined outside of any function) may only be used for module-global data, and therefore must be read-only. It is recommended that all such variables be declared as static constants using the C language `const` and `static` keywords.

While executing in a region, module-global data space (including static variables with function scope as well as global variables) is read-only, but dynamically allocated region data is read-write.

5.2 Data Objects

In general, the term *UDI data objects* refers to allocated data objects which are obtained via a call to a UDI allocation interface. UDI data objects include driver-addressable memory areas, metalanguage control blocks, and opaque objects referenced via handles. UDI data objects have the following properties associated with them: scope, transferability, and opaqueness. The scope can either be *module-global* or *region-local* as described above. Secondly, region-local objects can either be *transferable* or *non-transferable* as described previously. Thirdly, UDI data objects can be *visible*, *semi-opaque*, or *opaque*. Allocated driver structures are *visible*; control blocks are *semi-opaque*; handles reference *opaque* objects. Visible and semi-opaque objects are both referenced by pointers; however, semi-opaque objects are defined such that the environment may—and probably will—store additional data, which is not available to the driver, before or after the driver-visible fields of the object.

5.2.1 Memory Objects

Blocks of driver-addressable memory may be allocated by the driver at any time, using `udi_mem_alloc`. Most allocated memory is private to the region that allocated it and cannot be transferred to other regions. However, drivers may also allocate *movable memory* blocks, which can be passed as arguments to channel operations and thus transferred to other regions. Once a movable memory block is “given away”, however, the original driver must no longer access it. Only one region at a time “owns” a movable memory block. Movable memory is allocated using `udi_mem_alloc` with the `UDI_MEM_MOVABLE` flag.

5.2.1.1 Using Memory Pointers with Asynchronous Service Calls

Some asynchronous service calls take pointers to driver memory objects as parameters. Since the environment might continue to access these objects after returning to the calling code in the driver (any time until the environment completes the service call by calling the driver’s callback routine), special care must be taken to avoid race conditions and corruptions that might happen if both the environment and driver were using the memory at the same time.

To avoid the possibility of such race conditions, UDI requires drivers to obey the following rules for all memory object pointers passed as explicit parameters to asynchronous service calls. These rules do not apply to parameters that expect specific types of semi-opaque objects such as control blocks or UDI buffers.

The memory pointed to by such pointer parameters (if non-NULL) must be either movable memory, part of the control block’s scratch space, or part of a module-global (and thus, read-only) variable. (See Section 5.2.2 for more details on control blocks and scratch space.) In particular, memory allocated on the stack in local variables must not be passed to asynchronous service calls because the stack frame may no longer exist when the pointer is finally dereferenced. If movable memory is used, the pointer must point to the beginning of the movable memory block and the driver must not read, write, or pass to other environment service calls or channel operations any portion of the movable memory block until the completion callback has been called.

For some service calls, pointer parameter values may be NULL. See the definition of each service call to determine whether or not it accepts NULL pointers.

5.2.2 Control Blocks

A control block is a structure used within UDI to represent an asynchronous request to or from the driver. Control blocks provide the context and associated data to describe each request. All region-context entry points into a driver are called with an associated control block.

Control blocks are used for all metalanguage channel operations and asynchronous service calls. Each time a channel operation is performed, the requesting driver region passes a control block specific to the request; the receiving region receives that control block and uses it to maintain the context for the request, typically returning the control block to the requesting region via an acknowledgment operation once the associated task has been completed. Likewise, when a driver makes a UDI service call that may not complete immediately it provides a control block that will be passed back to the driver in the callback operation to provide the context for that call.

5.2.2.1 Scratch Space

Each control block contains additional space that may be used by the driver to store information related to a 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 contents will be preserved across asynchronous service calls (see Section 4.8.1.2) and the driver's callback will always be invoked with the same control block that was originally passed to the asynchronous service call.

When the current operation is completed by transferring the control block to another region, ownership of the corresponding scratch space is also relinquished and the contents will not be preserved. The driver should not expect to receive the same control block back for any future operations, nor should it expect the same scratch space or scratch space contents to be maintained for that control block.

Drivers specify their scratch space requirements through the `udi_cb_init_t` structure as part of the `udi_init_info` initialization information (see **udi_cb_init_t** on page 10-11). The scratch space for a control block may actually change size (invisibly to the driver) as it is passed from region to region and is adjusted to meet the requirements of the receiving region. If the driver's scratch requirement is zero, the value of the scratch pointer is unspecified and it must not be dereferenced.

5.2.2.2 Inline Data

Some control block types have *inline data* elements associated with them. These are blocks of memory pointed to by fields within the visible portion of the control block that are automatically allocated when the control block is allocated. Drivers specify the size and, in some cases, the structure of inline elements through additional fields in the `udi_cb_init_t` structure. Inline memory pointers in control block structures are initialized by the environment and must not be modified by drivers.

5.2.2.3 Control Block Groups

Each metalanguage defines the format and contents of the various control blocks used for the interface operations used in that metalanguage. As part of this definition, the metalanguage organizes these control blocks into one or more *control block groups*, each with a corresponding control block group number. The control block group defines the allocation granularity for control blocks; the `udi_cb_alloc` operation (see **udi_cb_alloc** on page 11-5) is passed a control block index which the driver has correlated to the metalanguage's defined control block group number by including them in a `udi_cb_init_t` structure (see **udi_cb_init_t** on page 10-11). Thus, when a control block is allocated the result can be used as any of the control block types defined for that group (using appropriate type casts); the specific type is determined by the driver's initialization of that control block and the subsequent channel operation to which that control block is passed.

By using control block groups a metalanguage can reduce the overall cost of managing control block types for that metalanguage; frequently there are only a few control block groups defined within a metalanguage whereas there may be a large number of individual control block types to match the channel interface operations.

5.2.2.4 Control Block Synchronization

It is important to note that using a control block for a service call does *not* transfer ownership of that control block to another region. The driver must not use any part of the control block, including the scratch space, for other activities (i.e. passing it to another asynchronous service call or channel operation, or accessing any of its contents) until returned via the callback routine, but the contents of the control block's visible portion and scratch space are preserved and unmodified by the service call. In this way the driver may maintain its internal context for a request across an asynchronous service call.

5.2.2.5 Control Block Recycling

Channel operations are typically defined in pairs. For each type of initiating request or indication operation there are one or more corresponding response operation types. When a driver receives a control block as part of a request or indication operation, it must use the same control block in the response operation. The initiating driver may then free the control block, but if it expects to initiate additional operations of the same type it should instead maintain a pool of control blocks which it reuses for subsequent operations.

When a driver forwards a request, in some form, to another region, it must use a new control block for the layered request, rather than attempting to forward the original control block directly. This ensures that the original initiator's context is preserved.

5.2.2.6 Control Block Pointer Invariance

The pointer value used to identify a control block and to access its visible fields remains valid as long as the control block is owned by the same region, even across asynchronous service calls and callbacks. Once a control block is given away via a channel operation, however, the pointer value is valid only for purposes of aborting outstanding operations.

When a control block is returned to the initiating region as part of a response operation, the pointer value may or may not be the same as the original control block pointer, even though it refers to the same underlying control block. The environment may choose to reallocate and/or re-map the memory for the control block when it passes between regions.

5.2.3 Region Data

Each region created for the driver will have an associated block of memory referred to as *region data*. This region data is 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: state variables, request queues, PIO handles for accessing the device, and information about the channels connected to that region.

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** on page 10-17). The driver may choose to preserve this structure or overwrite it; once the region data has been created and initialized in this manner its subsequent use and contents are determined entirely by the associated driver region code.

5.3 Channel Context

A *channel context* is a driver-defined context value that is associated with a specific channel. The channel context is a single pointer value and is typically used to point to the region-global data structure for the region associated with that channel or to a more specific structure which in turn points to the region-global data. 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.

On entry to a region via a channel operation, the control block's context pointer is set to the channel context for the channel over which the operation was received.

5.4 Transferable Objects

Most of the data objects provided to or allocated by a region are not transferable to other regions. This allows the management and handling of those data objects to be optimized.

Specific data objects may be identified as transferable (or allocated that way as in the case of the `udi_mem_alloc` operation). When an object is transferable, there are further considerations which the driver and the environment must make in using and transferring those objects, including remapping or copying of parameters and associated data when the transfer crosses a domain boundary as well as various constraints and alignment issues for the memory associated with the transferable object.

In addition, once the driver has transferred an object to another region it may no longer use or reference that object, even if it still has a local pointer or variable reference to that object. Only one region may "own" a transferable object at any one time.

5.5 Implicit MP Synchronization

As indicated above, a region consists of a set of allocated data private to that region and a current thread of execution (unless it is idle). At most one thread may be active in any given region at one time; once a driver region is entered, all other attempts to enter the region will be deferred until after the first call returns. This deferral may be achieved through spin-waiting (on another CPU), queueing, or other implementation-specific methods.

Three factors in the UDI execution and data models combine to achieve implicit MP synchronization. These are:

1. All region data accessible to the driver is private to the region and may not be accessed from other regions or other entry point routines.
2. All module-global data is read-only.
3. Only one thread may be executing in a region at one time.

This guarantees that all data accesses within a UDI driver are single-threaded, so no explicit locking primitives are needed to run the driver in a Multi-Processor environment.

At the same time, a UDI driver can still take advantage of MP parallelism, since multiple driver instances can run in parallel and the entire driver can run in parallel with other drivers and other system activity. A driver may also increase its parallelism by using additional regions per driver instance (secondary regions) and dividing the work into mutually parallel pieces.

Configuration Model 6

6.1 Overview

There are two types of UDI driver configuration: static configuration and dynamic configuration. Static configuration specifies the operational characteristics of the UDI driver and its related device and is set in the distribution package for that driver. Dynamic configuration is the result of using that UDI driver along with its static configuration in a running system.

6.2 Static Configuration

Each UDI driver is provided to end-users by distributing a driver package. That driver package includes the driver itself along with information describing the driver and the associated device; this latter information is the *static configuration* information for that UDI driver. The static configuration is specified by the UDI driver developer as part of UDI driver development and included in the distribution with standard UDI utilities.

6.2.1 Static Driver Properties

All UDI drivers (and libraries) have a set of *static properties* associated with them and contained in a separate file (`udiprops.txt`) that must accompany the driver installation package. These static properties describe the driver and its associated device. They provide identification information about the driver, including the name of the driver, the description of the device(s) that are managed by the driver, and the relationship between UDI objects used by the driver.

To provide the required level of UDI portability, no assumption can be made about the target system beyond the general configuration specified by the ABI binding for that target. Therefore the UDI static properties supply all additional information needed to configure and control a device in the target system, including obtaining operational configuration parameters from the user or system adminstrator.

All UDI drivers (and libraries) have a set of static properties associated with them and contained in a separate file (`udiprops.txt`), which is intended to accompany the driver installation package. The static properties file also indicates which additional files are needed to complete the package, including specifying driver source code files and build rules for source code distributions.

6.2.2 Initialization Structures

Each driver must contain a set of static initialization structures that describe the internal structure of the driver and the corresponding environment objects needed or referenced by the driver, as defined in Chapter 10, *"Initialization"*. These structures are linked together and hung off of the driver's central initialization structure (`udi_init_info`). This allows the UDI environment to examine the structures before executing any driver code and determine and prepare for the operational needs of that driver. The initialization structures supplement the information specified in the static driver properties with information, such as sizes of structures, best contained in driver source code.

A similar approach is used in UDI metalanguage libraries, based on a `udi_meta_info` global variable in each library.

6.2.3 Building UDI Drivers

Each target environment for a UDI driver will probably have a different set of tools or names for those tools, as well as options appropriate to that toolset or environment that are required for building device drivers. To standardize this information, UDI defines the `udibuild` utility which is required for all UDI environments that support building UDI drivers from source code. The implementation of the `udibuild` utility and the operations it performs are defined by the target environment and will include compiling the various driver module source files to create the UDI driver.

The `udibuild` utility is designed to be portable and applies only to UDI driver builds; it replaces more conventional tools such as “make” and “build” which require additional system-specific information to operate correctly. The `udibuild` utility cannot be used for generic (non-UDI) purposes and uses the `udiprops.txt` static properties file to obtain information about the driver’s source files and corresponding build options.

6.2.4 UDI Packaging

To portably distribute UDI drivers and libraries, UDI defines the packaging format and a tool that may be used to generate that package. This packaging format is understood by various UDI tools to assist in creating and installing packages and may be placed onto distribution media for physical distribution.

UDI defines the format of the package itself, but does not specify the methods which are to be used in placing that package on the distribution media. Local media access methods and utilities may be used since these activities are not covered by the UDI portability guarantees, but ultimately the package must be delivered to the UDI tools in its original form, as a hierarchy of files.

To create packages, UDI specifies the `udimkpkg` utility, which must be available on all UDI development environments and will create a UDI package from a set of UDI device driver source code and/or binary objects. The `udimkpkg` utility uses the information in the driver’s static properties (from `udiprops.txt`) to locate the components of the package and construct the package itself. The `udiprops.txt` static properties file is also part of the package so that the proper information is available to the UDI package installation tools; for source distributions, it is included as a separate file; for binary distributions, is it encapsulated in the driver binary itself (often using a special section of the object file) by the `udimkpkg` tool.

6.2.5 UDI Package Installation

Once a package has been physically distributed to the target system, the system administrator uses the `udisetup` utility to install the package onto that system. The `udisetup` utility is defined to be present for all UDI target environments but its activities will be customized to the local environment to properly install the UDI driver and associated files.

The `udisetup` utility may invoke the udibuild utility for source code distributions. Not all environments are required to support source code distributions; thus, a particular environment might not include a `udibuild` utility. Such environments will not be able to utilize a UDI source code distribution.

6.3 Dynamic Configuration

6.3.1 Device Tree

The UDI target environment is typically described in terms of a “device tree”, which represents the hardware topology for that environment. For example, the base of the tree may be the system board or processor-memory interconnect, which has one or more buses as its branches (children), each of which may have an adapter plugged into them as leaves (grandchildren).

The depth of the tree is system-specific and the presence of multiplexers will make this routing significantly more complex. Each node in the tree is potentially managed by a different driver (UDI or otherwise), depending on the type of device represented by each node.

6.3.2 Driver Instantiation

For each device tree node that is managed by a UDI device driver, the UDI environment will instantiate an instance of that device driver. If multiple devices of the same type are present in the device tree, the UDI environment may choose to use the same code segment for all of those devices but is required to instantiate a separate logical instance of the driver for each device. This instantiation creates independent driver instances that separately manage and operate their corresponding devices.

6.3.3 Device Node Enumeration and Attributes

Each UDI driver (except “leaf” drivers) assists in the creation of the device tree by *enumerating* each of its child devices for the UDI environment. During this enumeration, the driver specifies the *enumeration attributes* of that child, which allows the UDI environment to match those attributes with the information supplied in the installed UDI drivers’ static properties, and therefore to locate the appropriate driver to manage each child device.

When each UDI driver instance is instantiated, that device/driver node will receive a set of instance attributes which describe that node. Some of these attributes will have been supplied by the parent in the form of enumeration attributes. Other attributes are supplied in the form of persistent attributes, which were set the last time this driver was instantiated (typically via system administrator input), and were preserved by the system in the persistent storage database (if it exists).

6.3.4 Driver Inter-Instance Binding

Each driver should use the information provided by its instance attributes to prepare for operation and then issue a metalanguage specific bind request to the parent device that enumerated it. Each metalanguage describes the details of the bind request for that metalanguage. Successful completion of that bind operation will supply all of the additional information needed by the UDI driver instance to manage its device. The UDI driver is then responsible for managing and operating its device until subsequently instructed to unbind, whereupon it will be removed from the device tree.

Calling Sequence and Naming Conventions 7

7.1 Overview

This chapter defines naming and calling conventions that apply to UDI environment interfaces in general. All calls of certain general types have common properties.

This chapter also defines conventions for metalanguage-specific interfaces. Some of these conventions are specified as strict requirements for all metalanguages; others are simply recommendations that may be overridden by metalanguage designers.

Generally, conventions covering required function parameters and types are strict requirements, while conventions covering function, parameter, and macro naming are recommendations. Metalanguage designers are free to use different naming conventions as long as the interface requirements defined below are met and the resulting names would be considered unique within the UDI interface namespace (at least as unique as the recommended conventions described in this chapter).

There are two function call categories to which these conventions apply: channel operations and asynchronous service calls. For more details on function call categorization see Section 4.8, “Function Call Classifications,” on page 4-4.

In addition, conventions apply to the naming of metalanguage-specific channel ops vector types and control block group numbers.

7.2 Channel Operations

Channel operations are invoked by a driver or by the environment and result in a procedure call to an operation entry point in another region. The calling sequence for the invocation of a channel operation (caller-side interface) is identical to the calling sequence for the corresponding entry point (callee-side interface). Caller-side functions have specific names, such as `udi_gio_xfer_req`. Callee-side functions have the same prototype as the caller-side equivalent, but will have names private to the driver, such as `my_gio_xfer_req`, and should be static symbols.

7.2.1 Channel Operation Invocations

Channel operations are metalanguage-specific. The invocation calls have declarations with the following form:

```
void <<meta>>_<<op>> (
   <<meta>>_<<cbtype>>_cb_t *cb,
   ...<<call-dependent parms>>... );
```

where:

<<meta>>	is a distinct prefix identifying the metalanguage, usually beginning with the prefix, "`udi_`".
<<op>>	identifies the particular channel operation within the metalanguage.
<<cbtype>>	identifies the particular control block type within the metalanguage.
cb	is a pointer to the semi-opaque metalanguage-specific control block, of type *<<meta>>*_*<<cbtype>>*`_cb_t`, for this operation.
<<call-dependent parms>>	are the zero or more metalanguage-dependent parameters for this particular channel operation.

Channel operation invocation calls are required to begin with an argument with the semantics described above for ***cb***. The name of this argument and the naming of *<<meta>>*, *<<op>>*, and *<<cbtype>>* are up to the metalanguage designer, but the above naming conventions are recommended.

The target channel over which to send the operation is determined by the value of `cb->gcb.channel`. The particular channel type to use for the operation is specified in the **TARGET CHANNEL** section of the reference page defining the operation.

7.2.2 Channel Operation Entry Points

The corresponding operation entry point in the target driver can have any name, but by convention has the same name as the invocation call with the initial "`udi`" prefix replaced by a driver-specific prefix, "*ddd*." In any case, the arguments will be as shown in the following declaration:

```
static void ddd_<<meta>>_<<op>> (
   <<meta>>_<<cbtype>>_cb_t *cb,
   ...<<call-dependent parms>>... );
```

This is the same as the calling sequence for the corresponding channel operation invocation.

For example, one driver may call:

```
udi_intr_event_ind(intr_event_cb, flags);
```

This would result in an invocation of the target driver's entry point routine for the target channel for this operation:

```
udi_intr_event_ind_op_t my_intr_event_ind;
   :
my_intr_event_ind(intr_event_cb, flags);
```

For convenience in the declaration of ops vector types and operation entry point forward declarations, a standard typedef shall be defined by the metalanguage header files for each operation type, in the form:

```
typedef void <<meta>>_<<op>>_op_t (
   <<meta>>_<<cbtype>>_cb_t *cb,
   ...<<call-dependent parms>>... );
```

7.3 Asynchronous Service Calls

Asynchronous service calls are calls to the environment in which the result may not be immediately available and is therefore supplied via a callback routine rather than as a direct return value of the service call itself (see Section 4.8.1.2). These types of calls are a core mechanism of the non-blocking UDI model of execution.

7.3.1 Asynchronous Service Call Invocations

Asynchronous service calls all have declarations with the following form:

```
void udi_<<category>>_<<service>> (
   udi_<<category>>_<<service>>_call_t *callback,
   udi_cb_t *gcb,
   ...<<call-dependent parms>>... );
```

where:

<<category>>	is a distinct prefix identifying the service category, such as “`buf`” for buffer management.
<<service>>	identifies the particular service within the category.
callback	is a pointer to the driver’s callback routine, of type `udi_<<category>>_<<service>>_call_t`.
gcb	is a pointer to a generic control block.
<<*call-dependent parms*>>	are the zero or more specific additional parameters for this service call.

7.3.2 Associated Callback Functions

Callback functions are called upon completion of the service request. The declaration for each callback type appears on the reference page along with the associated service call, in the following form:

```
typedef void udi_<<category>>_<<service>>_call_t (
   udi_cb_t *gcb,
   ...<<callback-dependent parms>>... );
```

where:

<<*callback-dependent parms*>> are zero or more additional parameters specific to this callback type.

In the driver’s code, the callback routine would appear as:

```
static void ddd_<<category>>_<<service>>_callback (
   udi_cb_t *gcb,
   ...<<callback-dependent parms>>... )
{
   ...
}
```

For example, a driver may call the environment as follows to obtain a new control block:

```
udi_cb_alloc(&my_cb_alloc_callback, gcb, my_cb_idx, chan);
```

which will result in calling the following callback when the allocation is complete:

```
udi_cb_alloc_call_t my_cb_alloc_callback;
   :
my_cb_alloc_callback(gcb, new_cb);
```

7.3.3 Control Block Type Conversion

Although the asynchronous service calls are defined to use a “generic control block”, the driver may use any control block for this purpose because all control blocks are a superset of the generic control block definition. The control block passed to a driver via a channel operation entry point is typically used for all asynchronous service calls and the subsequent channel operation made while processing that operation.

The `UDI_GCB()` macro (defined on page 11-11) is provided for convenience in converting a specific control block pointer to a generic control block pointer, in order to pass it to a service call. Using this macro, a typical asynchronous service call invocation becomes:

```
udi_<<category>>_<<service>>(callback, UDI_GCB(cb), ...);
```

The `UDI_MCB()` macro (defined on page 11-12) is provided for convenience in converting a generic control block pointer back to a specific control block type. Using this macro, a callback routine typically begins with:

```
<<meta>>_<<cbtype>>_cb_t *cb =
                     UDI_MCB(gcb, <<meta>>_<<cbtype>>_cb_t);
```

7.4 Channel Operations Vectors

The channel operations vector structure ("ops vector") used with each type of channel endpoint is defined in each metalanguage. These structures generally have declarations of the following form:[1]

```
typedef struct {
   udi_channel_event_ind_op_t *channel_event_ind_op;
   <<meta>>_<<op_1>>_op_t *<<op_1>>_op;
   ...
   <<meta>>_<<op_N>>_op_t *<<op_N>>_op;
} <<meta>>_<<role>>_ops_t;
```

where:

<<meta>> and *<<role>>*	are defined as above in the "ops_init" calling sequence.
<<op_1>>..<<op_N>>	identifies one or more channel operation entry point types that belong to this ops vector. These have the callee-side calling sequences defined in Section 7.2.2, "Channel Operation Entry Points".

Each entry in the ops vector is the driver's entry point for the corresponding channel operation.

Associated with each ops vector definition is a metalanguage-defined number that identifies this ops vector type with respect to others in the same metalanguage. Metalanguages must define both the numeric value and a mnemonic to use for that value. The mnemonic is typically named:

<<meta>>_<<role>>_OPS_NUM

where:

<<meta>> and *<<role>>*	are upper-case versions of those used in the above type definition.

7.5 Control Block Groups

Associated with each control block group is a metalanguage-defined number that identifies this control block group with respect to others in the same metalanguage. Metalanguages must define both the numeric value and a mnemonic to use for that value. The mnemonic is typically named:

<<meta>>_<<cbgroup>>_CB_NUM

1. The Management Metalanguage deviates from this form in that it doesn't have a `channel_event_ind_op` as the first member; all other metalanguages must have the first member of type `udi_channel_event_ind_op_t` as shown here.

UDI Core Specification

Section 3: Core Services

General Requirements 8

8.1 Versioning

All functions and structures defined in the UDI Core Specification, except for those defined in Chapter 25, *"Generic I/O Metalanguage"* and Chapter 28, *"Metalanguage-to-Environment Interface"*, are part of the "udi" interface, currently at version "0x101". A driver or library module that conforms to the UDI Core Specification, Version 1.01, must include the following declaration in its udiprops.txt file (see Chapter 30, *"Static Driver Properties"*):

```
requires udi 0x101
```

In each device driver or library source file, before including any UDI header files, the driver or library must define the preprocessor symbol, UDI_VERSION, to indicate the version of the UDI Core Specification to which it conforms, which must be the same as the interface version defined above:

```
#define UDI_VERSION         0x101
```

As defined in Section 30.4.6, "Requires Declaration," on page 30-6, the two least-significant hexadecimal digits of the interface version represent the minor number; the rest of the hex digits represent the major number. Versions that have the same "major version number" as an earlier version shall be backward compatible with that earlier version (i.e. a strict superset).[1]

8.2 Header Files

Each device driver source file must include the file "udi.h", as follows:

```
#include <udi.h>
```

This header file contains environment-specific definitions of standard UDI structures and types, as well as all function prototypes and other definitions needed to use the core UDI interfaces and services. Additional header files will need to be included, as required by other UDI specifications relevant to the device driver, for interfaces such as non-core services, metalanguages, bus bindings, etc. UDI drivers must not include any system header files not explicitly specified within a relevant UDI specification.

To maintain portability across UDI supportive platforms, device driver writers shall not assume any knowledge of the contents of udi.h with respect to implementation-dependent aspects of the UDI interfaces (such as the definition of handles or abstract types). Similarly, drivers shall not access any functions or objects external to the driver except those defined in the UDI Specifications to which they conform.

1. As an exception to this version compatibility, version 1.0 (0x100) is not forward compatible with any other versions bearing the major number of 1; version 1.0 of the specification cannot be wholly implemented as a functional product.

8.3 C Language Requirements

UDI device drivers that are written in C must be compiled using a *conforming freestanding implementation* of ISO C and must be *strictly conforming freestanding programs* in conformance with ISO/IEC 9899:1990.

All symbols with global scope will be treated uniquely to 31 characters for UDI implementations in accordance with the above ISO C specification.

8.4 Endianness Requirements

The ordering of bytes within a data value stored into memory directly by a UDI driver is referred to as the *driver endianness* of the driver. This ordering is typically based on the native byte ordering of the processor's instruction set, but can also be influenced by the storage model of the compiler with which the driver was compiled. UDI drivers must be compiled to execute with a driver endianness that is purely little endian or purely big endian. (See the definitions of **big endian** and **little endian** in Section 3.2.2, "Common Terms," on page 3-2.)

Fundamental Types 9

9.1 Overview

This chapter defines the C language type declaration conventions used by UDI. Other language bindings could be created for UDI; some of the syntax would differ, but the principles and the UDI-defined names listed here would be the same. In particular, UDI interfaces may be accessed from assembly language code, as long as the shape of data structures and calling conventions are made to match the C language conventions for the target platform.

For the most part, UDI avoids the use of standard C data types, since the sizes used for these data types are generally not specified by ISO C. Instead, UDI defines a set of *specific-length types* that are guaranteed to be specific sizes and a set of *abstract types* that are sized appropriately for a given class of environment implementations. UDI also defines *opaque types* that are used to refer to objects that may not be directly manipulated by drivers, and *semi-opaque types* that have visible parts and opaque parts. Header files provided by each environment implementation contain appropriate definitions of each of the above sets of data types.

All UDI interfaces and declarations are based (directly or indirectly) on the UDI-defined fundamental types listed in this chapter. The only standard ISO C types included as UDI fundamental types are ***char***, ***void***, and the varargs types listed in Section 9.2.4.

UDI drivers must use the UDI-defined types for data objects and interfaces specified by UDI. It is also recommended that UDI drivers use UDI-defined types *even for driver-internal variables and structures*, to avoid platform-dependent size assumptions. It still may be useful, however, to use the ***int*** type for a driver-internal variable that needs *the most efficient size that isn't particularly large* (clearly a very vague definition); if used, it should not be assumed that the size of an ***int*** is bigger than 16 bits, but it is reasonable to assume that an ***int*** is at least 16 bits since this is guaranteed by the ISO C standard. The ISO C standard also guarantees that the size of a ***long*** is at least 32 bits. For maximum portability, only quantities that fit into 32 bits should be stored in ***long*** variables.

While recommended for all drivers, drivers distributed as source code are particularly required to avoid non-portable use of ISO C data types, as described above.

UDI drivers may use floating point arithmetic or data types only in very restricted circumstances. The driver must indicate in its region attributes that floating point will be used (see Section 30.6.8, "Region Declaration," on page 30-18). When this attribute is present, the environment will, if possible, load the region into a domain that can support floating-point operations; otherwise, this driver will be rejected. Not all environments support the use of floating point in UDI drivers. Some environments may only support floating point in user-space domains. In all cases, use of floating-point types is limited to code within a region; there are no UDI service calls or channel operations that support floating-point types.

Note – Separate ABI specifications (see **Chapter 2, *"Document Organization"*** and "Section 6: MEI Services") define binary bindings for the UDI interfaces, including such things as the sizes of data types, calling conventions, and object file formats. The UDI Core Specification and other non-ABI UDI specifications support the capability of binary portability, but themselves provide source portability.

9.2 Usage of Standard ISO C Data Types and Macros

The following standard ISO C types and macros are used by UDI, and are available to UDI drivers and libraries by including the UDI-defined header file, `<udi.h>`. UDI drivers and libraries must not include `<stddef.h>` or other ISO C header files.

9.2.1 ISO C `char` Type

UDI supports the standard ISO C **char** type to refer to an 8-bit byte value.

Pointers to ***char*** (`char *`) are used to represent text strings, as in ISO C. Strings are null-terminated and may use Unicode characters encoded as a UTF-8 byte stream. ASCII as a subset of this encoding (that is, characters that are included in the ASCII set are encoded as separate successive 8-bit bytes using the zero-extended 7-bit ASCII encodings, and no combination of characters outside this set result in encodings that include bytes with the high bit clear). All specific string constants specified by UDI shall contain only ASCII characters.

9.2.2 ISO C `void` Type

UDI supports the standard ISO C **void** type.

There are two uses in UDI for the ***void*** type. The first is as the "return value" of a function that has no value to return, or to indicate a null argument list. This is standard ISO C usage and is very common in UDI.

The other use of the ***void*** type is as a pointer `(void *)` to an unformatted block of memory in the driver's virtual address space. Such pointers, called *generic pointers*, may be cast to (or from) any other pointer type, but may not be dereferenced directly.

9.2.2.1 Null Pointers

The special symbol, `NULL`, is an implementation-defined null pointer constant, as defined by ISO C. It is guaranteed to compare equal to zero and unequal to any valid pointer to any statically or dynamically allocated memory object. Some UDI service calls attach special meaning to a pointer value of NULL, as called out in the documentation of those functions. Where not otherwise mentioned, NULL is treated as any other illegal value: it must not be passed to any UDI service call nor should it ever be expected to be returned by UDI services.

