



Uniform Driver Interface

UDI Core Specification

Version 0.90

Note – This version (Version 0.90) is intended to be functionally complete. It is presented to the industry at large for broad public review. Upon completion of a two-month review period, review comments will be incorporated to form the final 1.0 specification. Review comments may be submitted to <http://www.sco.com/UDI/review.html>.

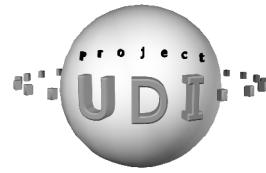


Table Of Contents

List of Reference Pages by Chapter	xi
Alphabetical List of Reference Pages	xvii
Abstract.....	xxi

Section 1: Overview

1 Introductory Material	1-1
1.1 Introduction	1-1
1.2 Scope	1-1
1.3 Normative References	1-1
1.4 Conformance	1-2
1.4.1 Environment Conformance	1-2
1.4.2 Device Driver Conformance	1-2
2 Document Organization	2-1
2.1 Overview of UDI Documentation	2-1
2.2 Overview of the UDI Core Specification	2-2
2.2.1 Core Specification Sections	2-2
2.2.2 Core Specification Topics	2-2
3 Terminology	3-1
3.1 Introduction	3-1
3.2 Definitions	3-1
3.2.1 Directive Terms	3-1
3.2.2 Basic Terms	3-2
3.3 Acronyms	3-5

Table of Contents

Section 2: Architecture

4 Execution Model.....	4-1
4.1 Introduction	4-1
4.2 Driver Object Modules	4-1
4.3 Driver Instances	4-1
4.4 Regions	4-1
4.4.1 Driver Partitioning	4-2
4.5 Multi-Module Drivers	4-2
4.5.1 Relationship of Regions to Modules	4-2
4.5.2 Primary Module Requirements	4-2
4.6 Channels	4-2
4.7 Driver Execution Environments	4-3
4.7.1 Non-Blocking Model	4-3
4.8 Function Call Classifications	4-4
4.9 Module-Global Entry Points	4-4
4.10 Module-Global Service Calls	4-5
4.10.1 Module-Global Initialization Functions	4-5
4.10.2 Debugging Services	4-5
4.11 Region-Context Service Calls	4-5
4.11.1 Synchronous Service Calls	4-5
4.11.2 Asynchronous Service Calls	4-5
4.12 Channel Operations	4-6
4.13 Location Independence	4-6
4.14 Driver Faults/Recovery	4-6
4.15 Metalanguage Model	4-7
4.15.1 Metalanguage Roles	4-7
5 Data Model.....	5-1
5.1 Overview	5-1
5.2 Data Objects	5-2
5.2.1 Memory Objects	5-2
5.2.2 Control Blocks	5-2
5.2.2.1 Scratch Space	5-2
5.2.2.2 Inline Data	5-3
5.2.2.3 Control Block Synchronization	5-3
5.2.2.4 Control Block Groups	5-3
5.2.3 Region Data	5-3
5.3 Channel Context	5-3
5.4 Transferable Objects	5-3
5.5 Implicit MP Synchronization	5-3

Table of Contents

6 Standard Calling Sequences	6-1
6.1 Overview	6-1
6.2 Module-Global Entry Points	6-1
6.3 Module-Global Initialization Functions	6-2
6.3.1 Channel Ops Properties Initialization	6-2
6.3.1.1 Channel Operations Vectors	6-2
6.3.2 Control Block Properties Initialization	6-3
6.4 Channel Operations	6-4
6.4.1 Channel Operation Invocations	6-4
6.4.2 Channel Operation Entry Points	6-4
6.5 Asynchronous Service Calls	6-6
6.5.1 Asynchronous Service Call Invocations	6-6
6.5.2 Associated Callback Functions	6-6
6.5.3 Control Block Type Conversion	6-7

Section 3: Core Services

7 General Requirements.....	7-1
7.1 UDI_VERSION	7-1
7.2 Header Files	7-1
7.3 C Language Requirements	7-1
8 Fundamental Types	8-1
8.1 Overview	8-1
8.2 Usage of Standard ISO C Data Types	8-2
8.2.1 ISO C char Type	8-2
8.2.2 ISO C void Type	8-2
8.2.2.1 Null Pointers	8-2
8.2.3 Varargs Types	8-2
8.3 Notation for Implementation-Dependent Types and Constants	8-3
8.4 Specific-Length Types	8-3
8.5 Abstract Types	8-5
8.5.1 Size Type	8-5
8.5.2 Index Type	8-5
8.5.2.1 Control Block Index	8-5
8.5.2.2 Metalanguage Index	8-6
8.5.2.3 Ops Index	8-6
8.5.2.4 Region Index	8-6
8.6 Opaque Types	8-7
8.6.1 Opaque Handles	8-7
8.6.1.1 Buffer Handle	8-8
8.6.1.2 Channel Handle	8-8

Table of Contents

8.6.1.3	Constraints Handle	8-8
8.6.2	Self-Contained Opaque Types	8-9
8.6.2.1	Timestamp Type	8-9
8.7	Semi-Opaque Types	8-9
8.8	Common Derived Types	8-10
8.8.1	UDI Status	8-10
8.8.1.1	Common Status Codes	8-11
8.8.2	Data Layout Specifier	8-14
8.9	Structures Requiring a Fixed Binary Representation	8-18
9	Initialization.....	9-1
9.1	Overview	9-1
9.1.1	Per-Driver Initialization	9-1
9.1.2	Per-Instance Initialization	9-2
9.2	Entry Points	9-3
9.3	Service Calls	9-5
9.4	Initial Region Data Structures	9-13
10	Control Block Management	10-1
10.1	Overview	10-1
10.2	Control Block Service Calls and Macros	10-2
11	Memory Management.....	11-1
11.1	Overview	11-1
11.2	Memory Management Service Calls	11-2
12	Constraints Management	12-1
12.1	Overview	12-1
12.1.1	Constraints Attributes	12-1
12.2	Constraints Attributes Service Calls and Structures	12-3
12.3	Constraints Propagation Service Calls	12-13
13	Buffer Management	13-1
13.1	Overview	13-1
13.2	Buffer Constraints	13-2
13.3	Buffer Management Macros	13-5
13.4	Buffer Management Service Calls	13-10
13.4.1	Buffer Usage Models	13-10
13.4.2	Buffer Recovery Mechanism	13-11
13.5	Buffer Tags	13-20
13.5.1	Buffer Tag Categories	13-20
13.5.2	Buffer Tag Utilities	13-29

Table of Contents

14 Time Management	14-1
14.1 Timer Services	14-2
14.1.1 Timed Delays	14-2
14.1.2 Timer Context	14-2
14.2 Timestamp Services	14-6
15 Instance Attribute Management.....	15-1
15.1 Overview	15-1
15.2 Instance Attribute Names	15-1
15.3 Persistence of Attributes	15-1
15.4 Classes of Attributes	15-1
15.4.1 Instance-Private Attributes	15-2
15.4.2 Enumeration Attributes	15-2
15.4.2.1 Locator Enumeration Attribute	15-2
15.4.2.1.1 Locator Enumeration Attribute Example	15-2
15.4.3 Sibling Group Attributes	15-3
15.4.4 Parent Visible Attributes	15-4
15.4.5 Message String Attributes	15-4
15.4.6 Attribute Classification	15-4
15.5 Instance Attribute Services	15-5
16 Inter-Module Communication.....	16-1
16.1 Overview	16-1
16.2 Service Calls	16-1
17 Tracing and Logging	17-1
17.1 Overview	17-1
17.2 Tracing and Logging Service Calls	17-1
17.2.1 Tracing Calls	17-1
17.2.2 Logging Calls	17-1
17.2.3 Trace Event Types	17-2
17.3 Tracing and Logging Entry Points	17-9
17.3.1 Formatting of Trace and Log Entries	17-9
18 Debugging Services	18-1
18.1 Overview	18-1
18.2 Debugging Service Calls	18-2
19 Utility Functions.....	19-1
19.1 Overview	19-1
19.2 String/Memory Functions	19-1
19.3 Output Formatting Functions	19-9

Table of Contents

19.4 Queue Management	19-14
19.4.1 Queue Element Structure	19-14
19.4.2 Queuing Functions	19-16
19.4.3 Queuing Macros	19-19
19.5 Endian Management	19-26
19.5.1 Rules for C Structure Definitions	19-26
19.5.1.1 Byte-by-byte structure layout	19-27
19.5.2 Helper Macros	19-29
19.5.2.1 Bit-field Macros	19-29
19.5.2.2 Multi-Byte Macros	19-29
19.5.3 Endian-Swapping Utilities	19-31

Section 4: Core Metalanguages

20 Introduction to UDI Metalanguages.....	20-1
20.1 Overview	20-1
20.2 Standard Metalanguage Functions and Parameters	20-1
20.3 Channel Operation Suffixes	20-2
20.4 General Rules for Handling Channel Operations	20-3
20.4.1 Normal Operation Handling	20-3
20.4.2 Not Understood Operations	20-3
20.4.3 Unsupported Operations	20-3
20.4.4 Invalid State Operations	20-3
20.5 Channel Event Indication Operation	20-4
21 Management Metalanguage	21-1
21.1 Overview	21-1
21.2 Management Agent	21-1
21.2.1 Driver Instantiation	21-2
21.3 Management Metalanguage Considerations	21-3
21.4 Initialization	21-5
21.4.1 Tracing Control Operations	21-5
21.4.2 Resource Management	21-5
21.5 Binding Operations	21-15
21.6 Unbinding Operations	21-22
21.7 Enumeration Operations	21-27
21.7.1 Enumeration Attributes	21-27
21.7.2 Child Context	21-27
21.7.3 Enumeration Filters	21-27
21.7.4 Parent Context	21-28
21.7.5 Unenumeration	21-28
21.7.6 Notification of Topology Changes	21-28

Table of Contents

21.7.7	Directed Enumeration	21-29
21.8	Device Management Operations	21-42
21.8.1	Prepare To Suspend	21-42
21.8.2	Suspend	21-43
21.8.3	Shutdown	21-43
21.8.4	Parent Suspended	21-44
21.8.5	Resume	21-44
21.9	Management Metalanguage States	21-49
21.9.1	Management Metalanguage States	21-51
21.9.1.1	Operational Sub-States	21-51
22	Internal Management Metalanguage.....	22-1
22.1	Overview	22-1
22.1.1	Roles	22-1
22.2	Initialization	22-2
22.3	Interface Operations	22-3
23	Generic I/O Metalanguage.....	23-1
23.1	Overview	23-1
23.1.1	Roles	23-1
23.2	Generic I/O Initialization	23-2
23.3	Generic I/O Operations and Control Blocks	23-11
24	Diagnostics Support.....	24-1
24.1	Diagnostics State	24-1
Section 5: MEI Services		
25	Introduction to MEI	25-1
25.1	Overview	25-1
25.2	Metalanguage Design Rules	25-1
25.3	Data Model	25-1
25.4	Execution Model	25-1
26	Metalanguage-to-Environment Interface	26-1
26.1	Overview	26-1
26.2	Structures	26-2
26.3	Marshalling	26-4
26.4	Initialization Interfaces	26-5
26.5	Front-End Stub Interfaces	26-6
26.6	MEI Stub Implementation Macros	26-8

Table of Contents

Section 6: Packaging and Distribution

27 Introduction to Packaging and Distribution.....	27-1
27.1 Introduction	27-1
28 Static Driver Properties	28-1
28.1 Overview	28-1
28.1.1 UDI Modules	28-1
28.2 Basic Syntax	28-3
28.3 Property Declaration Syntax	28-4
28.4 Common Property Declarations	28-5
28.5 Property Declarations for Libraries	28-9
28.6 Property Declarations for Drivers	28-11
28.7 Sample Static Driver Properties File	28-21
29 Packaging & Distribution Format.....	29-1
29.1 Overview	29-1
29.2 Packaging Format	29-1
29.2.1 Directory Structure	29-1
29.3 Distribution Format	29-2
29.3.1 Floppy Storage Format	29-2
29.3.2 CD-ROM Storage Format	29-2
30 Build & Packaging Utilities	30-1
30.1 Overview	30-1
30.2 The udimkpkg Utility	30-1

Section 7: ABI Bindings

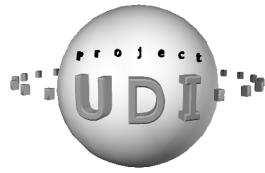
31 Introduction to ABI Bindings	31-1
31.1 Introduction	31-1
31.2 Processor Architecture	31-1
31.3 Runtime Architecture	31-1
31.4 API-Level Bindings	31-2
31.4.1 Sizes of UDI Data Types	31-2
31.4.2 Implementation-Dependent Macros	31-3
31.5 Building the Driver Object	31-3
31.5.1 Object File Format	31-3
31.5.2 Static Driver Properties Encapsulation	31-3

Table of Contents

Section 8: Appendices

A Glossary.....	A-1
Index.....	X-1

Table of Contents



List of Reference Pages by Chapter

Chapter 9 Initialization

init_module -----	<i>Module initialization entry point routine</i>	9-4
udi_cb_select -----	<i>Select control block properties to use with incoming channel ops</i>	9-6
udi_gcb_init -----	<i>Initialize generic control block properties.....</i>	9-7
udi_primary_region_init -----	<i>Initialize primary region properties.....</i>	9-8
udi_secondary_region_init -----	<i>Initialize secondary region properties</i>	9-10
udi_secondary_region_prepare -----	<i>Prepare for secondary region creation.....</i>	9-12
udi_init_context_t -----	<i>Initial context for new regions</i>	9-14
udi_limits_t -----	<i>Platform-specific allocation and access limits</i>	9-15

Chapter 10 Control Block Management

udi_cb_t -----	<i>Generic, least-common-denominator control block ..</i>	10-3
udi_cb_alloc -----	<i>Allocate a new control block.....</i>	10-4
udi_cb_free -----	<i>Deallocates a previously obtained control block.....</i>	10-6
UDI_GCB -----	<i>Convert any control block to udi_cb_t representation</i>	10-7
UDI_MCB -----	<i>Convert a generic control block to a specific one</i>	10-8
udi_cancel -----	<i>Cancel a pending allocation request</i>	10-9

Chapter 11 Memory Management

udi_mem_alloc -----	<i>Allocate memory for a virtually-contiguous object....</i>	11-3
udi_mem_free -----	<i>Free memory object</i>	11-5

Chapter 12 Constraints Management

udi_constraints_attr_t -----	<i>Constraints attribute type.....</i>	12-4
udi_constraints_attr_spec_t -----	<i>Specify attribute/value pair</i>	12-6
udi_constraints_attr_set -----	<i>Set constraints attributes.....</i>	12-7
udi_constraints_attr_reset -----	<i>Reset a constraints attribute to default</i>	12-9
udi_constraints_attr_combine -----	<i>Merge constraints attributes objects</i>	12-10
udi_constraints_free -----	<i>Free a constraints object.....</i>	12-12
udi_constraints_propagate -----	<i>Constraints propagation request</i>	12-14