9.2.3 ISO C `sizeof` and `offsetof` operators

UDI drivers and libraries may use `sizeof` and `offsetof`, as defined by ISO C. When used with UDI-conformant data structures, the values resulting from these operators shall be compatible with the UDI-defined `udi_size_t` type (see Section 9.5, “Abstract Types,” on page 9-6). That is, these values can be passed as parameters or assigned to variables of type `udi_size_t` without loss of information.

9.2.4 Varargs Types

UDI supports the standard ISO C variable argument list types.

The varags types supported in UDI are provided by the `<udi.h>` include file:

- **va_list** is a type defined for the variable used to traverse the list.

In addition to supporting the above varargs types, UDI environments shall provide the following macros and functions to manipulate these argument list variables:

- **va_start** is called to initialize a variable of type `va_list` to the beginning of the variable argument list.
- **va_arg** will return the next argument in the list pointed to by a `va_list` variable.
- **va_end** is used to terminate processing of a variable argument list by a `va_list` variable.

For additional information on using variable argument lists the ISO C documentation should be consulted; UDI deviates from that document only in the name of the header file used to obtain the type and macro declarations. UDI drivers must not #include `<stdarg.h>` directly; any definitions needed for varags support will be provided by #including `<udi.h>`.

Warning – ISO C `va_arg` has unspecified behavior when used with integral types smaller than ***int***, and many compilers will disallow this. Since UDI data types are defined as fixed sizes (e.g. `udi_ubit32_t`), a portable UDI driver cannot know whether or not some of these sizes are smaller than `sizeof(int)`. Therefore, instead of using `va_arg` directly with UDI data types, UDI drivers must use the `UDI_VA_ARG` macro (defined on page 9-30). The ISO C `va_arg` may still be used for standard types whose size is equal to or larger than the size of ***int*** (notably `int` and pointers), although for orthogonality the `UDI_VA_ARG` may be used for those types as well.

9.3 Notation for Implementation-Dependent Types and Constants

Wherever possible, UDI-defined types and interfaces are represented in the text of the specification by their actual declarations, in standard ISO C syntax, as they would appear in UDI header files. In cases where the details are implementation-specific (usually because of platform differences in sizes of integral data types) a *placeholder* designator is used in place of the missing detail. In actual header files the placeholder would be replaced with the appropriate valid C syntax.

Placeholder designators are shown with angle brackets, and will be one of the following:

Table 9-1 Placeholder Designators

Designator	Meaning
<INTEGRAL>	signed or unsigned integral type of appropriate size
<OPAQUE>	self-contained opaque type
<HANDLE>	handle type for environment-internal opaque objects
<NULL_HANDLE>	implementation-dependent null handle constant value

"**<INTEGRAL>**" is used with specific-length types and abstract types, "**<OPAQUE>**" is used with self-contained opaque types, and "**<HANDLE>**" is used with handle types, as described below.

Mnemonic constants (C preprocessor macros) are defined using the `#define` syntax. Mnemonic constants defined in UDI specifications are defined with specific values, with the exception of null handle constants, such as `UDI_NULL_CHANNEL`. Since the underlying handle type for null handle constants are implementation-dependent, the constant expression used to create a null handle constant is also implementation-dependent. "<NULL_HANDLE>" is used to represent such a constant expression.

9.4 Specific-Length Types

UDI *specific-length types* are defined to provide basic integer types, both signed and unsigned, which are guaranteed to be of the specified size and the specified range of valid values. These are all integral types, to which arithmetic and logical operations may be applied.

Implementations of the UDI environment will provide typedefs for the following types that will maintain the size and semantic definitions given below:

```
typedef <INTEGRAL>   udi_sbit8_t;   /* signed 8-bit: -2^7..2^7-1 */
typedef <INTEGRAL>   udi_sbit16_t;  /* signed 16-bit: -2^15..2^15-1 */
typedef <INTEGRAL>   udi_sbit32_t;  /* signed 32-bit: -2^31..2^31-1 */
typedef <INTEGRAL>   udi_ubit8_t;   /* unsigned 8-bit: 0..2^8-1 */
typedef <INTEGRAL>   udi_ubit16_t;  /* unsigned 16-bit: 0..2^16-1 */
typedef <INTEGRAL>   udi_ubit32_t;  /* unsigned 32-bit: 0..2^32-1 */
typedef udi_ubit8_t  udi_boolean_t; /* 0=False; 1..2^8-1=True */
```

Note – There are by design no 64-bit specific-length types. UDI is designed to work with compilers that do not support 64-bit integral types. In the few rare cases where 64-bit quantities are needed (such as for physical addresses) they are represented either as a pair of 32-bit values or as a self-contained opaque type (see Section 9.6.2 below).

The following constants are defined for use with `udi_boolean_t`:

```
#define FALSE                          0
#define TRUE                           1
```

These are intended for use in assignment statements. It is not safe to compare a boolean value against the constant TRUE since, for example, 57 is also a valid true value and 57 does not equal 1. Boolean variables should instead be tested by direct application of the ***if*** statement in ISO C:

```
if (boolean_variable) /* then true */
if (!boolean_variable) /* then false */
```

This is guaranteed to work since any non-zero value of the tested expression causes ***if*** to take the "then" branch.

Similarly, boolean variables can be tested in conditional expressions; e.g.,

```
x = (boolean_variable || (some_other_expression)) ? a_value : b_value;
x = (!boolean_variable && !(some_other_expression)) ? b_value : a_value;
```

without comparing the boolean variable against `TRUE` or `FALSE`.

Care must also be taken when assigning values to `udi_boolean_t` variables. For example, the following assignment statement could cause trouble:

```
boolean_variable = (flags & FLAG);
```

If the value of FLAG were 0x100 or greater, a true value (FLAG set in flags word) would be truncated in order to fit into the 8-bit `boolean_variable`. The value would then incorrectly become FALSE. To avoid this problem you must either know that FLAG would never be 0x100 or greater, or use one of the following constructs:

```
boolean_variable = !(!(flags & FLAG));
boolean_variable = ((flags & FLAG) != 0);
```

9.5 Abstract Types

UDI *abstract types* are integral types whose size is implementation-dependent. Each environment implementation chooses a size for each of these types that is appropriate for the way in which it can be used on a given platform. By keeping the sizes abstract, UDI can efficiently adapt to the needs of different platforms, and can evolve over time as needs change.

UDI abstract types are all integral types, to which arithmetic and logical operations may be applied.

Note – As ABIs are defined for binary portability, the sizes of abstract types will become part of each ABI definition. All implementations supporting the same ABI will have to use the same sizes. If a size were to change at some point, that effectively produces a new ABI, and all affected modules would require recompilation to use the new ABI.

9.5.1 Size Type

A driver refers to the number of bytes needed in, being read from, or written to, a buffer by using a size type. The size type is also used for buffer offsets, device memory offsets, and memory object sizes (zero offset refers to the first byte position). This type will be used in many places and it may need to vary across different classes of platforms depending on platform needs and constraints. Therefore, UDI refers to size with the following type:

```
typedef <INTEGRAL>    udi_size_t;    /* buffer size */
```

Because of architectural minimums on some of the defined size limits (e.g., see **udi_limits_t** on page 10-18) `udi_size_t` is guaranteed to be at least 16 bits in size. Since the `udi_size_t` type can represent different ranges of values in different domains, `udi_size_t` variables are not transferable between regions.

9.5.2 Index Type

A driver refers to a (“relatively small”) zero-based index value via the `udi_index_t` type (i.e., zero corresponds to the first element). The `udi_index_t` type is guaranteed to be able to hold values from 0 to 255, inclusive; only values in this range shall be used.

```
typedef <INTEGRAL>    udi_index_t;   /* zero-based index type */
```

When index values are used to refer to environment objects, as in the sub-sections below, the values are global to an entire driver and all of its instances, even if a driver consists of multiple modules. `udi_index_t` variables are transferable between regions.

9.5.2.1 Control Block Index

A `udi_index_t` variable may be used to hold a *control block index*. A control block index is used to identify a control block group registered via a `udi_cb_init_t` structure (see **udi_cb_init_t** on page 10-11) so it can be subsequently used to allocate control blocks or select scratch sizes and other control block properties.

Zero is reserved for future use as a special control block index value. It is illegal to use the value zero anywhere a control block index is expected.

9.5.2.2 Metalanguage Index

A `udi_index_t` variable may be used to hold a *metalanguage index*. A metalanguage index is used to identify one of possibly several metalanguages used by a driver. Metalanguages are associated with metalanguage indexes value via “meta” declarations in the driver’s Static Driver Properties (see Chapter 30). A metalanguage index of zero indicates the Management Metalanguage. Metalanguage index values are used with the tracing and logging services to associate certain types of events with specific metalanguages.

9.5.2.3 Ops Index

A `udi_index_t` variable may be used to hold an *ops index*. An ops index is used to identify a channel operations vector registered via a `udi_ops_init_t` function (see **udi_ops_init_t** on page 10-9) so it can be subsequently used to anchor channels or set default channel types.

Zero is reserved as a special ops index value that indicates that no ops are specified. This is used to spawn unanchored channels (see **udi_channel_spawn** on page 16-4), or to terminate a list of structures containing ops index values.

9.5.2.4 Region Index

A `udi_index_t` variable may be used to hold a *region index*. A region index is used to identify a driver-defined region type, so that region attributes can be associated with regions created by or on behalf of a driver instance. Region index zero always refers to the primary region of a driver instance. Secondary regions must use non-zero values for region index.

9.6 Opaque Types

UDI defines *opaque types* for objects whose contents are implementation-specific but whose semantics are strictly specified. Opaque objects are not directly visible to drivers, but are instead managed entirely by the environment. Drivers may only use opaque values by passing them from one environment interface to another.

Opaque types must not have arithmetic or logical operations applied to them and they must not be dereferenced. The only type of operation which may be applied to an opaque type is assignment (which includes argument passing and function return values). It is not even legal to directly compare two opaque values for equality.[1]

There are two sub-categories of opaque types: handles and self-contained opaque types.

Note – To facilitate binary portability across the same instruction set architecture, UDI environment implementations are likely to use an ABI-specified size for each opaque type, even though that may be larger than needed by some environments. (This refers to the size of the opaque type itself, not the sizes of any objects that might be referenced by opaque handles.)

9.6.1 Opaque Handles

Since opaque objects cannot be accessed directly, most are referenced indirectly via *opaque handles*. Opaque handles have “reference” semantics like C language pointers, but the actual type used to implement handles is implementation-specific. Only the environment knows how to directly interpret an opaque handle or the object to which it refers.

If a handle is assigned to two different variables and the object is modified (via an environment routine) using one variable, the other variable still refers to the same, modified object. The objects themselves are owned and managed by the environment.

Each handle type has a corresponding “null” value, for which a unique UDI_NULL_XXX mnemonic constant is defined. This null value is different from any values for handles that reference actual objects, and is reserved for special circumstances when it is necessary to indicate “no object.” Drivers must not compare handle values for equality, but the UDI_HANDLE_IS_NULL macro can be used to determine if a handle variable currently holds a null value. (Pointers, on the other hand, may be compared against NULL directly.)

A handle whose contents have been zeroed is considered equivalent to the corresponding UDI_NULL_XXX null handle value. The zeroing may be a result of the initial allocation of the handle variable (as initial region data, or by using `udi_mem_alloc` without the UDI_MEM_NOZERO flag), or may be done explicitly by the driver, using `udi_memset`.

Service calls that free opaque objects through their handles act as no-ops when passed null handles. Where not otherwise mentioned, null handles are treated as any other illegal value: they must not be passed to any UDI service call nor should they ever be expected to be returned by UDI services.

1. In most cases, testing for equality should not be needed; drivers store opaque values in their own structures and pass them to environment routines later. In cases where an equality check is useful, an environment routine is provided.

Some opaque handle types are *transferable*, others are not. An object of a transferable opaque handle type may be passed from one region to another via a channel operation. Non-transferable opaque objects are local to the region in which they were allocated and may not be passed between regions.

Many UDI objects are manipulated via handles.

NAME **udi_channel_t** *UDI inter-module communications handle*

SYNOPSIS

```
#include <udi.h>

typedef <HANDLE> udi_channel_t;

/* NULL channel handle constant */
#define UDI_NULL_CHANNEL                <NULL_HANDLE>
```

DESCRIPTION UDI Drivers communicate with other drivers and with certain environment modules (e.g. the Management Agent) via bi-directional communication channels established during configuration. Channels are point-to-point and have two ends. The object which keeps track of a particular end of a communication channel between two modules is called the channel object, which is referred to by a channel handle.

Channel handles are transferable between regions if and only if they refer to *loose ends*. (See “Channels” on page 4-2.)

WARNINGS Drivers must not compare handle values for equality, but the UDI_HANDLE_IS_NULL macro can be used to determine if a handle variable currently holds a null value.

REFERENCES udi_channel_event_ind, udi_channel_anchor, udi_channel_close, UDI_HANDLE_IS_NULL

NAME **udi_buf_path_t** *Buffer path routing handle*

SYNOPSIS

```
#include <udi.h>

typedef <HANDLE> udi_buf_path_t;

/* NULL buffer path handle constant */
#define UDI_NULL_BUF_PATH                    <NULL_HANDL
                                             E>
```

DESCRIPTION When a driver allocates a UDI buffer, it associates it with a *buffer path*. Buffer paths indicate intended destinations for data buffers, typically associated with the allocating driver's parent. Drivers refer to buffer paths via opaque *buffer path handles*.

Path handles are explicitly allocated (via `udi_buf_path_alloc`), or provided to a driver instance via a `UDI_CHANNEL_BOUND` channel event indication; the driver indicates how many path handles are needed on a per-parent basis and the environment provides that number of handles each time a parent is bound to the driver.

Buffer path handles are not transferable between regions.

WARNINGS Drivers must not compare handle values for equality, but the `UDI_HANDLE_IS_NULL` macro can be used to determine if a handle variable currently holds a null value.

REFERENCES `UDI_HANDLE_IS_NULL`, `udi_channel_event_ind`, `udi_buf_copy`, `udi_buf_write`

NAME **udi_origin_t** *Request origination handle*

SYNOPSIS

```
#include <udi.h>

typedef <HANDLE> udi_origin_t;

/* NULL origin handle constant */
#define UDI_NULL_ORIGIN                <NULL_HANDL
                                       E>
```

DESCRIPTION Environments may use the *origin handle* to maintain information about the origination of a user request. Each driver is responsible for copying the origin handle from received control blocks into any control blocks generated on behalf of that received control block. This origin handle may be used by the environment to maintain tracking, quota, or other information for the original request from its point of origin. The origin handle is an opaque handle.

The driver may set the `UDI_NULL_ORIGIN` value for a control block's origin field instead of copying an origin handle from another control block, but the driver cannot not create or allocate origin handles itself.

Origin handles are transferable between regions.

WARNINGS Drivers must not compare handle values for equality, but the `UDI_HANDLE_IS_NULL` macro can be used to determine if a handle variable currently holds a null value.

REFERENCES `UDI_HANDLE_IS_NULL`, `udi_cb_t`

9.6.2 Self-Contained Opaque Types

A *self-contained opaque type* holds data that can be interpreted only by the environment. Unlike opaque handles, these types have “value” semantics rather than “reference” semantics. That is, assignment makes a copy of the entire object. If a self-contained opaque value is assigned to two different variables and one is modified (via an environment routine), the other will retain the original value.

This means that allocation calls are not needed for self-contained opaque types; drivers simply declare variables of this type and assign values to them.

Self-contained opaque types are not transferable between regions.

9.6.2.1 Timestamp Type

The timestamp type refers to a point in time, relative to an arbitrary starting point, in implementation-specific units. The timestamp type has the following type definition:

```
typedef    <OPAQUE>    udi_timestamp_t;
```

As with abstract types, the size of the `udi_timestamp_t` type is expected to vary according to the needs of different environments.

Detailed usage of this type is described under **udi_time_current** on page 14-8.

9.7 Semi-Opaque Types

UDI defines *semi-opaque types* for objects that have driver-visible fields, but also have implementation-specific contents that are not visible to drivers. The driver-visible part of a semi-opaque object is defined as a C structure; drivers refer to the object using a pointer to this structure.

Semi-opaque objects must only be allocated by the environment, since the driver doesn’t know how big the whole object is. This is typically done by calling an environment-provided service call such as `udi_cb_alloc` to allocate the object.

9.7.1 Control Blocks

UDI defines a *control block* type to provide context for asynchronous service calls and channel operations. UDI control blocks are semi-opaque objects and are transferable between regions.

See Chapter 11, “*Control Block Management*” for more details on control blocks.

9.7.1.1 Buffers

UDI defines a *buffer* type, which contains a variable-length collection of application or protocol data. UDI buffer data consists of a byte string that is logically contiguous, but which may be both physically and virtually segmented. In many cases, the actual storage will be of one or more structure types in the embedding system. UDI hides these machine- and OS-dependencies within the buffer object.

UDI buffers are semi-opaque objects and are transferable between regions.

See Chapter 13, “*Buffer Management*” for more details on UDI buffers.

9.8 Structures Requiring a Fixed Binary Representation

While drivers must specify the structure layout of certain driver-defined structures which are passed between regions (as indicated in the previous subsection), drivers need not concern themselves with the actual binary layout of such structures, or in general with the binary layout of UDI-defined structures or other software-defined structures. However, *hardware-defined structures*, defined by the device, bus, or hardware protocol, generally require a fixed binary representation. UDI drivers, which are portable across a range of platforms and operating environments, must carefully follow certain rules to create these structures in a manner that will guarantee correct layout in all environments. Such structures are required to be laid out in the appropriate endianness, with fixed alignment of multi-byte fields handled in a platform-independent manner, and that each byte in the structure be accounted for.

Any C structure definitions used to represent hardware structures must be constructed at least according to the following rules:

1. Must use only UDI specific-length types on naturally aligned boundaries (offsets) within the structure. Bit-fields in the C language are not portable and must not be used (see the warning below).

2. Every byte in the structure must be accounted for.

These rules must be restricted somewhat for protocol-defined structures, as defined in the section on “Endianness Management” on page 22-2. Refer to that section for additional details on the construction of hardware-defined structures.

Warning – Bit-fields in the C language are not portable and must never be used in the definition of hardware-defined structures or in interfaces between independent software components. This is because C is ambiguous about the ordering of bits in a bit-field, allowing compiler implementations to order the bits differently even within a given endianness. Therefore, bit-fields cannot be relied upon to reliably specify a placement of bits in a portable manner.

9.9 Common Derived Types

The types listed in this section are not, strictly speaking, fundamental types; they are derived from other UDI-defined types and are not in any way implementation-dependent. However, they are common to many areas of the UDI specification and so are described here.

9.9.1 UDI Status

Purpose:

To provide a uniform means of reporting status or error conditions within the I/O system. When an error has occurred, provide a means of tracing dependent errors to root causes.

NAME **udi_status_t** *UDI status code*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit32_t udi_status_t;

/* Mask Values and Flags for udi_status_t */
#define UDI_STATUS_CODE_MASK            0x0000FFFF
#define UDI_STAT_META_SPECIFIC          0x00008000
#define UDI_SPECIFIC_STATUS_MASK        0x00007FFF
#define UDI_CORRELATE_OFFSET            16
#define UDI_CORRELATE_MASK              0xFFFF0000

/* Common Status Values */
#define UDI_OK                          0
#define UDI_STAT_NOT_SUPPORTED          1
#define UDI_STAT_NOT_UNDERSTOOD         2
#define UDI_STAT_INVALID_STATE          3
#define UDI_STAT_MISTAKEN_IDENTITY      4
#define UDI_STAT_ABORTED                5
#define UDI_STAT_TIMEOUT                6
#define UDI_STAT_BUSY                   7
#define UDI_STAT_RESOURCE_UNAVAIL       8
#define UDI_STAT_HW_PROBLEM             9
#define UDI_STAT_NOT_RESPONDING         10
#define UDI_STAT_DATA_UNDERRUN          11
#define UDI_STAT_DATA_OVERRUN           12
#define UDI_STAT_DATA_ERROR             13
#define UDI_STAT_PARENT_DRV_ERROR       14
#define UDI_STAT_CANNOT_BIND            15
#define UDI_STAT_CANNOT_BIND_EXCL       16
#define UDI_STAT_TOO_MANY_PARENTS       17
#define UDI_STAT_BAD_PARENT_TYPE        18
#define UDI_STAT_TERMINATED             19
#define UDI_STAT_ATTR_MISMATCH          20
```

DESCRIPTION

UDI status values are 32-bit integers that are logically subdivided into a 16-bit *status code* field, and a 16-bit *correlation* field. Modules within the UDI environment must report status using this format. ("Modules" in the context of this section refers to drivers and environment services.) A module reports successful completion by setting the status value to `UDI_OK`.

To separate and distinguish between common status codes and metalanguage-specific status codes the *status code*, in the low-order 16-bits, is further sub-divided into a 1-bit "metalanguage-specific status flag" (0 = common status, 1 = metalanguage-specific status)—designated by `UDI_STAT_META_SPECIFIC`, and a 15-bit "specific status code"—designated by `UDI_SPECIFIC_STATUS_MASK`.

However, drivers do not generally need to be aware of this additional subdivision because the status code values are defined to include the flag bit, and drivers can just assign the UDI-defined status identifier into the `udi_status_t` (taking into account the correlation field, as described below). Metalanguage designers must make sure that `UDI_STAT_META_SPECIFIC` is or'ed into each of their metalanguage-specific status mnemonic constants.

When an error must be signalled, the reporting module selects an appropriate *status code* value (either one of the common ones shown below, or a call-specific or metalanguage-specific code appropriate to the context in which the error is encountered) and assigns it into a `udi_status_t` parameter. This status value shall contain zero in the correlation field in order to indicate that this is a new error, rather than a derivative error. The `udi_status_t` value is used in a call to `udi_log_write` (see page 17-7), which will record all the data pertinent to the error in a logging file and assign a correlation value to the error. This correlation value will be placed in the 16 most-significant bits of the `udi_status_t` on return. This combined value will be passed by the driver to other entities that are affected by the error. When this error in turn results in a derivative error worth logging (e.g., lost link connection results in a file access error) the next reporting module will replace the 16 least-significant bits of the `udi_status_t` with a new appropriate value but will maintain the *correlation* field contents. When called with a derivative status, `udi_log_write` will record that same correlation value, together with all the data pertinent to the new error. In this way, the individual entries in the log file can be threaded together by the correlation value to trace back to the original error for the root cause.

To check for a specific status code value, the driver writer can mask off the correlation field and compare the remaining value:

```
if ((status & UDI_STATUS_CODE_MASK) == UDI_STAT_XXX)
     handle_error();
```

9.9.1.1 Common Status Codes

The UDI environment defines several status codes for use in reporting various common problems and conditions within UDI drivers and metalanguages. It is important to note that any driver internal errors will have indeterminate results but very likely will result in the driver instance being killed without ever being re-entered, therefore there are no corresponding error codes defined for related conditions (e.g. invalid argument errors). For related information see Section 4.10 on page 4-6.

UDI status error codes are defined with mnemonic constants as shown in the following table.

Table 9-2 Common UDI Status Codes

Status Code	**Value**	**Meaning**	**Description**
UDI_OK	0	Success	The request completed properly without any exceptional conditions.
UDI_STAT_NOT_SUPPORTED	1	Not supported	This operation is not supported by this UDI environment implementation or the combination of parameters specified for this operation cannot be supported by this environment or hardware.
UDI_STAT_NOT_UNDERSTOOD	2	Request not understood	The parameters specified for this operation exceed valid ranges, or the combination of parameters does not make sense for this operation. Used only for channel operations; if out-of-range values are used with service calls, the driver instance may be terminated with extreme prejudice.
UDI_STAT_INVALID_STATE	3	Invalid state	The request is understood and implemented, but is not valid in the current state. This is typically used for channel operations; UDI service operations are typically not stateful.
UDI_STAT_MISTAKEN_IDENTITY	4	Mistaken identity	The request is understood and implemented, but is inappropriate for the device or other object to which it refers. This is typically used when a parameter selects a physical resource that is not present for a particular device.
UDI_STAT_ABORTED	5	Operation aborted	The operation was successfully aborted as a result of a `udi_cancel` or `udi_channel_op_abort` service call or a metalanguage-specific cancellation request.
UDI_STAT_TIMEOUT	6	Operation exceeded specified time period	The operation had an associated timeout value which was exceeded causing this operation to be aborted.

Table 9-2 Common UDI Status Codes

Status Code	Value	Meaning	Description
UDI_STAT_BUSY	7	Resource busy	The device or associated resource is currently busy and cannot handle this request at this time (or queue this request for later handling).
UDI_STAT_RESOURCE_UNAVAIL	8	No resources available	There are insufficient resources to satisfy this request. There is no guarantee or expectation that sufficient resources will become available in the future.
UDI_STAT_HW_PROBLEM	9	Hardware problem	A problem has been detected with the associated hardware that prevents this request from being executed successfully and is not covered by a more specific error indication.
UDI_STAT_NOT_RESPONDING	10	Device not responding	The device is not present or not responding.
UDI_STAT_DATA_UNDERRUN	11	Data underrun	A data transfer from a device transferred less data than expected.
UDI_STAT_DATA_OVERRUN	12	Data overrun	A data transfer from a device attempted to transfer more data than expected.
UDI_STAT_DATA_ERROR	13	Data error	Data corruption was detected during a transfer, typically by a parity or checksum check.
UDI_STAT_PARENT_DRV_ERROR	14	Parent driver error	A parent (or ancestor) of the driver reporting this condition has encountered an error that has prevented the request from being successfully executed. There will typically be a correlated error log issued by the parent driver.
UDI_STAT_CANNOT_BIND	15	Cannot bind to parent	The driver tried to bind to its parent, but was rejected by the parent driver. Also used by the parent to indicate such rejection to its child.
UDI_STAT_CANNOT_BIND_EXCL	16	Cannot bind exclusively to parent	A request to bind exclusively to a driver cannot be satisfied because another child instance is already bound.
UDI_STAT_TOO_MANY_PARENTS	17	Too many parents for this driver	The request to bind to a parent cannot be supported because this driver instance is already bound to the maximum number of parents that it can support.

Table 9-2 Common UDI Status Codes

Status Code	Value	Meaning	Description
UDI_STAT_BAD_PARENT_TYPE	18	Cannot bind to this type of parent device	The request to bind to a parent cannot be satisfied because the parent metalanguage or device properties (as determined by the parent-specified enumeration attributes) for the binding is not a type supported by this driver instance in its current state.
UDI_STAT_TERMINATED	19	Region was abruptly terminated	The request failed because the target region was abruptly terminated. Drivers must not generate this status code directly. It is used when the environment generates responses on behalf of a terminated driver instance. When a driver receives an operation with this status code, it must ignore all other metalanguage-specific control block fields and parameters except buffer pointers.
UDI_STAT_ATTR_MISMATCH	20	Driver/device cannot comply with custom attribute setting.	The driver has been given a custom attribute value that it cannot set on its device. This status can be used during either a parent or child binding.

The status codes here may be supplemented by various Physical I/O status codes or metalanguage-specific status codes as defined in the corresponding UDI specification books.

9.9.2 Data Layout Specifier

Purpose:

The data layout specifier type is used to describe the layout of control blocks and other driver data structures that may be transferred between regions by using channel operations. Data layout specifiers are primarily used by metalanguage libraries to describe the layout of all fixed structures passed via channel operations. Drivers may in some cases need to declare layout specifiers themselves, to use with `udi_cb_init_t` or `udi_cb_alloc_dynamic`; this allows a driver to register the layout of inline memory structures that aren’t strictly typed by the metalanguage.

NAME **udi_layout_t** *Data layout specifier*

SYNOPSIS

```
#include <udi.h>

typedef const udi_ubit8_t udi_layout_t;

/* Specific-Length Layout Type Codes */
#define UDI_DL_UBIT8_T                  1
#define UDI_DL_SBIT8_T                  2
#define UDI_DL_UBIT16_T                 3
#define UDI_DL_SBIT16_T                 4
#define UDI_DL_UBIT32_T                 5
#define UDI_DL_SBIT32_T                 6
#define UDI_DL_BOOLEAN_T                7
#define UDI_DL_STATUS_T                 8

/* Abstract Element Layout Type Codes */
#define UDI_DL_INDEX_T                  20

/* Opaque Handle Element Layout Type Codes */
#define UDI_DL_CHANNEL_T                30
#define UDI_DL_ORIGIN_T                 32

/* Indirect Element Layout Type Codes */
#define UDI_DL_BUF                      40
#define UDI_DL_CB                       41
#define UDI_DL_INLINE_UNTYPED           42
#define UDI_DL_INLINE_DRIVER_TYPED      43
#define UDI_DL_MOVABLE_UNTYPED          44

/* Nested Element Layout Type Codes */
#define UDI_DL_INLINE_TYPED             50
#define UDI_DL_MOVABLE_TYPED            51
#define UDI_DL_ARRAY                    52
#define UDI_DL_END                      0
```

DESCRIPTION

A data layout specifier consists of an array of one or more `udi_layout_t` layout elements. Each element contains a type code indicating one of the UDI data types that can be passed into a channel operation, either as a field in the control block or as an additional parameter. Each successive element of the array represents successive offsets within the described structure, with padding automatically inserted for alignment purposes as if the specified data types had appeared in a C `struct` declaration.

Since channel operations are based on strongly typed function calls, the environment usually has sufficient information to handle data transformations such as endian conversions when channel operations cross between domains of differing data formats. However, there are some cases where one or more parameters to a channel operation call are not strongly typed, but are simply `void *` pointers to chunks of memory. Such pointers must point either to movable memory (allocated by `udi_mem_alloc` with the `UDI_MEM_MOVABLE` flag set) or to inline memory permanently associated with a control block when it was allocated.

If such untyped memory is also unstructured—that is, it is to be treated as an array of bytes—then no data transformations need be performed. If the memory is structured, however, drivers must inform the environment of that structure, since it cannot be determined a priori from the channel operation definition. In that case, the driver supplies a data layout specifier as a parameter to the “cb_init” function with which the operation is associated.

Layout element type values for `udi_layout_t` are defined with mnemonic constants as shown in the following table:

Table 9-3 Specific-Length Element Type Codes

Element Type Code	**Value**	**Corresponding Data Type**
UDI_DL_UBIT8_T	1	`udi_ubit8_t`
UDI_DL_SBIT8_T	2	`udi_sbit8_t`
UDI_DL_UBIT16_T	3	`udi_ubit16_t`
UDI_DL_SBIT16_T	4	`udi_sbit16_t`
UDI_DL_UBIT32_T	5	`udi_ubit32_t`
UDI_DL_SBIT32_T	6	`udi_sbit32_t`
UDI_DL_BOOLEAN_T	7	`udi_boolean_t`
UDI_DL_STATUS_T	8	`udi_status_t`

Table 9-4 Abstract Element Type Codes

Element Type Code	**Value**	**Corresponding Data Type**
UDI_DL_INDEX_T	20	`udi_index_t`

Table 9-5 Opaque Handle Element Type Codes

Element Type Code	**Value**	**Corresponding Data Type**
UDI_DL_CHANNEL_T	30	`udi_channel_t` (must be a loose end or `UDI_NULL_CHANNEL`)
UDI_DL_ORIGIN_T	32	`udi_origin_t` (may be `UDI_NULL_ORIGIN`)

Table 9-6 Indirect Element Type Codes

Element Type Code	Value	Corresponding Data Type
UDI_DL_BUF	40	`udi_buf_t *` (may be `NULL`). The next three layout elements (3 unsigned bytes) provide detailed information on the use of this buffer type, as described below.
UDI_DL_CB	41	This element is a pointer to a metalanguage-specific control block of the same type as the control block in which this element is embedded. This is used for control block chaining. (May be `NULL`.)
UDI_DL_INLINE_UNTYPED	42	`void *` (untyped array of inline memory bytes; may be `NULL`)
UDI_DL_INLINE_DRIVER_TYPED	43	`void *` (structure determined by driver; corresponding "cb_init" call supplies layout; may be `NULL`)
UDI_DL_MOVABLE_UNTYPED	44	`void *` (untyped array of movable memory bytes; may be `NULL`)

Table 9-7 Nested Element Type Codes

Element Type Code	Value	Description
UDI_DL_INLINE_TYPED	50	Pointer to inline memory whose structure is determined by the metalanguage definition. Subsequent layout elements describe structure. (May be `NULL`)
UDI_DL_MOVABLE_TYPED	51	Pointer to movable memory whose structure is determined by the metalanguage definition. Subsequent layout elements describe structure. (May be `NULL`)
UDI_DL_ARRAY	52	Begins an embedded fixed-length array. The next layout element (one unsigned byte) is interpreted as the number of array elements, and must not be zero. Subsequent layout elements describe the structure of one array element.
UDI_DL_END	0	End of current nested element. Pop up one level. If used at top level, terminates layout array.

For all nested element types, the layout elements following the nested type, up until the matching `UDI_DL_END`, describe the structure of that element. For driver-type inline structures, indicated by `UDI_DL_INLINE_DRIVER_TYPED`, the driver-provided layout array is logically inserted as a nested element.

Since nested objects might be variable length arrays of structures, the layout elements for a nested object may need to be repeated to cover the whole nested object, as if the nested layout were describing the structure of one element of such an array. Partial repeats are not allowed; the object must be covered by zero or more complete repeats of the nested layout.

The `udi_layout_t` array must end with a `UDI_DL_END` element; the first `UDI_DL_END` not used to match a nested element type is interpreted as the end of the array.

Note – The various `INLINE` element types and `UDI_DL_CB` must not be used as part of the layout description for inline memory contents, since nested inline structures are not supported. They are only legal for control block visible layouts. At most one UDI_DL_CB element may be used within a single layout array.

The `UDI_DL_BUF` type is used for a pointer to a `udi_buf_t` buffer. It is followed by three unsigned bytes providing further details. These are used to describe related control block fields that affect the way the buffer and its data are handled during domain crossings and during abrupt driver termination ("region kill"). The meaning of each of these bytes is listed below.

byte 0 designates the control block field, if any, that holds a flag or type code that can be used to distinguish between buffers flowing in the "forward" direction (i.e. carrying significant data—typically a "write" request or a "read" acknowledgement) and those flowing in the "reverse" direction (i.e. carrying data that does not need to be preserved—typically a "read" request or a "write" acknowledgement). This field is referred to as the *preserve flag* for purposes of this layout element specifier. The value in byte 0 is used as a zero-based index into the layout specifier for the control block to which this `UDI_DL_BUF` applies; this selects the layout element that corresponds to the preserve flag.

byte 1 supplies a *mask value* to apply to the low order byte of the preserve flag value.

byte 2 supplies a *match value* to compare with the masked preserve flag value. If they compare equal, the environment must preserve any previously preserved data content and tags in the buffer, up to the buffer's current ***buf_size***, when the data is transferred via channel operation to the new region. If the masked preserve flag does not equal the match value, then all of the buffer's contents in the new region are unspecified, and buffer tags are removed, but the ***buf_size*** value is unchanged.

If the above criteria indicate that buffer data is to be preserved, then environments that might use "region kill" must track this buffer and return it to the sending region (by failing the request with a status of UDI_STAT_TERMINATED) if the receiving region is abruptly terminated while still holding this buffer.

9.10 Implementation-Dependent Macros

UDI defines a number of implementation-dependent macros. That is, macros whose parameters and semantics are defined generically, but whose implementation is environment-specfic (often associated with a particular ABI). This section lists implementation-dependent macros related to fundamental types.

NAME **UDI_HANDLE_IS_NULL** *Determine whether a handle value is null*

SYNOPSIS

```
#include <udi.h>

#define UDI_HANDLE_IS_NULL(handle, handle_type)
```

ARGUMENTS

handle is the handle value to check.

handle_type is the type specification for that handle.

DESCRIPTION This macro is used to check if an opaque handle value is null (i.e. all zeroes). This is the only way in which a handle value can be compared against any other value.

RETURN VALUES This macro returns a `udi_boolean_t` value that is `TRUE` if the handle value is a null handle value.

NAME

UDI_HANDLE_ID *Get identification value for specified handle*

SYNOPSIS

```
#include <udi.h>

#define UDI_HANDLE_ID(handle, handle_type)
```

ARGUMENTS

handle is the handle for which an ID value is to be obtained.

handle_type is the type specification for that handle.

DESCRIPTION

For tracing, logging, and debugging purposes it is often useful to be able to identify and differentiate between handles that are passed to the driver. Handles themselves are opaque structures that the driver has no information about. To obtain an ID value that can be used in tracing, logging, or debugging output, the `UDI_HANDLE_ID` macro should be used.

The ID value will be unique with respect to all other handle IDs for the same ***handle_type*** in the same region. Subsequent uses of UDI_HANDLE_ID for the same handle value will produce the same ID value.

This macro is useful in conjunction with `udi_snprintf`.

RETURN VALUES

This macro returns a `(void *)` value that can be formatted using the `%p` format code for `udi_snprintf` calls.

EXAMPLES

```
udi_snprintf("Got channel handle %p\n",
            UDI_HANDLE_ID(chan, udi_channel_t));
```

REFERENCES

`udi_snprintf, udi_trace_write, udi_log_write`

NAME **UDI_VA_ARG** *Varargs macro for UDI data types*

SYNOPSIS

```
#include <udi.h>

#define UDI_VA_ARG(pvar, type, va_code)


#define UDI_VA_UBIT8_T
#define UDI_VA_SBIT8_T
#define UDI_VA_UBIT16_T
#define UDI_VA_SBIT16_T
#define UDI_VA_UBIT32_T
#define UDI_VA_SBIT32_T
#define UDI_VA_BOOLEAN_T
#define UDI_VA_INDEX_T
#define UDI_VA_SIZE_T
#define UDI_VA_STATUS_T
#define UDI_VA_CHANNEL_T
#define UDI_VA_ORIGIN_T
#define UDI_VA_POINTER
```

ARGUMENTS

pvar is a pointer into the argument list, as for ISO C `va_arg()`.

type is one of the UDI data types from the table below.

va_code is a code corresponding to a UDI data type or class of UDI data types, from the table below. This must be from the row of the table that includes the type, ***type***.

DESCRIPTION

This macro acts as a wrapper around the ISO C `va_arg()` macro, allowing it to be used portably with UDI data types. The supported data types and their corresponding ***va_code*** values are listed in the following table:

Table 9-8 UDI_VA_ARG Data Type Codes

UDI Data Type	***va_code*** Value
`udi_ubit8_t`	UDI_VA_UBIT8_T
`udi_sbit8_t`	UDI_VA_SBIT8_T
`udi_ubit16_t`	UDI_VA_UBIT16_T
`udi_sbit16_t`	UDI_VA_SBIT16_T
`udi_ubit32_t`	UDI_VA_UBIT32_T
`udi_sbit32_t`	UDI_VA_SBIT32_T
`udi_boolean_t`	UDI_VA_BOOLEAN_T
`udi_index_t`	UDI_VA_INDEX_T
`udi_size_t`	UDI_VA_SIZE_T
`udi_status_t`	UDI_VA_STATUS_T

Table 9-8 UDI_VA_ARG Data Type Codes

UDI Data Type	***va_code* Value**
`udi_channel_t`	UDI_VA_CHANNEL_T
`udi_origin_t`	UDI_VA_ORIGIN_T
any pointer type	UDI_VA_POINTER

Initialization 10

10.1 Overview

There are two general phases to the initialization process for a UDI device driver: per driver initialization, and per instance (device) initialization.

10.1.1 Per-Driver Initialization

Per-driver initialization starts once the driver has been loaded and/or linked into the system, is handled entirely by the environment, and completes before any driver code is called. The driver communicates its per-driver or per-module initialization requirements to the environment by declaring a global symbol named `udi_init_info` in each separately loadable module of the driver. The `udi_init_info` structure specifies all of the parameters required to create the primary region and any secondary regions used by this driver, all channel operations vectors for each metalanguage used by the driver module, and parameters for metalanguage-specific control block groups as well as generic control blocks.

10.1.2 Per-Instance Initialization

Per-instance initialization for the driver starts when the driver receives the `udi_usage_ind` operation on its management channel. Following this general resource level and tracing indication, the driver instance will receive a `udi_channel_event_ind` operations of type `UDI_CHANNEL_BOUND` for each statically-allocated secondary region and for binding to its parent. The driver will usually respond to this with a metalanguage-specific bind operation to its parent driver and the parent will respond by propagating its constraints to the child (see Chapter 12, "*Constraints Management*") and then acknowleding the bind operation. Per-instance initialization is required to be complete when the new driver instance calls `udi_channel_event_complete` with the original control block(s).

See Chapter 24, "*Management Metalanguage*", for more details on the management operations mentioned above.

10.1.3 Per-Region Initialization

Each driver instance is composed of one or more regions. Each region is automatically created with an initial region data area which begins with a `udi_init_context_t` structure. This structure helps bootstrap the region to the point were it can allocate its own data structures.

10.2 Per-Driver Initialization Structure

Every UDI driver module must contain a global variable named udi_init_info, of type udi_init_t. This structure contains information describing the module's entry points, control block usage, and other information necessary to initialize the driver. The environment processes the information contained in this structure before executing any driver code.

This section contains descriptions of the various components of the udi_init_info structure.

NAME **udi_init_info** *Module initialization structure*

SYNOPSIS

```
#include <udi.h>

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

udi_init_t udi_init_info;
```

MEMBERS

primary_init_info is a pointer to a structure containing information about the driver's primary region, used in the driver's primary module. For secondary modules, this must be set to NULL.

secondary_init_list is a pointer to a list of structures containing information about each type of secondary region implemented in this module, if any. The list is terminated with an entry containing a zero `region_idx`. A NULL pointer is treated the same as an empty list.

ops_init_list is a pointer to a list of structures containing information about channel operations usage for each ops vector implemented in this module. The list is terminated with an entry containing a zero `ops_idx`. ***ops_init_list*** must contain at least one entry, and must include at least one entry for each metalanguage used in this module.

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`. A NULL pointer is treated the same as an empty list.

gcb_init_list is a pointer to a list of structures containing information about generic control block usage in this module, if any. The list is terminated with an entry containing a zero `cb_idx`. A NULL pointer is treated the same as an empty list.

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 ops vectors. The list is terminated with an entry containing a zero `cb_idx`. A NULL pointer is treated the same as an empty list.

DESCRIPTION

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 Static Driver Properties as the module with a "region" declaration for region index zero, which is the primary region. The primary module of any driver must have a non-NULL ***primary_init_info***. If the primary module also manages some secondary regions, the module must also include a non-empty ***secondary_init_list***.

See `udi_cb_init_t` for details on how ***cb_select_list*** is used.

REFERENCES

`udi_primary_init_t, udi_secondary_init_t, udi_ops_init_t, udi_cb_init_t, udi_gcb_init_t, udi_cb_select_t`

NAME **udi_primary_init_t** *Primary region initialization structure*

SYNOPSIS

```
#include <udi.h>

typedef const struct {
     udi_mgmt_ops_t *mgmt_ops;
     const udi_ubit8_t *mgmt_op_flags;
     udi_size_t mgmt_scratch_requirement;
     udi_ubit8_t enumeration_attr_list_length;
     udi_size_t rdata_size;
     udi_size_t child_data_size;
     udi_ubit8_t per_parent_paths;
} udi_primary_init_t;

/* Maximum Legal Scratch Requirement */
#define UDI_MAX_SCRATCH                    4000

/* Operation Flags */
#define UDI_OP_LONG_EXEC                   (1U<<0)
```

MEMBERS

mgmt_ops is a pointer to an ops vector of driver entry points for the required Management Metalanguage channel operation routines. See Chapter 24, "*Management Metalanguage*" for details on Management Metalanguage and its channel operations.

mgmt_op_flags is a pointer to an array of flag values with a one-for-one correspondence between entries in the ***mgmt_op_flags*** array and entries in the ***mgmt_ops*** array. This array may be used to indicate characteristics of the implementation of the corresponding operation to the environment:

UDI_OP_LONG_EXEC - indicates to the environment that the corresponding operation will execute for an extended period of time (relative to normal operations within this driver). This is a hint to the environment and allows the environment to make implementation specific decisions when scheduling the driver to perform this operation.

mgmt_scratch_requirement specifies in bytes the driver's requirements for scratch area size in Management Metalanguage control blocks received over its management channel. The scratch size specified here must be the maximum of the driver's scratch size needs across the various control blocks in the Management Metalanguage. This value must not exceed `UDI_MAX_SCRATCH` (4000 bytes).

enumeration_attr_list_length is the number of elements of the `udi_instance_attr_list_t` array that the driver requires for ***attr_list*** in `udi_enumerate_cb_t` control blocks during udi_enumerate_req. This may be zero if the driver uses udi_enumerate_no_children for its udi_enumerate_req entry point routine.

rdata_size is the size, in bytes, of the region data area to be allocated for the primary region of each driver instance. ***rdata_size*** must be at least sizeof(udi_init_context_t) and must not exceed UDI_MIN_ALLOC_LIMIT (see **udi_limits_t** on page 10-18).

child_data_size is the size, in bytes, of child data to be allocated for each call to udi_enumerate_req. ***child_data_size*** must not exceed UDI_MIN_ALLOC_LIMIT.

per_parent_paths specifies the number of path handles that the environment should supply for each parent that is bound to this driver instance. The path handles are supplied to the driver in the udi_channel_event_cb_t control block for each UDI_CHANNEL_BOUND event.

DESCRIPTION

The udi_primary_init_t structure contains information the environment needs to subsequently create a new primary region and associated management channel when each driver instance for this driver is instantiated. This structure is part of udi_init_info.

The primary region's *region data area* is a memory area that is ***rdata_size*** bytes in size. It contains a udi_init_context_t structure at the front of this data area. When the region is created the bytes following the init context structure will be initialized to zero. The driver must never free this memory; it will be freed automatically when the region is destroyed.

When the Management Agent in the environment creates a driver instance, it will automatically create a primary region according to the parameters provided in the udi_primary_init_t structure. 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.

The primary region's end of the management channel will be anchored using ***mgmt_ops*** as the ops vector and ***mgmt_scratch_requirement*** as the scratch size requirement for all Management Metalanguage control blocks.

Drivers may also request the creation of secondary regions within a driver instance, by using udi_secondary_init_t.

REFERENCES

udi_init_info, udi_mgmt_ops_t, udi_init_context_t, udi_secondary_init_t, udi_instance_attr_list_t, udi_enumerate_cb_t, udi_enumerate_req, udi_limits_t

NAME **udi_secondary_init_t** *Secondary region initialization structure*

SYNOPSIS

```
#include <udi.h>

typedef const struct {
     udi_index_t region_idx;
     udi_size_t rdata_size;
} udi_secondary_init_t;
```

MEMBERS

region_idx is a non-zero driver-dependent index value that indirectly identifies a late-bound set of platform-dependent properties (address and capability domains, memory residence, priority, etc.) that will be attached to new secondary regions of this type, or zero to terminate the ***secondary_init_list*** list to which this structure belongs (see `udi_init_info`). If ***region_idx*** is zero, all other members of this structure are ignored. These properties are derived from the driver's region attributes (see Section 30.6.8, "Region Declaration," on page 30-18). Drivers typically define mnemonic constants associated with each region index that name the type of region being created (e.g., `MY_INT_REGION`, `MY_INBOUND_REGION`).

rdata_size is the size, in bytes, of the region data area to be allocated for secondary regions created with the given region index. ***rdata_size*** must be at least `sizeof(udi_init_context_t)` and must not exceed `UDI_MIN_ALLOC_LIMIT` (see **udi_limits_t** on page 10-18).

DESCRIPTION

The `udi_secondary_init_t` structure contains information the environment needs to subsequently create a new secondary region and associated internal bind channel when each driver instance for this driver is instantiated or when new parents or children are bound to that instance. This structure is part of `udi_init_info`.

If non-zero, the ***region_idx*** value must match a region index in a "region" declaration in the driver's static driver properties that is associated with this module and must be unique with respect to all other `udi_secondary_init_t` structures for the same driver (even for separate modules in a multi-module driver). Each module's `udi_init_info` must include a set of `udi_secondary_init_t` structures exactly matching the set of secondary region types serviced by that module.

The secondary region's *region data area* is a memory area that is ***rdata_size*** bytes in size. It contains a `udi_init_context_t` structure at the front of this data area. When the region is created the bytes following the init context structure will be initialized to zero. The driver must never free this memory; it will be freed automatically when the region is destroyed.

The environment will automatically create secondary regions as needed, according to the parameters provided in the `udi_secondary_init_t` structure and associated static driver properties (see Chapter 30, "*Static*

Driver Properties"). If the corresponding "region" declaration has its `binding` attribute set to `dynamic`, secondary regions of this type will be created as needed, dynamically, after the driver instance has been created, when a parent or child instance of the appropriate type is bound to this driver instance. Otherwise, exactly one secondary region of this type will be created as part of the initial driver instantiation.

In either case, a channel will also be created and anchored between the primary region and this secondary region as the secondary region's *internal bind channel*. The channel context for the primary end of this channel will be set to point to the primary region's region data area. The channel context for the secondary end of this channel will be set to point to the secondary region's region data area.

The primary region's end of the channel will be anchored using the ops vector selected by <primary_ops_idx> in the "internal_bind_ops" declaration that has the same region index value. The secondary region's end of the channel will be anchored using the ops vector selected by <secondary_ops_idx> from the same "internal_bind_ops" declaration. See `udi_ops_init_t` for details on how ***ops_idx*** values are used.

When the new region has been fully initialized, the environment delivers a `udi_channel_event_ind` with a `UDI_CHANNEL_BOUND` event code to one end of the new internal bind channel. Depending upon the metalanguage definition for the metalanguage indicated by the above ***ops_idx*** values, one end or the other will be considered the *initiator* and the other end will be the *responder*. It is the *initiator* end that receives the `UDI_CHANNEL_BOUND` event. This allows the *initiator* to receive the channel handle for its end of this channel (from the ***channel*** member of the control block structure).

As a recommended and expected (but not required) convention in the driver-internal metalanguage definition, the *initiator* should send some form of initialization operation (bind operation) to the *responder* over the internal bind channel. This allows the *responder* to determine the channel handle for its end of the channel, and can also be used to pass parameters that will help set up the region, choose structure sizes, initialize fields, etc., so that it can become ready to be "open for business".

REFERENCES `udi_init_info, udi_init_context_t, udi_ops_init_t, udi_primary_init_t, udi_channel_event_ind, udi_cb_t, udi_limits_t`

NAME **udi_ops_init_t** *Ops vector initialization structure*

SYNOPSIS

```
#include <udi.h>

typedef void udi_op_t(void);

typedef udi_op_t * const udi_ops_vector_t;

typedef const 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;
     const udi_ubit8_t *op_flags;
} udi_ops_init_t;
```

MEMBERS

ops_idx is 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 (see `udi_init_info`). If ***ops_idx*** is zero, all other members of this structure are ignored.

meta_idx is a non-zero metalanguage index number, assigned by the driver to uniquely identify a set of metalanguage related properties for a particular metalanguage.

meta_ops_num is a metalanguage-specific number, defined in the corresponding metalanguage specification, that uniquely identifies a type of ops vector with respect to other ops vector types in the same metalanguage.

chan_context_size is the size, in bytes, of a context area that will be automatically allocated, if non-zero, whenever this ***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_child_chan_context_t)` for child bind channels or `sizeof(udi_chan_context_t)` for other bind channels and must not exceed `UDI_MIN_ALLOC_LIMIT` (see **udi_limits_t** on page 10-18).

ops_vector is a pointer to the metalanguage-specific `<<`*`meta`*`>>_<<`*`role`*`>>_ops_t` channel ops vector structure containing pointers to the various entry point routines for this type of channel. The structure pointed to by ***ops_vector*** 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 an initializer for ***ops_vector***.

op_flags is a pointer to an array of flag values with a one-for-one correspondence between entries in the ***op_flags*** array and entries in the ***ops_vector*** array. This member is used in the same way as the ***mgmt_op_flags*** member of the `udi_primary_init_t` structure.

DESCRIPTION

The `udi_ops_init_t` structure contains information the environment needs to subsequently create channel endpoints for a particular type of ops vector and control block usage. This structure is part of `udi_init_info`.

If non-zero, the ***ops_idx*** value must be unique with respect to all other `udi_ops_init_t` structures for the same driver (even for separate modules in a multi-module driver). Each module's `udi_init_info` for a particular metalanguage must include a set of `udi_ops_init_t` structures exactly matching the set of ops vectors used in that module.

If non-zero, the ***meta_idx*** value must match a `meta_idx` value from a "meta" declaration in the driver's static driver properties and must be unique with respect to all other ***meta_idx*** values for the same driver.

The ***meta_ops_num*** values in metalanguage definitions are typically named *<<meta>>_<<role>>*_OPS_NUM and each correspond to an ops vector type, typically named *<<meta>>_<<role>>*_ops_t. The ***ops_vector*** member must point to a constant initialized structure of this ops vector type.

When ***chan_context_size*** is non-zero and this ***ops_idx*** is used for a child, parent, or internal binding (as indicated by a matching "child_bind_ops", "parent_bind_ops", or "internal_bind_ops" declaration in the driver's static driver properties), a new context structure of this size is automatically allocated. For the parent's end of a parent-child bind channel, the new context structure will begin with a `udi_child_chan_context_t`; for all other channels it will begin with a `udi_chan_context_t`. The requested ***chan_context_size*** must be at least as large as the size of the appropriate header structure. Any remaining bytes will be initialized to zero.

chan_context_size is ignored for driver-spawned channels.

REFERENCES

`udi_init_info, udi_limits_t, udi_chan_context_t, udi_child_chan_context_t`

NAME **udi_cb_init_t** *Control block initialization structure*

SYNOPSIS

```
#include <udi.h>

typedef const 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;
     udi_layout_t *inline_layout;
} udi_cb_init_t;

/* Maximum Legal Scratch Requirement */
#define UDI_MAX_SCRATCH                    4000
```

MEMBERS

cb_idx is a non-zero control block index number, assigned by the driver to uniquely identify this set of control block related properties for use in other initialization structures and service calls, or zero to terminate the ***cb_init_list*** list to which this structure belongs (see `udi_init_info`). If ***cb_idx*** is zero, all other members of this structure are ignored.

meta_idx is a non-zero metalanguage index number, assigned by the driver to uniquely identify a set of metalanguage related properties for a particular metalanguage.

meta_cb_num is a metalanguage-specific number, defined in the corresponding metalanguage specification, that uniquely identifies a control block group with respect to other control block groups in the same metalanguage.

scratch_requirement specifies in bytes the driver's requirements for scratch area size in control blocks allocated with this ***cb_idx*** or received via a channel operation of the appropriate type. The scratch size specified here must be the maximum of the driver's scratch size needs across the various control blocks in the control block group indicated by ***meta_cb_num***. This value must not exceed `UDI_MAX_SCRATCH` (4000 bytes).

inline_size is the size, in bytes, of a piece of inline memory to allocate and associate with the control block. This value must not exceed `UDI_MIN_ALLOC_LIMIT` (see **udi_limits_t** on page 10-18)

inline_layout is a pointer to a data layout specifier that describes the structure of the inline memory, if necessary.

DESCRIPTION

The `udi_cb_init_t` structure contains information the environment needs to subsequently create and manage control blocks of a particular type. This structure is part of `udi_init_info`.

If non-zero, the ***cb_idx*** value must be unique with respect to all other `udi_cb_init_t` and `udi_gcb_init_t` structures for the same driver (even for separate modules in a multi-module driver). Each module's `udi_init_info` metalanguage must include a set of `udi_cb_init_t` structures exactly matching the set of control block groups used in that module. Even if the corresponding ***cb_idx*** is never directly referenced in the module, a `udi_cb_init_t` must be included for any ***meta_cb_num*** that might be received with a channel operation.

If non-zero, the ***meta_idx*** value must match a `meta_idx` value from a "meta" declaration in the driver's static driver properties and must be unique with respect to all other ***meta_idx*** values for the same driver.

The ***meta_cb_num*** values in metalanguage definitions are typically named *<<meta>>_<<cbgroup>>*_CB_NUM and each correspond to a control block group which consists of one or more control block types, typically named *<<meta>>_<<cbtype>>*_cb_t.

When control blocks are first allocated, their scratch size is determined by the ***scratch_requirement*** value for the specified ***cb_idx***. When a control block is passed across a channel (using a channel operation), its scratch area may need to grow to meet the requirements of the target region. The environment determines the required minimum size by examining the ***cb_select_list*** for the `udi_ops_init_t` corresponding to the receiving channel endpoint. If an entry is found that matches the control block group for the control block that is being passed, then the ***scratch_requirement*** value for that ***cb_idx*** will be used; otherwise, the maximum of all ***scratch_requirement*** values for all `udi_cb_init_t` structures with the appropriate ***meta_cb_num*** will be used.

Inline memory pieces, if any, are allocated when a control block is first allocated, and are not resized or reshaped for the life of the control block, regardless of the ***cb_idx*** values with which the control block becomes associated when it is passed across channels. This memory is automatically transferred to the target region with the control block.

The ***inline_size*** member is used if the control block group includes a control block type that has any inline pointers (i.e. one whose layout specifier includes `UDI_DL_INLINE_UNTYPED`, `UDI_DL_INLINE_TYPED`, or `UDI_DL_INLINE_DRIVER_TYPED`) and the driver allocates a control block using this ***cb_idx***; otherwise ***inline_size*** must be zero. The corresponding inline pointer in each allocated control block will be set to point to memory of the appropriate size, or NULL if ***inline_size*** is zero. Drivers must not modify inline pointers.

The ***inline_layout*** member is used if the structure of the inline memory is driver-dependent (as indicated by a `UDI_DL_INLINE_DRIVER_TYPED` in the control block layout specifier) and ***inline_size*** is not zero, otherwise, ***inline_layout*** must be NULL.

If ***inline_size*** is zero and the layout specifier contains a corresponding UDI_DL_INLINE_DRIVER_TYPED layout element, the ***cb_idx*** may be used with udi_cb_alloc_dynamic instead of udi_cb_alloc, to provide the inline size and layout dynamically.

REFERENCES udi_init_info, udi_meta_init_t, udi_ops_init_t, udi_gcb_init_t, udi_layout_t, udi_mei_op_template_t, udi_limits_t, udi_cb_alloc, udi_cb_alloc_dynamic

NAME **udi_cb_select_t** *Control block selections for incoming channel ops*

SYNOPSIS

```
#include <udi.h>

typedef const struct {
     udi_index_t ops_idx;
     udi_index_t cb_idx;
} udi_cb_select_t;
```

MEMBERS

cb_idx is a control block index number that must match a ***cb_idx*** value in a udi_cb_init_t associated with the udi_meta_init_t from which this structure is referenced, or must be zero to terminate a ***cb_select_list*** list to which this structure belongs (see udi_init_info).

DESCRIPTION

The udi_cb_select_t structure contains information the environment needs to subsequently manage scratch requirements of control blocks that are passed across channels. This structure is part of udi_init_info.

udi_cb_select_t entries can be used to override the default algorithm for determining scratch requirements for control blocks that are received via channel operations when there are multiple udi_cb_init_t structures for the same ***meta_cb_num*** and ***meta_idx***. By default, the scratch requirement is computed as the maximum from all matching udi_cb_init_t structures. However, if a udi_cb_select_t entry is present for the appropriate ***ops_idx*** that has a ***cb_idx*** matching one of the candidate udi_cb_init_t structures, then the scratch requirement from that structure is used instead.

udi_cb_select_t entries are optional and will not be needed by most drivers.

In all cases, control block allocation (with udi_cb_alloc) uses the specific properties associated with a control block index parameter, and is unaffected by udi_cb_select_t entries.

REFERENCES

udi_init_info, udi_cb_init_t, udi_ops_init_t, udi_cb_alloc

NAME **udi_gcb_init_t** *Generic control block initialization properties*

SYNOPSIS

```
#include <udi.h>

typedef const struct {
     udi_index_t cb_idx;
     udi_size_t scratch_requirement;
} udi_gcb_init_t;

/* Maximum Legal Scratch Requirement */
#define UDI_MAX_SCRATCH                4000
```

MEMBERS

cb_idx is a non-zero control block index number, assigned by the driver to uniquely identify this set of control block related properties for use in other initialization structures and service calls, or zero to terminate the ***gcb_init_list*** list to which this structure belongs (see udi_init_info). If ***cb_idx*** is zero, all other members of this structure are ignored.

scratch_requirement specifies in bytes the driver's requirements for scratch area size in generic control blocks allocated with this ***cb_idx***. This value must not exceed UDI_MAX_SCRATCH (4000 bytes).

DESCRIPTION

Control blocks that are to be used only for asynchronous environment service calls, such as udi_mem_alloc, and not for any channel operations, may be allocated using a control block index that is initialized with a udi_gcb_init_t. This structure is part of udi_init_info.

Such control blocks have no metalanguage-specific visible part and are directly referenced by the udi_cb_t generic control block pointer. As a result, these control blocks must not be used (or defined for use) in channel operations.

If non-zero, the ***cb_idx*** value must be unique with respect to all other udi_cb_init_t and udi_gcb_init_t structures for the same driver (even for separate modules in a multi-module driver).

REFERENCES

udi_init_info, udi_cb_init_t, udi_cb_t, udi_cb_alloc

10.3 Initial Region Data Structures

The initial region data structure provided to the driver include a system limits structure (udi_limits_t) along with space for the driver's private per-region data. This structure is allocated by the UDI environment according to the ***rdata_size*** member of the appropriate udi_primary_init_t or udi_secondary_init_t in udi_init_info.

A pointer to the initial region data structure (udi_init_context_t) 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. The channel handle is available via cb->gcb.channel.

NAME **udi_init_context_t** *Initial context for new regions*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_index_t  region_idx;
     udi_limits_t  limits;
} udi_init_context_t;
```

MEMBERS

region_idx is a region index value that indicates the type of this region. For a driver's primary region, it will be zero. For secondary regions, it will be the value from the `udi_secondary_init_t` that was used to create this region.

limits is a structure that describes system resource limits. See **udi_limits_t** on page 10-18.