Chapter 13 Buffer Management

udi_constraints_attr_t (Buffer) - -	<i>Buffer constraints attributes.....</i>	13-3
--	---	------

List of Reference Pages by Chapter

UDI_BUF_ALLOC -----	<i>Allocate and initialize a new buffer</i>	13-6
UDI_BUF_INSERT -----	<i>Insert bytes into a logical buffer</i>	13-7
UDI_BUF_DELETE -----	<i>Delete bytes from a logical buffer</i>	13-8
UDI_BUF_DUP -----	<i>Copy a logical buffer in its entirety.....</i>	13-9
udi_buf_copy -----	<i>Copy data from one logical buffer to another.....</i>	13-12
udi_buf_read -----	<i>Read data bytes from a logical buffer.....</i>	13-15
udi_buf_write -----	<i>Write data bytes into a logical buffer.....</i>	13-16
udi_buf_free -----	<i>Free a logical buffer.....</i>	13-18
udi_buf_get_size -----	<i>Get the valid data size of a UDI buffer</i>	13-19
udi_tagtype_t -----	<i>Buffer tag type</i>	13-21
udi_buf_tag_t -----	<i>Buffer tag structure</i>	13-25
udi_buf_tag_set -----	<i>Sets a tag for a portion of buffer data</i>	13-27
udi_buf_tag_get -----	<i>Gets one or more tags from a buffer</i>	13-28
udi_buf_tag_compute -----	<i>Compute values from tagged buffer data</i>	13-30
udi_buf_tag_apply -----	<i>Apply modifications to tagged buffer data</i>	13-31

Chapter 14 Time Management

udi_time_t -----	<i>Time value structure.....</i>	14-3
udi_timer_start -----	<i>Start a callback timer</i>	14-4
udi_timer_start_repeating -----	<i>Start a repeating timer</i>	14-5
udi_time_current -----	<i>Return indication of the current relative time</i>	14-7
udi_time_between -----	<i>Return time interval between two points</i>	14-8
udi_time_since -----	<i>Return time interval since a starting point.....</i>	14-9

Chapter 15 Instance Attribute Management

udi_instance_attr_type_t -----	<i>Instance attribute data-type type</i>	15-6
udi_instance_attr_get -----	<i>Read an attribute value for a driver instance</i>	15-7
udi_instance_attr_set -----	<i>Set a driver instance attribute value</i>	15-9
UDI_INSTANCE_ATTR_DELETE -----	<i>Driver instance attribute delete macro</i>	15-11

Chapter 16 Inter-Module Communication

udi_secondary_region_create -----	<i>Create a secondary region and its first channel</i>	16-2
udi_channel_anchor -----	<i>Anchor a channel to the current region</i>	16-4
udi_channel_spawn -----	<i>Spawn a new channel with loose ends.</i>	16-6
udi_channel_set_context -----	<i>Attach a new context to a channel endpoint.....</i>	16-8
udi_channel_close -----	<i>Close a channel</i>	16-9

Chapter 17 Tracing and Logging

udi_trevent_t -----	<i>Trace event type definition</i>	17-3
udi_trace_write -----	<i>Record trace data</i>	17-5
udi_log_write -----	<i>Record log data</i>	17-6
ddd_trace_log_format -----	<i>Format log or trace data to printable form</i>	17-10
UDI_HANDLE_ID -----	<i>Get identification value for specified handle</i>	17-12

List of Reference Pages by Chapter

Chapter 18 Debugging Services

udi_assert -----	<i>Perform driver internal consistency check</i>	18-3
udi_debug_break -----	<i>Request a debug breakpoint at the current location ..</i>	18-4

Chapter 19 Utility Functions

udi_strcat, udi_strncat -----	<i>String concatenation</i>	19-2
udi_strcmp, udi_strncmp,		
udi_memcmp -----	<i>String/memory comparison</i>	19-3
udi strcpy, udi_strncpy,		
udi_memcpy, udi_memmove ---	<i>String/memory copy</i>	19-4
udi_strchr,		
udi strrchr,		
udi_memchr -----	<i>String/memory searching</i>	19-5
udi strtou32 -----	<i>Convert string to unsigned 32-bit value.....</i>	19-6
udi_strlen -----	<i>Determine string length</i>	19-7
udi_memset -----	<i>Memory initialization</i>	19-8
udi_snprintf -----	<i>Format printable string.....</i>	19-10
udi_vsnprintf -----	<i>Format printable string with varargs</i>	19-13
udi_queue_t -----	<i>Queue element structure</i>	19-15
udi_enqueue -----	<i>Insert a queue element into a queue.....</i>	19-17
udi_dequeue -----	<i>Dequeue a queue element</i>	19-18
UDI_QUEUE_INIT,		
UDI_QUEUE_EMPTY -----	<i>Initialize queue; check if it's empty.....</i>	19-20
UDI_ENQUEUE_XXX,		
UDI_QUEUE_INSERT_XXX -----	<i>Insert an element into a queue</i>	19-21
UDI_DEQUEUE_XXX,		
UDI_QUEUE_REMOVE -----	<i>Remove an element from a queue</i>	19-23
UDI_FIRST/ LAST/		
NEXT/ PREV_ELEMENT -----	<i>Get first/last/next/previous element in queue</i>	19-24
UDI_QUEUE_FOREACH -----	<i>Safe mechanism to walk a queue.....</i>	19-25
UDI_ENDIAN_SWAP16/32 -----	<i>Byte-swap 16 or 32-bit integers</i>	19-32
udi_endian_swap -----	<i>Byte-swap data in place</i>	19-33
udi_endian_swap_copy -----	<i>Byte-swap data and copy the results</i>	19-34
UDI_ENDIAN_SWAP_ARRAY -----	<i>Byte-swap each element in an array</i>	19-35

Chapter 20 Introduction to UDI Metalanguages

udi_channel_event_cb_t -----	<i>Control block for channel_event_ind.....</i>	20-5
udi_channel_event_ind -----	<i>Channel event notification (env-to-driver)</i>	20-7

Chapter 21 Management Metalinguage

udi_mgmt_ops_t -----	<i>Management Meta channel ops vector</i>	21-6
udi_mgmt_cb_t -----	<i>Common Management Control Block.....</i>	21-7
udi_usage_cb_t -----	<i>Resource indication and trace level control block....</i>	21-8
udi_usage_ind -----	<i>Indicate desired resource usage and trace levels</i>	21-9
udi_usage_res -----	<i>Resource usage and trace level response operation</i>	21-11

List of Reference Pages by Chapter

udi_region_attach_cb_t	-----	<i>Control block for region attachments</i>	21-12
udi_region_attach_ind	-----	<i>Secondary region attachment event indication.....</i>	21-13
udi_region_attach_res	-----	<i>Secondary region attach event response</i>	21-14
udi_bind_cb_t	-----	<i>Prep_for_child/bind_to_parent control block</i>	21-16
udi_prep_for_child_req	-----	<i>Prepare for a child bind request</i>	21-17
udi_prep_for_child_ack	-----	<i>Acknowledge prep_for_child operation</i>	21-18
udi_bind_to_parent_req	-----	<i>Request to bind to a parent driver.....</i>	21-19
udi_bind_to_parent_ack	-----	<i>Acknowledge completion of parent/child binding....</i>	21-21
udi_unbind_parent_req	-----	<i>Request driver to unbind from h/w or parent.....</i>	21-23
udi_unbind_parent_ack	-----	<i>Acknowledge completion of an unbind</i>	21-24
udi_final_cleanup_req	-----	<i>Request driver to release all resources and context prior to instance unload.</i>	21-25
udi_final_cleanup_ack	-----	<i>Acknowledge completion of a final cleanup request</i>	21-26
udi_instance_attr_list_t	-----	<i>Enumeration instance attribute list.....</i>	21-30
udi_filter_element_t	-----	<i>Enumeration filter element structure</i>	21-31
udi_enumerate_cb_t	-----	<i>Enumeration operation control block</i>	21-33
udi_enumerate_req	-----	<i>Request information regarding a child instance</i>	21-34
udi_enumerate_ack	-----	<i>Provide child instance information</i>	21-37
udi_child_release_ind	-----	<i>Request to release child context</i>	21-40
udi_child_release_res	-----	<i>Acknowledge child release request.</i>	21-41
udi_devmgmt_req	-----	<i>Device Management request</i>	21-45
udi_devmgmt_ack	-----	<i>Acknowledge a device management request</i>	21-47

Chapter 22 Internal Management Metalanguage

Chapter 23 Generic I/O Metalanguage

udi_gio_provider_ops_init	-----	<i>Register client-to-provider ops</i>	23-3
udi_gio_client_ops_init	-----	<i>Register provider-to-client ops</i>	23-5
udi_gio_bind_cb_init	-----	<i>Register GIO bind control block properties.....</i>	23-7
udi_gio_xfer_cb_init	-----	<i>Register GIO transfer control block properties</i>	23-8
udi_gio_abort_cb_init	-----	<i>Register GIO abort control block properties</i>	23-9
udi_gio_event_cb_init	-----	<i>Register GIO event control block properties</i>	23-10
udi_gio_bind_cb_t	-----	<i>Control block for GIO binding operations.....</i>	23-12
udi_gio_bind_req	-----	<i>Request a binding to a GIO provider driver.....</i>	23-13
udi_gio_bind_ack	-----	<i>Acknowledge a GIO binding</i>	23-14
udi_gio_unbind_req	-----	<i>Request to unbind from a GIO provider driver.....</i>	23-15
udi_gio_unbind_ack	-----	<i>Acknowledge a GIO unbind request</i>	23-16
udi_gio_xfer_cb_t	-----	<i>Control block for GIO transfer operations</i>	23-17
udi_gio_op_t	-----	<i>GIO operation type</i>	23-19
udi_gio_rw_params_t	-----	<i>Parameters for standard GIO read/write ops</i>	23-20
udi_gio_xfer_req	-----	<i>Request a Generic I/O transfer</i>	23-21
udi_gio_xfer_ack	-----	<i>Acknowledge a GIO transfer request</i>	23-22
udi_gio_xfer_nak	-----	<i>Abnormal completion of a GIO transfer request</i>	23-23
udi_gio_abort_cb_t	-----	<i>Control block for GIO abort operations</i>	23-24
udi_gio_abort_req	-----	<i>Abort a pending transfer request</i>	23-25
udi_gio_abort_ack	-----	<i>Acknowledge a GIO abort operation</i>	23-26

List of Reference Pages by Chapter

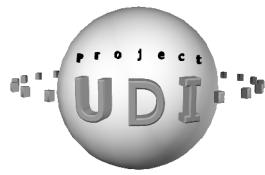
udi_gio_event_cb_t -----	<i>Control block for GIO event operations</i>	23-27
udi_gio_event_ind -----	<i>GIO event indication</i>	23-28
udi_gio_event_res -----	<i>GIO event response</i>	23-29

Chapter 24 Diagnostics Support

udi_gio_op_t -----	<i>Diagnostics control operations</i>	24-3
udi_gio_diag_params_t -----	<i>Parameters for standard GIO diagnostic ops</i>	24-5

Chapter 26 Metalanguage-to-Environment Interface

List of Reference Pages by Chapter



Alphabetical List of Reference Pages

ddd_trace_log_format	17-10
init_module	9-4
udi_assert	18-3
udi_bind_cb_t	21-16
udi_bind_to_parent_ack	21-21
udi_bind_to_parent_req	21-19
UDI_BUF_ALLOC	13-6
udi_buf_copy	13-12
UDI_BUF_DELETE	13-8
UDI_BUF_DUP	13-9
udi_buf_free	13-18
udi_buf_get_size	13-19
UDI_BUF_INSERT	13-7
udi_buf_read	13-15
udi_buf_tag_apply	13-31
udi_buf_tag_compute	13-30
udi_buf_tag_get	13-28
udi_buf_tag_set	13-27
udi_buf_tag_t	13-25
udi_buf_write	13-16
udi_cancel	10-9
udi_cb_alloc	10-4
udi_cb_free	10-6
udi_cb_select	9-6
udi_cb_t	10-3
udi_channel_anchor	16-4
udi_channel_close	16-9
udi_channel_event_cb_t	20-5
udi_channel_event_ind	20-7
udi_channel_set_context	16-8
udi_channel_spawn	16-6
udi_child_release_ind	21-40
udi_child_release_res	21-41
udi_constraints_attr_combine	12-10
udi_constraints_attr_reset	12-9
udi_constraints_attr_set	12-7
udi_constraints_attr_spec_t	12-6
udi_constraints_attr_t	12-4
udi_constraints_attr_t (Buffer)	13-3

Alphabetical List of Reference Pages

udi_constraints_free	12-12
udi_constraints_propagate	12-14
udi_debug_break	18-4
udi_dequeue	19-18
UDI_DEQUEUE_XXX,	
UDI_QUEUE_REMOVE	19-23
udi_devmgmt_ack	21-47
udi_devmgmt_req	21-45
udi_endian_swap	19-33
UDI_ENDIAN_SWAP_ARRAY	19-35
udi_endian_swap_copy	19-34
UDI_ENDIAN_SWAP16/32	19-32
udi_enqueue	19-17
UDI_ENQUEUE_XXX,	
UDI_QUEUE_INSERT_XXX	19-21
udi_enumerate_ack	21-37
udi_enumerate_cb_t	21-33
udi_enumerate_req	21-34
udi_filter_element_t	21-31
udi_final_cleanup_ack	21-26
udi_final_cleanup_req	21-25
UDI_FIRST/ LAST/	
NEXT/ PREV_ELEMENT	19-24
UDI_GCB	10-7
udi_gcb_init	9-7
udi_gio_abort_ack	23-26
udi_gio_abort_cb_init	23-9
udi_gio_abort_cb_t	23-24
udi_gio_abort_req	23-25
udi_gio_bind_ack	23-14
udi_gio_bind_cb_init	23-7
udi_gio_bind_cb_t	23-12
udi_gio_bind_req	23-13
udi_gio_client_ops_init	23-5
udi_gio_diag_params_t	24-5
udi_gio_event_cb_init	23-10
udi_gio_event_cb_t	23-27
udi_gio_event_ind	23-28
udi_gio_event_res	23-29
udi_gio_op_t	23-19
udi_gio_op_t	24-3
udi_gio_provider_ops_init	23-3
udi_gio_rw_params_t	23-20
udi_gio_unbind_ack	23-16
udi_gio_unbind_req	23-15
udi_gio_xfer_ack	23-22
udi_gio_xfer_cb_init	23-8

Alphabetical List of Reference Pages

udi_gio_xfer_cb_t	23-17
udi_gio_xfer_nak	23-23
udi_gio_xfer_req	23-21
UDI_HANDLE_ID	17-12
udi_init_context_t	9-14
UDI_INSTANCE_ATTR_DELETE	15-11
udi_instance_attr_get	15-7
udi_instance_attr_list_t	21-30
udi_instance_attr_set	15-9
udi_instance_attr_type_t	15-6
udi_limits_t	9-15
udi_log_write	17-6
UDI_MCB	10-8
udi_mem_alloc	11-3
udi_mem_free	11-5
udi_memset	19-8
udi_mgmt_cb_t	21-7
udi_mgmt_ops_t	21-6
udi_prep_for_child_ack	21-18
udi_prep_for_child_req	21-17
udi_primary_region_init	9-8
UDI_QUEUE_FOREACH	19-25
UDI_QUEUE_INIT, UDI_QUEUE_EMPTY	19-20
udi_queue_t	19-15
udi_region_attach_cb_t	21-12
udi_region_attach_ind	21-13
udi_region_attach_res	21-14
udi_secondary_region_create	16-2
udi_secondary_region_init	9-10
udi_secondary_region_prepare	9-12
udi_snprintf	19-10
udi_strcat, udi_strncat	19-2
udi_strchr, udi_strrchr, udi_memchr	19-5
udi_strcmp, udi_strncmp, udi_memcmp	19-3
udi strcpy, udi_strncpy, udi_memcpy, udi_memmove	19-4
udi_strlen	19-7
udi strtou32	19-6
udi_tagtype_t	13-21
udi_time_between	14-8
udi_time_current	14-7
udi_time_since	14-9
udi_time_t	14-3

Alphabetical List of Reference Pages

udi_timer_start	14-4
udi_timer_start_repeating	14-5
udi_trace_write	17-5
udi_trevent_t	17-3
udi_unbind_parent_ack	21-24
udi_unbind_parent_req	21-23
udi_usage_cb_t	21-8
udi_usage_ind	21-9
udi_usage_res	21-11
udi_vsnprintf	19-13



UDI Core Specification

Abstract

The UDI Core Specification defines the core set of interfaces 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 Core Specification.