DESCRIPTION

The `udi_init_context_t` structure is stored at the front of the region data area of each newly created region, providing initial data that a driver will need to begin executing in the region. A pointer to this structure (and therefore the region data area as a whole) is made available to the driver as the initial channel context for its first channel.

For primary regions, the first channel will be the management channel for the driver instance (see Chapter 24, "*Management Metalanguage*"). For secondary regions, the first channel will be the initial channel between this secondary region and the primary region, using either the Generic I/O Metalanguage (see Chapter 25) or a custom metalanguage.

REFERENCES

`udi_init_info, udi_limits_t`

NAME **udi_limits_t** *Platform-specific allocation and access limits*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_size_t max_legal_alloc;
     udi_size_t max_safe_alloc;
     udi_size_t max_trace_log_formatted_len;
     udi_size_t max_instance_attr_len;
     udi_ubit32_t min_curtime_res;
     udi_ubit32_t min_timer_res;
} udi_limits_t;

/* architectural minimums */
#define UDI_MIN_ALLOC_LIMIT               4000
#define UDI_MIN_TRACE_LOG_LIMIT           200
#define UDI_MIN_INSTANCE_ATTR_LIMIT       64
```

MEMBERS

max_legal_alloc is the maximum legal memory allocation size for this system. Any larger size passed to `udi_mem_alloc` will produce indeterminate results, which could include termination of the driver or region, or even a complete system abort.

max_safe_alloc is the maximum memory allocation size that must be passed to `udi_mem_alloc` without being prepared to cancel an unsuccessful allocation (see definition of "safe limits" below).

max_trace_log_formatted_len is the maximum legal size of the formatted result of a call to `udi_trace_write` or `udi_log_write`.

max_instance_attr_len is the maximum legal size of an instance attribute value.

min_curtime_res is the minimum time difference, in nanoseconds, between successive unique values returned by `udi_time_current`.

min_timer_res is the minimum resolution, in nanoseconds, of timers registered with `udi_timer_start_repeating` or `udi_timer_start`. See **udi_timer_start** on page 14-4 for details on how ***min_timer_res*** affects timer operation.

DESCRIPTION

`udi_limits_t` reflects implementation-dependent system limits, such as memory allocation and timer resolution limits, for a particular region. These limits may vary from region to region, but will remain constant for the life of a region.

The `udi_limits_t` structure is passed to a driver instance via the `udi_init_context_t` in its initial region data.

Since UDI can be implemented on a wide variety of systems from small embedded systems to large server systems, the memory available for drivers can vary widely. `udi_limits_t` allows drivers to adjust their allocation algorithms to best fit their environment.

There are two types of allocation limits: *legal* limits and *safe* limits. Legal limits represent the absolute upper bound on a single allocation. Drivers must not make requests that would exceed the legal limits.

Safe limits represent the maximum amount that a driver may safely request without arranging to deal with unsuccessful allocations. For any size greater than the safe limit (but not exceeding the legal limit), drivers must cancel the request (using `udi_cancel`) after a reasonable amount of time has expired. To do this, the driver may set a timer using `udi_timer_start` or `udi_timer_start_repeating`. Drivers are expected to be coded as if allocations below the safe limit will always eventually succeed.

The ***max_legal_alloc*** and ***max_safe_alloc*** limits affect the size of virtually-contiguous driver memory allocations via `udi_mem_alloc`. These allocation limits are guaranteed to be greater than or equal to `UDI_MIN_ALLOC_LIMIT` in all UDI environments. This means drivers don't need to check these limits for requests that don't exceed `UDI_MIN_ALLOC_LIMIT` bytes. C language structures that need to be dynamically allocated should be limited to `UDI_MIN_ALLOC_LIMIT` bytes in size, so they can be allocated directly with a simple `udi_mem_alloc` call.

The ***max_trace_log_formatted_len*** limit specifies the maximum size, in bytes, of strings resulting from formatting messages passed to `udi_trace_write` or `udi_log_write`. This limit is guaranteed to be greater than or equal to `UDI_MIN_TRACE_LOG_LIMIT` in all UDI environments.

The ***max_instance_attr_len*** parameter specifies the maximum size, in bytes, of a device instance attribute value (see Chapter 15, "*Instance Attribute Management*") that can be handled by the environment. This limit is guaranteed to be greater than or equal to `UDI_MIN_INSTANCE_ATTR_LIMIT` in all UDI environments.

The ***min_curtime_res*** and ***min_timer_res*** parameters specify the corresponding resolution of the system chronological timer and system timeout timer, respectively (see Chapter 14, "*Time Management*"). Current time values will change no faster than the amount of time specified by ***min_curtime_res***, and timers will not be scheduled with any better resolution or granularity than the ***min_timer_res*** specification.

REFERENCES

`udi_mem_alloc, udi_cancel, udi_timer_start, udi_timer_start_repeating, udi_init_context_t, udi_instance_attr_set, udi_trace_write, udi_log_write`

NAME **udi_chan_context_t** *Initial context for bind channels*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     void *rdata;
} udi_chan_context_t;
```

MEMBERS

rdata is a pointer to the driver instance's initial region data.

DESCRIPTION

The `udi_chan_context_t` structure is stored at the front of the channel context structure pre-allocated for new bind channels whose ***chan_context_size*** is non-zero (see **udi_ops_init_t** on page 10-9), except for child-bind channels, which use `udi_child_chan_context_t`.

A pointer to this structure is made available to the driver as the initial channel context for the corresponding bind channel. Drivers may subsequently change the channel context, but must not free this structure; the environment will free it when the channel is unbound.

REFERENCES

`udi_init_info, udi_child_chan_context_t, udi_init_context_t, udi_ops_init_t`

NAME **udi_child_chan_context_t** *Initial channel context for child-bind channels*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     void *rdata;
     udi_ubit32_t child_ID;
} udi_child_chan_context_t;
```

MEMBERS

rdata is a pointer to the driver instance's initial region data.

child_ID is the child ID value initially supplied by this driver in the `udi_enumerate_ack` operation which enumerated this child. This value allows the parent to uniquely determine which child this new bind channel is connected to.

DESCRIPTION

The `udi_child_chan_context_t` structure is stored at the front of the channel context structure pre-allocated for new bind channels whose ***chan_context_size*** is non-zero (see **udi_ops_init_t** on page 10-9).

A pointer to this structure is made available to the driver as the initial channel context for the corresponding bind channel. Drivers may subsequently change the channel context, but must not free this structure; the environment will free it when the channel is unbound.

REFERENCES

`udi_init_info`, `udi_chan_context_t`, `udi_init_context_t`, `udi_ops_init_t`, `udi_enumerate_ack`

Control Block Management *11*

11.1 Overview

The UDI service calls available to the driver can be divided into two classes:

1) service calls not requiring external resources
2) service calls that (may) require external resources

Service calls of the first type resemble conventional system service calls, however service calls of the second type may require the environment to obtain resources to complete the service request. When resources must be obtained, the service call cannot complete immediately with the requested resources because they may not be presently available; a callback is used instead to handle completion for these types of requests so that the driver may be re-entered once the resources are available.

Service calls of the first type are referred to as *synchronous service calls*, whereas those of the second type are referred to as *asynchronous service calls*. See also the discussion of "Asynchronous Service Calls" on page 7-4 and the "Function Call Classifications" on page 4-4.

The UDI *control block* provides the context for the second type of service call. The control block can be used to marshall and unmarshall the parameters for a request and to allow the environment to queue the request internally and maintain context-oriented status. While the driver owns the control block it can be used for similar queuing and status/context purposes within the driver.

Metalanguage-specific channel operations (see Chapter 23, *"Introduction to UDI Metalanguages"*) also use UDI control blocks for similar purposes.

The generic control block is a representation of the basic elements common to all UDI control blocks. Most UDI service calls requiring a control block will accept *any* control block but are defined in terms of the generic control block; convenience macros are also provided to obtain a generic control block reference for any specific control block and vice versa.

11.2 Control Block Service Calls and Macros

The service calls and macros used to manipulate control blocks are described in the paragraphs that follow.

NAME **udi_cb_t** *Generic, least-common-denominator control block*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_channel_t channel;
     void *context;
     void *scratch;
     void *initiator_context;
     udi_origin_t origin;
} udi_cb_t;
```

MEMBERS

channel is a handle to the channel currently associated with this control block. When used in a channel operation, the main control block's ***channel*** member is used as the target channel, and must not—at that time—be UDI_NULL_CHANNEL. For environment implementation reasons, ***channel*** must never be explicitly set by the driver to UDI_NULL_CHANNEL.

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.

See udi_channel_set_context for details on how channel context is determined.

scratch is a pointer to the control block's scratch area. Drivers must not change this pointer, but may change any of the bytes in the space pointed to by ***scratch***, up to the required scratch size specified by the appropriate udi_cb_init_t in the driver's udi_init_info.

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 it needs to complete the operation.

Any driver receiving a request or indication operation must use the same control block in its (acknowledgement, nak, or response) reply, and must not modify the ***initiator_context*** value. In fact, the value of ***initiator_context*** is unspecified except when the control block is owned by the initiating region, so must not be compared, dereferenced, or otherwise used from any other region.

origin is a handle to the origination information for the current request. This is set in the original control block by the environment; each module must copy this field from input control blocks to any other control blocks used to complete work requested by the input control block. Any control block used in an asynchronous service call or channel operation that is not associated with an incoming request control block must set origin to the `UDI_NULL_ORIGIN` value.

DESCRIPTION

The `udi_cb_t` structure is used for generically handling control blocks and accessing their common members. All metalanguage-specific control blocks have a `udi_cb_t` structure as their first structure member.

The `udi_cb_t` structure is a semi-opaque type, and must only be allocated by environment service calls. Control blocks are transferable between regions, when used as the main control block for a channel operation, or chained from that control block as part of a linked list of identically-typed control blocks.

When a new control block is allocated, its ***context*** and ***origin*** members are initialized to the ***context*** value from the original control block, its ***channel*** member is initialized according to the ***default_channel*** argument block passed to `udi_cb_alloc`, and its ***initiator_context*** value is unspecified.

The driver that currently owns the control block may change the ***channel*** and ***context*** members at any time while the control block is not in use with an environment service call. If the control block is not already part of an in-progress request/response sequence (that is, not transferred to this region from another region as part of a request or indication operation), the controlling driver may also change the ***initiator_context*** value.

All members of `udi_cb_t` and other visible fields in a metalanguage-specific control block, as well as the scratch area contents, are preserved across asynchronous service calls, but not across channel operations. The only member of `udi_cb_t` that is preserved across a channel operation is ***initiator_context***, and that only when the control block is returned to the initiating region.

When a control block or a chain of control blocks is passed to another region via a channel operation, the ***channel*** and ***context*** members of each control block are automatically set to the channel handle for the target region's end of the channel and the channel context for that endpoint, respectively, before the target region's entry point is invoked.

REFERENCES

`udi_init_info, udi_cb_init_t, udi_cb_alloc`

NAME **udi_cb_alloc** *Allocate a new control block*

SYNOPSIS

```
#include <udi.h>

void udi_cb_alloc (
     udi_cb_alloc_call_t *callback,
     udi_cb_t *gcb,
     udi_index_t cb_idx,
     udi_channel_t default_channel );

typedef void udi_cb_alloc_call_t (
     udi_cb_t *gcb,
     udi_cb_t *new_cb );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

cb_idx is a control block index that indicates required properties of the control block, such as metalanguage type and scratch size.

default_channel is a channel handle that, if set to a value other than UDI_NULL_CHANNEL, is used as the initial value for the new control block's channel member. If set to UDI_NULL_CHANNEL, the environment is free to initialize the channel member with some other value, so the driver must not depend on it containing UDI_NULL_CHANNEL.

new_cb is a pointer to the newly allocated control block.

DESCRIPTION

udi_cb_alloc allocates a new control block for use by the driver. The new control block can be used to allocate other resources using any UDI service request or to invoke channel operations appropriate to the specified control block type.

While such allocations are usually performed using a specific control block already associated with a channel operation, the new control block returned by udi_cb_alloc provides a way to continue or complete the channel operation without waiting for a service call to complete. This is particularly useful when initiating delayed callbacks with udi_timer_start or udi_timer_start_repeating.

When a new control block is allocated, its ***context*** member is initialized to the ***context*** value from ***gcb***, its ***origin*** member is initialized to the ***origin*** value from ***gcb***, its ***channel*** member is initialized according to the ***default_channel*** argument block passed to udi_cb_alloc, and its ***initiator_context*** value is unspecified.

The scratch pointer of the new control block is initialized to point to the associated scratch area and the pointer must not be modified by the driver. If the driver's scratch requirement is zero, the value of the scratch pointer is unspecified and it must not be dereferenced.

The initial values in the new control block's scratch space are unspecified; they are not guaranteed to be zero. Similarly, for metalanguage-specific control blocks that have additional visible structure members, the initial value of these structure members are also unspecified.

WARNINGS

The control block obtained with this call must not be used with metalanguage-related channel operations other than those appropriate for the control block type associated with ***`cb_idx`***. If the control block index was associated with a `udi_gcb_init_t` in `udi_init_info`, rather than a metalanguage-specific `udi_cb_init_t`, then the new control block must not be used with any channel operations.

The driver must not explicitly set the channel member of the returned control block to `UDI_NULL_CHANNEL` at any time and must not expect `UDI_HANDLE_IS_NULL` to return TRUE for the channel member of a control block even if ***`default_channel`*** was `UDI_NULL_CHANNEL`.

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

REFERENCES

`udi_cb_t, udi_cb_free, udi_timer_start, udi_timer_start_repeating, udi_init_info, udi_cb_init_t`

NAME **udi_cb_alloc_dynamic** *Allocate a control block with variable inline layout*

SYNOPSIS

```
#include <udi.h>

void udi_cb_alloc_dynamic (
     udi_cb_alloc_call_t *callback,
     udi_cb_t *gcb,
     udi_index_t cb_idx,
     udi_channel_t default_channel,
     udi_size_t inline_size,
     udi_layout_t *inline_layout );

typedef void udi_cb_alloc_call_t (
     udi_cb_t *gcb,
     udi_cb_t *new_cb );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

cb_idx
default_channel
new_cb are the same arguments as used in `udi_cb_alloc`.

inline_size is the size of the previously unspecified inline structure for this cb_idx.

inline_layout is the layout of the previously unspecified inline structure for this cb_idx. Must be `NULL` if the control block layout does not include `UDI_DL_INLINE_DRIVER_TYPED`.

DESCRIPTION

`udi_cb_alloc_dynamic` behaves like `udi_cb_alloc`, except that it allows the driver to specify the size and layout of an inline structure for the control block that was left unspecified in the driver's `udi_cb_init_t` structure with the given ***cb_idx***.

The ***inline_size*** and ***inline_layout*** members of the corresponding `udi_cb_init_t` structure (see page 10-11) must have been set to zero and NULL, respectively, and the control block layout must include exactly one `UDI_DL_INLINE_UNTYPED`, `UDI_DL_INLINE_TYPED`, or `UDI_DL_INLINE_DRIVER_TYPED` layout element.

It is recommended that `udi_cb_alloc` be used instead of `udi_cb_alloc_dynamic` if possible, as it's likely to be faster, but if the layout is not known statically, `udi_cb_alloc_dynamic` must be used.

WARNINGS

Use of the ***inline_layout*** parameter must conform to the rules described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls".

REFERENCES

`udi_cb_t, udi_cb_alloc, udi_layout_t, udi_init_info, udi_cb_init_t`

NAME **udi_cb_alloc_batch** *Allocate a batch of control blocks with buffers*

SYNOPSIS

```
#include <udi.h>

void udi_cb_alloc_batch (
	udi_cb_alloc_batch_call_t *callback,
	udi_cb_t *gcb,
	udi_index_t cb_idx,
	udi_index_t count,
	udi_boolean_t with_buf,
	udi_size_t buf_size,
	udi_buf_path_t path_handle );

typedef void udi_cb_alloc_batch_call_t (
	udi_cb_t *gcb,
	udi_cb_t *first_new_cb );
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the "Asynchronous Service Calls" section of "*Calling Sequence and Naming Conventions*".

cb_idx is a control block index that indicates required properties of the control block, such as metalanguage type and scratch size. All of the control blocks allocated will be of the same type as indicated by this ***cb_idx***.

count is the number of control blocks of this type to allocate in the batch operation.

with_buf is true if buffers should be allocated along with the control blocks. If true, a buffer of size ***buf_size*** will be allocated for each `udi_buf_t` pointer (`UDI_DL_BUF` layout entry) that exists in each allocated control block. If false, no buffers will be allocated and the values of the corresponding control block buffer pointers are undefined.

buf_size is the size of the buffers to be allocated if ***with_buf*** is true. This argument is ignored if ***with_buf*** is false.

path_handle is the handle identifying the intended use and dispatching of the allocated buffers. Path handle usage is determined by the driver, but by associating the use of a specific ***path_handle*** with buffers allocated for a specific purpose, the driver allows the environment to predict and optimize the allocated buffer requirements. This field is ignored if ***with_buf*** is false.

first_new_cb is a pointer to the first allocated control block in the list of returned control blocks. If ***count*** is zero, ***first_new_cb*** will be NULL.

DESCRIPTION

This service combines the use of `udi_cb_alloc` with `UDI_BUF_ALLOC` to allocate batches of one or more control blocks with optional associated buffers.

Consult `udi_cb_alloc` for more specifics on how the individual control blocks will be allocated and initialized.

Consult `UDI_BUF_ALLOC` and `udi_buf_write` for more specifics on how the individual buffers will be allocated and initialized.

The control blocks are returned to the caller by passing them as a chain or list. If the control block type allows control block chaining (i.e. the control block contains a pointer to another control block of the same type) then the chain field within the control blocks are used to link the returned control blocks: each control block's chain field will point to the next control block in the chain. If the control block type does not support chaining, then the ***initiator_context*** field of the returned control blocks is used to link the control blocks; the ***callback*** function should unlink the control blocks and reset the ***initiator_context*** as appropriate. The link pointer in the last control block shall be set to NULL.

WARNINGS

See the warnings for `udi_cb_alloc`.

Batch allocated control blocks must be unlinked before use unless actually used as a chain. Passing a control block to a channel operation or system service call relenquishes ownership of that control block and any chained control blocks. If the list is maintained via the ***initiator_context***, the driver is assured that the ***initiator_context*** will be returned unchanged, but is not guaranteed that the ***initiator_context*** will not be changed (or deallocated) while the driver does not own the control block.

REFERENCES

`udi_cb_t`, `udi_cb_alloc`, `udi_buf_t`, `UDI_BUF_ALLOC`, `udi_buf_write`

NAME **udi_cb_free** *Deallocates a previously obtained control block*

SYNOPSIS

```
#include <udi.h>

void udi_cb_free ( udi_cb_t *cb );
```

ARGUMENTS

cb is a pointer to the control block to be deallocated. If NULL, this function is a no-op.

DESCRIPTION

`udi_cb_free` releases the specified control block, including any metalanguage-specific parts, along with any associated resources back to the environment. ***cb*** must be NULL or must have been previously obtained by a call to `udi_cb_alloc`, or passed to the driver via a channel operation.

Note that `udi_cb_free` may be used to free any type of control block.

WARNING

The control block must not currently have any service call or callback pending. Any pending requests must first be cancelled with `udi_cancel`.

Management metalanguage control blocks and channel event control blocks must not be passed to `udi_cb_free`.

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*

REFERENCES

`udi_cb_alloc, udi_cancel, udi_channel_event_cb_t`

NAME **UDI_GCB** *Convert any control block to generic udi_cb_t*

SYNOPSIS

```
#include &lt;udi.h&gt;

#define UDI_GCB(mcb) (&(mcb)->gcb)
```

ARGUMENTS

mcb is a pointer to the control block reference to be converted.

DESCRIPTION

This macro is used to convert any UDI control block pointer into its generic control block representation (`udi_cb_t *`) suitable for use with a UDI service request. The original control block is not copied or re-allocated.

This macro is provided for convenience only. Its use is highly recommended but not required.

REFERENCES

`udi_cb_t`

NAME **UDI_MCB** *Convert a generic control block to a specific one*

SYNOPSIS

```
#include <udi.h>

#define UDI_MCB(gcb, cb_type) ((cb_type *)(gcb))
```

ARGUMENTS

gcb is a pointer to the control block reference to be converted.

cb_type is the type name for the desired specific control block type.

DESCRIPTION This macro is used to convert a generic control block pointer to a metalanguage-specific control block type. The original control block is not copied or re-allocated. The control block itself must already be of the type appropriate to ***cb_type***.

This macro is provided for convenience only. Its use is highly recommended but not required.

WARNINGS The control block referenced by ***gcb*** must have been previously obtained by a call to `udi_cb_alloc` with a ***cb_idx*** appropriate to ***cb_type***.

Control block usage must follow the rules described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*

REFERENCES `udi_cb_t`

NAME **udi_cancel** *Cancel a pending asynchronous service call*

SYNOPSIS

```
#include <udi.h>

void udi_cancel (
     udi_cancel_call_t *callback,
     udi_cb_t *gcb );

typedef void udi_cancel_call_t (
     udi_cb_t *gcb );
```

ARGUMENTS ***callback, gcb*** are standard arguments described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

DESCRIPTION Any service request with a pending callback can be canceled by this call, except timer requests (which must be canceled with `udi_timer_cancel`). The control block must be the same one specified when the service was requested, and must be active (i.e. the callback has not yet been called, regardless of whether or not allocations have actually completed).

`udi_cancel` must not be used with control blocks that have been passed to channel operations, but some channel operations can be aborted by using `udi_channel_op_abort`.

When a service request is cancelled, any whole or partially-allocated resources or data structures that would have been returned with that callback upon normal completion will be discarded (i.e. there will be no resource leaks). Further, any resources or data structures that would have been consumed by the original request (e.g. movable structs and objects referenced by transferable handles) will be consumed (and discarded), since there is no way to pass the object back to the original caller. Another way to look at this is that `udi_cancel` does not provide an undo operation, but rather an abort operation; any objects (such as a data buffer for `udi_buf_write`) being modified or created by the original request are destroyed by the abort.

Once the request has been cancelled, and any partial allocations released, the specified ***callback*** routine will be called instead of the original callback routine from the outstanding request. Ownership of the control block is transferred back to the requestor with this callback, and the control block is available for reuse.

WARNINGS `udi_cancel` must be called from the region that owned the control block at the time of the original request. It cannot be used to cancel a pending request in another region.

A driver must keep track of its in-progress requests to avoid canceling a different request than intended. See the example below for details. A good rule of thumb is that `udi_cancel` must not be used to cancel a request without first checking to see if the corresponding callback has been called.

If a driver issues a `udi_cancel` for a control block that is not active the driver is in error. See the "Driver Faults/Recovery" section of *"Execution Model"* for an explanation of how the environment may react to this driver error.

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*

Since ownership of control blocks are transferred away from the driver upon issuing a channel operation, any attempt to use `udi_cancel` to cancel a channel operation will be considered an error and will be handled as an environment-detected error in accordance with the "Driver Faults/Recovery" section of *"Execution Model"*.

EXAMPLES

The first example shows how ***not*** to use `udi_cancel`. The `udi_cancel` call in `ddd_step1` will not necessarily cancel the `udi_mem_alloc` call that immediately precedes it. In fact, it could even cancel the subsequent `udi_cb_alloc` request in `ddd_step2`, or even some further subsequent allocation in the callback sequence. This example is somewhat contrived in that there would typically be some reason the driver is canceling the request; it wouldn't simply do an allocation followed immediately by a cancel, but it illustrates the issues.

```
void
ddd_step1(ddd_context_t *context)
{
     ...
     udi_mem_alloc(ddd_step2, UDI_GCB(cb1), size, 0);
     udi_cancel(ddd_step1a, UDI_GCB(cb1));
}

void
ddd_step1a(
     udi_cb_t *gcb)
{
     /* Something has been canceled,
          but it's unclear what */
     ...
}

void
ddd_step2(
     udi_cb_t *gcb,
     void *new_mem)
{
     ...
     udi_cb_alloc(ddd_step3, UDI_GCB(cb1), idx, chan);
}

void
ddd_step3(
     udi_cb_t *gcb,
```

```
     udi_cb_t *new_cb)
{
     ...
}
```

To fix this problem, the driver must first check to see if the corresponding allocation callback has been received before calling `udi_cancel`. Adding such a check to the above code produces the following, which will cancel the immediately preceding `udi_mem_alloc` call if and only if the allocation doesn't complete immediately (i.e. isn't complete upon return). (Note that some environments may be designed to never do the callback immediately before returning. So this would not in general be a useful thing to do in the driver, but it does illustrate the issues.)

```
void
ddd_step1(
     ddd_context_t *context)
{
     ...
     context->mem_alloc_done = FALSE;
     udi_mem_alloc(ddd_step2, UDI_GCB(cb1), size, 0);
     if (!context->mem_alloc_done)
          udi_cancel(ddd_step1a, UDI_GCB(cb1));
}

void
ddd_step1a(
     udi_cb_t *gcb)
{
     /* udi_mem_alloc in step1 has been cancelled. */
     ...
}

void
ddd_step2(
     udi_cb_t *gcb,
     void *new_mem)
{
     ddd_context_t *context = gcb->context;
     ...
     context->mem_alloc_done = TRUE;
     udi_cb_alloc(ddd_step3, UDI_GCB(cb1), idx, chan);
}

void
ddd_step3(
     udi_cb_t *gcb,
     udi_cb_t *new_cb)
{
     ...
}
```

Note that the region serialization rules prevent reentrancy in the region code and therefore prevent the race conditions related to accesses and modifications of the `mem_alloc_done` variable that would normally need to be considered.

Memory Management 12

12.1 Overview

The UDI memory management services allow drivers to allocate and free blocks of region-local, virtually-contiguous memory. There are two types of virtually-contiguous memory allocation provided in UDI: (1) allocation of memory which is transferable, or movable, between regions (provides for memory which can either directly or indirectly be passed as an argument to a channel operation), and (2) allocation of memory which is not transferable between regions (may only be used within the context of the caller's region). In both cases the environment returns a region-local pointer to the allocated memory. In the transferable case it is up to the environment implementation of the channel operations to translate the region-local pointer being transferred to a region-local pointer in the target region. Non-transferable memory should be used wherever possible, as allocation of transferable memory may be more expensive in some environments.

Note – For memory copy, compare, and initialization utility functions, see Chapter 20, *"String/Memory Utility Functions"*.

When using memory allocation services, care must be taken to avoid excessive use of memory resources. Memory is a finite system resource. It is the device driver's responsibility to allocate, track and release memory back to the environment in a responsible manner.

The driver must also be careful to keep its memory demands in tune with the capabilities of the platform. Since UDI can be implemented on a wide variety of systems from small embedded systems to large server systems, the memory available for drivers can vary widely. When a driver region is created, it is provided with a set of platform-specific allocation limits (see **udi_limits_t** on page 10-18) to which it must conform. The resource managements operations in the Management Metalanguage provide additional resource utilization guidelines to the driver (see Section 24.4.2, "Resource Management," on page 24-6).

Warning – Memory allocated by `udi_mem_alloc` is intended for access by driver software and must not be used for Direct Memory Access from devices. DMA-addressable memory allocation is described in the DMA chapter of the *UDI Physical I/O Specification*, for those environments that support DMA and other physical I/O.

12.2 Memory Management Service Calls

The memory management service calls, which consist of `udi_mem_alloc` and `udi_mem_free`, are described in the paragraphs that follow.

NAME

udi_mem_alloc *Allocate memory for a virtually-contiguous object*

SYNOPSIS

```
#include <udi.h>

void udi_mem_alloc (
     udi_mem_alloc_call_t *callback,
     udi_cb_t *gcb,
     udi_size_t size,
     udi_ubit8_t flags );

typedef void udi_mem_alloc_call_t (
     udi_cb_t *gcb,
     void *new_mem );

/* Values for flags */
#define UDI_MEM_NOZERO                    (1U<<0)
#define UDI_MEM_MOVABLE                   (1U<<1)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

size is the number of bytes of space requested. See **udi_limits_t** on page 10-18 for limits on allocation sizes.

flags is a bitmask of optional flags, which may include zero or more of the following:

UDI_MEM_NOZERO — Don't zero memory contents.

UDI_MEM_MOVABLE — Allocate movable memory.

new_mem is a pointer to the new memory object. The driver is expected to cast this to the appropriate type of struct, array, etc. If ***size*** is zero, ***new_mem*** will be NULL.

DESCRIPTION

`udi_mem_alloc` allocates memory for a new virtually-contiguous object capable of storing at least ***size*** bytes. The newly allocated memory will be zeroed unless UDI_MEM_NOZERO is set, in which case the initial values are undefined.

The newly allocated memory will be aligned on the most restrictive alignment of the platform's natural alignments for *long* and pointer data types, allowing the allocated memory to be directly accessed as C structures.

If the UDI_MEM_MOVABLE flag is set, the memory will be allocated as *movable* memory. This means that it can be passed outside of the region from which it was allocated. Only movable memory may be pointed to by control block fields or channel operation parameters. UDI_MEM_MOVABLE should be used only if needed, as movable memory may be a more limited resource.

WARNINGS

The memory allocated by this routine has no particular physical or I/O bus-related properties. It is intended only for access by driver software.

Control block usage must follow the rules described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

The usage of memory allocated by this routine must follow the rules described in the “Memory Objects” section of “*Data Model*”.

REFERENCES `udi_mem_free, udi_cancel, udi_limits_t`

NAME **udi_mem_free** *Free a memory object*

SYNOPSIS

```
#include <udi.h>

void udi_mem_free (
     void *target_mem );
```

ARGUMENTS ***target_mem*** is a pointer to the memory object being deallocated.