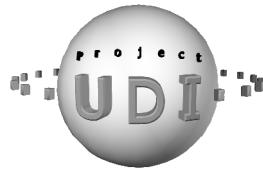
The intended audience for this book is UDI driver writers, environment implementors, and metalanguage implementors, as well as developers of additional UDI definitions such as bus bindings and ABI bindings.

Abstract



UDI Core Specification

Section 1: Overview



Introductory Material

I

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 OSes and platforms and are thus isolated from driver code.

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 may 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. ANSI/ISO 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. ISO 9960 (CDROM filesystem specification).
7. IETF RFC 1071 “*Computing the Internet checksum*”
8. IETF RFC 1141 “*Incremental updating of the Internet checksum*”
9. IETF RFC 1624 “*Computation of the Internet Checksum via Incremental Update*”
10. 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, with their associated rules and semantics, as defined in the UDI Core Specification, 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 specification.

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

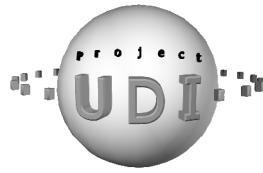
Note – UDI environment implementations may vary in the way that they implement a particular UDI interface, the amount of internal debugging and interface consistency checking provided, the underlying address or protection or synchronization domain in which UDI drivers execute, etc. However, as defined above, all UDI environments must implement the full set of interfaces defined in this Core Specification, and must adhere to the requirements of the UDI architectural model as defined in this Specification.¹

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 that are not defined in the UDI specifications. A conforming UDI device driver shall also follow all the rules and semantics defined for the use of these UDI interfaces.

1. Some environment implementations are static in that it is not possible to load new drivers or otherwise modify the configuration of the system. These *static environments* may choose to utilize UDI for their I/O subsystem support but due to size constraints, functionality limitations, or other practical considerations they may implement only the subset of the UDI specification. Such a system may be said to “use UDI” or “implement UDI”, but cannot state that it “conforms to the UDI Specification”.

A conforming UDI driver must also adhere to the general requirements regarding UDI_VERSION, header files, and the use of ISO C, as defined in Chapter 7, “*General Requirements*”.



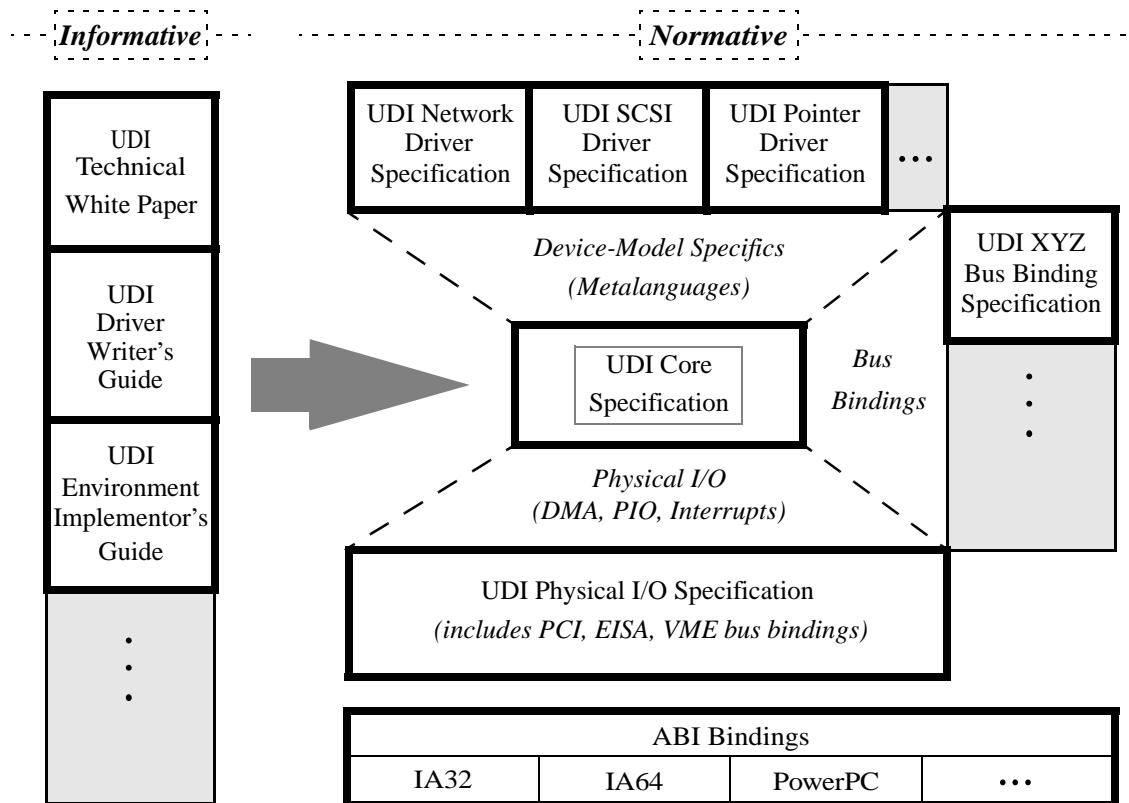
Document Organization

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 provide no additional interface definitions. 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 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
- 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.)

Documents Overview of the UDI Core Specification

These topics are not found in this UDI Core Specification because they are not strictly needed for all UDI driver implementations. A typical UDI driver for a PCI adapter would make use of UDI definitions for Physical Device Access and the PCI Bus Binding, whereas a USBDI compliant UDI driver for a USB device would use the USB Metalanguage Interface but no physical accesses or physical I/O bus bindings.



Terminology

3

3.1 Introduction

This chapter defines common terminology which has special or particular usage in UDI, and acronyms which are used in the UDI specifications. There are two categories of terminology defined here: basic terms whose purpose is to provide directives on the behavior, features and semantics of the UDI Specification, called *directive terms*; and other common terms with special or particular meaning in UDI. Other terms which are more UDI-specific and UDI architectural terms are defined in the Glossary in Appendix A.

3.2 Definitions

3.2.1 Directive Terms

can	indicates a permissible behavior available to the UDI driver.
ignored	indicates that the contents of a particular field cannot be usefully examined.
illegal	indicates a violation of the Specification.
implementation-dependent	indicates that a particular feature or behavior is not consistent across all environment implementations and must not be relied upon by the driver.
invalid	indicates a condition which is not valid within a given context.
may	indicates that a particular feature or behavior of the UDI environment is optional; UDI drivers must not rely on the existence of the feature. To avoid ambiguity, the antonym of <i>may</i> is expressed as <i>need not</i> , instead of <i>may not</i> .
must	indicates a requirement on the UDI driver.
shall	indicates that the behavior described is a requirement on the UDI environment and that UDI drivers can rely on its existence.
should	When used with respect to UDI environments: indicates that a feature is recommended but not mandatory; UDI drivers must not rely on its existence. When used with respect to UDI drivers: indicates strongly recommended behavior.
unspecified	Same as <i>implementation-dependent</i> ?
will	Same as <i>shall</i> .

3.2.2 Basic Terms

adapter	I/O hardware which provides a specific function or connectivity/bridging capability and which is accessed via a system bus. Also called a <i>card</i> , a <i>controller</i> or an <i>HBA</i> . 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.)
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 used by the Motorola 680x0, HP PA-RISC, 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 <i>sleeping</i> . Can also be used as an adjective describing OS service calls that can cause such a suspension.
	UDI uses <i>asynchronous service calls</i> instead of blocking service calls, so there is no way for a UDI-compliant driver to block. Note that this does <i>not</i> affect the embedding OS and its users' applications, since native synchronous or asynchronous interfaces are provided by external mappers, not directly by UDI drivers.
byte	A unit of data storage made up of eight binary digits (bits). An <i>octet</i> . UDI does not use the archaic meaning of "byte" to refer to other than 8-bit data units.
capability domain	see <i>protection domain</i> .
card	see <i>adapter</i> .
controller	see <i>adapter</i> .

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 <i>device tree</i> .
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 <i>device driver</i> .
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 <i>driver endianness, device endianness, protocol_endianness</i>

Definitions

Terminology

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.
handle	an opaque reference to an environment object that must not be directly referenced by drivers. Drivers must only act on such objects by passing their handles to environment service calls. Handles provide an address domain-independent, protected reference to an object (see <i>local handle</i> , <i>private handle</i> , and <i>transferable handle</i>). UDI handles are all region-local handles.
HBA	Host Bus Adapter. Another name for an <i>adapter</i> , but most commonly used for SCSI adapters.
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”).
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.
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. 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	see <i>blocking</i> .
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.
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 protection level. 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.

sleeping	see <i>blocking</i> .
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: the title for this project. 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.

3.3 Acronyms

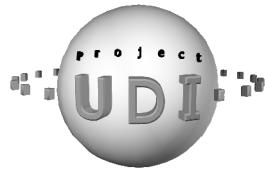
ANSI	American National Standards Institute
FIFO	First In, First Out
IEC	International Electrotechnical Commission
IETF	International ???
ISO	International Organization for Standardization
RFC	Request For Comment ???
OS	Operating System
UDI	Uniform Driver Interface

Comment – Add a section “1.4 Conventions” and title the chapter “Terminology and Conventions”?



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 resulting object files represent, as defined by the driver in its static driver properties file (see Chapter 28), one or more independent executable modules, called *driver modules*.

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). The driver specifies its secondary modules via the “module” property in the driver's `udiprops.txt` configuration file (see Chapter 28). A simple driver may be composed of a single module; more complex drivers may be composed of multiple modules. Each driver has one module, called the *primary module*, which contains initial per-instance startup code and the driver's `ddd_trace_log_format` routine. Additional modules, called *secondary modules*, may be defined by the driver. 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 UDI driver module has a single well-known entry point routine, named `init_module`. (See `init_module` on page 9-4.) All other entry points are registered dynamically by calls to environment or metalanguage services from the `init_module` routine.

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, 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 instances 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. In conjunction with this, from the driver's perspective, there is no concurrent sharing of memory among 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 below in section 4.13 for a discussion of *location-independence*).

4.4.1 Driver Partitioning

Driver writers need to consider the partitioning of their driver into regions when the driver is designed. 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

4.5.1 Relationship of Regions to Modules

Drivers are separable into multiple *modules* (as defined in Section 4.2 above) only to the degree that they have also been separated into multiple regions. Since driver modules can be separately loaded and executed in different domains, driver modules cannot share regions; each module must have its own distinct set of regions that it operates on.

Note that while a driver module is simply a sub-division of the driver’s executable code and thus has no per-instance component per-se, it does have code that executes in a per-instance context. This code must be designed with regions in mind, and the code from two different modules in a driver must not both operate on the same region.

4.5.2 Primary Module Requirements

The primary module must contain all of the code and static data for the primary region, including the driver’s `ddd_trace_log_format` routine. The primary module’s `init_module` routine must be the one that invokes `udi_primary_region_init`.

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 *metalinguage*.

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 21, “*Management Metalanguage*”.)

4.7 Driver Execution Environments

There are two general categories of UDI driver execution: execution that is not associated with any per-instance state, referred to as *module-global* execution, and execution that is associated with per-instance state, referred to as *per-instance* execution. All entries into the driver are either *module-global* or *per-instance* in nature. Per-instance entry points are said to execute in the *context of a region* or in *region context*, since they have access to driver and environment state associated with the particular region for which they were 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.

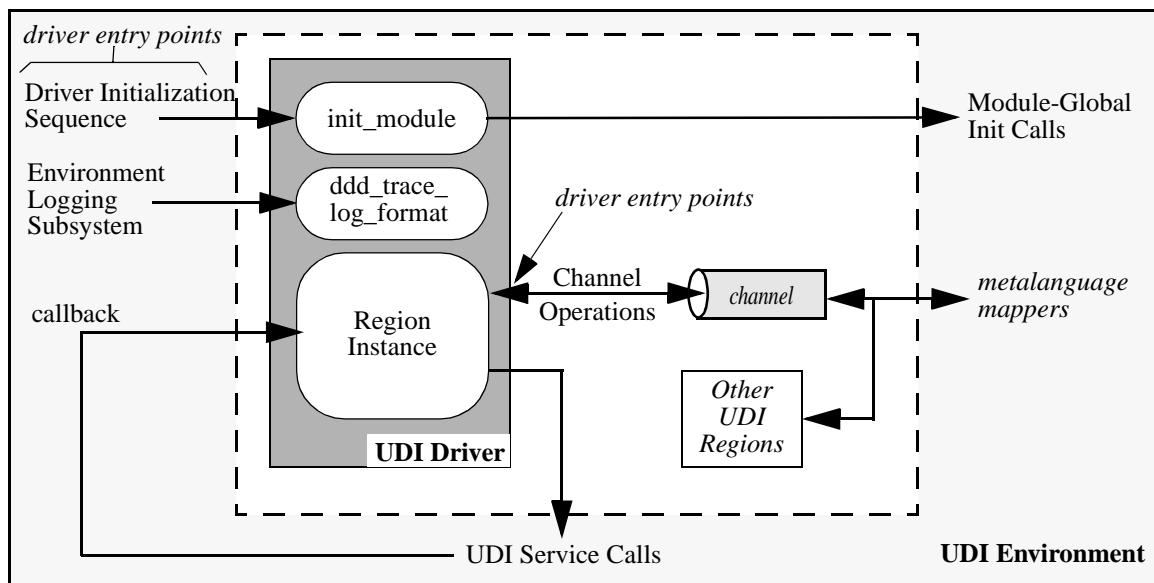
4.7.1 Non-Blocking Model

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*), service calls from the driver to the environment or metalanguage (environment or metalanguage *service calls*), or calls over channels between driver or environment regions (metalanguage-specific *channel operations*). Service calls can be further sub-divided into *module-global service calls* (calls that do not execute within the context of a region) and *region-context service calls* (calls that have per-instance context). Channel operations are always called from within a region context and are specified in terms of both the caller side (*channel operation invocation*) and callee side (*channel operation entry point*).

The following figure illustrates these function call categories.



4.9 Module-Global Entry Points

Two driver entry points are called without regard to a particular driver instance or region, and are thus global to the driver module. These are `init_module`, which is called once for each module loaded (statically or dynamically) into a particular domain, and an optional formatting function for tracing and logging (see `ddd_trace_log_format` on page 17-10). In both cases, the driver is severely limited in the external function calls that it may make, since it is not executing in a region context.

Functions callable from module-global entry points include those in Chapter 18, “*Debugging Services*” and Chapter 19, “*Utility Functions*”. In addition, the module-global initialization functions described in the next section may be called from `init_module`.

4.10 Module-Global Service Calls

4.10.1 Module-Global Initialization Functions

A set of module-global initialization functions are available for use from `init_module` to initialize driver and module state that is not associated with a driver instance or region. These functions must only be called from `init_module`. Generic module-global initialization functions are described in Chapter 9, “Initialization”. Metalanguage-specific module-global initialization functions are defined in each metalanguage chapter, but are discussed in general terms in Chapter 6, “Standard Calling Sequences”.

4.10.2 Debugging Services

The debugging services listed in Chapter 18 may be called from any driver entry point, including all module-global entry points.

4.11 Region-Context 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. Except for the module-global initialization functions, all service calls must only be invoked from within the context of a region.

There are no metalanguage-specific region-context service calls. The only metalanguage-specific service calls are the module-global initialization functions described in Chapter 6, “Standard Calling Sequences”.

4.11.1 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 designed as *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 control block argument.

4.11.2 Asynchronous Service Calls

Service calls that may not complete “immediately” are designed as *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 may require allocation, access to external resources, or delayed completion must be asynchronous. (This is required by UDI’s non-blocking model—drivers executing in a region context cannot block threads 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. Callbacks will not be processed while other driver code is actually executing in the region.

In the case of a delayed callback, the environment needs to be able to queue the pending request. In order to provide space for queuing the request, if needed, a control block is passed to each asynchronous service call.

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 10-9). 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 10-3).

4.12 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.

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 unmarshal¹ other parameters and data passed to the operations if they need to be queued or transferred between separate domains.

4.13 Location Independence

TBS

4.14 Driver Faults/Recovery

Any improper usage of the UDI service calls documented in the UDI specifications will be considered an error on the driver's part. The action that the environment takes upon encountering such an error is dependent on the environment implementation: some environments may ignore the error entirely whereas others may detect the error and remove the driver from the current execution environment. In either case, the driver should not be coded to expect to resume execution or be scheduled for future operations after generating an error of this type. There are no specified indications for such errors; since the driver's code is already known to be incorrect, it cannot be expected to be able to recover from its own errors; further, it doesn't make sense to incur a performance cost in a correct driver to make such a check. Instead, the UDI environment may deal with these errors in any way it sees fit. The UDI architecture enables UDI environments, if they so desire, to completely isolate a driver instance and kill it when it misbehaves.

Improper channel operation usage will be handled differently depending on which entity detects or encounters the error. UDI channel operations involve three distinct entities: the associated metalanguage library, the environment's metalanguage support code, and the target region. If the environment support

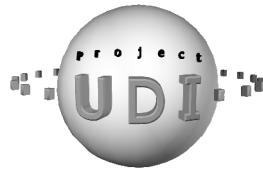
1. *Marshalling* is the activity of identifying and possibly collecting all of the information related to the request so that the information may be moved to a different domain (via an unspecified mechanism) where it will be *unmarshalled* back into operational form to deliver to the recipient.

code detects the error, the operation will be handled in a similar manner to an improper service call and the driver may be removed from the system. If the metalanguage library or the target region detects an error, an error code will be passed back to the driver in a status indication operation.

4.15 Metalanguage Model

4.15.1 Metalanguage Roles

TBS



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*. Module-global data is read-only throughout the execution of a UDI driver. Environment implementations may share a single copy of module-global data between all instances of a driver in a given 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 CPUs (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 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 `init_module` or `ddd_trace_log_format`, all data is read-only except for local automatic variables. The only permanent state that can be affected is in environment-internal state, set through environment calls such as `udi_primary_region_init`. This and other environment calls are used to register the driver's additional entry points, memory requirements, and other module-global properties.

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

5.2 Data Objects

In general, when we talk about UDI data objects, we're referring 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 *domain-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*; and 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.

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 passed control blocks.

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 which 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 per-request control block contains additional space that may be used by the driver to store information related to the request. This space is referred to as the *scratch space* of the control block and its contents are determined by the driver. When the current request is completed by transferring it to another region, ownership of the corresponding scratch space is also relinquished; the driver should not expect to be returned the same control block with its associated scratch space, nor should it expect the contents of the scratch space to be preserved.

Drivers specify their scratch space requirements through standard arguments to metalanguage-specific “`cb_init`” functions. The scratch space for a control block may actually change size as it is passed from region to region and is adjusted to meet the requirements of the receiving region.

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 arguments to metalanguage-specific “cb_init” functions.

5.2.2.3 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.

5.2.2.4 Control Block Groups

TBS

5.2.3 Region Data

TBS

5.3 Channel Context

A *channel context* is a driver-defined context value that is associated with a specific channel. The channel context is typically used to locate the region-global data structure (and is therefore typically set to point to that structure). The channel context is the only information available to the target region for a channel operation beyond the operation-specific data in the control block or associated parameters.

5.4 Transferable Objects

[TO BE SUPPLIED]

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.



Standard Calling Sequences

6

6.1 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 and parameter 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).

The sections that follow are organized by the function call categories described in Section 4.8, “Function Call Classifications,” on page 4-4. Requirements and conventions for module global entry points are defined in Section 6.2; those for module global service calls (initialization functions) are defined in Section 6.3. Channel operations are always called from within a region context and are specified in terms of both the caller side (channel operation invocation) and callee side (channel operation entry point) in Section 6.4. Lastly, region-context service calls that have general requirements (such as the asynchronous service calls with their associated callback functions) are dealt with in Section 6.5.

6.2 Module-Global Entry Points

There are only a small number of module-global entry points, and there are no particular conventions that apply to them as a whole. There can be no metalanguage-specific module-global entry points.

Module-Global Initialization Functions

6.3 Module-Global Initialization Functions

There are two types of module-global initialization functions that are metalanguage-specific: (1) the “ops_init” functions which are called by the driver’s `init_module` routine to set its entry points for a particular ops vector, and (2) the “cb_init” functions which are called by the driver’s `init_module` routine to initialize properties related to a control block group. These two sets of functions must be defined in each metalanguage within the requirements and semantics given below.

A closely-related module-global initialization function, called `udi_cb_select`, is not described here because it is a single environment function (rather than a set of metalanguage-specific functions). It is used to select which control block properties to use when receiving control blocks over channels using particular ops index values. For details, see the definition on page 9-6.

6.3.1 Channel Ops Properties Initialization

All metalanguages contain functions to initialize (driver-specific and metalanguage-specific) properties associated with channel ops vectors. These generally have declarations with the following form:

```
void <<meta>>_<<role>>_ops_init (
    udi_index_t ops_idx,
    <<meta>>_<<role>>_ops_t *ops );
```

where:

<<meta>>	is a distinct prefix identifying the metalanguage, usually beginning with the prefix, “udi_”.
<<role>>	identifies the type of channel ops vector within the metalanguage, usually reflecting a role and, if necessary, the type of channel within that role.
ops_idx	is a non-zero channel ops index number, assigned by the driver to uniquely identify this set of entry-point related properties to subsequent function calls. Each call to any “ops_init” function from the same driver must use a different ops_idx value.
ops	is a pointer to the metalanguage-specific channel operation entry point routines for this type of channel. The structure pointed to by ops must be a static (read-only) variable.

While it is recommended that metalanguages follow the above conventions, the only interface requirements on these “ops_init” functions are that there be one “ops_init” function per ops vector type, that the function contain at least two arguments whose semantics are the same as the two arguments (**ops_idx** and **ops**) above, and that the requirements defined for ops vectors in the next sub-section below are followed.

6.3.1.1 Channel Operations Vectors

The channel operations vector structure (“ops vector”) pointed to by **ops** is defined in each metalanguage in association with the corresponding “ops_init” definition. These structures generally have declarations of the following form (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):

```
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 6.4.2, “Channel Operation Entry Points”.

Each entry in the ops vector is the driver’s entry point for the corresponding channel operation. Since the function prototypes for these driver entry points can be derived from the caller-side operations (see Section 6.4, “Channel Operations”), metalanguage specifications generally do not show the function prototypes for the operation entry points. In any case, they will be declared in the metalanguage header file as the *callee* version of the corresponding channel operation definition.

6.3.2 Control Block Properties Initialization

All metalanguages contain functions to initialize (driver-specific and metalanguage-specific) properties of control block groups (metalanguage-defined groups of one or more control blocks with similar properties). These generally have declarations with the following form:

```
void <<meta>>_<<cbgroup>>_cb_init (
    udi_index_t cb_idx,
    udi_size_t scratch_requirement,
    ...<<call-dependent parms>>... );
```

where:

- <<meta>> is a distinct prefix identifying the metalanguage, usually beginning with the prefix, “udi_”.
- <<cbgroup>> identifies a control block group within the metalanguage.
- cb_idx** is a non-zero control block index number, assigned by the driver to uniquely identify this set of control block properties to subsequent function calls. Each call to any “cb_init” function from the same driver must use a different **cb_idx** value.
- scratch_requirement** is the number of bytes of scratch space the driver requires when using control blocks associated with this index. This value must not exceed UDI_MAX_SCRATCH (4000 bytes).
- <<call-dependent parms>> are the zero or more metalanguage-dependent parameters for this particular initialization function. Some metalanguages use such parameters to allow for driver-specific sizes for variable parts of control blocks.

While it is recommended that metalanguages follow the above conventions, the only interface requirements on these “cb_init” functions are that they contain at least two arguments whose semantics are the same as the first two arguments (**cb_idx** and **scratch_requirement**) above.

6.4 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) differs slightly from the calling sequence for the corresponding entry point (callee-side interface), due to the indirection involved. The entry point calling sequence can be derived from the invocation calling sequence by removing the first parameter to the invocation. For simplicity, each reference page in the UDI specifications that defines a channel operation shows only the invocation form, since the operation entry point form can be directly derived from the invocation form. Header files provided by the build environment must, however, include type definitions for both forms.

6.4.1 Channel Operation Invocations

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

```
void <<meta>>_<<op>> (
    udi_channel_t target_channel,
    <<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.
target_channel	is a handle for the channel over which this operation should be invoked. The particular channel type to use for the operation is specified in the TARGET CHANNEL section of the reference page defining the operation.
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 have the first two arguments (**target_channel** and **cb**) in the order and with the semantics shown above. The naming of <<meta>>, <<op>>, and <<cbtype>> are up to the metalanguage designer, but the above naming conventions are recommended.

6.4.2 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>>... );
```

The only difference between the calling sequence for a channel operation invocation and a corresponding operation entry point is that the first parameter to the invocation, the target channel, is not present in the entry point; the entry point doesn't need the target channel handle since it can derive any data it needs from the channel context which is placed in the control block.

For example, one driver may call:

```
udi_intr_event_ind(target_channel, 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:

```
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>>... );
```

6.5 Asynchronous Service Calls

Many environment service calls 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. This is required by UDI’s non-blocking model—drivers executing in a region context cannot block threads in order to wait for resources or I/O events.

Thus, callbacks are required on any service call that may require allocation, access to external resources, or delayed completion. Such service calls are called “asynchronous service calls”. 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. In the case of a delayed callback, the environment needs to be able to queue the pending request. In order to provide space for queuing the request, if needed, a control block is passed to each asynchronous service call.

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 10-9). 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 10-3).

6.5.1 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 <i>udi_<<category>>_<<service>>_call_t</i> .
gcb	is a pointer to a generic control block.
<<call-dependent parms>>	are the zero or more specific additional parameters for this particular service call.

6.5.2 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 control block:

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

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

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

6.5.3 Control Block Type Conversion

The UDI_GCB() macro (defined on page 10-7) 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), ...);
```

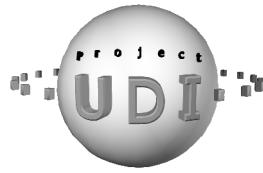
The UDI_MCB() macro (defined on page 10-8) is provided for convenience in converting a generic control block pointer back to a specific control block type. Using this macro, a typical callback routine begins with:

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




UDI Core Specification

Section 3: Core Services



General Requirements

7

7.1 UDI_VERSION

In each device driver source file, before including any UDI header files, the driver must define the preprocessor symbol, UDI_VERSION, to indicate the version of the UDI Core Specification to which it conforms. For the current version of the specification, UDI_VERSION must be set to 0x090, as follows:

```
#define UDI_VERSION 0x090
```

7.2 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.

7.3 C Language Requirements

UDI device drivers that are written in C must be compiled using an ISO C compiler.



Fundamental Types

8

8.1 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 8.2.3.

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.

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 28.6.6, “Region Declaration,” on page 28-14). 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: Packaging and Distribution”) 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 to provide source portability.

8.2 Usage of Standard ISO C Data Types

The following standard ISO C types are used by UDI.

8.2.1 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.

8.2.2 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.

8.2.2.1 Null Pointers

The special symbol, `NULL`, is defined by `<udi.h>` and is guaranteed to be different from any valid pointer value (generic or otherwise). 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.

8.2.3 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 used as the parameter list in a function header

- **va_dcl** is a declaration for va_alist. No semicolon should follow va_dcl.
- **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.

8.3 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 8-1 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 NULL_BUF. 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.

8.4 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: -27..27-1 */
typedef <INTEGRAL> udi_sbit16_t; /* signed 16-bit: -215..215-1 */
typedef <INTEGRAL> udi_sbit32_t; /* signed 32-bit: -231..231-1 */
typedef <INTEGRAL> udi_ubit8_t; /* unsigned 8-bit: 0..28-1 */
typedef <INTEGRAL> udi_ubit16_t; /* unsigned 16-bit: 0..216-1 */
typedef <INTEGRAL> udi_ubit32_t; /* unsigned 32-bit: 0..232-1 */
typedef udi_ubit8_t udi_boolean_t; /* 0=False; 1..28-1=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 8.6.2 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);
```

8.5 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.