DESCRIPTION udi_mem_free frees all resources associated with the specified memory object. The driver must not dereference the ***target_mem*** pointer once this function is called.

If ***target_mem*** is equal to NULL, explicitly or implicitly (zeroed by initial value or by using udi_memset), this function acts as a no-op. Otherwise, ***target_mem*** must have been allocated by udi_mem_alloc or passed to the driver as a movable memory block via a channel operation.

Note – The udi_init_context_t structure, the rest of the initial region data area, and any channel context structures pre-allocated by the environment, must not be freed by the driver and are not transferrable between regions.

REFERENCES udi_mem_alloc

Buffer Management 13

13.1 Overview

The service calls in this chapter are used to manage the data buffers that are used to carry “application” or “wire” data within the UDI environment. Any form of data transfer either to or from the device or between UDI modules will use a UDI buffer construct to reference that data.

In order to facilitate various device and DMA requirements and to avoid copying data, UDI buffers implement a layer of abstraction between the driver and the actual data. A device driver does not typically need to access data with the exception of various headers or tags, so the lack of direct access to the data is typically not even noticed in the UDI driver.

Using this abstraction, drivers are presented with a “logical” view of the data as a single contiguous block of data accessible via UDI buffer read/write operations. The implementation of the UDI buffer is determined by the UDI environment implementation and a single UDI environment may have several different buffer implementations supporting the UDI driver-to-buffer interface. This facility allows buffer data to be distributed into multiple virtual and physical segments as needed and desired to achieve the aforementioned goals of copy avoidance and natural DMA presentation.

Another valuable effect of representing buffers logically rather than using direct virtual access is that data may be added to or removed from any part of the buffer without requiring extra copy or buffer chaining operations. New sections of data may be chained into the existing buffer “behind the scenes” by the environment without disturbing the present buffer contents. Likewise the environment can adjust the buffer’s representation to ignore deleted portions of data without requiring the actual data to be rewritten. The extent to which these practices are performed is determined entirely by the UDI environment implementation; the driver is not concerned with these minutiae.

No endianness conversion is performed on the data in UDI buffers when they are transferred between regions. UDI buffer data is managed by the environment as an untyped string of bytes.

13.2 Buffer Type

UDI buffers are represented by the following semi-opaque type.

NAME **udi_buf_t** *Logical buffer type*

SYNOPSIS

```
#include <udi.h>

typedef struct {
    udi_size_t buf_size;
} udi_buf_t;
```

MEMBERS

buf_size is the current size of the buffer data, in bytes. The environment will adjust this as necessary as a result of service calls that change the buffer's content. The driver may also change the value, to indicate a desired size change, which will affect subsequent service calls.

DESCRIPTION

The `udi_buf_t` structure is used to reference a collection of data that is passing through a driver, typically between an application and a device or communications medium.

The `udi_buf_t` structure is a semi-opaque type, and must only be allocated by environment service calls. UDI buffers are transferable between regions.

If a buffer is used with a service call that retrieves the contents of some or all of the buffer data (such as `udi_buf_read` or `udi_buf_copy`) and the ***buf_size*** value is larger than the extent of data explicitly written into the buffer, the values retrieved for the un-written range are unspecified.

If data is written into the buffer (such as with `udi_buf_write` or `udi_buf_copy`), and the starting offset at which the data is written is greater than the extent of data previously explicitly written into the buffer, an values subsequently retrieved for the un-written range are unspecified.

Any service call that potentially modifies a buffer's contents returns a new buffer pointer in the corresponding callback. While this pointer may in many cases be equal to the original buffer pointer, the environment may in fact have reallocated the buffer, so drivers must always replace all subsequent use of the original buffer pointer with the new pointer. This is also true of buffers passed to channel operations.

13.3 Transfer Constraints

UDI buffer allocation and usage is subject to various constraints specifications. This section describes those constraints that relate to data transfer operations. The UDI Physical I/O Specification specifies additional constraints related to DMA operations.

NAME **udi_xfer_constraints_t** *Transfer constraints structure*

SYNOPSIS

```
#include <udi.h>

typedef struct {
    udi_ubit32_t udi_xfer_max;
    udi_ubit32_t udi_xfer_typical;
    udi_ubit32_t udi_xfer_granularity;
    udi_boolean_t udi_xfer_one_piece;
    udi_boolean_t udi_xfer_exact_size;
    udi_boolean_t udi_xfer_no_reorder;
} udi_xfer_constraints_t;
```

MEMBERS

udi_xfer_max is the maximum # of bytes for an I/O transfer that can be supported by the device and/or driver. Zero indicates that there is no restriction on transfer size.

udi_xfer_typical is the typical # of bytes for an I/O transfer to this device. This value may be used by the environment to optimize pre-allocation decisions. Zero indicates that the device has no typical pattern. Only drivers that do have a typical pattern should set this attribute. This constraint is typically used to assist the environment in implementing pre-allocation strategies.

udi_xfer_granularity is the transfer granularity. The total transfer size must be a multiple of this number of bytes. For random access devices, it is also required that the starting device offset for a transfer must be a multiple of the transfer granularity. A value of one effectively means no restriction. Transfer size is a function of metalanguage-specific operations, and may or may not be related to the size of buffers used to pass the data.

udi_xfer_one_piece is a flag indicating (if TRUE) that the transfer must be handled as a single request; it cannot be broken up. This is typically used for drivers that use the transfer size as an implicit attribute; for example, a tape driver might use the transfer size to control the size of the block written to a tape. Also acts as if ***UDI_XFER_EXACT_SIZE*** were TRUE.

udi_xfer_exact_size is a flag indicating (if TRUE) that transfer requests that don't conform to transfer granularity constraints must be failed instead of being passed to the driver. Even if this flag is not set, the request that is passed to the driver will still meet the transfer granularity constraints, but it may have been modified from the original request in order to do so (using a blocking/de-blocking algorithm).

udi_xfer_no_reorder is a flag indicating (if TRUE) that transfer requests must be passed to the driver in FIFO order. Any fine-grained breakup into smaller requests must also preserve ascending device offset order and must not insert new requests into the stream.

DESCRIPTION

The `udi_xfer_constraints_t` structure is used to describe the various transfer constraints for a specific operation. These transfer constraints may be passed to a child via a metalanguage-specific bind acknowledgement channel operation to communicate the transfer requirements to the child driver; a metalanguage may alternatively explicitly specify an applicable subset of these constraints in a manner unique to that metalanguage.

The transfer constraints information may be used by a driver to determine how to divide requests into appropriate individual control blocks and buffers for handling by the parent driver.

13.4 Buffer Management Macros

The macros specified in this section are standard buffer management macros provided for convenience in using the buffer management service calls. These macros are built on top of the buffer management service calls in Section 13.5 on page 13-12.

NAME **UDI_BUF_ALLOC** *Allocate and initialize a new buffer*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_BUF_ALLOC( \
     callback, gcb, init_data, size, path_handle) \
          udi_buf_write(callback, gcb, init_data, \
                        size, NULL, 0, \
                         0, path_handle)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

init_data is a pointer to the initial data to use to fill the buffer. If set to NULL, the initial data values are unspecified.

size is the initial size of the buffer data, in bytes.

path_handle is the handle identifying the intended use and dispatching of this buffer. Path handle usage is determined by the driver, but by associating the use of a specific path_handle with buffers allocated for a specific purpose, the driver allows the environment to predict and optimize the allocated buffer requirements.

DESCRIPTION

UDI_BUF_ALLOC allocates a new logical buffer with a valid data length of ***size***. The initial data will be copied from ***init_data*** if non-NULL. If ***init_data*** is NULL, the buffer will still have ***size*** bytes of valid data, but the initial value of these bytes is unspecified.

The macro UDI_BUF_ALLOC must be called as if it had the following functional interface, as can be derived from the above macro definition and the definition of udi_buf_write:

```
void UDI_BUF_ALLOC (
     udi_buf_write_call_t *callback,
     udi_cb_t *gcb,
     void *init_data,
     udi_size_t size,
     udi_buf_path_t path_handle );

typedef void udi_buf_write_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_buf );
```

REFERENCES udi_buf_write

NAME **UDI_BUF_INSERT** *Insert bytes into a logical buffer*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_BUF_INSERT( \
     callback, gcb, new_data, size, \
                   dst_buf, dst_off) \
          udi_buf_write(callback, gcb, new_data, \
                        size, dst_buf, dst_off, \
                         0, UDI_NULL_BUF_PATH)
```

ARGUMENTS

callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

new_data is a pointer to the new data bytes to insert into the buffer data. If set to NULL, the ***size*** bytes inserted into ***dst_buf*** at ***dst_off*** shall have unspecified values.

size is the number of bytes to insert into ***dst_buf***.

dst_buf is a handle to the logical buffer into which to insert bytes.

dst_off is the logical offset from the first valid data byte in the buffer to the start of the insertion, in bytes.

DESCRIPTION

UDI_BUF_INSERT inserts ***size*** bytes into ***dst_buf*** at offset ***dst_off***, logically moving any data currently at ***dst_off*** “down” by ***size*** bytes.

The macro UDI_BUF_INSERT must be called as if it had the following functional interface, as can be derived from the above macro definition and the definition of udi_buf_write:

```
void UDI_BUF_INSERT (
     udi_buf_write_call_t *callback,
     udi_cb_t *gcb,
     void *new_data,
     udi_size_t size,
     udi_buf_t *dst_buf,
     udi_size_t dst_off );

typedef void udi_buf_write_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

REFERENCES

udi_buf_write

NAME **UDI_BUF_DELETE** *Delete bytes from a logical buffer*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_BUF_DELETE( \
     callback, gcb, size, dst_buf, dst_off) \
          udi_buf_write(callback, gcb, NULL, \
                        0, dst_buf, dst_off, \
                         size, UDI_NULL_BUF_PATH)
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

size is the number of bytes to delete from ***dst_buf***.

dst_buf is a handle to the logical buffer from which to delete bytes.

dst_off is the logical offset from the first valid data byte in the buffer to the start of the deletion, in bytes.

DESCRIPTION

UDI_BUF_DELETE deletes ***size*** bytes from ***dst_buf*** starting at offset ***dst_off***, logically moving any additional data "up" to fill the gap.

The macro UDI_BUF_DELETE must be called as if it had the following functional interface, as can be derived from the above macro definition and the definition of udi_buf_write:

```
void UDI_BUF_DELETE (
     udi_buf_write_call_t *callback,
     udi_cb_t *gcb,
     udi_size_t size,
     udi_buf_t *dst_buf,
     udi_size_t dst_off );

typedef void udi_buf_write_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

REFERENCES udi_buf_write

NAME **UDI_BUF_DUP** *Copy a logical buffer in its entirety*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_BUF_DUP( \
     callback, gcb, src_buf, path_handle) \
          udi_buf_copy(callback, gcb, src_buf, \
                       0, (src_buf)->buf_size, \
                       NULL, 0, 0, path_handle)
```

ARGUMENTS

callback*, *gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

src_buf is a handle to the logical buffer to copy.

path_handle is the handle identifying the intended use and dispatching of this buffer. Path handle usage is determined by the driver, but by associating the use of a specific path_handle with buffers allocated for a specific purpose, the driver allows the environment to predict and optimize the allocated buffer requirements.

DESCRIPTION

UDI_BUF_DUP makes a logical copy of ***src_buf*** and passes the new buffer to the driver with a callback.

The macro UDI_BUF_DUP must be called as if it had the following functional interface, as can be derived from the above macro definition and the definition of udi_buf_copy:

```
void UDI_BUF_DUP (
     udi_buf_copy_call_t *callback,
     udi_cb_t *gcb,
     udi_buf_t *src_buf,
     udi_buf_path_t path_handle );

typedef void udi_buf_copy_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

REFERENCES

udi_buf_copy

13.5 Buffer Management Service Calls

The functions in this section provide basic UDI buffer management services. These services include the ability to copy one UDI buffer to another, to transfer (read and write) data bytes between driver memory and a UDI buffer, and to free a UDI buffer. A UDI buffer may be allocated by copying or writing without an initial buffer (e.g., see the `UDI_BUF_ALLOC` macro).

13.5.1 Buffer Usage Models

The UDI buffer is used to pass user data from one UDI region to another, typically for the purpose of performing I/O with that buffer. This I/O path may involve several layers of either native OS or UDI modules and some of those modules may wish to implement “retransmit” functionality based on various conditions such as timeouts or failed acknowledgements. Buffer management therefore needs to be implemented in a highly efficient manner. Native OS buffer handling has been optimized over time to avoid copying or relocating data during the high-performance paths in the driver. UDI allows the same types of optimization to be performed as part of the environment implementation although the specification of how the metalanguage manages these buffers is a critical part of this model.

Most I/O designs can be roughly grouped into one of two buffer models:

1) The *command/response* model where there is no asynchronous or unsolicited data from the device, and
2) The *push* model where data is pushed from either end but there’s no direct “acknowledgement” or “completion” of that data transfer.

The most typical example of the command/response model is the SCSI storage protocol. In this protocol, the application supplies the data buffer that either contains data to be written to the device or specifies a buffer region into which data is to be read from the device. The buffer is associated with a command which instructs the adapter and remote device to perform the data transfer, and a response which indicates the success or failure of that transfer. Any retransmissions are usually as a result of a failure indication for the transfer.

The common example of a push model is a network protocol. For most (LAN-based) network protocols, the application supplies a buffer which is manipulated by various protocol entities and then transmitted on a best-case basis. Various amounts of lossage are expected and protocols or applications are typically constructed to expect this lossage and initiate retransmissions if the data is not acknowledged within a specific period of time. Likewise, incoming data may arrive asynchronously and unsolicited from any network partner and may need to be delivered to any one or more applications after appropriate protocol processing.

As a general rule, UDI metalanguages will manage the usage of UDI buffers based one of the above buffer models:

- For a command/response model, the buffer will be passed down to the UDI driver along with the initial command and the (possibly modified) buffer will be passed back with the response. On write failures, metalanguages generally require the driver to pass the buffer back with its contents unmodified; this allows the requester to retransmit the buffer if it so desires.
- For the push module, the buffer is passed down with the request and always deallocated by the UDI driver after being transmitted, regardless of the success or failure of the transmission. If an upper level module wishes to implement a retransmit algorithm based on timers or remote acknowledgements, it must create a copy of the buffer before passing it to

the lower level driver.

It is important at this point to note that the "copy" of the buffer is not necessarily a full copy of the data portion. The UDI environment may simply create another buffer handle that refers to the same data for the copy; this is implementation dependent and is acceptible as long as the environment insures that any buffer modifications through one handle are not visible through another handle (usually by performing a "late-copy" at the time the modification occurs).

Each UDI metalanguage is free to manage buffers in a manner appropriate to that metalanguage (and may even manage different buffers in a different manner for different metalanguage operations) but must specify the methodology to be used in the metalanguage specification and as part of the metalanguage library interface.

13.5.2 Buffer Recovery Mechanism

For most situations in the UDI environment, ownership of a resource such as a buffer is passed to the target region whenever the corresponding handle is passed to that target as part of a metalanguage operation. Any module wishing to preserve the data will typically create a copy of the buffer as described above.

However, in the command/response buffer usage model, the buffer is not copied by the child UDI module before being passed to the parent for processing. Instead, the child module expects the parent to return the buffer when an error occurs. Under normal operating conditions the parent can satisfy this expectation but in the event of an abrupt removal of the parent device (e.g. a hot swap condition) the parent will be unable to return the buffer to the child.

In this situation the child still needs the buffer returned to it in order to perform retransmissions or perhaps perform a failover operations. This is supported in UDI through the operation recovery mechanism described in Section 4.10, "Driver Faults/Recovery". In this situation, the UDI environment will return any buffers held by the parent region to the child as part of the recovery process. Each metalanguage specification shall indicate which operations and their associated buffers are handled in this manner.

NAME **udi_buf_copy** *Copy data from one logical buffer to another*

SYNOPSIS

```
#include <udi.h>

void udi_buf_copy (
     udi_buf_copy_call_t *callback,
     udi_cb_t *gcb,
     udi_buf_t *src_buf,
     udi_size_t src_off,
     udi_size_t src_len,
     udi_buf_t *dst_buf,
     udi_size_t dst_off,
     udi_size_t dst_len,
     udi_buf_path_t path_handle );

typedef void udi_buf_copy_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

ARGUMENTS

callback, gcb are the standard arguments described in *the "Asynchronous Service Calls" section of "Calling Sequence and Naming Conventions"*

src_buf is a pointer to the buffer containing data to be copied. This must not be set to NULL.

src_off is the offset, in bytes, from the first logical data byte to the start of the copy area in the source buffer. This must not exceed the current size of the buffer:

0 ≤ src_off < src_buf->buf_size

src_len is the number of bytes to be copied from the source buffer. ***src_len*** must be at least 1, and ***src_off*** + ***src_len*** must not extend beyond the current buffer size:

0 < src_len ≤ (src_buf->buf_size - src_off)

For ***src_len*** of zero, use `udi_buf_write` instead.

dst_buf is a pointer to the buffer that is the target of the data copy. If set to `NULL`, a new, empty buffer will be allocated before copying the source data.

dst_off is the offset, in bytes, from the first logical data byte to the start of the copy area in the destination buffer. The buffer will be extended if necessary to accommodate the data.

0 ≤ dst_off ≤ dst_buf->buf_size

dst_len is the number of bytes in ***dst_buf*** to be replaced with data copied from the source buffer.

0 ≤ dst_len ≤ (dst_buf->buf_size - dst_off)

If ***dst_buf*** is NULL, both ***dst_off*** and ***dst_len*** must be zero.

path_handle is the handle identifying the intended use and dispatching if a new buffer must be allocated for this request. Path handle usage is determined by the driver, but by associating the use of a specific ***path_handle*** with buffers allocated for a specific purpose, the driver allows the environment to predict and optimize the allocated buffer requirements. If ***dst_buf*** is not NULL on entry, its existing path will continue to be used and this parameter must be set to UDI_NULL_BUF_PATH; otherwise it must be non-null.

new_dst_buf is a pointer to the new, modified destination buffer.

DESCRIPTION

udi_buf_copy logically replaces ***dst_len*** bytes of data starting at offset ***dst_offset*** in ***dst_buf*** with a copy of ***src_len*** bytes of data starting at ***src_offset*** in ***src_buf***. When the data has been copied, the ***callback*** routine is called.

Table 13-1 Common actions for udi_buf_copy/udi_buf_write arguments

Action	src_buf/src_mem	src_len	dst_buf	dst_len
Allocate/initialize	non-null	N	NULL	0
Overwrite	non-null	N	non-null	N
Delete	NULL/NULL	0	non-null	N
Insert	non-null	N	non-null	0
Ensure space	NULL/NULL	N	NULL	0

If ***dst_len*** is zero, the ***src_len*** bytes of source data will be inserted in the destination buffer at ***dst_off***. If ***dst_len*** is positive, the ***dst_len*** bytes will be replaced by ***src_len*** bytes from the source buffer. The ***src_len*** parameter must be > 0 bytes. (For a ***src_len*** of zero, udi_buf_write must be used.)

This routine is very similar to udi_buf_write, except that the data source is another buffer, rather than a virtually-contiguous data structure.

If ***dst_buf*** is NULL, a new buffer will be allocated to hold the data.

The destination buffer will be extended or reallocated as necessary to hold any new data being added to the buffer. This extension or reallocation is performed by the environment as part of the udi_buf_copy operation and all data in the destination buffer not described by the ***dst_off*** and ***dst_len*** region will be preserved. This reallocation may result in a new buffer being returned in the callback, therefore the ***dst_buf*** should no longer be used after passing it to udi_buf_copy and the driver must use the ***new_dst_buf*** value following the callback.

It is expected that this routine will efficiently duplicate buffers (e.g., when multiple higher levels above a multiplex point must receive the same inbound buffer). Because UDI implementations may avoid copying data whenever possible, the actual allocation of space for the copied data may be delayed until the shared data is written via either buffer.

WARNINGS

Control block usage must follow the rules described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

src_buf and ***dst_buf*** must not reference the same buffer.

On successful completion, ***dst_buf*** will no longer be valid and ***new_dst_buf*** is substituted, even if ***dst_buf*** was not specified as `NULL`. ***new_dst_buf*** may be set to the same handle value as the input value of ***dst_buf***, but the driver must not depend on this.

If this operation is cancelled with `udi_cancel`, any pre-existing ***dst_buf*** buffer will be discarded (see `udi_cancel` for an explanation of why this is so).

REFERENCES

`udi_buf_write, udi_cancel`

NAME **udi_buf_write** *Write data bytes into a logical buffer*

SYNOPSIS

```
#include <udi.h>

void udi_buf_write (
     udi_buf_write_call_t *callback,
     udi_cb_t *gcb,
     const void *src_mem,
     udi_size_t src_len,
     udi_buf_t *dst_buf,
     udi_size_t dst_off,
     udi_size_t dst_len,
     udi_buf_path_t path_handle );

typedef void udi_buf_write_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_dst_buf );
```

ARGUMENTS

callback*, *gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

src_mem is a pointer to caller memory where the first byte of data is to be copied from. If NULL, the resulting data values are unspecified.

src_len Number of bytes to be copied from ***src_mem***, replacing the specified ***dst_len*** bytes in ***dst_buf***. If ***src_mem*** is NULL the ***dst_len*** bytes in ***dst_buf*** are replaced by ***src_len*** bytes of unspecified data values. If ***src_len*** is zero, ***src_mem*** is ignored.

dst_buf
dst_off
dst_len
path_handle
new_dst_buf are the same arguments as used in `udi_buf_copy`.

DESCRIPTION

`udi_buf_write` copies data bytes from virtually contiguous driver memory area to a logical buffer. This function works like `udi_buf_copy` except that the data source is a virtually-contiguous memory area, rather than another buffer. No endianness conversion will be performed by `udi_buf_write`.

If ***src_mem*** is NULL, data in the resulting range of the destination buffer will have unspecified values. This is useful for ensuring that a buffer is instantiated to a certain size, without taking the expense of copying data into the buffer. This mechanism should only be used when the instantiated data *must* exist.

WARNINGS

A NULL ***src_mem*** with nonzero ***src_len*** and ***dst_len*** can produce unspecified data values in the middle of valid data (*e.g.*, ***src_mem***=NULL, ***src_len***=6, and ***dst_len***=4 produces at least two bytes of unspecified data within the valid data area of ***dst_buf***). While this is a legal operation, the results may be unexpected.

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

Use of the ***src_mem*** parameter must conform to the rules described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls".

If this operation is cancelled with `udi_cancel`, any pre-existing ***dst_buf*** buffer will be discarded (see `udi_cancel` for an explanation of why this is so).

REFERENCES `udi_buf_copy, udi_cancel`

NAME **udi_buf_read** *Read data bytes from a logical buffer*

SYNOPSIS

```
#include <udi.h>

void udi_buf_read (
     udi_buf_t *src_buf,
     udi_size_t src_off,
     udi_size_t src_len,
     void *dst_mem );
```

ARGUMENTS

src_buf is a pointer to a buffer containing data to be read.

src_off is the offset, in bytes, into the logical data of ***src_buf*** at which to start reading data.

src_off must be ≤ ***src_buf->buf_size***.

src_len The number of bytes to be read from ***src_buf***.

src_off+***src_len*** must not exceed ***src_buf->buf_size***.

dst_mem pointer to caller's memory where data is to be copied.

DESCRIPTION

`udi_buf_read` non-destructively reads data bytes from a logical buffer to a virtually contiguous driver memory area pointed to by ***src_buf***. No endianness conversion will be performed by `udi_buf_read`.

If ***src_buf->buf_size*** was previously extended to include bytes not explicitly written, the resulting values in ***dst_mem*** for these bytes are unspecified.

NAME **udi_buf_free** *Free a logical buffer*

SYNOPSIS

```
#include <udi.h>

void udi_buf_free ( udi_buf_t *buf );
```

ARGUMENTS

buf is a pointer to the buffer to be deallocated. If ***buf*** is NULL on entry, this routine is a no-op.

DESCRIPTION

`udi_buf_free` is called to indicate that a UDI buffer is no longer needed. The buffer and all associated resources will be released and the caller must no longer use the buffer handle, ***buf***.

If ***buf*** is equal to NULL, explicitly or implicitly (zeroed by initial value or by using `udi_memset`), this function acts as a no-op. Otherwise, ***buf*** must have been allocated by `udi_buf_copy` or `udi_buf_write`, or passed to the driver via a channel operation.

REFERENCES

`udi_buf_copy, udi_buf_write`

13.6 Buffer Paths

UDI buffers are used to transport data between various UDI modules for processing by those modules. Ultimately, if this data is destined for a physical device, the buffer will be passed to a physical I/O driver so the data can be presented to or retrieved from the associated hardware device, often via a DMA mechanism. (For more information on physical I/O drivers and DMA, see the UDI Physical I/O Specification.)

The various physical I/O devices and I/O buses in the system may have various DMA constraints and capabilities. To avoid additional processing overhead, it is desireable to ensure that the buffers presented for DMA processing are already conformant to the constraints and capabilities of the associated DMA engine. Within UDI, this is implemented by the environment by association with a buffer path handle.

Each time a buffer is allocated by a UDI module, a buffer path is specified for that allocation request (even if the module itself is not involved in the DMA operation). The path handle that is used for the allocation request should be associated with the expected path through the UDI modules that the buffer is likely to take: ideally, buffers presented to different DMA engines will have been allocated with different path handles.

When a buffer is mapped for DMA, the environment may, if it so chooses, update the associated buffer path object (remembered in the buffer handle from the original allocation) with the constraint and capability information of the DMA engine. By accumulating the most-restrictive combination of capabilities in the path object, the environment can optimize future allocations made with the corresponding path handle to ensure that newly-allocated buffers already conform to the accumulated DMA constraints and capabilities, avoiding subsequent reallocations and copies.

The buffer path mechanism is an optimization provided by UDI for performance improvements in DMA and buffer management. The module allocating a buffer is not required to use different path handles and likewise the UDI environment is not required to update the constraints associated with those path handles; the UDI specification requires the UDI environment to perform the needed buffer adjustments at the time that the buffer is mapped if it does not already conform to the DMA constraints, so any buffer may be passed along any “path” at the cost of the loss of these optimizations.

The UDI module allocating a buffer should choose a path handle based on the information available to it. It is valid to pass buffers allocated with different handles to the same parent (and ultimately the same DMA engine), and it is also valid to pass a buffers allocated with a single path handle to different DMA engines; however, the more closely the module can associate a path handle with a destination DMA engine the better the optimization opportunities for the UDI environment. Examples of buffer path selection heuristics include: the parent channel to which a multiplexing module passes the buffer, the destination IP address for an IP or TCP module allocating a network packet buffer, or the controller number for a SCSI command buffer.

13.6.1 Buffer Path Multiplexing

For a UDI multiplexer module with multiple parents, an additional facility is provided to assist in selecting the parent to which a buffer is passed. If the multiplexer has evaluated the various parents to which a particular buffer could be passed according to the implementation of that multiplexer and has arrived at a list of more than one possible parent, it may be advantageous for the multiplexer to pass the buffer to the parent whose DMA engine (or whose penultimate parent’s DMA engine) is most capable of handling that buffer.

In this situation, the path handle is used in a slightly different manner than for buffer allocation. The multiplexer will typically maintain a path handle for each parent channel, internally maintaining a one-to-one association between a specific path handle and the corresponding parent channel. When the list of possible parents has been determined by the multiplexer by internal means, the `udi_buf_best_path` service may be called with the buffer and the list of path handles corresponding to the list of possible parent channels. The UDI environment will then select one or more of the paths to which the buffer should be passed, presumably based on the constraints associated with the specified paths.

The `udi_buf_best_path` service will return an array of indices into the path handle array, where the returned indices represent the best path or paths to which the buffer may be passed. The environment must return at least one path, but may determine that multiple paths are equivalent (or roughly equivalent) and therefore return an array of more than one indices. The UDI driver must also pass in the index of the most-recently used path; the UDI environment will begin selecting paths at the array position following the previously-matched index (wrapping as necessary) and terminating the search when it has reached the previously-matched index (which may also be included in the returned array of valid indices). If the environment continually finds multiple matches for buffers, the use of the previous index value will cause the first return match to indicate a round-robin algorithm for equitable load balancing scenarios.

As with the path handles used for buffer allocation, the UDI environment may choose how much information to maintain and update for the path handles used with `udi_buf_best_path`, and may, at one extreme, treat all paths as equally good, regardless of actual costs.

NAME **udi_buf_best_path** *Select best path(s) for a data buffer*

SYNOPSIS

```
#include <udi.h>

void udi_buf_best_path (
     udi_buf_t *buf,
     udi_buf_path_t *path_handles,
     udi_ubit8_t npaths,
     udi_ubit8_t last_fit,
     udi_ubit8_t *best_fit_array );

/* Terminator for best_fit_array */
#define UDI_BUF_PATH_END                    255
```

ARGUMENTS

buf is a pointer to a UDI data buffer.

path_handles is an array of candidate buffer path handles which correspond to parent instances to which the buffer might be sent..

npaths is the number of entries to use from the ***path_handles*** array. ***npaths*** must be greater than zero and less than 256.

last_fit is an index into the ***path_handles*** array (starting from zero) indicating the least preferred choice (typically, the one that was selected last time). ***last_fit*** must be less than ***npaths***.

best_fit_array is an array of index values, which is filled in with the indices of one or more ***path_handles*** entries that best fit the data buffer. The list is terminated with an entry containing `UDI_BUF_PATH_END`. ***best_fit_array*** must point to enough space for (***npaths***+1) entries.

DESCRIPTION

`udi_buf_best_path` is used to choose between multiple alternative path handles, each associated with a particular data path over which a request might be sent, and find those that can be expected to result in the best performance, all other aspects of the data path being equal. The environment may consider multiple choices to be equally suitable, and thus the result is returned as a list, in ***best_fit_array***.

Some drivers may wish to factor in other criteria to further narrow down the choice; such drivers would scan the entire returned list. Others may simply take the first entry in ***best_fit_array***, unconditionally. In the latter case, these drivers may want to load-balance among the otherwise-equal alternatives; this is achieved by setting ***last_fit*** to the index that was chosen in the previous call to `udi_buf_best_best_path`.

The index values returned in ***best_fit_array*** are provided in ascending order starting from the first one that is strictly greater than ***last_fit*** modulo ***npaths***, and wrapping around once ***npaths*** is reached.