8.5.1 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;
```

Because of architectural minimums on some of the defined size limits (e.g., see `udi_limits_t` on page 9-15) `udi_size_t` is guaranteed to be at least 16 bits in size.

8.5.2 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;
```

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.

8.5.2.1 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 `<<meta>>_<<cbgroup>>_cb_init` function (see Section 6.3.2, “Control Block Properties Initialization,” on page 6-3) 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.

8.5.2.2 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 “child_meta”, “parent_meta”, and “internal_meta” declarations in the driver’s Static Driver Properties (see Chapter 28). A metalanguage index of zero indicates the Management Metalanguage. Metalanguage index values are used with the Tracing & Logging Facility to associate certain types of events with specific metalanguages.

8.5.2.3 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 <>_{meta}>_<>_{role}>>_ops_init function (see Section 6.3.1, “Channel Ops Properties Initialization,” on page 6-2) 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 16-6), or to specify unanchored default channels (see udi_primary_region_init, udi_secondary_region_init, and udi_secondary_region_prepare in Chapter 9, “Initialization”).

8.5.2.4 Region Index

A udi_index_t variable may be used to hold a *region index*. A region index is used to identify a 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.

8.6 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 use opaque values only to pass 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.)

8.6.1 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 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, however, cannot compare handle values for equality or otherwise directly determine if a handle variable currently holds a null value. (Pointers, on the other hand, can be compared against NULL.)

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

8.6.1.1 Buffer Handle

Application 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 which is manipulated via the following handle type:

```
typedef <HANDLE> udi_buf_t;
```

The following `udi_buf_t` constant provides a null value for buffer handles:

```
#define NULL_BUF <NULL_HANDLE>
```

Buffer handles are transferable between regions.

8.6.1.2 Channel Handle

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. This handle is of the following type:

```
typedef <HANDLE> udi_channel_t;
```

The following `udi_channel_t` constant provides a null value for channel handles:

```
#define NULL_CHANNEL <NULL_HANDLE>
```

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

8.6.1.3 Constraints Handle

When buffer data is passed to a driver, it may be required to satisfy a set of constraints, such as maximum transfer size or device offset alignment. In order for drivers to express their constraints UDI defines a constraints object, which is accessed via an opaque *constraints handle*:

```
typedef <HANDLE> udi_constraints_t;
```

The following `udi_constraints_t` constant provides a null value for constraints handles:

```
#define NULL_CONSTRAINTS <NULL_HANDLE>
```

Constraints handles are passed from parent to child driver along an I/O path to ensure that when a buffer is allocated by the parent, the environment can optimize it appropriately to meet all the constraints along the path, potentially avoiding copies later. The environment may also add its own hidden constraints based on knowledge of how the buffer will be used in the rest of the operating system.

Constraints handles are transferable between regions.

8.6.2 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.

8.6.2.1 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 14-7.

8.7 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. All control blocks are semi-opaque. See Chapter 10, “Control Block Management” for more details on control blocks.

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.

8.8 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.

8.8.1 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.

Format:

```
typedef     udi_ubit32_t      udi_status_t;

#define      UDI_STATUS_CODE_MASK    0x0000FFFF
#define      UDI_CORRELATE_OFFSET    16
#define      UDI_CORRELATE_MASK     0xFFFF0000
```

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), and a 15-bit “specific status code”:

```
#define      UDI_SPECIFIC_STATUS_MASK 0xEFFF
#define      UDI_STAT_META_SPECIFIC   0x8000
```

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.

Use:

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`.

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 17-6), 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 ((my_status & UDI_STATUS_CODE_MASK) == UDI_STAT_XXX)
    handle_error();
```

8.8.1.1 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 4.14 on page 4-6.

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

Table 8-2 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 are valid values for the corresponding parameter types but the combination of parameters does not make sense for this operation.
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 metalanguage interface 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.

Table 8-2 Common UDI Status Codes

Status Code	Value	Meaning	Description
UDI_STAT_ABORTED	5	Operation aborted	The operation was successfully aborted as a result of a <code>udi_cancel</code> operation 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.
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_UNDERUN	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.

Table 8-2 Common UDI Status Codes

Status Code	Value	Meaning	Description
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 <code>udi_bind_to_parent_req</code> cannot be supported because this driver instance is already bound to the maximum number of parents that it can support.
UDI_STAT_BAD_PARENT_TYPE	18	Cannot bind to this type of parent device	The <code>udi_bind_to_parent_req</code> 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.

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.

8.8.2 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 pass as arguments to certain “cb_init” functions; this allows a driver to register the layout of inline memory structures that aren’t strictly typed by the metalanguage.

Format:

```
typedef udi_ubit8_t *udi_layout_t;
```

A `udi_layout_t` data layout specifier consists of an array of one or more `udi_ubit8_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.

Use:

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 8-3 Specific-Length Element Type Codes

Element Type Code	Value	Corresponding Data Type
<code>UDI_DL_UBIT8_T</code>	1	<code>udi_ubit8_t</code>
<code>UDI_DL_SBIT8_T</code>	2	<code>udi_sbit8_t</code>
<code>UDI_DL_UBIT16_T</code>	3	<code>udi_ubit16_t</code>
<code>UDI_DL_SBIT16_T</code>	4	<code>udi_sbit16_t</code>
<code>UDI_DL_UBIT32_T</code>	5	<code>udi_ubit32_t</code>

Table 8-3 Specific-Length Element Type Codes

Element Type Code	Value	Corresponding Data Type
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 8-4 Other Self-Contained Element Type Codes

Element Type Code	Value	Corresponding Data Type
UDI_DL_VOID_PTR	20	void * (only the pointer is transferred, not any referenced object; therefore the value is only meaningful if eventually transferred back to the originating region; typically used for transaction context pointers)
UDI_DL_SIZE_T	21	udi_size_t
UDI_DL_INDEX_T	22	udi_index_t
UDI_DL_TIMESTAMP_T	23	udi_timestamp_t

Table 8-5 Opaque Handle Element Type Codes

Element Type Code	Value	Corresponding Data Type
UDI_DL_BUF_T	30	udi_buf_t (may be NULL_BUF)
UDI_DL_BUF_RETURN	31	udi_buf_t (may be NULL_BUF). In this case, the buffer is to be returned to the sending region (via udi_channel_event_ind over the same channel) if the receiving region is abruptly terminated while still holding this buffer. The orig_tr_context for the udi_channel_event_cb_t will be NULL.
UDI_DL_BUF_CONTEXT	32	Same as UDI_DL_BUF_RETURN, but orig_tr_context will be derived from a UDI_DL_VOID_PTR field in the control block used to transfer the buffer. The next layout element (one unsigned byte) is interpreted as the offset of this void pointer field within the control block (relative to the start of the visible fields, including the udi_cb_t header).
UDI_DL_CHANNEL_T	33	udi_channel_t (must be a loose end or NULL_CHANNEL)
UDI_DL_CONSTRAINTS_T	34	udi_constraints_t (may be NULL_CONSTRAINTS)

Table 8-6 Indirect Element Type Codes

Element Type Code	Value	Corresponding Data Type
UDI_DL_CB	40	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	41	<code>void *</code> (untyped array of inline memory bytes; may be NULL)
UDI_DL_INLINE_DRIVER_TYPED	42	<code>void *</code> (structure determined by driver; corresponding “cb_init” call supplies layout; may be NULL)
UDI_DL_MOVABLE_UNTYPED	43	<code>void *</code> (untyped array of movable memory bytes; may be NULL)

Table 8-7 Nested Element Type Codes

Element Type Code	Value	Description
UDI_DL_MOVABLE_TYPED	50	Pointer to movable memory whose structure is determined by the metalanguage definition. Subsequent layout elements describe structure. (May be NULL)
UDI_DL_INLINE_TYPED	51	Pointer to inline 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. 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.

Structures Requiring a Fixed Binary Representation

8.9 Structures Requiring a Fixed Binary Representation

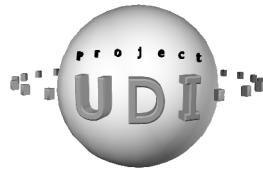
While drivers must specify the structure layout of certain driver-defined structures which are passed between regions (as indicated in the previous subsection), drivers need not concern themselves with the actual binary layout of such structures, or in general with the binary layout of UDI-defined structures 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 “Endian Management” on page 19-26. 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.



Initialization

9

9.1 Overview

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

9.1.1 Per-Driver Initialization

Per-driver initialization starts once the driver has been loaded and/or linked into the system and the driver's `init_module` routine is called. The `init_module` routine is responsible for all of the per driver initialization, which includes specifying all of the parameters required to create the primary region and any secondary regions used by this driver, specifying channel operations vectors for each metalanguage used by the driver module, and initializing properties of control block groups.

Each metalanguage ops vector is a structure listing function pointers for each entry point routine needed for a particular metalanguage role. The driver must register these ops vectors with the environment so its entry points are called when channel operations are invoked on its channels. The driver registers its Management Metalanguage ops vector by passing a pointer to a `udi_mgmt_ops_t` structure to `udi_primary_region_init`, and registers ops vectors for other metalanguages using the appropriate `<<meta>>_<<role>>_ops_init` function defined in each metalanguage. Each `<<meta>>_<<role>>_ops_init` call associates an ops vector with an ops index number provided by the driver that can be used in subsequent service calls.

The device driver uses `udi_primary_region_init` to specify the parameters required by the primary region of each instance for this driver. These include selection of ops indexes or ops vectors for parent and child bind channels and the management channel, and the size of the initial region data.

If the device driver is to be implemented using multiple regions, its additional (*secondary*) regions can be created either statically (at the same time as the primary region) or dynamically, as required, during driver execution. To create a region statically, the driver calls `udi_secondary_region_init` from `init_module` once for each region to be created. The parameters to `udi_secondary_region_init` include the region type index, ops indexes for the initial channel between the primary and secondary regions, and the size of the initial region data.

To create secondary regions dynamically, the driver calls `udi_secondary_region_prepare` from `init_module` once for each type of secondary region. Then, when the driver determines that it is time to dynamically create a secondary region of this type, it calls `udi_secondary_region_create` once for each secondary region of this type to be created.

9.1.2 Per-Instance Initialization

Per-instance initialization for the driver starts when the driver receives the `udi_usage_ind` on its management channel. Following this general resource level and tracing indication, the driver typically receives one or more `udi_region_attach_ind` (if the driver uses statically-allocated secondary regions) or a `udi_bind_to_parent_req` on its management channel.

If the driver uses statically-allocated secondary regions, it will receive one `udi_channel_attach_ind` for each secondary region specified by the driver calling `udi_secondary_region_init` from its `init_module` routine. For each `udi_channel_attach_ind`, the driver must at least save the channel handle to the secondary region being attached before calling `udi_region_att_res`. The channel handle is passed to the driver in the `udi_region_att_cb`.

When all dynamically-allocated secondary regions, if any, have been attached to the driver primary region, per-instance initialization continues when the driver receives a Management Metalanguage channel operation (`udi_bind_to_parent_req`) requesting that the driver bind itself to its hardware or parent driver (see Chapter 21, “*Management Metalanguage*”). The per-instance initialization is required to be complete when the driver calls `udi_bind_to_parent_ack`, at which point the UDI environment assumes that the driver is “open for business”. In between, the driver must perform any required initialization, such as verifying the existence of its hardware, initializing that hardware, and allocating any memory or timer resources it may require. Since UDI allocation routines return their resources by means of a callback, a driver can either proceed by using a chained sequence of separate callback routines, or by using one callback routine with a state variable to indicate where the driver is in the initialization process.

9.2 Entry Points

NAME	init_module	<i>Module initialization entry point routine</i>
SYNOPSIS	<pre>#include <udi.h> void init_module (void);</pre>	
DESCRIPTION	<p>The <code>init_module</code> routine performs the initialization that must be done once per driver module. This initialization includes registering all channel ops vectors used by this driver and specifying all of the parameters required to create all regions used by this driver. Channel ops vectors are registered using the <code><<meta>>_<<role>>_ops_init</code> functions specific to each metalanguage, except for the management metalanguage ops which are passed directly to <code>udi_primary_region_init</code>.</p> <p>Since all drivers must have at least a primary region, all drivers must call <code>udi_primary_region_init</code> to specify the parameters used to create their primary region. If the driver is implemented using multiple regions, those regions can be created either statically, as the driver is instantiated, or dynamically at run time. In either case, the appropriate calls to specify the parameters for these secondary regions must be made from the driver's <code>init_module</code> routine.</p> <p>Secondary regions can be created statically, using <code>udi_secondary_region_init</code>, or dynamically, by first specifying the parameters required for that region type by calling <code>udi_secondary_region_prepare</code> from the driver's <code>init_module</code> routine and then, at run time, when the driver determines more secondary regions of this type are desirable, by calling <code>udi_secondary_region_create</code> to create the region. Region types are specific to each driver and are specified by the region index parameter.</p> <p>The only functions that are callable from <code>init_module</code> are the functions documented in this chapter, the metalanguage-specific “ops_init” and “cb_init” routines, and the debugging and utility functions defined in chapters 18 and 19 respectively. See “Driver Execution Environments” on page 4-3 for additional information.</p>	
WARNINGS	<p>While in <code>init_module</code>, the driver must not modify any global or static variables. It may only modify automatic (“stack”) variables and parameters.</p>	

9.3 Service Calls

NAME	udi_cb_select	<i>Select control block properties to use with incoming channel ops</i>
SYNOPSIS	<pre>#include <udi.h> void udi_cb_select (udi_index_t ops_idx, udi_index_t cb_idx);</pre>	
ARGUMENTS	<p>ops_idx is a channel ops index number, as previously passed to <code><<meta>>_<<role>>_ops_init</code> (see “Channel Ops Properties Initialization” on page 6-2).</p> <p>cb_idx is a control block index number, as previously passed to <code><<meta>>_<<cbgroup>>_cb_init</code> (see “Control Block Properties Initialization” on page 6-3).</p>	
DESCRIPTION	<p>The driver calls this function from its <code>init_module</code> routine to select which control block properties (specified by cb_idx) to use for a particular control block group when receiving control blocks over channels using a particular channel ops index (as specified by ops_idx). Calls to this function are optional and will not be needed by many drivers. They are only relevant when multiple cb_idx values are associated with the same control block group (by calling the same <code><<meta>>_<<cbgroup>>_cb_init</code> multiple times). In such a case, a control block received through a channel operation will by default use the maximum of all properties (such as scratch requirements) associated with each relevant control block index. If, instead, the driver has called <code>udi_cb_select</code> for the ops index through which the control block is received, the specified control block properties will be used exclusively. See Section 5.2.2, “Control Blocks,” on page 5-2 for more details on how <code>udi_cb_select</code> affects control blocks.</p> <p>In all cases, control block allocation uses the specific properties associated with a control block index parameter, and is unaffected by <code>udi_cb_select</code> calls.</p>	
WARNINGS	<p><code>udi_cb_select</code> is only callable from the driver’s <code>init_module</code> routine.</p>	
REFERENCES	<code>init_module</code> , <code>udi_cb_alloc</code>	

NAME	udi_gcb_init	<i>Initialize generic control block properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gcb_init (udi_index_t cb_idx, udi_size_t scratch_requirement);</pre>	
		#define UDI_MAX_SCRATCH 4000
ARGUMENTS		cb_idx , scratch_requirement are standard arguments described in “Control Block Properties Initialization” on page 6-3.
DESCRIPTION		Control blocks which are to be used only for asynchronous environment service calls, such as <code>udi_mem_alloc</code> , and not for any channel operations, may be allocated using a control block index that is initialized using <code>udi_gcb_init</code> . These control blocks have no metalanguage-specific visible part and are directly referenced by the <code>udi_cb_t</code> generic control block pointer. As a result, such control blocks cannot be used (or defined for use) in channel operations.
		Scratch requirements for this and any metalanguage-specific <code><<meta>>_<<cbgroup>>_cb_init</code> function must not exceed <code>UDI_MAX_SCRATCH</code> (4000 bytes).
WARNINGS		<code>udi_gcb_init</code> is only callable from the driver’s <code>init_module</code> routine.
REFERENCES		<code>init_module</code> , <code>udi_cb_alloc</code>

NAME	udi_primary_region_init	<i>Initialize primary region properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_primary_region_init (udi_mgmt_ops_t *mgmt_ops, void (*trace_log_format_func)(), udi_index_t child_bind_ops_idx, udi_index_t parent_bind_ops_idx, udi_size_t mgmt_scratch_size, udi_size_t rdata_size, void *init_params);</pre>	
ARGUMENTS	<p>mgmt_ops is a pointer to an ops vector of driver entry points for the required Management Metalanguage channel operation routines. See Chapter 21, “Management Metalanguage” for details on Management Metalanguage and its channel operations.</p> <p>trace_log_format_func is the address of a routine provided by the driver to format the tracing and logging entries previously provided by the driver in the <code>udi_trace_write</code> or <code>udi_log_write</code> operations. This is the address of a driver-global function that is called directly by the UDI environment and does not execute as part of any region. This function pointer may be passed as NULL if the driver chooses not to implement this function. See the <code>ddd_trace_log_format</code> definition in Section 17.3 on page 17-9 for more information.</p> <p>child_bind_ops_idx is an ops index to use to anchor this driver’s end of bind channels from its children. If this is set to zero, child bind channels will be created with loose ends that may be anchored later. Otherwise, at child instantiation the driver’s end will be anchored using this ops index and its channel context will be set to point to the region data area.</p> <p>parent_bind_ops_idx is an ops index used to anchor the driver’s end of bind channels to its parents. If zero, parent bind channels will be created with loose ends that may be anchored later. Otherwise, prior to binding with the parent, the driver’s end will be anchored using this ops index and its channel context will be set to point to the region data area.</p> <p>mgmt_scratch_size specifies in bytes the driver’s requirements for scratch area size in Management Metalanguage control blocks. The scratch size specified here should be the maximum of the driver’s scratch size needs across the various control blocks in the Management Metalanguage.</p> <p>rdata_size is the size, in bytes, of the region data area to be allocated (as if by <code>udi_mem_alloc</code>) for the primary region of each driver instance. rdata_size must be at least <code>sizeof(udi_init_context_t)</code>.</p>	

Initialization

udi_primary_region_init

DESCRIPTION

init_params is a driver-private pointer value passed into each region's initial context, via the `udi_init_context_t`.

udi_primary_region_init registers information the environment needs to create a new primary region, then creates that new primary region when the driver is instantiated.

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 and the **first_channel**'s context will point to the region data area.

If **child_bind_ops_idx** is non-zero, it must refer to an ops vector that is for the parent role of the metalanguage specified by the one and only "child_meta" declaration for this driver. (See Chapter 28, "Static Driver Properties".)

If **parent_bind_ops_idx** is non-zero, it must refer to an ops vector that is for the child role of the metalanguage specified by the one and only "parent_meta" declaration for this driver. (See Chapter 28, "Static Driver Properties".)

Since all Management Metalanguage operations are initiated from the environment to the driver, the environment will allocate any associated control blocks with the specified scratch size before sending them to the driver.

WARNINGS

`udi_primary_region_init` is only callable from the driver's primary module's `init_module` routine.

`udi_primary_region_init` must only be called once for each driver.

REFERENCES

`init_module`, `udi_init_context_t`, `udi_channel_anchor`,
`udi_secondary_region_init`, `ddd_trace_log_format`

NAME	udi_secondary_region_init	<i>Initialize secondary region properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_secondary_region_init (udi_index_t region_idx, udi_index_t primary_ops_idx, udi_index_t secondary_ops_idx, udi_size_t rdata_size, void *init_params);</pre>	
ARGUMENTS	<p>region_idx is a non-zero driver-dependent index value that indirectly identifies a late-bound set of platform-dependent properties that will be attached to the new region (address and capability domains, memory residence, priority, etc.) for the region being created. These properties are derived from the driver's region attributes (see Section 28.6.6, "Region Declaration," on page 28-14). 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). A region index value of zero is reserved for the primary region of a driver instance.</p> <p>primary_ops_idx is a non-zero driver-dependent index value for the channel ops vector that the driver wants to associate with the primary region's end of the initial channel for the secondary region.</p> <p>secondary_ops_idx is a non-zero driver-dependent index value for the channel ops vector that the driver wants to associate with the secondary region's end of its initial channel (see first_channel in <i>udi_init_context_t</i>).</p> <p>rdata_size is the size, in bytes, of the <i>region data area</i> to be allocated (as if by <i>udi_mem_alloc</i>) for secondary regions created with the given region index. rdata_size must be at least <i>sizeof(udi_init_context_t)</i>.</p> <p>init_params is a driver-private pointer value passed into each region's initial context, via the <i>udi_init_context_t</i>.</p>	
DESCRIPTION	<p><i>udi_secondary_region_init</i> registers information the environment needs to create a new secondary region and its first channel, based upon the region_idx parameter. A new secondary region corresponding to this region_idx will be automatically created and attached to the primary region when the driver is instantiated.</p> <p>The region_idx must match a region index in the region attributes of the driver's static driver properties. The channel ops and other parameters will be used to create the new region and first channel when the corresponding driver instance is created by the Management Agent. UDI initializes the new region's</p>	

first channel's ops vector with the set of entry points identified by `secondary_ops_idx`, which is the ops index used to communicate between the primary region and this secondary region.

When the new region has been fully initialized, UDI passes a handle for the primary region's end of the new channel to the primary region via `udi_region_attach_ind`. This end of the channel is created already anchored in the primary region using the entry points identified by `primary_ops_idx`.

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 and the `first_channel`'s context will point to the region data area.

As a recommended and expected (but not required) convention in the driver-internal metalanguage definition, the primary region should send some form of initialization operation (bind operation) to the new secondary region's first channel. This operation is intended 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".

WARNINGS

`udi_secondary_region_init` is only callable from the driver's `init_module` routine.

REFERENCES

`init_module`, `udi_init_context_t`,
`udi_secondary_region_create`,
`udi_secondary_region_prepare`

NAME	udi_secondary_region_prepare	<i>Prepare for secondary region creation</i>
SYNOPSIS	<pre>#include <udi.h> void udi_secondary_region_prepare (udi_index_t region_idx, udi_index_t first_ops_idx, udi_size_t rdata_size, void *init_params);</pre>	
ARGUMENTS	<p>region_idx is a driver-dependent index value that must match a subsequent call to <i>udi_secondary_region_create</i>. This must also match the index number of a region attribute descriptor in the driver's static driver properties. A region index value of zero is reserved for the primary region of a driver instance.</p> <p>first_ops_idx is a non-zero driver-dependent index value for the channel ops vector that the driver wants to associate with the initial channel on the secondary region (see first_channel in <i>udi_init_context_t</i>). It must match the ops index previously passed to a metalanguage-specific "ops_init" call. This index will refer to an ops vector for the Internal Management Metalanguage or a custom driver-internal metalanguage used to talk to the primary region within the driver instance. See also the definition of the ops index in the <i>Fundamental Types</i> Chapter of this Specification.</p> <p>rdata_size is the size, in bytes, of the <i>region data area</i> to be allocated (as if by <i>udi_mem_alloc</i>) for each secondary region created with the given region index. rdata_size must be at least <i>sizeof(udi_init_context_t)</i>.</p> <p>init_params is a driver-private pointer value passed into each region's initial context, via the <i>udi_init_context_t</i>.</p>	
DESCRIPTION	<p><i>udi_secondary_region_prepare</i> registers information the environment will eventually need to create a new region as a result of a call to <i>udi_secondary_region_create</i>. The combination of these two calls behaves like <i>udi_secondary_region_init</i>.</p> <p>This function is used when secondary regions are to be created dynamically by the driver.</p>	
WARNINGS	<p><i>udi_secondary_region_prepare</i> is only callable from the driver's <i>init_module</i> routine.</p>	
REFERENCES	<p><i>init_module</i>, <i>udi_init_context_t</i>, <i>udi_secondary_region_create</i>, <i>udi_secondary_region_init</i></p>	

9.4 Initial Region Data Structures

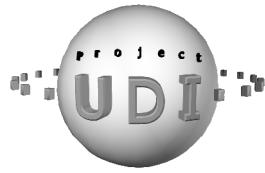
NAME	udi_init_context_t	<i>Initial context for new regions</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_channel_t first_channel; udi_limits_t limits; void *init_params; } udi_init_context_t;</pre>	
MEMBERS	<p>first_channel is the handle for the first channel for the region. This channel is used to communicate with the creator of this region; for the primary region of a driver instance, this would be the Management Agent and this channel would use the Management Metalanguage (via the channel operations registered with <code>udi_primary_region_init</code> during <code>init_module</code>); for other regions, this channel uses the Internal Management Metalanguage or a custom driver-internal metalanguage to talk to the primary region of the same driver instance (via the channel operations from the ops index registered with <code>udi_secondary_region_init</code> or <code>udi_secondary_region_prepare</code> during <code>init_module</code>). The local channel endpoint is anchored in the newly created region, using the appropriate channel operations, and with its channel context set to point to this <code>udi_init_context_t</code> structure.</p> <p>limits is a structure that describes system resource limits. See “<code>udi_limits_t</code>” on page 9-15.</p> <p>init_params is a driver-private value passed from <code>init_module</code> into each region’s initial context. This can be used for driver-internal partitioning in which multiple similar drivers share common code; this common code could then be designed to obtain driver-specific parameters from a structure pointed to by <code>init_params</code>.</p> <p>When <code>init_params</code> is used as a pointer to data, it must point to read-only (static const) data, since it refers to driver-global data (as opposed to per-instance data).</p>	
DESCRIPTION	The <code>udi_init_context_t</code> structure is stored at the front of the region data area of a 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 passed to the driver as the initial context for <code>first_channel</code> .	
REFERENCES	<code>init_module</code> , <code>udi_limits_t</code> , <code>udi_primary_region_init</code> , <code>udi_secondary_region_init</code> , <code>udi_secondary_region_prepare</code>	

NAME	udi_limits_t	<i>Platform-specific allocation and access limits</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_size_t max_legal_alloc; udi_size_t max_safe_alloc; udi_size_t max_trace_data_len; udi_size_t max_log_data_len; udi_size_t max_format_data_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</pre>	
MEMBERS	<p>max_legal_alloc is the maximum legal memory allocation size for this system. Any larger size passed to <code>udi_mem_alloc</code> will produce indeterminate results, which could include termination of the driver or region, or even a complete system abort.</p> <p>max_safe_alloc is the maximum memory allocation size that should be passed to <code>udi_mem_alloc</code> without being prepared to cancel an unsuccessful allocation (see definition of “safe limits” below).</p> <p>max_trace_data_len is the maximum legal size of the data buffer passed to <code>udi_trace_write</code>.</p> <p>max_log_data_len is the maximum legal size of the data buffer passed to <code>udi_log_write</code>.</p> <p>max_format_data_len is the maximum legal size of the formatted buffer for logged or traced data. The max_format_data_len value shall be greater than or equal to the maximum of max_trace_data_len and max_log_data_len.</p> <p>max_instance_attr_len is the maximum legal size of an instance attribute value.</p> <p>min_curtime_res is the minimum time difference, in nanoseconds, between successive unique values returned by <code>udi_time_current</code>.</p> <p>min_timer_res is the minimum resolution, in nanoseconds, of timers registered with <code>udi_timer_start</code>. Interval values passed to <code>udi_timer_start</code> will be rounded up to the nearest multiple of this value.</p>	

DESCRIPTION	<p>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.</p> <p>The udi_limits_t structure is passed to a driver instance via the udi_init_context_t in its initial region data.</p> <p>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.</p> <p>There are two types of allocation limits: <i>legal</i> limits and <i>safe</i> limits. Legal limits represent the absolute upper bound on a single allocation. Drivers must not make requests that would exceed the legal limits.</p> <p>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.</p> <p>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.</p> <p>The max_trace_data_len, max_log_data_len, and the max_format_data_len limits specify the maximum size, in bytes, of trace or log data that shall be passed to udi_trace_write, udi_log_write, and outputted by ddd_trace_log_format, respectively. These limits are guaranteed to be greater than or equal to UDI_MIN_TRACE_LOG_LIMIT in all UDI environments.</p> <p>The max_instance_attr_len parameter specifies the maximum size, in bytes, of a device instance attribute value (see Chapter 15, “<i>Instance Attribute Management</i>”) 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.</p> <p>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, “<i>Time Management</i>”). 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.</p>
-------------	--

REFERENCES

udi_mem_alloc, udi_cancel, udi_timer_start,
udi_timer_start_repeating, udi_constraints_attr_t,
udi_init_context_t, udi_instance_attr_set,
udi_trace_write, udi_log_write, ddd_trace_log_format



10.1 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 which (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 6-6 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 20, “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.

10.2 Control Block Service Calls and Macros

NAME	udi_cb_t	<i>Generic, least-common-denominator control block</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { void *context; void *scratch; } udi_cb_t;</pre>	
MEMBERS	<p>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.</p> <p>See <code>udi_channel_set_context</code> for details on how channel context is determined.</p> <p>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 scratch size specified by each driver using <code>udi_gcb_init</code> (see page 9-7) or a metalanguage-specific “<i>cb_init</i>” function.</p>	
DESCRIPTION	<p>The <code>udi_cb_t</code> structure is used for generically handling control blocks and accessing their scratch area and context pointer. All meta-language specific control blocks have a <code>udi_cb_t</code> structure as their first structure member.</p> <p>Both context and scratch, as well as the scratch area contents, are preserved across asynchronous service calls, but not across channel operations.</p>	

NAME	udi_cb_alloc	<i>Allocate a new control block</i>
SYNOPSIS	<pre>#include <udi.h> void udi_cb_alloc (udi_cb_alloc_call_t *callback, udi_cb_t *gcb, udi_index_t cb_idx); typedef void udi_cb_alloc_call_t (udi_cb_t *gcb, udi_cb_t *new_cb);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>cb_idx is a control block index that indicates required properties of the control block, such as metalanguage type and scratch size.</p> <p>new_cb is a pointer to the newly allocated control block.</p>	
DESCRIPTION	<p>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.</p> <p>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.</p> <p>As with all control block allocations, including metalanguage-specific control block allocations, the initial values in the scratch space are undefined; 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 undefined, except for those which point to other control block components, such as the scratch space pointer and the context pointer. The scratch pointer is initialized to point to the associated scratch area and the pointer must not be modified by the driver. The context pointer is cloned from the incoming gcb; i.e. it is initialized to the value of context currently contained in gcb.</p>	
WARNINGS	<p>The control block obtained with this call may 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 initialized with udi_gcb_init (see page 9-7) rather than a metalanguage-specific “cb_init” call, then the new control block must not be used with any channel operations.</p> <p>Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p>	

REFERENCES

`udi_cb_t`, `udi_cb_free`, `udi_gcb_init`, `udi_timer_start`,
`udi_timer_start_repeating`

NAME	udi_cb_free	<i>Deallocates a previously obtained control block</i>
SYNOPSIS	<pre>#include <udi.h> void udi_cb_free (udi_cb_t *cb);</pre>	
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.	
REFERENCES	udi_cb_alloc, udi_cancel	

NAME	UDI_GCB	<i>Convert any control block to udi_cb_t representation</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_GCB(mcb) (&(mcb)->gcb)</pre>	
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 (<i>udi_cb_t *</i>) 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		<i>udi_cb_t</i>

NAME	UDI_MCB	<i>Convert a generic control block to a specific one</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_MCB(gcb, cb_type) ((cb_type *)(gcb))</pre>	
ARGUMENTS	<p>gcb is a pointer to the control block reference to be converted.</p> <p>cb_type is the type name for the desired specific control block type.</p>	
DESCRIPTION	<p>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.</p> <p>This macro is provided for convenience only. Its use is highly recommended but not required.</p>	
WARNINGS	The control block referenced by gcb must have been previously obtained by a call to <code>udi_cb_alloc</code> with a cb_idx appropriate to cb_type .	
REFERENCES	udi_cb_t	

NAME	udi_cancel	<i>Cancel a pending allocation request</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS		callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “ <i>Standard Calling Sequences</i> ”.
DESCRIPTION		<p>Any service request with a pending callback, including timer requests, 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, regardless of whether or not allocations have actually completed).</p> <p>When a 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 <code>udi_cancel</code> does not provide an undo operation, but rather an abort operation; any objects (such as a data buffer for <code>udi_buf_write</code>) being modified or created by the original request are destroyed by the abort.</p> <p>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.</p>
WARNINGS		<p><code>udi_cancel</code> 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.</p> <p>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 <code>udi_cancel</code> must not be used to cancel a request without first checking to see if the corresponding callback has been called.</p> <p>If a driver issues a <code>udi_cancel</code> for a control block that is not active the driver is in error. See the “Driver Faults/Recovery” section of “<i>Execution Model</i>” for an explanation of how the environment may react to this driver error.</p>

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 which immediately precedes it. In fact, it could even cancel the subsequent `udi_timer_start` 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_timer_start(ddd_step3, UDI_GCB(cb1), intvl);
}

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

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));
}

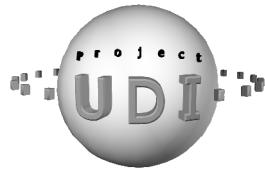
void
ddd_step1a(
    udi_cb_t *gcb)
{
    /* udi_mem_alloc in step1 has been cancelled. */
    ...
}

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

void
ddd_step3(
    udi_cb_t *gcb)
{
    ...
}