REFERENCES

`UDI_BUF_ALLOC, udi_channel_event_cb_t`

NAME **udi_buf_path_alloc** *Buffer path handle allocation*

SYNOPSIS

```
#include <udi.h>

void udi_buf_path_alloc (
     udi_buf_path_alloc_call_t *callback,
     udi_cb_t *gcb );

typedef udi_buf_path_alloc_call_t (
     udi_cb_t *gcb,
     udi_buf_path_t new_buf_path );
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the “Asynchronous Service Calls” section of “*Calling Sequence and Naming Conventions*”.

new_buf_path is a newly allocated buffer path handle.

DESCRIPTION

The `udi_buf_path_alloc` service is used to allocate a new buffer path handle to be used for describing a new buffer path. Buffer path usage is defined by the driver performing the allocation operation.

REFERENCES

`udi_buf_copy, udi_buf_path_t`

NAME **udi_buf_path_free** *Buffer path handle deallocation*

SYNOPSIS

```
#include <udi.h>

void udi_buf_path_free ( udi_buf_path_t buf_path );
```

ARGUMENTS

path is a buffer path handle to be deallocated.

DESCRIPTION

The `udi_buf_path_free` call is used to deallocate a buffer path handle when it will no longer be used by the driver.

REFERENCES

`udi_buf_path_alloc`, `udi_buf_copy`, `udi_buf_path_t`

13.7 Buffer Tags

Along with the actual buffer data content, there may be additional information related to a buffer that needs to be maintained along with that buffer and available to any UDI driver that is currently operating on the buffer. This is done by attaching one or more *buffer tags* to a UDI buffer. These buffer tags are used to provide additional descriptions of the data contained in the buffer without placing those descriptions in the data of the buffer itself.

Each buffer tag specifies the tag type, the portion of the buffer to which the tag applies, and the value (if any) associated with that tag. A buffer may have zero or more tags attached to that buffer and the tags may overlap, even for tags of the same type (although two tags that specify the exact same type and identify the same portion of the buffer will be reduced to a single tag whose value is that of the latter tag assignment). The tag will remain associated with the buffer until the buffer is deleted or until the tag is invalidated.

Buffer tags are related to specific data within the buffer and are used to describe that data. Because of this relationship, a tag will always indicate the same section of data in a buffer regardless of insertions or deletions before or after that section of the buffer. If the section of the buffer described by the tag is directly modified, the tag (along with all other tags associated with that buffer section) is invalidated and will be removed. Because of this behavior, tags should not be used to communicate critical information unless the UDI modules can provide assurances that the buffer will not be modified.

There is no limit to the number of tags that may be assigned to a buffer.

When a buffer is copied to another buffer or to a newly created buffer, any tags contained entirely within the copied section are duplicated in the destination buffer automatically.

13.7.1 Buffer Tag Categories

Buffer tags are divided into a number of categories which are used to assist in examining and processing the tags. The specific meaning and appropriate handling of a tag is defined individually for each tag; however, tags can be grouped into categories where the tags in each category perform related functionality. The following tag categories are defined:

- *Value Tags*. These tags are used to store a numeric value associated with the portion of the buffer that the tag applies to. A common example of this is a checksum value.
- *Update Tags*. These tags are used to request an update of the buffer based on a computation or scan of the associated portion of the buffer. The tag value for these tags usually represents a location in the buffer where the result of the computation or scan is to be written. A common example of this category of tag is for calculating a buffer data checksum and writing the result into a buffer header.
- *Status Tags*. These tags are used to indicate the status of the associated portion of the buffer. These tags are useful when the hardware is able to supply additional status information about buffer data that may need to be communicated to other modules. Status tags should not be used to store critical status due to the transitory nature of tags.
- *Driver-internal Tags*. These tags are defined and processed by UDI drivers and are ignored by the UDI environment. This category of tags may be used by the driver to store temporary information or inter-region information. This category of tags is driver-specific and driver-internal tags set by one driver will not be visible to any other driver that the buffer is passed to.

NAME **udi_tagtype_t** *Buffer tag type*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit32_t udi_tagtype_t;

/* Tag Category Masks */
#define UDI_BUFTAG_ALL                   0xffffffff
#define UDI_BUFTAG_VALUES                0x000000ff
#define UDI_BUFTAG_UPDATES               0x0000ff00
#define UDI_BUFTAG_STATUS                0x00ff0000
#define UDI_BUFTAG_DRIVERS               0xff000000

/* Value Category Tag Types */
#define UDI_BUFTAG_BE16_CHECKSUM         (1U<<0)

/* Update Category Tag Types */
#define UDI_BUFTAG_SET_iBE16_CHECKSUM    (1U<<8)
#define UDI_BUFTAG_SET_TCP_CHECKSUM      (1U<<9)
#define UDI_BUFTAG_SET_UDP_CHECKSUM      (1U<<10)

/* Status Category Tag Types */
#define UDI_BUFTAG_TCP_CKSUM_GOOD        (1U<<17)
#define UDI_BUFTAG_UDP_CKSUM_GOOD        (1U<<18)
#define UDI_BUFTAG_IP_CKSUM_GOOD         (1U<<19)
#define UDI_BUFTAG_TCP_CKSUM_BAD         (1U<<21)
#define UDI_BUFTAG_UDP_CKSUM_BAD         (1U<<22)
#define UDI_BUFTAG_IP_CKSUM_BAD          (1U<<23)

/* Drivers Category Tag Types */
#define UDI_BUFTAG_DRIVER1               (1U<<24)
#define UDI_BUFTAG_DRIVER2               (1U<<25)
#define UDI_BUFTAG_DRIVER3               (1U<<26)
#define UDI_BUFTAG_DRIVER4               (1U<<27)
#define UDI_BUFTAG_DRIVER5               (1U<<28)
#define UDI_BUFTAG_DRIVER6               (1U<<29)
#define UDI_BUFTAG_DRIVER7               (1U<<30)
#define UDI_BUFTAG_DRIVER8               (1U<<31)
```

DESCRIPTION The `udi_tagtype_t` type definition specifies the tag type used to specify a bitmask of one or more tags. These tags are subdivided into categories according to the general meaning of the tag. Each category can be easily identified or selected by using the appropriate category mask defined above.

The value tags defined in the Values category are typically used to store a numeric value associated with the portion of the buffer that the tag applies to. Since buffer data is stored in raw form any value tag must indicate the endianness interpretation of the buffer data as part of the tag type where appropriate. The value associated with the tag itself is passed to/from environment service calls in the driver's endianness regardless of the endianness of the buffer data.

UDI_BUFTAG_BE16_CHECKSUM - This tag's value is a 16-bit checksum that has been computed for the tagged range of the buffer. The tag value is in the driver's endianness but the checksum is computed as if the buffer contents are in big-endian 16-bit format.

The checksum is calculated by treating the specified portion of the buffer as an array of `udi_ubit16_t` elements and computing the sum of all elements modulo 2^{16}. If the length of the buffer portion is odd the "missing" low order byte of the last array element is treated as zero.

The update tags defined in the Updates category are used to request an update of the buffer based on a computation or scan of the associated portion of the buffer. The tag value for these tags usually represents a location in the buffer where the result of the computation or scan is to be written.

UDI_BUFTAG_SET_TCP_CHECKSUM - This tag is used to indicate that the associated portion of the buffer is a TCP/IP packet for which the TCP checksum is to be set before transmission, The associated buffer section includes the data and both the TCP and IP headers. The tag's value is ignored.

The TCP checksum is computed by taking the unsigned sum of 16-bit elements modulo 2^{16}, then applying a ones-complement; the following elements are included in this checksum: the TCP header and data areas, the IP source and destination addresses, the IP specified length, and the IP protocol byte (0 extended). The TCP checksum is written as a 16-bit big-endian value at bytes 16 and 17 of the TCP header.

More information regarding the TCP checksum algorithm may be obtained by consulting the following IETF RFCs:RFC 1071 "*Computing the Internet checksum*"; RFC 1141 *"Incremental updating of the Internet checksum"*; RFC 1624 *"Computation of the Internet Checksum via Incremental Update"*; and RFC 1936 *"Implementing the Internet Checksum in Hardware"*.

UDI_BUFTAG_SET_UDP_CHECKSUM - This tag is used to indicate that the associated portion of the buffer is a UDP/IP packet for which the UDP checksum is to be set before transmission. The associated buffer section includes the data and both the UDP and IP headers. The tag's value is ignored.

The UDP checksum is the ones-complement of a 16-bit big-endian checksum of: the UDP header and data areas, the IP source and destination addresses, an additional copy of the UDP specified length, and the IP protocol byte (0 extended). The UDP checksum is written as a 16-bit big-endian value at bytes 6 and 7 of the UDP header.

More information regarding the UDP checksum algorithm may be obtained by consulting the IETF RFCs described above for the `UDI_BUFTAG_SET_TCP_CHECKSUM` buffer tag.

`UDI_BUFTAG_SET_iBE16_CHECKSUM` - This tag is used to indicate that a 16-bit big-endian ones-complement checksum is to be generated for the tagged portion of the buffer and that the result must be written into the buffer at the offset specified by the tag's value field before transmitting the buffer.

This buffer tag is commonly used to request that the IP header checksum is to be set before transmitting the buffer.

The status tags defined in the Status category are used to indicated the status of the associated portion of the buffer.

`UDI_BUFTAG_TCP_CKSUM_GOOD` - This tag is used to indicate that the associated portion of the buffer contains a TCP header and data portion and that the checksum contained in the header has been validated as correct for that buffer. This tag is typically set by a Network Adapter whose hardware validates TCP checksums for received packets. The checksum value itself, if known, may be specified as the ***`tag_value`*** for this tag; the header may no longer contain the checksum and this value in the packet header should not be reference.

`UDI_BUFTAG_UDP_CKSUM_GOOD` - This tag is used to indicate that the associated portion of the buffer contains a UDP header and data portion and that the checksum contained in the header has been validated as correct for that buffer. This tag is typically set by a Network Adapter whose hardware validates UDP checksums for received packets. The checksum value itself, if known, may be specified as the ***`tag_value`*** for this tag; the header may no longer contain the checksum and this value in the packet header should not be reference.

`UDI_BUFTAG_IP_CKSUM_GOOD` - This tag is used to indicate that the associated portion of the buffer contains an IP header (including options) and that the checksum contained in the header has been validated as correct for that buffer. This tag is typically set by a Network Adapter whose hardware validates IP checksums for received packets. The checksum value itself, if known, may be specified as the ***`tag_value`*** for this tag; the header may no longer contain the checksum and this value in the packet header should not be reference.

`UDI_BUFTAG_TCP_CKSUM_BAD` - This tag is used to indicate that the associated portion of the buffer contains a TCP header and data portion and that the checksum contained in the header does ***not*** match the calculated checksum (as typically determined by the driver or the hardware).

UDI_BUFTAG_UDP_CKSUM_BAD - This tag is used to indicate that the associated portion of the buffer contains a UDP header and data portion and that the checksum contained in the header does ***not*** match the calculated checksum (as typically determined by the driver or the hardware).

UDI_BUFTAG_IP_CKSUM_BAD - This tag is used to indicate that the associated portion of the buffer contains an IP header (including options) and that the checksum contained in the header does ***not*** match the calculated checksum.

The driver tags defined in the Drivers category are available for use by the driver for temporary or driver-internal use. This is especially useful when passing buffers in a multi-region driver. These tags are not visible to any other drivers; this protects against inter-driver confusion or tag assumptions but also means that these tags are not suitable for passing buffer information to other drivers in the UDI environment. Driver tags attached to a buffer which is passed to other drivers and subsequently returned will still have the current driver's tags attached and visible unless the associated region of the buffer was modified before being returned to the current driver; driver-specific tags set by other drivers will have no effect on the driver-specific tags set by the current driver.

REFERENCES udi_buf_tag_t

NAME **udi_buf_tag_t** *Buffer tag structure*

SYNOPSIS

```
#include <udi.h>

typedef struct {
    udi_tagtype_t tag_type;
    udi_ubit32_t tag_value;
    udi_size_t tag_off;
    udi_size_t tag_len;
} udi_buf_tag_t;
```

MEMBERS

tag_type is the type of tag represented by this tag structure. Although `udi_tagtype_t` is a bitmask type only one tag type may be specified in the `udi_buf_tag_t` structure (i.e. only one bit may be set).

tag_value is the value associated with this tag.

tag_off is the starting buffer data offset for which the tag applies.

tag_len is the length of data (in bytes) for which the tag applies. The ***tag_len*** value must not be zero.

DESCRIPTION

The `udi_buf_tag_t` structure is used to describe a buffer tag. The range of data to which the tag applies is specified by the ***tag_off*** and ***tag_len*** fields; the ***tag_type*** specifies which type of tag is being described. The ***tag_value*** is the associated value for this tag (if any) as defined by the ***tag_type***.

Table 13-2 Tag structure field usage

tag_type **UDI_BUFTAG_xxx**	**tag_value**	**tag_off**	**tag_len**
BE16_CHECKSUM	16-bit checksum	start of region checksummed as big-endian 16-bit values	number of bytes to checksum (if odd, an extra byte value of 0 is assumed: 16-bit array length = (***tag_len***+1)/2)
SET_iBE16_CHECKSUM	buffer offset at which to write the 16-bit big-endian checksum of the buffer data	start of region to generate a 16-bit big-endian checksum over	number of bytes to checksum

Table 13-2 Tag structure field usage

tag_type UDI_BUFTAG_xxx	**tag_value**	**tag_off**	**tag_len**
SET_TCP_CHECKSUM	unused	start of IP header (including options) followed by TCP header and TCP data	total byte length of IP header (including options), TCP header, and TCP data
SET_UDP_CHECKSUM	unused	start of IP header (including options) followed by UDP header and UDP data	total byte length of IP header (including options), UDP header, and UDP data
TCP_CKSUM_GOOD	checksum value (if known, otherwise zero)	start of TCP header	total byte length of TCP header and TCP data
UDP_CKSUM_GOOD	checksum value (if known, otherwise zero)	start of UDP header	total byte length of UDP header and UDP data
IP_CKSUM_GOOD	checksum value (if known, otherwise zero)	start of IP header	total byte length of IP header including options
TCP_CKSUM_BAD	unused	start of TCP header	total byte length of TCP header and TCP data
UDP_CKSUM_BAD	unused	start of UDP header	total byte length of UDP header and UDP data
IP_CKSUM_BAD	unused	start of IP header	total byte length of IP header including options
DRIVER1...DRIVER8	driver-defined	driver-defined	driver-defined

REFERENCES `udi_buf_tag_set`, `udi_buf_tag_get`

NAME

udi_buf_tag_set *Sets a tag for a portion of buffer data*

SYNOPSIS

```
#include <udi.h>

void udi_buf_tag_set (
     udi_buf_tag_set_call_t *callback,
     udi_cb_t *gcb,
     udi_buf_t *buf,
     udi_buf_tag *tag_array,
     udi_ubit16_t tag_array_length );

typedef void udi_buf_tag_set_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_buf );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

buf is the buffer for which the new tag is to be set.

tag_array is a pointer to an array of udi_buf_tag_t structures that are to be set in the current buffer.

tag_array_length is the number of entries in the ***tag_array***.

new_buf is a pointer to the buffer with the new tag value set.

DESCRIPTION

The udi_buf_tag_set operation is used to set one or more tags for the associated buffer. The tags to be set are specified in the ***tag_array*** and each tag will be set individually. If a tag in the input array is not a driver-specific tag and matches an existing buffer tag of the same type, offset, and length, the ***tag_value*** from the input array replaces the current tag value and the tag is otherwise unchanged. If no exactly matching type, offset, and length tag already exists for the buffer, a new tag will be created from the information in the array element.

The range specified by the tag offset and length must consist entirely of valid data.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

On successful completion, ***buf*** will no longer be valid and ***new_buf*** must be used instead.

REFERENCES

udi_buf_tag_t, udi_buf_tag_get

NAME **udi_buf_tag_get** *Gets one or more tags from a buffer*

SYNOPSIS

```
#include <udi.h>

udi_ubit16_t udi_buf_tag_get (
     udi_buf_t *buf,
     udi_tagtype_t tag_type,
     udi_buf_tag_t *tag_array,
     udi_ubit16_t tag_array_length,
     udi_ubit16_t tag_start_idx );
```

ARGUMENTS

buf is the buffer for which the tag information is to be returned

tag_type is a bitmask of tag types; only tags which correspond to bits set in this bitmask will be returned. For convenience, the tag category mask values may be used for this argument.

tag_array is a pointer to an array of `udi_buf_tag_t` structures that are to be filled in with the obtained tag information.

tag_array_length is the number of entries that may be written to ***tag_array***.

tag_start_idx is the number of tags of the specified type to skip before returning tag information.

DESCRIPTION

The `udi_buf_tag_get` operation is used to obtain information about tags which are attached to the buffer. Any available tags matching of of the requested ***tag_type*** bit values will be written into the ***tag_array*** (after skipping the first ***tag_start_idx*** tags) until either all tags of the target types or ***tag_array_length*** number of tags have been written.

RETURN VALUES

This function returns the actual number of tags of the selected types, regardless of the input ***tag_start_idx***. The ***tag_start_idx*** may be used to iterate through all tags if ***tag_array_length*** is less than the number of defined tags.

REFERENCES

`udi_buf_tag_t, udi_buf_tag_set`

13.7.2 Buffer Tag Utilities

This section defines a set of utility routines that may be used to efficiently make use of buffer tags. The functionality provided by these utility routines could alternatively be implemented by discrete operations using `udi_buf_tag_get` and `udi_buf_tag_set` and other buffer management service calls. These utility routines are provided to assist in implementing and supporting the most common set of buffer tag operations, such as calculating network data checksums.

NAME **udi_buf_tag_compute** *Compute values from tagged buffer data*

SYNOPSIS

```
#include <udi.h>

udi_ubit32_t udi_buf_tag_compute (
     udi_buf_t *buf,
     udi_size_t off,
     udi_size_t len,
     udi_tagtype_t tag_type );
```

ARGUMENTS

buf is the buffer for which the tag value is to be computed.

off is the offset into the buffer at which the computation is to begin. The offset specified must point to valid buffer data.

len is the number of bytes in the buffer to be used for the computation. All bytes in the buffer specified by ***off*** and ***len*** must be valid buffer data.

tag_type is the tag value to be computed. Only one tag type may be specified (only one bit may be set for this argument) and it must be one of the Value category tags (i.e. one of the tag types in the `UDI_BUFTAG_VALUES` category).

DESCRIPTION

The `udi_buf_tag_compute` utility routine is used to calculate the specified tag value for a portion of data contained in the buffer; the most common tag value computed is the 16-bit big-endian checksum value used for network packets.

The buffer range specified must consist entirely of valid data bytes.

The ***tag_type*** argument specifies what type of tag value is to be calculated. It is assumed (but not required) that this utility will take advantage of existing tags attached to the buffer to optimize the computation of the tag values.

This utility function does not actually set a tag of the corresponding ***tag_type*** on the buffer itself; that activity is left to the caller if needed.

Note – This function could be implemented entirely as a series of calls to various UDI service calls such as `udi_buf_read` and `udi_buf_tag_get`, but is expected in most environments to be implemented more directly in terms of the underlying implementation-specific data structures for greater efficiency.

RETURN VALUE

The computed tag value.

REFERENCES

`udi_buf_tag_apply`, `udi_buf_tag_get`

NAME

udi_buf_tag_apply *Apply modifications to tagged buffer data*

SYNOPSIS

```
#include <udi.h>

void udi_buf_tag_apply (
     udi_buf_tag_apply_call_t *callback,
     udi_cb_t *gcb,
     udi_buf_t *buf,
     udi_tagtype_t tag_type );

typedef void udi_buf_tag_apply_call_t (
     udi_cb_t *gcb,
     udi_buf_t *new_buf );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

buf is the buffer for which tag values are to be computed and set.

tag_type is a bitmask of tag types for which the tag values are to be set. Only bit values corresponding to the `UDI_BUFTAG_UPDATES` mask may be used; for convenience the mask value itself may be specified.

new_buf is the buffer returned to the caller after tags have been computed and have been written into the buffer.

DESCRIPTION

The `udi_buf_tag_apply` utility routine is used to process any Update category tags in the buffer. These buffer tags specify various tag values that are to be generated and inserted into the buffer as part of the handling of that buffer (e.g. for TCP/IP network checksum generation before transmitting the buffer).

This utility will process all tags attached to the buffer which correspond to bits set in the specified ***tag_type***. For each tag it will compute the tag value for the indicated section of the buffer (as if by a call to `udi_buf_tag_compute`) and then write the result into the buffer according to the description of that ***tag_type***. The requested update tags will not be processed in any particular order; if a specific order of computation is desired multiple calls to `udi_buf_tag_apply` should be made with the required sequence of tag types.

This utility function is typically used by Network Interface Card (NIC) Drivers which do not provide a checksum off-load capability and need to insert various TCP or other protocol-specific checksums into the packet before it is transmitted.

Note – This function could be implemented entirely as a series of calls to various UDI service calls such as `udi_buf_write` and `udi_buf_tag_get`, but is expected in most environments to be implemented more directly in terms of the underlying implementation-specific data structures for greater efficiency.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

On successful completion, ***buf*** will no longer be valid and ***new_buf*** is substituted, even if ***buf*** was not specified as `NULL`; ***new_buf*** may return the same handle value as the input value of ***buf***.

If this operation is cancelled with `udi_cancel`, any pre-existing ***buf*** buffer will be discarded (see `udi_cancel` for an explanation of why this is so).

REFERENCES

`udi_buf_tag_t, udi_buf_tag_get, udi_buf_tag_compute`

Time Management 14

UDI supports two types of time-related services: Timer Services, which allow driver callback routines to be called at specific times; and Timestamp Services, which allow drivers to measure elapsed time. These are described in more detail in separate sections below.

14.1 Timer Services

14.1.1 Timed Delays

UDI timer services provide a set of operations that can be used to schedule future events for handling. The UDI timer services are very similar to legacy timer services found in most operating systems and provide a mechanism to schedule the call of a driver's timeout routine at some point in the future (relative to the current time). UDI timers may be of either the one-shot variety or may be invoked as repeating timers where the timeout routine will be called repeatedly until cancelled.

UDI timer services shall be implemented with the expectation that the normal operation of most timers is to start the timer to accompany a request and then cancel the timer when the request is successfully handled by the device. Timer startup and cancellation shall therefore be implemented by the environment with minimal overhead to allow their use in the datapath in this manner.

14.1.2 Timer Context

The UDI timer services are performed using a control block structure (e.g., `udi_cb_t`) to provide a context to the timer operations. The control block provides context information about the original request that can be used in the timeout routine. However, there are cases where the timer is not directly related to any current request and a specific control block is needed to manage the timeout operation. One example of this is a *watchdog timer* routine where the timeout routine is called periodically to check the general health of the device independent of any current requests. To handle these general timeout situations, a control block will be needed. Any available control block may be used so long as it is not needed for any other purpose; in practice, however, this usually means that a new control block will have to be allocated with `udi_cb_alloc`. In this case, a control block index associated with `udi_gcb_init` can be used to allocate a generic control block.

Control blocks are a finite system resource. It is a responsibility of a device driver to allocate, track and return control blocks to the UDI environment in a responsible manner.

NAME **udi_time_t** *Time value structure*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_ubit32_t  seconds;
     udi_ubit32_t  nanoseconds;
} udi_time_t;
```

MEMBERS

seconds is the number of seconds of time.

nanoseconds is the number of additional nanoseconds of time. Thus ***nanoseconds*** ranges from zero to one less than one thousand million (10^9) nanoseconds.

DESCRIPTION

The `udi_time_t` structure is used to specify a timeout interval for use with the UDI Timer Services or an elapsed time interval returned by UDI Timestamp Services. The fields in this structure allow very precise specification of time values relative to the current time; the `udi_limits_t` values should be consulted to determine the actual granularity of the environment’s timers, as all specified `udi_time_t` values will be rounded up to integral multiples of the minimum system timer resolution.

This structure is not used to represent absolute (“wall-clock”) times. UDI provides no facility to determine absolute time.

REFERENCES

`udi_limits_t`

NAME **udi_timer_start** *Start a callback timer*

SYNOPSIS

```
#include <udi.h>

void udi_timer_start (
     udi_timer_expired_call_t *callback,
     udi_cb_t *gcb,
     udi_time_t interval );

typedef void udi_timer_expired_call_t (
     udi_cb_t *gcb );
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

interval is the desired minimum interval that should elapse between the time the event is initiated with `udi_timer_start` and the time ***callback*** is called. The actual interval will depend on system activity, platform implementation (*e.g.* clock interrupt interval), timer resolution (***min_timer_res***), and the availability of processor resources. Under normal system activity the actual interval will be at least as long as the specified interval and not usually more than ***interval*** plus ***min_timer_res***.

DESCRIPTION

`udi_timer_start` schedules a delayed callback according to the parameters specified. The ***callback*** routine will be called at some time in the future, as specified by ***interval***.

As with other control block operations, the ownership of the control block passes from the driver to the environment until such time as the callback is invoked and the control block is passed back. Re-using the specified control block for this or any other request before it has been returned to the driver via the ***callback*** routine is illegal. This may require the driver to obtain another control block by calling `udi_cb_alloc` in order to be able to dedicate it to this purpose.

A `udi_timer_start` request may be cancelled at any time by calling the `udi_timer_cancel` routine with the original control block pointer.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

REFERENCES

`udi_time_t, udi_limits_t, udi_cb_alloc, udi_timer_cancel`

NAME **udi_timer_start_repeating** *Start a repeating timer*

SYNOPSIS

```
#include &lt;udi.h&gt;

void udi_timer_start_repeating (
     udi_timer_tick_call_t *callback,
     udi_cb_t *gcb,
     udi_time_t interval );

typedef void udi_timer_tick_call_t (
     void *context,
     udi_ubit32_t nmissed );
```

ARGUMENTS

callback*, *gcb are standard arguments described in **the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.**

interval is the repeating period for this timer (see udi_timer_start). For udi_timer_start_repeating, ***interval*** must be greater than zero.

context is the context pointer from the original control block, ***gcb***.

nmissed is the number of timeout callbacks missed.

DESCRIPTION

udi_timer_start_repeating behaves like udi_timer_start except that the ***callback*** routine is called repeatedly at each successive occurrence of ***interval***. Repeated callbacks are timed relative to the original starting time, rather than the last callback time.

Each time the specified ***interval*** timeout period has elapsed (within system timer resolution capability) the ***callback*** function is called. If the ***callback*** routine is currently scheduled or active or the environment otherwise is unable to call the ***callback*** on schedule, the environment will increment an internal counter representing the number of missed timeout calls for a particular timeout control block. This missed timeout count is passed to the ***callback*** function as the ***nmissed*** argument; this indicator allows the driver to determine if it has missed callbacks and take appropriate action. The ***nmissed*** value will only reflect missed callbacks since the last delivered callback. After a missed callback, the next callback may be delivered any time between the scheduled time of the missed callback and the normally scheduled time for the next callback.

The repeating timer can be stopped by calling udi_timer_cancel from either the ***callback*** timeout routine or from other code within the region that started the timer with the original control block.

Unlike other callback functions, udi_timer_tick_call_t does not pass the control block back to the driver, since it remains busy until the repeating timer is cancelled with udi_timer_cancel.

REFERENCES

udi_time_t, udi_limits_t, udi_cb_alloc, udi_timer_cancel

NAME

udi_timer_cancel *Cancel a pending timer*

SYNOPSIS

```
#include <udi.h>

void udi_timer_cancel (
     udi_cb_t *gcb );
```

ARGUMENTS

gcb is a pointer to a control block that was passed to a prior `udi_timer_start` or `udi_timer_start_repeating` service call.

DESCRIPTION

Any timer service request with a pending callback can be canceled by this call. The control block must be the same one specified when the service was requested, and must be active (i.e. the callback has not yet been called).

Once `udi_timer_cancel` has returned, the original ***callback*** routine is guaranteed not to be called. Ownership of the control block is transferred back to the requestor, and the control block is available for reuse.

WARNINGS

`udi_timer_cancel` must be called from the region that owned the control block at the time of the original request. It cannot be used to cancel a pending request in another region.

A driver must keep track of its in-progress requests to avoid canceling a different request than intended. A good rule of thumb is that `udi_timer_cancel` must not be used to cancel a request without first checking to see if the corresponding callback has been called.

If a driver issues a `udi_timer_cancel` for a control block that is not active the driver is in error. See the “Driver Faults/Recovery” section of *“Execution Model”* for an explanation of how the environment may react to this driver error.

REFERENCES

`udi_timer_start, udi_timer_start_repeating, udi_cancel`

14.2 Timestamp Services

Timestamp services allow drivers to measure elapsed time. This is accomplished by taking snapshots, or *timestamps*, of the current time, using `udi_time_current()` and comparing multiple timestamps with `udi_time_between()` or `udi_time_since()`. Timestamps are represented using the self-contained opaque type, `udi_timestamp_t`, defined in Section 9.6.2.1, “Timestamp Type,” on page 9-13.

NAME **udi_time_current** *Return indication of the current relative time*

SYNOPSIS

```
#include &lt;udi.h&gt;

udi_timestamp_t udi_time_current ( void );
```

DESCRIPTION `udi_time_current` returns the current time (relative to some arbitrary starting point), in implementation-specific units. The system time resolution can be determined from the ***min_curtime_res*** field in the `udi_limits_t` structure.

No UDI services are provided to directly convert a `udi_timestamp_t` value to standard units, such as in a `udi_time_t`. Instead, timestamp values can be compared using `udi_time_since` or `udi_time_between`. `udi_timestamp_t` is a self-contained opaque type, and is therefore not transferable between regions.

In many environments, timestamp values are only useful for accurate comparisons for a limited amount of time. That is, when compared to another timestamp value they may appear to be more recent than the actual time at which they were obtained, since underlying time counters may wrap around. In all environments, `udi_timestamp_t` values are guaranteed to be useful for at least 24 hours.

RETURN VALUES The current time stamp is returned to the caller.

WARNINGS There are no guaranteed "invalid" values for `udi_timestamp_t`. In order to represent an invalid or uninitialized timestamp value, an external flag must be used.

Drivers must not assume that repeated calls to `udi_time_current` without returning from the driver will ever return different values; environments may choose to update the underlying time value only between calls into the driver. Delays must be implemented with timer services not timestamp services.

REFERENCES `udi_time_t, udi_limits_t, udi_time_since, udi_time_between`

See also Section 9.6.2.1, "Timestamp Type," on page 9-13.

NAME **udi_time_between** *Return time interval between two points*

SYNOPSIS

```
#include <udi.h>

udi_time_t udi_time_between (
     udi_timestamp_t start_time,
     udi_timestamp_t end_time );
```

ARGUMENTS

start_time is a timestamp value marking the starting point of the interval.

end_time is a timestamp value marking the ending point of the interval.