```

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.



11.1 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 Section 19.2, “String/Memory Functions,” on page 19-1.

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 9-15) to which it must conform. The resource management operations in the Management Metalanguage provide additional resource utilization guidelines to the driver (see Section 21.4.2, “Resource Management,” on page 21-5).

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.

11.2 Memory Management Service Calls

NAME	udi_mem_alloc	<i>Allocate memory for a virtually-contiguous object</i>
SYNOPSIS	<pre>#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 (1 << 0) #define UDI_MEM_MOVABLE (1 << 1)</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>size is the number of bytes of space requested. See udi_limits_t on page 9-15 for limits on allocation sizes.</p> <p>flags is a bitmask of optional flags, which may include zero or more of the following:</p> <ul style="list-style-type: none"> UDI_MEM_NOZERO — Don’t zero memory contents. UDI_MEM_MOVABLE — Allocate movable memory. <p>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.</p>	
DESCRIPTION	<p>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.</p> <p>The newly allocated memory will be aligned on the most restrictive alignment of the platform’s natural alignments for <i>long</i> and pointer data types, allowing the allocated memory to be directly accessed as C structures.</p> <p>If the UDI_MEM_MOVABLE flag is set, the memory will be allocated as <i>movable</i> 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.</p>	
WARNINGS	<p>Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p>	

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.

REFERENCES

`udi_mem_free`, `udi_cancel`, `udi_limits_t`

NAME	udi_mem_free	<i>Free memory object</i>
SYNOPSIS	<pre>#include <udi.h> void udi_mem_free (void *target_mem);</pre>	
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.
	<hr/> <p>Note – The udi_init_context_t structure and the rest of the initial region data area are not movable memory, and must not be freed by the driver.</p> <hr/>	
REFERENCES	udi_mem_alloc	



12.1 Overview

The service calls in this chapter are used to manage *constraints objects* that are used to reflect drivers' constraints on buffer data and transfer properties. Constraints are used to indicate the data access capabilities and restrictions of the hardware and associated drivers as well as the various capabilities of software modules and drivers in the I/O handling path.

Constraints are first constructed in the environment by starting with a completely unconstrained set of constraints attributes, and are subjected to any restrictions imposed by the native bridge or any intervening bus bridges and then passed to the driver as its initial constraints object. Constraints may subsequently be set to more restrictive values by that driver or any other module in the datapath, including the environment, the bus and any bus bridges, and any children of the driver.

To provide this capability, UDI defines a set of services used to set the values of various constraints and a related set of services and channel operations used to propagate the constraints throughout the datapath. The constraints propagation includes the ability to update the constraints at any time due to hardware events (*e.g.* hot swap), reconfiguration events (hardware or software), or changes in software parameters.

12.1.1 Constraints Attributes

Constraints may be imposed from several sources and are specified in terms of *constraints attributes*. Some constraints attributes are general properties such as maximum transfer sizes and as such are defined here in the UDI Core Specification. Some attributes are associated with DMA operations and are therefore defined in the UDI Physical I/O Specification.

At the lowest level, constraints attributes indicate the host data access capabilities of the device. Any restrictions of the DMA engine on the device should be indicated by restricting one or more constraints attributes accordingly. For example, a device with only a 16-bit DMA address generation capability would indicate this by modifying the initial address-size constraints settings.

The constraints may then be used by all modules allocating memory or buffers that will be accessed by the device to ensure that the buffer or memory obtained will comply with the constraints. This is done by having the parent driver *propagate* the constraints settings to all children. The children then reference this constraints object when allocating any memory or buffers to allow the UDI environment to be aware of the usage of that allocated object. Drivers typically hold constraints objects for the duration of the driver instance's existence, to use whenever needed for buffer allocation or DMA operations. The driver must deallocate the constraints object to return the resources to the environment before the driver instance is destroyed (*i.e.* during the unbind process).

Constraints attributes may be used to indicate both software and hardware requirements and restrictions in the datapath. For example, a UDI adapter driver may set the transfer constraints in accordance with the maximum data buffer size that the hardware can handle. If an upper level module provides segmentation and re-assembly capabilities then it can export a much larger transfer constraint than was allowed by the lower level adapter driver.

The UDI environment need not adhere to constraints at resource allocation time. Instead, constraints may be ignored by the environment (if desired) up to such time as the constraint is applicable. A DMA constraint does not need to be explicitly adhered to until such time as a DMA mapping operation is performed. If the constraint is not honored at the time of initial resource allocation, the resource may need to be re-allocated at the time the constraint must be honored.

12.2 Constraints Attributes Service Calls and Structures

The functions in this section provide constraints attribute management services. These services include the ability to set attributes on a constraints object, to combine constraints objects by merging attributes, and to make copies of existing constraints objects.

Note that there is no function to allocate a new empty constraints object, since these are always allocated by the Management Agent in the environment and passed in to the driver. Instead, the driver is limited to obtaining copies of constraints objects for which it already holds handles; these multiple copies may then be individually modified and used separately, to specify different constraints for different child instances or different components of this driver instance.

NAME	udi_constraints_attr_t				<i>Constraints attribute type</i>
SYNOPSIS	#include <udi.h> typedef udi_ubit8_t udi_constraints_attr_t;				
DESCRIPTION	<p>This type is used to select a particular attribute of a constraints object. Constraints objects, referenced by constraints handles (see “Constraints Handle” on page 8-8), are used to constrain data and transfer properties to most optimally meet the requirements of all drivers handling the data.</p> <p>A constraints attribute is a mnemonic constant (UDI_XFER_XXX in the table below) which is used to set an associated value in a constraints handle. The valid range of associated values for a particular attribute is specified in the table below. Definitions of use and meaning of each attribute follow the table.</p> <p>Note - Additional constraints will be defined throughout the UDI specifications to facilitate the control and management of the associated resources. For example the <i>Buffer Management</i> chapter describes a set of constraints used for buffer allocation operations and the <i>UDI Physical I/O Specification</i> defines additional attributes which can be used by drivers that support DMA to constrain the physical attributes of memory passed to the device.</p>				
	Table 12-1 Transfer Constraints Attributes and Associated Value Ranges				
Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0	
	#define UDI_XFER_MAX 1				
0..2 ³² -1	0	1	0	no restriction	
	#define UDI_XFER_TYPICAL 2				
0..2 ³² -1	N/A	N/A	0	no typical pattern	
	#define UDI_XFER_GRANULARITY 3				
1..2 ³² -1	1	2 ³² -1	1	N/A	
	#define UDI_XFER_ONE_PIECE 4				
0..1	0	1	0	N/A	
	#define UDI_XFER_EXACT_SIZE 5				
0..1	0	1	0	N/A	
	#define UDI_XFER_NO_REORDER 6				
0..1	0	1	0	N/A	

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 not be related to the size of buffers used to pass the data.

UDI_XFER_ONE_PIECE is a flag indicating (if non-zero) 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 set, even if it is not.

UDI_XFER_EXACT_SIZE is a flag indicating (if non-zero) 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 non-zero) 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.

NAME	udi_constraints_attr_spec_t	<i>Specify attribute/value pair</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_constraints_attr_t attr_type; udi_ubit32_t attr_value; } udi_constraints_attr_spec_t;</pre>	
MEMBERS	<p>attr_type is the attribute being specified.</p> <p>attr_value is the value for the attribute.</p>	
DESCRIPTION	The <code>udi_constraints_attr_spec_t</code> structure is used to associate an attribute with its corresponding value. An array of these structures is used as an argument for the <code>udi_constraints_attr_set</code> operation to specify a list of attributes and their values to be set.	
REFERENCES	<code>udi_constraints_attr_t</code> , <code>udi_constraints_attr_set</code>	

NAME	udi_constraints_attr_set	<i>Set constraints attributes</i>
SYNOPSIS	<pre>#include <udi.h> void udi_constraints_attr_set (udi_constraints_attr_set_call_t *callback, udi_cb_t *gcb, udi_constraints_t constraints, const udi_constraints_attr_spec_t *attr_list, udi_ubit16_t list_length); typedef void udi_constraints_attr_set_call_t (udi_cb_t *gcb, udi_constraints_t new_constraints, udi_status_t status);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>constraints is a constraints handle for the constraints object to be modified.</p> <p>attr_list is the list of attributes and values to set. Once attr_list is passed into udi_constraints_attr_set the data area it points to cannot be written into by the driver until the corresponding callback is received. The memory pointed to by attr_list must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, attr_list must point to the beginning of the memory block.</p> <p>list_length is the number of valid entries in the attr_list array.</p> <p>new_constraints is a new constraints handle for the modified constraints object. This replaces the constraints handle.</p> <p>status is a UDI status code indicating the success or failure of the constraints modification operation.</p>	
DESCRIPTION	<p>udi_constraints_attr_set sets or changes one or more attribute values in the constraints object referenced by the constraints handle. The attr_list argument indicates this driver’s requirements for the specified constraints attributes.</p> <p>Most attributes have a defined range of least restrictive to most restrictive values, as specified with the description of the attribute. In these cases, udi_constraints_attr_set sets the new value for an attribute to be the <i>more</i> restrictive of the specified attr_list entry’s attr_value and the attribute’s current value.</p>	

For example, if the current value of UDI_XFER_MAX were 8192 and the **attr_list** entry's attr_value were 4096, the new value would be 4096; but if **attr_list** entry's attr_value were 32768, the attribute value would remain set to 8192, since 8192 is more restrictive than 32768 for UDI_XFER_MAX.

A few attributes have no defined sense of less or more restrictive. These are marked as “N/A” (not applicable). For these attributes, the new value is simply set to the value specified in the **attr_list** entry.

UDI_BUF_HEADER_SIZE and UDI_BUF_TRAILER_SIZE are special: they each cumulate the total value of all *udi_constraints_attr_set* calls for that parameter.