DESCRIPTION

`udi_time_between` returns the time delta between two previously recorded times, in ***start_time*** and ***end_time***. The previously recorded times must have been obtained via `udi_time_current`.

start_time must reflect a time that occurred no later than ***end_time***.

The system time resolution can be determined from the ***min_curtime_res*** field in the `udi_limits_t` structure.

RETURN VALUES

The time interval, in seconds and nanoseconds, is returned to the caller.

REFERENCES

`udi_time_t, udi_limits_t, udi_time_current, udi_time_since`

NAME **udi_time_since** *Return time interval since a starting point*

SYNOPSIS

```
#include <udi.h>

udi_time_t udi_time_since (
    udi_timestamp_t start_time );
```

ARGUMENTS ***start_time*** is a timestamp value marking the starting point of the interval.

DESCRIPTION udi_time_since returns the time delta between a previously recorded time, in ***start_time***, and the current time. The previously recorded time must have been obtained via udi_time_current.

The system time resolution can be determined from the ***min_curtime_res*** field in the udi_limits_t structure.

udi_time_since is equivalent to:

```
    udi_time_between(start_time, udi_time_current())
```

RETURN VALUES The time interval, in seconds and nanoseconds, is returned to the caller.

WARNINGS Drivers must not assume that repeated calls to udi_time_since without returning from the driver will ever return different values; environments may choose to update the underlying time value only between calls into the driver. Delays must be implemented with timer services not timestamp services.

REFERENCES udi_time_t, udi_limits_t, udi_time_current

Instance Attribute Management 15

15.1 Overview

UDI provides the capability of associating *attributes* (information) with driver instances. These are called *driver instance attributes*. These attributes may be stored in a system-wide persistent storage database to allow the driver to maintain configuration and topology information across driver and system restarts. A driver restart is defined within the context of a particular driver instance, and is the period between when the driver detaches and then later reattaches to that instance. This may occur when the host system restarts or may be during a period when the host reclaims the resources of a driver instance that is not actively being used. A system restart is defined to be the period between when the host terminates and then later resumes operation. This most commonly is a system reboot, which may include system power cycling.

This section defines the interfaces used to read and modify the various driver attributes.

15.2 Instance Attribute Names

Instance attribute names may be composed of up to 31 ASCII characters plus a null terminator. Legal characters for attribute names consist of upper and lower case letters, digits, and the underscore character (‘_’). In addition, the first character may be a percent-sign (‘%’), a dollar-sign (‘$’), a caret (‘^’), or an at-sign (‘@’); these prefix characters have special meanings, described below. All other characters are illegal.

Upper and lower case ASCII letters are treated identically when looking up existing attribute names (i.e. the matching is case-insensitive). It is environment implementation-specific whether or not alphabetic case is preserved in attribute names when creating or changing attributes. By convention, specific attribute names defined in UDI specifications are written in all lower case.

Each distinct name, even if it differs from another attribute name only by a prefix character, identifies a distinct attribute.

15.3 Persistence of Attributes

Attributes may be specified to be either *persistent* or *volatile* (non-persistent). Persistent attributes will be maintained in a persistent storage database and will be available across system restarts, whereas volatile attributes are only guaranteed to persist for the duration of the corresponding driver instance.

Certain environments will not be able to supply a modifiable persistent storage database (e.g. an embedded ROM-based environment). For these types of environments, any attempt to modify a persistent attribute value will result in a `UDI_STAT_NOT_SUPPORTED` error code. The driver may choose to ignore or otherwise handle this return value as determined by the driver implementation requirements.

Accesses to the persistent storage database will be implemented in an atomic manner. This means that any of the attribute management service calls documented in this section may be issued without concern about collision with other operations, although there is no guarantee as to the sequence of individual operations relative to operations issued by other driver regions.

15.4 Classes of Attributes

There are four principle classes of driver instance attributes:

1. Instance-private attributes
2. Enumeration attributes
3. Sibling group attributes
4. Parent-visible attributes

15.4.1 Instance-Private Attributes

These attributes are persistent or volatile attributes that are read and written via the service operations defined in this section. They are visible only to the driver instance to which they apply. These attributes may be used for any driver-related information.

Private persistent attribute names must begin with a percent-sign (‘%’) prefix character. Private volatile attribute names must begin with a dollar-sign (‘$’) prefix character.

15.4.2 Enumeration Attributes

Enumeration attributes are those attributes used in the enumeration operation to uniquely identify a child instance and its initial parameters. These attributes are typically specified in the Metalanguage Specification, and are provided by the driver’s parent during enumeration. The enumeration attributes are set on the child instance before that instance is enabled; the enumeration attributes are set atomically (i.e. none of the attributes can be read or changed until all of the enumeration attributes for that child instance have been set).

Enumeration attributes may be read but not modified by the driver instance with which they are associated.

Enumeration attributes are not visible to the parent once enumerated.

Enumeration attribute names must begin without a special prefix character.

15.4.2.1 Generic Enumeration Attributes

There are four generically-accessible enumeration attributes: “`identifier`”, “`address_locator`”, “`physical_locator`”, and “`physical_label`”. These attributes, of type UDI_ATTR_STRING, are defined so as to allow environments to use these attributes in generic algorithms to identify and compare information about the devices in the system. This is useful in keeping the UDI environment isolated from the specifics of metalanguages and bus bindings.

15.4.2.1.1 identifier attribute

The contents of the "`identifier`" attribute must be defined in all metalanguages and bus bindings, and an appropriate value for this attribute must be provided on any child enumeration. This attribute is defined on a per-metalanguage/bus basis to provide information that can be used to uniquely identify a device as much as possible for the given I/O technology. In most cases, this will simply identify a type of device and multiple devices of the same type will have the same value, but where available, a serial number could be used to make the string truly unique.

15.4.2.1.2 address_locator attribute

The contents of the "`address_locator`" attribute must be defined in all metalanguages and bus bindings, and an appropriate value for this attribute must be provided on any child enumeration. This attribute is defined on a per-metalanguage/bus basis to provide information which can be used to address the device, relative to the enumerating parent.

15.4.2.1.3 physical_locator attribute

The "`physical_locator`" attribute is an optional attribute which may be non-existent for some metalanguages or bus bindings. Metalanguages or buses whose children are physical devices should specify this attribute whenever possible. When defined, this attribute is used to provide information about the physical location of a device, such as a slot number.

15.4.2.1.4 physical_label attribute

The "`physical_label`" attribute is an optional attribute which may be non-existent for some metalanguages or bus bindings. Metalanguages or buses whose children are physical devices should specify this attribute whenever possible. When defined, this attribute is used to provide information about the physical location of a device in terms of user-visible labeling, when known by the enumerating parent.

15.4.2.1.5 Generic Enumeration Attribute Example

As an example of the usage and combination of these attributes, the following environment is hypothesized:

1. Child enumerated by Bus XYZ:
   - identifier="<productid,vendorid>"
   - address_locator="<dev_num,func_num>"
   - physical_locator="<slot#>"
   - physical_label="<chassis location>"
2. Matches SCSI HBA in slot 3 with a dev_num,func_num of 0x1234,2 and a productid,vendorid of 0x8178,0x9004, which enumerates:
   - identifier="<subset of INQUIRY data>"
   - address_locator="<bus><target><lun>"
   - physical_locator not used
   - physical_label not used

3. Matches External SCSI Disk Storage Unit at bus 1, target 4, luns 0,1
    - no enumeration is done at this level
4. Matches External SCSI Tape Device
    - no enumeration is done at this level

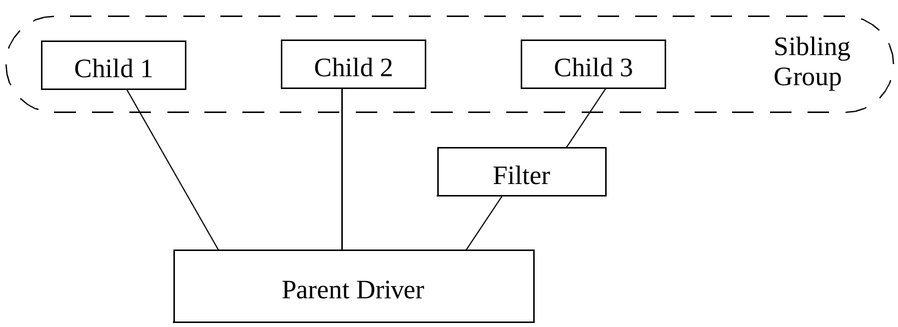
For an operating system that represents the device node tree to the user via filesystem notation, the above locators might result in the following identification:

/devices/xyz/scsi3-1234,devbay2/tgt2-0,disk
/devices/xyz/scsi3-1234,devbay2/tgt2-1,disk
/devices/xyz/scsi3-1234,devbay2/tgt2-255,ses
/devices/xyz/scsi3-1234,devbay2/tgt6-0,tape

Note – The above example does not reflect actual definitions for enumeration attributes nor actual devices and presents only one style of combining and representing locator attributes. The UDI drivers and metalanguages will define the actual device attributes and locator attributes and the environment is free to use this locator information for any style of representation that it chooses.

15.4.3 Sibling Group Attributes

Sibling group attributes are volatile attributes only; unlike instance-private attributes, they are global to all sibling instances in a *sibling group*. A sibling group is defined as the set of driver instances that share the same parent instance; i.e. siblings are the set of child instances enumerated by the parent at device enumeration time. It is important to note that sibling instances do not have to be instances of the same driver. For example, a PCI network adapter and a PCI SCSI adapter may be siblings if enumerated by the same PCI parent bus device. It is expected that filter and multiplexer modules will not appear in the sibling/parent relationship.



The attributes for the sibling group are effectively associated with the parent instance, although they are not visible to the parent itself. Instead, the sibling group attributes are visible to all members of that sibling group. Each sibling group member may read and write sibling attributes, although all sibling attributes are volatile and will not be available across system boots.

Sibling group attribute names must begin with a caret (‘^’) prefix character. Since they are global to all sibling group members, it is recommended that enumeration locator information for the relevant device be included in the attribute name in order to make the name unique. It is assumed that all siblings that need to share information using sibling attributes will use the same algorithm for unique differentiation of their attribute names and will therefore be able to locate these shared attributes.

15.4.4 Parent-Visible Attributes

Parent-visible attributes are attributes that are set on a child instance but which may be read (but not written) only by that instance's parent (and read and written by the environment). These attributes are used by the system administrator to specify configuration information about the child that may be needed by the parent. These types of attributes are defined by the parent driver instance or by the associated metalanguage.

Parent-visible attribute names must begin with an at-sign ('@') prefix character and are persistent.

15.4.5 Attribute Classification

The properties of each instance attribute class are summarized in the following table. For each class, the table specifies the prefix character for the class and whether or not attributes in that class are persistent. It also specifies which driver instances are allowed to write (set) and read (get) the attributes, relative to the driver instance with which the attributes are associated ("self"), and whether or not the attributes are intended to be customized by system administrators or other aspects of the environment. Drivers must provide reasonable default action in cases where custom attributes are not set. The "custom" declaration in the driver's static properties provides a way to guide administrative input of custom attributes.

Table 15-1 Instance Attribute Classification Table

Attribute Class	**Prefix**	**Persistent?**	**Writable by Whom?**	**Readable by Whom?**	**Customized by Environment?**
Private Persistent	%	√	self	self	√
Private Volatile	$		self	self	
Enumeration			parent[1]	self	
Sibling Group	^		child	child	
Parent-Visible	@	√	–	parent	√

1. Enumeration attributes are writable only at enumeration time (write-once semantics) by the parent via `udi_enumerate_ack`, not via `udi_instance_attr_set`.

15.5 Instance Attribute Services

This section describes the structural representation of the instance attributes and how they are manipulated by a UDI driver. The method and location of storing attributes is up to the environment implementation so long as it supports the requirements defined by this specification.

NAME **udi_instance_attr_type_t** *Instance attribute data-type type*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit8_t udi_instance_attr_type_t;

/* Instance Attribute Types */
#define UDI_ATTR_NONE                     0
#define UDI_ATTR_STRING                   1
#define UDI_ATTR_ARRAY8                   2
#define UDI_ATTR_UBIT32                   3
#define UDI_ATTR_BOOLEAN                  4
#define UDI_ATTR_FILE                     5
```

DESCRIPTION This type is used to identify the data type of an instance attribute. Instance attribute data types determine the storage requirements, encodings, and semantics of instance attribute values.

A list of supported instance attribute data type codes is given below, along with a description of each attribute.

UDI_ATTR_NONE indicates that an attribute has no current value. This type is only legal with an attribute length of zero.

UDI_ATTR_STRING identifies a null-terminated character string, consisting of Unicode characters encoded with the UTF-8 byte-stream character encoding. This encoding ensures that any byte in the string that has the 8th bit clear is in fact an ASCII character and not part of a multi-byte character. The null-terminator byte is considered part of the attribute value and is required.

UDI_ATTR_ARRAY8 identifies a sequence of `udi_ubit8_t` values.

UDI_ATTR_UBIT32 identifies a single `udi_ubit32_t` value. The attribute length for attributes of this type must be exactly `sizeof(udi_ubit32_t)`.

UDI_ATTR_BOOLEAN identifies a single `udi_boolean_t` value. The attribute length for attributes of this type must be exactly `sizeof(udi_boolean_t)`.

UDI_ATTR_FILE identifies a read-only attribute whose value is contained in a driver-provided external file. The attribute name must match a “`readable_file`” entry in the driver’s persistent configuration information, optionally suffixed with a colon (‘:’) followed by ASCII digits representing a decimal integer up to 2^{24}-1. The suffix indicates the beginning file offset to read from; zero is the default. If this offset suffix is provided, it does not count as part of the actual attribute name, so does not have to fit within the 63-character limit.

NAME **udi_instance_attr_get** *Read an attribute value for a driver instance*

SYNOPSIS

```
#include <udi.h>

void udi_instance_attr_get (
     udi_instance_attr_get_call_t *callback,
     udi_cb_t *gcb,
     const char *attr_name,
     udi_ubit32_t child_ID,
     void *attr_value,
     udi_size_t attr_length );

typedef void udi_instance_attr_get_call_t (
     udi_cb_t *gcb,
     udi_instance_attr_type_t attr_type,
     udi_size_t actual_length );
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

attr_name is a null-terminated string specifying the attribute name. See Section 15.2, "Instance Attribute Names", and the `UDI_ATTR_FILE` attribute type for rules on attribute names.

child_ID is the child ID associated with the specific child instance for which this attribute has been set if it is a parent-visible attribute (prefix character '@').

For parent-visible attributes, this argument must match a ***child_ID*** from a previous `udi_enumerate_ack` that has not been unenumerated; it is ignored for other types of attributes.

attr_value is a pointer to a memory area to receive the attribute value.

attr_length is the length in bytes of the memory area pointed to by ***attr_value***.

attr_type is the type specifier for the attribute value. See **udi_instance_attr_type_t** on page 15-7 for details.

actual_length is the actual length of the attribute value, even if it could not fit in the ***attr_value*** memory area.

DESCRIPTION

The `udi_instance_attr_get` function is used to obtain the value of a driver instance attribute. The returned attribute value will be written to the memory area specified by ***attr_value.***

If ***attr_name*** contains a colon (':'), the rest of the name must be an ASCII-encoded decimal number and ***attr_type*** must be UDI_ATTR_FILE. In this case, the number indicates the beginning file offset to read from, in bytes, starting from zero.

If the requested attribute does not exist, the ***callback*** routine will be called with an ***actual_length*** of 0 and an ***attr_type*** of UDI_ATTR_NONE.

Otherwise, ***actual_length*** will be set to the actual length of the attribute value, regardless of ***attr_length***; in the case of UDI_ATTR_FILE with an offset specified, this will be the remaining length relative to the specified file. For attribute types other than UDI_ATTR_FILE, if ***actual_length*** exceeds ***attr_length***, the contents of the ***attr_value*** memory area are unspecified; for UDI_ATTR_FILE, all valid bytes that fit will be filled in.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of "*Standard Calling Sequences*".

Use of the ***attr_name*** and ***attr_value*** parameters must conform to the rules described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls".

REFERENCES

`udi_instance_attr_type_t, udi_instance_attr_set`

NAME **udi_instance_attr_set** *Set a driver instance attribute value*

SYNOPSIS

```
#include <udi.h>

void udi_instance_attr_set (
    udi_instance_attr_set_call_t *callback,
    udi_cb_t *gcb,
    const char *attr_name,
    udi_ubit32_t child_ID,
    const void *attr_value,
    udi_size_t attr_length,
    udi_ubit8_t attr_type );

typedef void udi_instance_attr_set_call_t (
    udi_cb_t *gcb,
    udi_status_t status );
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

attr_name is the name of the attribute whose value is to be set.

child_ID is the child ID associated with the specific child instance for which this attribute is to be set if it is a parent-visible attribute (prefix character ‘@’).

For parent-visible attributes, this argument must match a ***child_ID*** from a previous `udi_enumerate_ack` that has not been unenumerated; it is ignored for other types of attributes.

attr_value is a pointer to the attribute value to set. ***attr_value*** must be NULL if and only if ***attr_length*** is 0.

attr_length is the length of the value pointed to by ***attr_value***.

attr_type is the type specifier for the attribute value. See **udi_instance_attr_type_t** on page 15-7 for details. UDI_ATTR_FILE is not allowed with `udi_instance_attr_set`. ***attr_type*** must be UDI_ATTR_NONE if and only if ***attr_length*** is zero.

DESCRIPTION

The `udi_instance_attr_set` function is used to set the value of a driver-instance attribute. The attribute to set is specified by ***attr_name*** and may be either a persistent or a volatile attribute depending on the attribute type (as indicated by a prefix character).

The ***attr_value***, ***attr_length***, and ***attr_type*** combine to specify the attribute value. If the attribute does not presently exist, it is created. If the current attribute type is different than ***attr_type***, the attribute type will be changed to the newly specified type. If the attribute length ***attr_length*** is specified as zero, the attribute may be deleted from the database. In general, a zero-length attribute is indistinguishable from a non-existent attribute.

The length of the attribute value specified by ***attr_value*** and ***attr_length*** must not exceed the maximum length specified by the ***max_instance_attr_len*** member of the `udi_limits_t` structure.

The ***status*** value indicates the success or failure of the attribute modification operation.

The `udi_instance_attr_set` service call must not be used with the UDI_ATTR_FILE attribute type.

WARNINGS Control block usage must follow the rules described in the "Asynchronous Service Calls" section of "*Standard Calling Sequences*".

Use of the ***attr_name*** and ***attr_value*** parameters must conform to the rules described in Section 5.2.1.1, "Using Memory Pointers with Asynchronous Service Calls".

STATUS VALUE `UDI_OK` the attribute value was successfully modified.

`UDI_STAT_RESOURCE_UNAVAIL` the persistent storage database is full and this attribute could not be created or set in the database. The driver is not expected to retry the operation; it should consider this a permanent failure.

`UDI_STAT_NOT_SUPPORTED` the current environment does not allow modification of the persistent storage database. This error can only occur with persistent attributes.

REFERENCES `udi_instance_attr_get`, `UDI_INSTANCE_ATTR_DELETE`, `udi_limits_t`

NAME

UDI_INSTANCE_ATTR_DELETE *Driver instance attribute delete macro*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_INSTANCE_ATTR_DELETE( \
     callback, gcb, attr_name) \
          udi_instance_attr_set( \
               callback, gcb, attr_name, NULL, \
               NULL, 0, UDI_ATTR_NONE);
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

attr_name is the name of the attribute to delete.

DESCRIPTION

The `UDI_INSTANCE_ATTR_DELETE` macro is a convenience macro which may be used to remove a driver instance attribute. As defined above, this macro utilizes the `udi_instance_attr_set` service call and sets the ***attr_length*** parameter to zero to effect the deletion of the corresponding attribute.

The callback function specified for this macro must be of the `udi_instance_attr_set_call_t` type.

REFERENCES

`udi_instance_attr_set`

NAME **udi_instance_attr_list_t** *Enumeration instance attribute list*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     char attr_name[UDI_MAX_ATTR_NAMELEN];
     udi_ubit8_t attr_value[UDI_MAX_ATTR_SIZE];
     udi_ubit8_t attr_length;
     udi_instance_attr_type_t attr_type;
} udi_instance_attr_list_t;

/* Instance attribute limits */
#define UDI_MAX_ATTR_NAMELEN            32
#define UDI_MAX_ATTR_SIZE               64
```

MEMBERS

attr_name is the name of the instance attribute.

attr_value is the value of this instance attribute.

attr_length is the valid length (in bytes) of the ***attr_value*** and must not be zero.

attr_type is the attribute type as specified for **udi_instance_attr_type_t** on page 15-7. Must not be UDI_ATTR_NONE or UDI_ATTR_FILE.

DESCRIPTION

The udi_instance_attr_list_t structure is used to hold a value used to pre-load an enumeration instance attribute. The MA allocates space for a contiguous array of these structures as a movable memory block in order to provide information describing a child instance in an enumeration operation (see “Enumeration Operations” on page 24-13).

If ***attr_type*** is UDI_ATTR_UBIT32, the 32-bit value is encoded as a little-endian value in the first four bytes of ***attr_value***, and ***attr_length*** must be 4. In this case, UDI_ATTR32_SET and UDI_ATTR32_GET must be used to access ***attr_value***, or UDI_ATTR32_INIT must be used to statically initialize such a value before copying it into this structure.

REFERENCES

udi_mem_alloc, udi_instance_attr_type_t,
UDI_ATTR32_SET, UDI_ATTR32_GET, UDI_ATTR32_INIT

NAME

UDI_ATTR32_SET/GET/INIT *Instance attribute encoding/decoding utilities*

SYNOPSIS

```
#include <udi.h>

#define UDI_ATTR32_SET(aval, v) \
         { udi_ubit32_t vtmp = (v); \
           (aval)[0] = (vtmp) & 0xff; \
           (aval)[1] = ((vtmp) >> 8) & 0xff; \
           (aval)[2] = ((vtmp) >> 16) & 0xff; \
           (aval)[3] = ((vtmp) >> 24) & 0xff; }

#define UDI_ATTR32_GET(aval) \
         ((aval)[0] + ((aval)[1] << 8) + \
          ((aval)[2] << 16) + ((aval)[3] << 24))

#define UDI_ATTR32_INIT(v) \
         { (v) & 0xff, ((v) >> 8) & 0xff, \
           ((v) >> 16) & 0xff, ((v) >> 24) & 0xff }
```

ARGUMENTS

aval is the ***attr_value*** array which holds an encoded UDI_ATTR_UBIT32 instance attribute value.

v is the udi_ubit32_t value for the instance attribute.

DESCRIPTION

These utility macros are used to access values in the ***attr_value*** member of a udi_instance_attr_list_t structure when ***attr_type*** is UDI_ATTR_UBIT32. In this case, the 32-bit unsigned integer value is encoded as a little-endian value in the first four bytes of ***attr_value***.

UDI_ATTR32_SET assigns a udi_ubit32_t value to an ***attr_value*** array, using the above encoding.

UDI_ATTR32_GET extracts a udi_ubit32_t value from an ***attr_value*** array, using the above encoding.

UDI_ATTR32_INIT initializes an ***attr_value*** from a udi_ubit32_t constant value as a compile-time initializer.

REFERENCES

udi_mem_alloc, udi_instance_attr_list_t, udi_instance_attr_type_t

Inter-Module Communication 16

16.1 Overview

The Inter-Module Communication (IMC) services allow drivers to create new *channels*, to anchor channels within a *region*, to dynamically set the *channel context*, and to close a channel. This chapter also defines the *channel event indication* operation, which is used to send channel-related events from the environment to a driver. See Chapter 4, *"Execution Model"*, for an introduction to regions and channels.

16.2 Service Calls

This section defines service calls that allow drivers to create and anchor channels, dynamically set a channel's context, close a channel, abort outstanding channel operations, and process channel-related events.

Note that the primary region and management channel for each driver instance are provided to the driver automatically by the Management Agent when the driver instance is created based on information provided by the driver in its `udi_init_info` variable. If the driver has indicated that it requires static secondary regions, they will also be created at this time, along with an internal bind channel between the primary region and each such secondary region. If the driver has requested dynamic secondary regions, additional secondary regions and corresponding internal bind channels will be created later, when appropriate child or parent instances are being bound to this driver instance.

Drivers can spawn new channels at any time, but it is the driver's responsibility to allocate, track and return these objects back to the environment in a responsible manner. Drivers must not free any channels not explicitly spawned by the driver, via calls to the services in this chapter.

NAME

udi_channel_anchor *Anchor a channel to the current region*

SYNOPSIS

```
#include &lt;udi.h&gt;

void udi_channel_anchor (
     udi_channel_anchor_call_t *callback,
     udi_cb_t *gcb,
     udi_channel_t channel,
     udi_index_t ops_idx,
     void *channel_context );

typedef void udi_channel_anchor_call_t (
     udi_cb_t *gcb,
     udi_channel_t anchored_channel );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

channel is the channel handle for the loose end to be anchored. Once udi_channel_anchor is called, the driver may no longer use this handle.

ops_idx is an ops index for the ops vector that the driver wants to associate with the specified channel, as indicated by the appropriate udi_ops_init_t in udi_init_info. ***ops_idx*** must be non-zero.

channel_context is a channel context pointer to be associated with the anchored channel endpoint.

anchored_channel is the new channel handle for the now-anchored channel endpoint. This handle must subsequently be used to access the channel, rather than the original handle passed to udi_channel_anchor.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

DESCRIPTION

udi_channel_anchor is used to anchor a loose channel end to the current region. Loose ends may be passed to a driver from another region, or as the result of a udi_channel_spawn request. Management channels, external bind channels between driver instances, and internal bind channels between primary and secondary regions are always pre-anchored.

Once anchored, the channel endpoint is permanently associated with the current region, and has an associated ops vector and channel context. Loose ends may be anchored, but anchored ends may not be made loose.

Loose ends may be passed between regions as parameters to channel operations. Anchored ends may not.

When the anchoring is complete, the UDI environment will invoke the ***callback*** to notify the requestor of the completion, and return ownership of the control block (***gcb***) to the driver.

Once both ends of the channel are anchored, the channel may be used for communication, by invoking channel operations. Drivers must ensure that both ends are anchored and ready to go before invoking any operations on the channel. This is typically done via metalanguage-specific handshaking on another channel.

REFERENCES udi_init_info, udi_cancel, udi_channel_spawn

NAME **udi_channel_spawn** *Spawn a new channel*

SYNOPSIS

```
#include <udi.h>

void udi_channel_spawn (
     udi_channel_spawn_call_t *callback,
     udi_cb_t *gcb,
     udi_channel_t channel,
     udi_index_t spawn_idx,
     udi_index_t ops_idx,
     void *channel_context );

typedef void udi_channel_spawn_call_t (
     udi_cb_t *gcb,
     udi_channel_t new_channel );
```

ARGUMENTS

callback, gcb are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*

channel is the channel handle for an existing anchored channel. The new channel will be spawned relative to this channel.

spawn_idx is a small integer which allows the environment to match two spawn requests (one from each end of the channel) together.

ops_idx is an ops index for the ops vector that the driver wants to associate with the specified channel, as indicated by the appropriate `udi_ops_init_t` in `udi_init_info`, or zero.

channel_context is a channel context pointer to be associated with the new anchored channel endpoint.

new_channel is the channel handle for the new channel's local endpoint, which will be a loose end. This handle must subsequently be passed to `udi_channel_anchor`, either in this region, or after passing it to another region via a channel operation.

WARNINGS

Control block usage must follow the rules described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*.

DESCRIPTION

`udi_channel_spawn` is used to create a new channel (initially) between the same two regions as an existing channel. Both ends must be created separately by their own calls to `udi_channel_spawn`.

If ***ops_idx*** is zero, the channel endpoint is created as a loose end, which must be anchored before it can be used. Loose ends may be passed between regions, and even between drivers, before being anchored.

The pair of the original channel handle and the spawn index uniquely identify an in-progress spawn operation. The ***callback*** routine is called once the local end of the channel has been created and, if specified, anchored. The other region may or may not yet have completed its end of the spawn.

Drivers must ensure that both ends have completed spawning and are anchored and ready to go before invoking any operations on the channel. This is typically done via metalanguage-specific handshaking on the original channel.

REFERENCES `udi_init_info, udi_cancel, udi_channel_anchor`

NAME **udi_channel_set_context** *Attach a new context to a channel endpoint*

SYNOPSIS

```
#include <udi.h>

void udi_channel_set_context (
     udi_channel_t target_channel,
     void *channel_context );
```

ARGUMENTS

target_channel is a channel handle for the channel endpoint to be modified.

channel_context is a generic pointer that will be returned as-is by UDI in any channel operations related to this channel.

DESCRIPTION

`udi_channel_set_context` attaches a new context pointer to the local end of a target channel. The new context pointer will be attached to the referenced channel (***target_channel***) by the time this call returns. It will then be passed to the driver with each channel operation in the ***gcb.channel_context*** member of the control block.

WARNINGS

`udi_channel_set_context` must be called from the region containing the channel endpoint. This endpoint must already be anchored.

NAME **udi_channel_op_abort** *Abort a previously issued channel operation*

SYNOPSIS

```
#include &lt;udi.h&gt;

void udi_channel_op_abort (
     udi_channel_t target_channel,
     udi_cb_t *orig_cb );
```

ARGUMENTS

target_channel is a channel handle for the channel to which the previously issued operation was sent.

orig_cb is a control block pointer for the control block that was sent with the original operation. Even though the driver no longer owns that control block, it is allowed to use the otherwise stale pointer only with this service call.

DESCRIPTION

udi_channel_op_abort delivers a UDI_CHANNEL_OP_ABORTED event via udi_channel_event_ind to the other end of the target channel, in order to request that a previously sent request, using ***orig_cb*** as its control block, be aborted.

The original operation must be of an operation type defined to be *abortable* by the relevant metalanguage definition. Metalanguage libraries indicate that operations are abortable by using the UDI_MEI_OP_ABORTABLE flag in the corresponding udi_mei_op_template_t.

The original control block, identified by ***orig_cb***, must previously have been sent on the target channel using an abortable operation, and must not yet have been returned (via a metalanguage-specific operation) to the initiating region. The control block is aborted and returned to the current driver via the normal metalanguage completion operation with a status of UDI_STAT_ABORTED to indicate that the operation was aborted; operations that have already completed will be passed back to the current module in the normal fashion without the abort status indication.

Even if the control block was originally sent as part of a chain of control blocks sent with one operation, only the specific control block indicated by ***orig_cb*** is aborted.

Drivers receiving abortable control blocks must not free them but must (eventually) return them over the same channel on which they were received.

REFERENCES

udi_channel_event_cb_t, udi_channel_event_ind, udi_mei_op_template_t

NAME **udi_channel_close** *Close a channel*

SYNOPSIS

```
#include <udi.h>

void udi_channel_close ( udi_channel_t channel );
```

ARGUMENTS ***channel*** is a channel handle for the channel endpoint being closed.

DESCRIPTION `udi_channel_close` deallocates and returns any channel-related resources to the UDI environment. Normally a driver calls this routine only as a result of receiving a `channel_event_ind` operation of type `UDI_CHANNEL_CLOSED`, to close its end of the channel.

The result of this routine is immediate: the channel endpoint will be closed and freed when this call returns. It is the responsibility of the driver to clean up all channel-related state and resources first, so as to maintain architectural integrity before destroying a channel. This must include the processing of all outstanding operations related to the channel. The driver should ensure, via proper channel operation handling, that all operations directed to this channel have been completed and that no more will be generated. Any operations previously sent to this channel but not yet delivered at the time this routine is called will be treated as having been initiated after the channel was closed.

When one end of a channel is closed, either by the driver explicitly calling `udi_channel_close` or by the environment if a driver is killed, the other end receives a `udi_channel_event_ind` operation of type `UDI_CHANNEL_CLOSED`. This tells the driver at the other end that one of its neighbors has gone away unexpectedly. (See page 16-13 for the definition of the `udi_channel_event_ind` operation).

`udi_channel_close` may be used on loose ends, as well as anchored channel endpoints.

If a driver calls `udi_channel_close` on a channel whose *other* end is loose, the `udi_channel_event_ind` operation will be delivered if and when that other end is anchored.

If a driver invokes an operation on a channel whose other end is closed, it will be ignored and any associated control blocks and data objects will be freed.

Once both ends of a channel are closed, all environment resources associated with the channel are released. Calling `udi_channel_close` on the single end of a half-spawned channel has this effect as well.

`udi_channel_close` acts as a no-op if ***channel*** is a null handle, but must not be called for a channel that has already been closed with a previous call to `udi_channel_close`.

WARNINGS `udi_channel_close` must not be used with a channel handle that has been passed to another region. `udi_channel_close` must not be used on management channels.

REFERENCES `udi_channel_event_ind`, `udi_channel_anchor`, `udi_channel_spawn`

16.3 Channel Event Indication Operation

The one channel operation common to all metalanguages (except the Management Metalanguage) is the `udi_channel_event_ind` operation. It is always the first operation in any channel ops vector. It is automatically invoked by the environment whenever one of several channel-related events occurs:

1. A `udi_channel_event_ind` operation of type `UDI_CHANNEL_CLOSED` is passed to the other end of a channel whenever a channel is closed. This can occur as a result of an explicit `udi_channel_close` (see page 16-8) or as a result of the region being prematurely terminated by the environment.

2. A `udi_channel_event_ind` operation of type `UDI_CHANNEL_BOUND` is passed to the *initiator* end of a newly created bind channel after it has been anchored (by the environment) on both ends. The *initiator*, as opposed to *responder*, is the driver (or environment entity) that generally initiates requests to the responder (see the description of ***relationship*** on page 28-4 for more details on these metalanguage roles). This allows the initiator to acquire the channel handle for its end of the channel (via the ***channel*** member of the control block), so it can send the first request. The responder acquires its channel handle as a result of this first request.

3. A `udi_channel_event_ind` operation of type `UDI_CHANNEL_OP_ABORTED` is passed to the other end of a channel whenever a driver calls `udi_channel_op_abort` (see page 16-7), in order to abort a previously sent channel operation.

The `udi_channel_event_ind` operation is used in all metalanguages except the Management Metalanguage, but is defined only once, here in this chapter in the reference pages that follow.

NAME **udi_channel_event_cb_t** *Channel event control block*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
     udi_ubit8_t event;
     union {
          struct {
               udi_cb_t *bind_cb;
          } internal_bound;
          struct {
               udi_cb_t *bind_cb;
               udi_ubit8_t parent_ID;
               udi_buf_path_t *path_handles;
          } parent_bound;
          udi_cb_t *orig_cb;
     } params;
} udi_channel_event_cb_t;

/* Channel event types */
#define UDI_CHANNEL_CLOSED              0
#define UDI_CHANNEL_BOUND               1
#define UDI_CHANNEL_OP_ABORTED          2
```

MEMBERS

gcb is the standard control block section providing scratch space and context information.

event is the type of event that is being indicated:

UDI_CHANNEL_CLOSED indicates that the remote end of the channel has been closed. The driver receiving this event must clean up any channel-related resources and call `udi_channel_close` on its end of the channel after calling `udi_channel_event_complete`.

UDI_CHANNEL_BOUND indicates that another region has been bound to this region as a result of a “parent_bind_ops” or “internal_bind_ops” declaration from this driver’s static driver properties (see Chapter 30, “*Static Driver Properties*”), and that this channel was created as a result of that binding. The driver receiving this event may now begin using the channel for normal operations, usually beginning with a metalanguage-specific bind request. (For internal bindings, only the region using the *initiator* role in the corresponding metalanguage receives the `UDI_CHANNEL_BOUND` indication.)

For parent bindings, the ***parent_ID*** field will contain an environment-assigned parent ID that identifies the particular parent instance. The ***parent_ID*** value will never be zero; zero is reserved for environment use. Single-parent drivers can ignore this field.

The driver must not call `udi_channel_event_complete` for this event until its entire bind sequence has completed.

See Chapter 24, *"Management Metalanguage"*, for more details on the bind sequence.

`UDI_CHANNEL_OP_ABORTED` indicates that an abort request has been generated (using `udi_channel_op_abort`) by the driver on the other end of the channel, with respect to a previous metalanguage-specific abortable request (the *original request*). The driver receiving this event must abort any outstanding processing for the original request and fail it with a status code of `UDI_STAT_ABORTED`; it may do this before or after calling `udi_channel_event_complete`. The ***orig_cb*** field will point to the control block for the original request, which is guaranteed to be an abortable control block currently owned by this region (though it may be in use with an environment service call on behalf of this region). Drivers receiving abortable control blocks must not free them but must (eventually) return them over the same channel on which they were received.

orig_cb is a pointer to the control block for the original request being aborted by a `UDI_CHANNEL_OP_ABORTED` channel event. For all other channel events, the value of ***orig_cb*** is unspecified and must not be used by the driver.

bind_cb is a pointer to the pre-allocated control block for the metalanguage-specific bind request that the driver will issue as a result of the `UDI_CHANNEL_BOUND` event. This control block type is indicated by the `<bind_cb_idx>` value of the corresponding "parent_bind_ops" declaration (see Section 30.6.3 on page 30-13) or "internal_bind_ops" declaration (see Section 30.6.4 on page 30-14). For all other channel events or if `<bind_cb_idx>` was zero then no control block will be allocated and the value of ***bind_cb*** is unspecified and must not be used by the driver.

parent_ID is a unique non-zero value supplied by the MA during a `UDI_CHANNEL_BOUND` event on a parent bind channel, to explicitly identify the parent driver instance being bound. Drivers that have multiple parents will be assigned a unique ***parent_ID*** value for each parent. This ***parent_ID*** is used for any operations that need to identify a specific parent to which those operations are related (*e.g.* the enumeration and device management operations of the Management Metalanguage). If the event was not for a parent binding, this member's value is unspecified and must be ignored.

The MA may assign parent IDs in any order.

path_handles is a pointer to an array of udi_buf_path_t handles that may be used by the driver when allocating buffers on behalf of the parent being bound. These handles are maintained in an inline array associated with the channel event control block and must be copied to instance-internal storage before the control block is passed to udi_channel_event_complete. If the event was not for a parent binding, this member's value is unspecified and must be ignored.

DESCRIPTION

The udi_channel_event_cb_t control block is a semi-opaque object used between the environment and the driver in channel event indication operations. When passed to the target driver, this control block provides a context for the operation and must be returned to the environment by calling udi_channel_event_complete.

Unlike with other control blocks, the value of ***gcb.channel*** for a channel event control block is unspecified and must not be modified.

Unlike with other control blocks, there is no way to list attributes of a udi_channel_event_cb_t in a udi_cb_init_t initialization structure, so the scratch space size for udi_channel_event_cb_t control blocks is always zero.

Drivers cannot allocate udi_channel_event_cb_t control blocks.

REFERENCES

udi_channel_event_complete, udi_channel_close, udi_channel_op_aborted, udi_constraints_propagate, udi_layout_t

NAME **udi_channel_event_ind** *Channel event notification (env-to-driver)*

SYNOPSIS

```
#include <udi.h>

void udi_channel_event_ind (
     udi_channel_event_cb_t *cb );

typedef void udi_channel_event_ind_op_t (
     udi_channel_event_cb_t *cb );
```

ARGUMENTS

cb is a channel event control block allocated by the environment and used to hold details of the specific channel event.

TARGET CHANNEL The channel over which the event is to be delivered.

DESCRIPTION This channel operation is used by the environment to signal that a generic event has occured on the other end of the channel. The type of event that has occured, and additional parameters for the event, are contained in the `udi_channel_event_cb_t` control block.

If a driver receives an unexpected `UDI_CHANNEL_CLOSED` event indication on a parent or child channel, it must treat it as an "abrupt unbind", as described in Section 24.6, "Device Management Operations," on page 24-27.

If a driver closes its end of the channel itself (with `udi_channel_close`) before the other end is closed or before a `udi_channel_event_ind` of type `UDI_CHANNEL_CLOSED` is serviced, it will not receive a `UDI_CHANNEL_CLOSED` indication.

Once a `UDI_CHANNEL_CLOSED` indication has been received on a given channel, no other operations will be received on that channel.

Once the driver has completed processing the channel event, it must return the control block to the environment using `udi_channel_event_complete`. Drivers must not directly free channel event control blocks.

WARNINGS Drivers must not invoke this operation.

Drivers handling `UDI_CHANNEL_OP_ABORTED` events should be careful to abort the referenced control block (or at least mark it as having an abort in progress) before returning from the `udi_channel_event_ind` operation, to avoid race conditions with normal completions.

REFERENCES `udi_channel_event_cb_t, udi_channel_event_complete`

NAME **udi_channel_event_complete** *Complete a channel event (driver-to-env)*

SYNOPSIS

```
#include <udi.h>

void udi_channel_event_complete (
    udi_channel_event_cb_t *cb,
    udi_status_t status );
```

ARGUMENTS

cb is a channel event control block from `udi_channel_event_ind`.

status is a UDI status code used to indicate the success or failure of some event types. Unless otherwise specified, drivers must set ***status*** to UDI_OK.

DESCRIPTION `udi_channel_event_complete` is called by a driver that has previously received a `udi_channel_event_ind` notification from the environment after that driver has processed the event. Drivers must not directly free channel event control blocks.

STATUS VALUES

UDI_OK The event was processed successfully.

UDI_STAT_CANNOT_BIND A parent binding triggered by a `UDI_CHANNEL_BOUND` event failed because the metalanguage specific bind process was rejected by the parent, or was otherwise unsuccessful.

UDI_STAT_TOO_MANY_PARENTS A parent binding triggered by a `UDI_CHANNEL_BOUND` event failed because this driver instance is already bound to the maximum number of parents that it can support.

UDI_STAT_BAD_PARENT_TYPE A parent binding triggered by a `UDI_CHANNEL_BOUND` event failed because the parent metalanguage or device properties (as determined by the parent-specified enumeration attributes) for the binding cannot be supported by this driver instance in its current state.

UDI_STAT_ATTR_MISMATCH A parent binding triggered by a `UDI_CHANNEL_BOUND` event failed because the child could not comply with one or more of the custom attribute settings already specified for the newly-created child instance.

WARNINGS The control block must be the same control block as passed to the driver in the corresponding `udi_channel_event_ind` operation.

REFERENCES `udi_channel_event_cb_t, udi_channel_event_ind`

Tracing and Logging 17

17.1 Overview

UDI environments are expected to provide facilities for drivers to record information about their operation. There are two such types of information: *log data*, which describes infrequent events read by a system administrator to determine the state of a running system, and *trace data*, which is divided into classes of information used by developers and systems analysts for debugging UDI modules or the UDI subsystem as a whole. A driver is required to provide log data; providing trace data is optional.

This chapter defines the tracing and logging service calls which are provided for use by the driver to record both trace data and log data. Additional operations to specify the level of tracing generated are defined in the Management Metalanguage (see “Tracing Control Operations” on page 24-6).

17.2 Tracing and Logging Service Calls

17.2.1 Tracing Calls

Tracing is initially disabled when a driver instance is initialized. The Management Metalanguage includes a channel operation to enable or disable tracing of specified types of events in a driver instance (see Section 24.4.1, “Tracing Control Operations”). Depending on the issue being debugged, the driver may be asked to trace only rare errors, all internal function calls, or somewhere in between. The actual set of events traced is up to the discretion of the driver implementation, but must in all cases be a subset of the currently enabled set of trace event types. Note that the environment may be filtering final trace output by some other criteria, even though the driver itself filters only by trace event type.

When the driver encounters an event to be traced, it calls `udi_trace_write`, passing the trace event code, a data buffer, and a pointer to its `udi_init_context_t` structure (identifying this driver region as the source of the data). The contents and format of trace data are driver implementation-dependent.

17.2.2 Logging Calls

Major events, including any situation where a `udi_status_t` value other than UDI_OK that indicates an exceptional condition is generated, should be logged by UDI drivers using `udi_log_write`. Logging is always active; it is not controlled by event masks as tracing is (though `udi_log_write` may be used to simultaneously log and trace data). As with tracing, however, the environment may choose to filter final log output on its own.

Although drivers can function without logging any data, making calls to `udi_log_write` when appropriate should not be considered optional. Such logging is particularly important because `udi_log_write` tags `udi_status_t` values with a correlation code (see Section 9.9.1, “UDI Status,” on page 9-15), allowing it to associate related errors as they are passed from driver to driver.

17.2.3 Trace Event Types

Trace events specify the types of trace data which the driver is to report at any given time. Setting the corresponding bit value in the ***trace_event_mask*** mask in a `udi_usage_ind` operation (see Section 24.4.1) enables tracing for all events of a particular type. Some event types are designed to trace metalanguage-specific information or operations and are thus selectable on a per-metalanguage basis; these are referred to as metalanguage-selectable trace event types.

The trace events are divided into four different classes as defined in this section:

1) *common trace event* codes, which apply to all drivers and metalanguages.
2) *common metalanguage-selectable trace event* codes, whose semantics are defined in each metalanguage. The general semantics of the *common metalanguage-selectable events* are defined in this chapter; metalanguages can define more specific semantics as they conform to the general semantics defined here.
3) *metalanguage-specific trace event* codes (`UDI_TREVENT_META_SPECIFIC_`*n*), with semantics defined by each metalanguage, are also metalanguage-selectable.
4) *driver-specific trace event* codes (`UDI_TREVENT_INTERNAL_`*n*), which are available for tracing events specific to a single driver implementation.

Note that drivers and metalanguages can define the use of the driver or metalanguage-selectable event codes, respectively, without having to worry about event code usage defined by other drivers or metalanguages, since each trace call is associated with a particular calling driver and a selected metalanguage.

Note – Environment implementations may trace other types of events transparently to the driver, such as incoming and outgoing channel operations and service calls.

NAME **udi_trevent_t** *Trace event type definition*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit32_t udi_trevent_t;

/* Common Trace Events */
#define UDI_TREVENT_LOCAL_PROC_ENTRY      (1U<<0)
#define UDI_TREVENT_LOCAL_PROC_EXIT       (1U<<1)
#define UDI_TREVENT_EXTERNAL_ERROR        (1U<<2)

/* Common Metalanguage-Selectable Trace Events */
#define UDI_TREVENT_IO_SCHEDULED          (1U<<6)
#define UDI_TREVENT_IO_COMPLETED          (1U<<7)

/* Metalanguage-Specific Trace Events */
#define UDI_TREVENT_META_SPECIFIC_1       (1U<<11)
#define UDI_TREVENT_META_SPECIFIC_2       (1U<<12)
#define UDI_TREVENT_META_SPECIFIC_3       (1U<<13)
#define UDI_TREVENT_META_SPECIFIC_4       (1U<<14)
#define UDI_TREVENT_META_SPECIFIC_5       (1U<<15)

/* Driver-Specific Trace Events */
#define UDI_TREVENT_INTERNAL_1            (1U<<16)
#define UDI_TREVENT_INTERNAL_2            (1U<<17)
#define UDI_TREVENT_INTERNAL_3            (1U<<18)
#define UDI_TREVENT_INTERNAL_4            (1U<<19)
#define UDI_TREVENT_INTERNAL_5            (1U<<20)
#define UDI_TREVENT_INTERNAL_6            (1U<<21)
#define UDI_TREVENT_INTERNAL_7            (1U<<22)
#define UDI_TREVENT_INTERNAL_8            (1U<<23)
#define UDI_TREVENT_INTERNAL_9            (1U<<24)
#define UDI_TREVENT_INTERNAL_10           (1U<<25)
#define UDI_TREVENT_INTERNAL_11           (1U<<26)
#define UDI_TREVENT_INTERNAL_12           (1U<<27)
#define UDI_TREVENT_INTERNAL_13           (1U<<28)
#define UDI_TREVENT_INTERNAL_14           (1U<<29)
#define UDI_TREVENT_INTERNAL_15           (1U<<30)

/* Logging Event */
#define UDI_TREVENT_LOG                   (1U<<31)
```

DESCRIPTION The `udi_trevent_t` type definition is used to specify a bitmask of trace events. These trace events are used in the tracing and logging service calls to specify the occurrence of events or to provide masks to filter the set of interesting trace events.

The following common trace event codes are defined independently of any metalanguage.

UDI_TREVENT_LOCAL_PROC_ENTRY – Trace entry to all procedures that are local to the driver. Include argument values in the trace output.

UDI_TREVENT_LOCAL_PROC_EXIT – Trace exit from all procedures that are local to the driver. Include return values in the trace output.

UDI_TREVENT_EXTERNAL_ERROR – Trace error conditions that are passed from this driver to other UDI drivers or modules. This happens when a `udi_status_t` value other than `UDI_OK` that indicates an exceptional condition is generated. Such events must be logged using `udi_log_write` (which will handle the tracing of this event as well).

The following trace event types are designed to trace metalanguage-specific information or operations, and can therefore be selectively enabled and disabled on a per-metalanguage basis. For these events, tracing is enabled or disabled only for the metalanguages indicated by ***meta_idx*** of the trace usage operation (see “Tracing Control Operations” on page 24-6). Each metalanguage defines its own rules and conventions for the use of these event types; therefore, the metalanguage specifications should be consulted before using these events.

UDI_TREVENT_IO_SCHEDULED - Trace the point at which the driver starts handling a specific I/O request. The use of this trace point is different for different types of drivers but should indicate the point at which the driver passes the I/O request to the hardware. (Example: submission of a SCSI command to the hardware to be sent on the SCSI bus.) This trace event applies only to the responder role of a request/response metalanguage (e.g. GIO provider).

UDI_TREVENT_IO_COMPLETED - Trace the point at which an I/O request has been completed. This is the counterpart to `UDI_TREVENT_IO_SCHEDULED` and in a similar fashion the use of this trace event is determined by type of UDI driver. (Example: Interrupt indicating SCSI command complete.) This trace event applies only to the responder role of a request/response metalanguage (e.g. GIO provider).

UDI_TREVENT_META_SPECIFIC_*n* - Trace metalanguage-specific events as defined in each metalanguage.

The driver-internal trace events, **UDI_TREVENT_INTERNAL_*n***, may be used to trace any driver-specific events desired. The interpretation of those events is determined by the driver implementor.

The logging event code is a special trace event code which is used to indicate that a logging event has occurred rather than one of the customary trace events. Logging events cannot be filtered.

UDI_TREVENT_LOG – Event code that is used for logging messages that are not associated with trace events. This event code must only be used with `udi_log_write` and not with `udi_trace_write`.

NAME **udi_trace_write** *Record trace data*

SYNOPSIS

```
#include <udi.h>

void udi_trace_write (
     udi_init_context_t *init_context,
     udi_trevent_t trace_event,
     udi_index_t meta_idx,
     udi_ubit32_t msgnum,
     ... );
```

ARGUMENTS

init_context is a pointer to the front of the driver's region data area and is used to uniquely identify this driver instance.

trace_event is the type of trace event being reported.

meta_idx is a metalanguage index number that identifies the metalanguage to which ***trace_event*** is relative, for metalanguage-selectable trace events. It must match the value of `<meta_idx>` in the corresponding "child_meta", "parent_meta", or "internal_meta" declaration of the driver's Static Driver Properties (see Chapter 30), or 0 for the Management Metalanguage. If the event is not metalanguage-selectable, ***meta_idx*** is ignored.

msgnum is the index value of a message string provided in the driver's static properties file (see Section 30.4.9, "Message Declaration," on page 30-7). This selects the text of the message to be traced. Any embedded formatting codes in the text of that message will be used to format the traced message with the remaining arguments supplied to this call. The formatting is performed as if the message string and remaining arguments were passed to the `udi_snprintf` utility function (see **udi_snprintf** on page 20-11).

... are the remaining arguments which provide the values used for the formatting codes contained in the message identified by ***msgnum***. Arguments formatted with %c or %s format codes must not contain newline ('`\n`') or other control characters.

DESCRIPTION

This routine traces data generated by the driver. Time-stamping of trace entries will be done at the discretion of the environment; the driver is not expected to supply timestamp information.

To simplify usage, `udi_trace_write` does not involve a callback. The environment will immediately copy the ***msgnum*** and remaining arguments into its own buffers for immediate or delayed processing. This may result in loss of trace data during unusually heavy usage. The driver writer is encouraged to keep trace entries short to minimize this possibility.

If the driver wishes to log and trace the same event, the `udi_log_write` operation should be used instead.

REFERENCES `udi_log_write`

NAME **udi_log_write** *Record log data*

SYNOPSIS

```
#include <udi.h>

void udi_log_write (
     udi_log_write_call_t *callback,
     udi_cb_t *gcb,
     udi_trevent_t trace_event,
     udi_ubit8_t severity,
     udi_index_t meta_idx,
     udi_status_t original_status,
     udi_ubit32_t msgnum,
     ... );

typedef void udi_log_write_call_t (
     udi_cb_t *gcb,
     udi_status_t correlated_status );

/* Values for severity */
#define UDI_LOG_DISASTER                  1
#define UDI_LOG_ERROR                     2
#define UDI_LOG_WARNING                   3
#define UDI_LOG_INFORMATION               4
```

ARGUMENTS

callback, ***gcb*** are standard arguments described in the "Asynchronous Service Calls" section of *"Calling Sequence and Naming Conventions"*. Note that no init context pointer is required (unlike with `udi_trace_write`), since region identity is established through the ***gcb***.

trace_event is the type of trace event to be logged. For log data that is not associated with trace events, use `UDI_TREVENT_LOG`.

severity specifies the severity level of the log data.

meta_idx is a metalanguage index number that identifies the metalanguage to which ***trace_event*** is relative, for metalanguage-selectable trace events. It must match the value of <meta_idx> in the corresponding "child_meta", "parent_meta", or "internal_meta" declaration of the driver's Static Driver Properties (see Chapter 30), or 0 for the Management Metalanguage. If the event is not metalanguage-selectable, ***meta_idx*** is ignored.

original_status is the UDI status value, if any, that was either generated by the driver or received from another driver. The environment will generate appropriate information in the log file for this status value; the driver may provided supplemental information with the ***msgnum*** and associated arguments.

msgnum is the index value of a message string provided in the driver's static properties file (see Section 30.4.9, "Message Declaration," on page 30-7). This selects the text of the message to be logged.

Any embedded formatting codes in the text of that message will be used to format the traced message with the remaining arguments supplied to this call. The formatting is performed as if the message string and remaining arguments were passed to the `udi_snprintf` utility function (see **udi_snprintf** on page 20-11).

... are the remaining arguments which provide the values used for the formatting codes contained in the message identified by ***msgnum***. Arguments formatted with %c or %s format codes must not contain newline (‘`\n`’) or other control characters.

correlated_status is the ***original_status*** value, possibly modified to include a new correlation value. (See the *Fundamental Types* Chapter for more information on the correlation field of the `udi_status_t` type.) The correlation value allows multiple log entries related to a single event to be correlated based on the correlation value assigned; if there is already a correlation value in the status code the `udi_log_write` call will preserve that original correlation.

WARNINGS Control block usage must follow the rules described in the “Asynchronous Service Calls” section of *“Calling Sequence and Naming Conventions”*.

DESCRIPTION This routine logs events that can affect functionality of the driver, controlled hardware or other subsystems using the driver. Each of the data records may be automatically time stamped by the environment; the driver is not expected to supply timestamp information.

The following severity levels are defined:

UDI_LOG_DISASTER This severity indicates that the driver detected a severe and unrecoverable error condition that will likely affect multiple users of a driver and may jeopardize system integrity. Environments may take actions that result in killing the driver or the system upon logging this severity.

UDI_LOG_ERROR This severity indicates that the driver encountered an error condition that might cause some error conditions in its users, but from which it was able to recover.

UDI_LOG_WARNING This severity indicates minor abnormal conditions, likely caused by other subsystems.

UDI_LOG_INFORMATION This severity is used for expected events such as driver start-up or shutdown.

If the ***trace_event*** is not `UDI_TREVENT_LOG`, an implicit call to `udi_trace_write` will be made if tracing for the corresponding event type is enabled.

udi_log_write allows the environment to associate related events in different drivers with each other. It can do this by modifying the status codes it is passed to include a correlation value. This allows errors related to the same event to be correlated.

REFERENCES

udi_trace_write

Debugging Services 18

18.1 Overview

This chapter defines several functions that can be used to help debug and verify UDI drivers and perform internal consistency checking. Contrary to conventions common in many legacy driver models, UDI does not allow a driver to directly invoke a system abort or reset; the UDI environment has the capability, if it desires, to detect a malfunctioning driver and kill or cease to use that driver without affecting the integrity of the rest of the system.

18.2 Debugging Service Calls

In this section, UDI defines one function for debugging (`udi_debug_break`) and one function for internal consistency checking (`udi_assert`). These represent the limit of the driver's ability to explicitly cause system-level exceptions, and the handling of these exceptions is dependent on the implementation and current execution mode of the environment under which the driver is running. It is still possible for the driver to perform architectural code violations (e.g. dereference a null pointer) but it is legitimate for the environment to intercept these violations and handle them by killing the driver rather than allowing a system crash as conventionally occurs.

NAME **udi_assert** *Perform driver internal consistency check*

SYNOPSIS

```
#include <udi.h>

void udi_assert ( udi_boolean_t expr );
```

ARGUMENTS

expr expression to evaluate for truthfulness

DESCRIPTION

The `udi_assert` function is used by the driver to perform an internal consistency check. The supplied expression ***expr*** is evaluated and if the result is false, the consistency check is interpreted as having failed. A failed consistency check indicates an unrecoverable condition within the driver and the UDI environment should take steps to kill the driver or mark it as not-executable. A failed assertion is tantamount to a suicide request on the part of the driver but not for the system as a whole.

The actual handling of an assertion failure is left to the environment implementation. It may be that a particular environment even has multiple execution modes (*e.g.* free *vs.* checked) where the failed assertions have different results depending on the mode.

While it is not actually guaranteed that `udi_assert` will not return to the driver if ***expr*** is false, it is expected that drivers will be coded as if that were the case.

As an esoteric note, an environment may choose not to directly handle the `udi_assert` call simply by returning to the calling code regardless of the success or failure of the evaluated expression. Although the results are indeterminate (and it is likely that the system will subsequently crash as a result of an ignored assertion) the environment implementation has chosen this as a valid outcome of a failed assertion. This is a very subtle environment implementation issue that should not affect driver code; as noted above, driver writers should write their code under the assumption that a failed `udi_assert` call will not return.

NAME **udi_debug_break** *Request a debug breakpoint at the current location*

SYNOPSIS

```
#include <udi.h>

void udi_debug_break (
     udi_init_context_t *init_context,
     const char *message );
```

ARGUMENTS

init_context is the initial context supplied to the driver on the primary region's management channel.

message is a string used to indicate the cause of the debug break.

DESCRIPTION

The `udi_debug_break` function is used for driver debugging purposes. In a debug configuration, calling this routine indicates that a system debugger, if present and available, should be entered at the current time for developer debugging operations.

The ***init_context*** argument is used to identify which driver region is issuing the breakpoint. This allows environments to selectively set breakpoints for specific regions as identified by their ***init_context*** values..

The implementation of this function is environment dependent and the actions taken may be defined by an operational mode of that UDI environment.

EXAMPLE

An example implementation of the `udi_debug_break` utility might distinguish between a debug and a non-debug environment, where the former is identified for driver development and the latter is typically the production environment.

In debugging mode, the ***message*** string would be output to the debug console and the debugger is entered in the context of the thread that called this function. The operator can then perform various debugging operations and then resume normal execution, which will cause this function to return to the caller for continued execution of UDI driver code.

In a non-debugging mode, the environment may completely ignore this request and simply return immediately to the UDI driver code.

NAME **udi_debug_printf** *Output a debugging message*

SYNOPSIS

```
#include <udi.h>

void udi_debug_printf (
     const char *format,
     ... );
```

ARGUMENTS

format is the format string, which controls the formatting of the output string, as described for **udi_snprintf** on page 20-11.

... are the remaining arguments, which provide the values used for the formatting codes.

DESCRIPTION

The `udi_debug_printf` function is intended for use in driver debugging, as a simplified alternative to the pair of `udi_snprintf` and `udi_trace_write`, in cases where the output is not needed in production environments. It is expected that `udi_debug_printf` calls would typically not appear in a (compiled) production driver.

Where required by this or other UDI specifications to trace or log events, drivers must use `udi_trace_write` or `udi_log_write` instead of `udi_debug_printf`, since `udi_debug_printf` may be a no-op in some environments. Use of `udi_debug_printf` may impair driver or system performance.

Environments may choose to ignore any or all calls to `udi_debug_printf`. Some environments may have different operational modes (e.g. debug mode vs. non-debug mode) that treat `udi_debug_printf` differently. Environments intended to facilitate driver debugging should include at least an option to enable output from `udi_debug_printf` calls. All environments must at least provide the `udi_debug_printf` function, even if it does nothing.

Output from `udi_debug_printf`, if any, will be sent to an environment implementation-defined device, file, or application. Newline (‘\n’) characters in the format string or any string or character arguments will be translated to an appropriate end of line character(s); other control characters must not be used. Output from each call to `udi_debug_printf` may be truncated to 99 bytes of text. Note that if the output is truncated, any terminating newline character may have been discarded.

Index