Once all attributes in the **attr_list** have been processed, the **callback** routine is invoked to notify the driver that the constraints object has been modified and the success or failure of that modification. The original handle, **constraints**, is consumed by the call; the driver must use **new_constraints** in its place.

The *udi_constraints_attr_set* operation may fail with **status** set to UDI_STAT_NOT_SUPPORTED if the requested attributes cannot be supported on a given system. The UDI_STAT_NOT_SUPPORTED error case is not intended to catch invalid attribute values. If attribute values are used that are outside the valid ranges documented for the attribute, the results are indeterminate.

If **status** indicates failure, the constraints object passed back in **new_constraints** shall have the same constraints attribute values as the original.

WARNINGS

Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “*Standard Calling Sequences*”.

STATUS VALUES

UDI_OK	successful modification of the constraints attributes
	UDI_STAT_NOT_SUPPORTED environment and/or platform cannot support the requested combination of attributes.

REFERENCES

udi_constraints_attr_spec_t,
udi_constraints_attr_reset

NAME	udi_constraints_attr_reset	<i>Reset a constraints attribute to default</i>
SYNOPSIS	<pre>#include <udi.h> void udi_constraints_attr_reset (udi_constraints_t constraints, udi_constraints_attr_t attr_type);</pre>	
ARGUMENTS	<p>constraints is a constraints handle for the constraints object to be modified.</p> <p>attr_type is the attribute to reset.</p>	
DESCRIPTION	<p>udi_constraints_attr_reset is used to reset a constraints attribute back to its default value (which is also usually the least restrictive). This is usually needed when a particular module provides special handling relative to the constraints attribute such that any restrictions imposed by parent or child drivers are not transferred through this driver.</p> <p>For example, a module that can segment and re-assemble buffers would use udi_constraints_attr_reset to set the UDI_XFER_MAX attribute to the default value before using udi_constraints_attr_set to restrict UDI_XFER_MAX to the largest buffer size which can be segmented. This would be done instead of simply propagating the parent driver's UDI_XFER_MAX constraint to the current module's children.</p> <p>If the udi_constraints_attr_reset operation were not used in this example sequence then if the UDI_XFER_MAX constraint of the parent modules were smaller than the buffer size supported by this module for segmentation the udi_constraints_attr_set would not be effective since the set value is less restrictive than the existing value.</p> <p><i>The upper-level driver in the above example is free to reset the UDI_XFER_MAX attribute only because it has taken on responsibility for applying its parents' constraints to the individual pieces that it passes on after segmenting the original buffers. Currently, the only way it can do this is by communicating each parent's maximum transfer size via a metalanguage-specific operation. [THIS IS EXPECTED TO BE CHANGED IN A LATER VERSION.]</i></p>	
REFERENCES	udi_constraints_attr_set, udi_constraints_attr_t	

NAME	udi_constraints_attr_combine	<i>Merge constraints attributes objects</i>
SYNOPSIS	<pre>#include <udi.h> void udi_constraints_attr_combine (udi_constraints_attr_combine_call_t *callback, udi_cb_t *gcb, udi_constraints_t src_constraints, udi_constraints_t dst_constraints, udi_ubit8_t attr_combine_method); typedef void udi_constraints_attr_combine_call_t (udi_cb_t *gcb, udi_constraints_t new_dst_constraints); #define UDI_ATTR_MOST_RESTRICTIVE 0 #define UDI_ATTR_LEAST_RESTRICTIVE 1</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>src_constraints is the handle for the first constraints object.</p> <p>dst_constraints is the handle for the other constraints object. This constraints object is consumed by this operation. If set to NULL_CONSTRAINTS, the src_constraints is simply copied into a new constraints object.</p> <p>attr_combine_method is used to select the type of combination algorithm to use. It may take one of the following values:</p> <ul style="list-style-type: none"> UDI_ATTR_MOST_RESTRICTIVE – Use the most restrictive of each pair of attribute values. UDI_ATTR_LEAST_RESTRICTIVE – Use the least restrictive of each pair of attribute values. <p>new_dst_constraints is the new constraints object handle whose attributes are the combination of the src_constraints and dst_constraints objects.</p>	
DESCRIPTION	<p>udi_constraints_attr_combine is used to obtain the merged combination of two individual constraints objects. This is most commonly done when a driver receives a udi_channel_event_ind operation indicating that a constraints propagation is being performed, to combine newly propagated constraints with any existing constraints.</p> <p>When combining the attributes, the specific algorithm used is controlled by the attr_combine_method parameter. Some attribute values may be incompatible and cannot be combined. In such cases, the original dst_constraints values will be preferred.</p>	

Although specific constraints management techniques will depend on the driver implementation, the commonly expected sequence of events that a driver will perform on receiving a constraints propagation indication are:

- 1) Combine the newly propagated constraints with any current constraints using `udi_constraints_attr_combine`.
- 2) Make individual adjustments to the newly combined constraints with `udi_constraints_attr_set` and/or `udi_constraints_attr_reset`.
- 3) Continue the constraints propagation operation by calling `udi_constraints_propagate` for any affected child or parent channel(s) in the direction of the propagation.
- 4) Clean up resources by deallocating `src_constraints` (if necessary) by calling `udi_constraints_free`.
- 5) Preserve the combined constraints objects in region data for use in any UDI service operation requiring a constraints object.

The constraints object represented by `src_constraints` may still be used after completion of this operation but the `dst_constraints` object is consumed and must not be used after calling this routine.

REFERENCES

`udi_constraints_t`, `udi_constraints_attr_set`

NAME	udi_constraints_free	<i>Free a constraints object</i>
SYNOPSIS	<pre>#include <udi.h> void udi_constraints_free (udi_constraints_t constraints);</pre>	
ARGUMENTS		<i>constraints</i> is a handle for the constraints object being deallocated.
DESCRIPTION		udi_constraints_free releases all resources associated with the constraints object. If constraints is equal to NULL_CONSTRAINTS, explicitly or implicitly (zeroed by initial value or by using udi_memset), this function acts as a no-op. Otherwise, constraints must have been allocated by udi_constraints_combine or passed to the driver via a channel operation.

12.3 Constraints Propagation Service Calls

Constraints must be communicated to the various modules in the datapath to be most effectively utilized. This is done in UDI by providing a method for constraints propagation. The most common need for constraints propagation is the boot time construction of the device tree wherein the datapath is built from the parent-most bus through the adapter driver and subsequent child modules to the OS interface. However, constraints may need to be re-propagated anytime there is an event which modifies the datapath (e.g. a hot-plug event or the introduction of a new parent or child connection to a multiplexing module).

Within UDI, the constraints propagation methodology is implemented using the following general algorithm:

1. The initiating driver updates the constraints object to indicate the modified constraints.
2. This driver calls `udi_constraints_propagate`, specifying the appropriate parent or child channel over which the constraints change is to be propagated.
3. The environment copies the initiator's constraints object, creating a duplicate copy for the selected driver, and passes it to that driver by initiating a `udi_channel_event_ind` operation on the destination channel with an event indication type of `UDI_CONSTRAINTS_CHANGED`.
4. The remaining steps may occur in any order:
 - a. The receiving driver internally preserves or merges the constraints object using the UDI services documented in this chapter, recursively propagating the new constraints along the datapath.
 - b. The environment acknowledges the constraints propagation by calling the `udi_constraints_propagate` callback operation, thereby returning the initiator's original constraints object.
 - c. The receiving driver may discard the `udi_channel_event_ind` control block received as part of the `udi_channel_event_ind` operation by calling `udi_cb_free` as documented in Chapter 20, “*Introduction to UDI Metalanguages*”.

In order to properly synchronize the constraints requirements when the drivers in the device tree are instantiated, it is required that `udi_constraints_propagate` be initiated by the parent driver after receiving the metalanguage-specific bind request from the child but before acknowledging that metalanguage-specific bind. In this way, the child will receive the `UDI_CONSTRAINTS_CHANGED` channel event and update its constraints accordingly before being enabled for operation by the receipt of the metalanguage-specific bind acknowledgment.

The functions in this section provide constraints propagation service calls. These calls are made from the initiator of a constraints propagation; the target module of a constraints propagation will receive a channel event operation as defined in Section 20.5 on page 20-4.

NAME	udi_constraints_propagate	<i>Constraints propagation request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_constraints_propagate (udi_constraints_propagate_call_t *callback, udi_cb_t *gcb, udi_channel_t target_channel, udi_constraints_t constraints);</pre> <pre>typedef void udi_constraints_propagate_call_t (udi_cb_t *gcb);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>target_channel is the channel over which the constraints modification is to be propagated.</p> <p>constraints is the constraints object to propagate to the target.</p>	
DESCRIPTION	<p>The <code>udi_constraints_propagate</code> operation is used to initiate the propagation of new constraints to a parent or child driver instance over the indicated channel. The environment shall create a copy of the constraints object and pass that newly created copy to the selected driver instance via a <code>udi_channel_event_ind</code> channel operation (see Section 20.5, “Channel Event Indication Operation”).</p> <p>Note that this is not a channel operation even though a channel handle is passed as one of the arguments.</p> <p>The constraints object should not be referenced or modified until ownership is returned to the initiating driver via the callback invocation.</p>	
REFERENCES	<code>udi_constraints_t</code> , <code>udi_channel_event_cb_t</code> , <code>udi_channel_event_ind</code>	



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.

Data in UDI buffers is transferred without regard to endianness.

13.2 Buffer Constraints

UDI buffer allocation is subject to various constraints specifications as described in Chapter 12, “*Constraints Management*”. This section describes additional constraints which are used specifically for buffer allocation operations.

NAME	udi_constraints_attr_t (Buffer)	<i>Buffer constraints attributes</i>																									
SYNOPSIS	<pre>#include <udi.h> typedef udi_ubit8_t udi_constraints_attr_t; /* Buffer constraints */ #define UDI_BUF_HEADER_SIZE 20 #define UDI_BUF_TRAILER_SIZE 21</pre>																										
DESCRIPTION	<p>This page lists additional attribute codes which can be used with the <code>udi_constraints_attr_t</code> type and <code>udi_constraints_attr_set</code> to modify buffer-related attributes of a constraints object.</p> <p>Constraints objects, referenced by constraints handles (<code>udi_constraints_t</code>), are used to constrain transfer properties as well as memory placement of <code>udi_buf_t</code> data and any related data storage characteristics. These constraints objects are passed to <code>udi_buf_copy</code> or <code>udi_buf_write</code> to allow the environment to create and manage the corresponding buffers so that they meet the specified constraints.</p> <p>A list of supported buffer constraints attribute codes is given below, along with the range of valid values for each attribute, which of these values are considered least and most restrictive, and the default value for the attribute. This is presented in the form of a table for each attribute category; each table is followed by detailed descriptions of each attribute.</p>																										
Table 13-1 Buffer Constraints Attributes and Associated Value Ranges																											
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center; padding-bottom: 5px;">Valid Range</th><th style="text-align: center; padding-bottom: 5px;">Least Restrictive Value</th><th style="text-align: center; padding-bottom: 5px;">Most Restrictive Value</th><th style="text-align: center; padding-bottom: 5px;">Default Value</th><th style="text-align: center; padding-bottom: 5px;">Special Case Behavior for 0</th></tr> </thead> <tbody> <tr> <td colspan="2" style="text-align: center; padding-top: 5px;">#define UDI_BUF_HEADER_SIZE 20</td><td></td><td></td><td></td></tr> <tr> <td style="text-align: center; padding-top: 5px;">0..$2^{16}-1$</td><td style="text-align: center; padding-top: 5px;">0</td><td style="text-align: center; padding-top: 5px;">$2^{16}-1$</td><td style="text-align: center; padding-top: 5px;">0</td><td style="text-align: center; padding-top: 5px;">N/A</td></tr> <tr> <td colspan="2" style="text-align: center; padding-top: 5px;">#define UDI_BUF_TRAILER_SIZE 21</td><td></td><td></td><td></td></tr> <tr> <td style="text-align: center; padding-top: 5px;">0..$2^{16}-1$</td><td style="text-align: center; padding-top: 5px;">0</td><td style="text-align: center; padding-top: 5px;">$2^{16}-1$</td><td style="text-align: center; padding-top: 5px;">0</td><td style="text-align: center; padding-top: 5px;">N/A</td></tr> </tbody> </table>		Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0	#define UDI_BUF_HEADER_SIZE 20					0.. $2^{16}-1$	0	$2^{16}-1$	0	N/A	#define UDI_BUF_TRAILER_SIZE 21					0.. $2^{16}-1$	0	$2^{16}-1$	0	N/A
Valid Range	Least Restrictive Value	Most Restrictive Value	Default Value	Special Case Behavior for 0																							
#define UDI_BUF_HEADER_SIZE 20																											
0.. $2^{16}-1$	0	$2^{16}-1$	0	N/A																							
#define UDI_BUF_TRAILER_SIZE 21																											
0.. $2^{16}-1$	0	$2^{16}-1$	0	N/A																							
<p>UDI_BUF_HEADER_SIZE is a hint provided to the environment that the current driver will typically add a header of the indicated size to <code>udi_buf_t</code> data buffers. Each call to <code>udi_constraints_attr_set</code> using this parameter adds to the total value, rather than the normal behavior for <code>udi_constraints_attr_set</code>. No guarantee is made that any portion of this space will be pre-allocated by the environment and it is not visible or available to the driver until actually made valid via a <code>udi_buf_copy</code> or <code>udi_buf_write</code> operation.</p> <p>UDI_BUF_TRAILER_SIZE is a hint which is used in a similar fashion as the UDI_BUF_HEADER_SIZE hint except that it is applied to a trailer region of bytes which will be added to the buffer. Note</p>																											

REFERENCES`udi_constraints_attr_t`

that some or all of the header or trailer size hints may be ignored, especially in the case where these hints would combine to create a very large buffer.

13.3 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 13.4 on page 13-10.

NAME	UDI_BUF_ALLOC	<i>Allocate and initialize a new buffer</i>
SYNOPSIS	<pre>#include <udi.h> #define \ UDI_BUF_ALLOC(\ callback, gcb, init_data, size, constraints) \ udi_buf_write(callback, gcb, init_data, \ size, NULL_BUF, 0, \ 0, constraints)</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>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.</p> <p>size is the initial size of the buffer data, in bytes.</p> <p>constraints is a non-null constraints handle used to ensure that the allocated buffer memory meets the needs of devices to which it may be passed.</p>	
DESCRIPTION	<p>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.</p> <p>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:</p> <pre>void UDI_BUF_ALLOC (udi_buf_write_call_t *callback, udi_cb_t *gcb, void *init_data, udi_size_t size, udi_constraints_t constraints);</pre> <pre>typedef void udi_buf_write_call_t (udi_cb_t *gcb, udi_buf_t new_buf);</pre>	
REFERENCES	udi_buf_write	

NAME	UDI_BUF_INSERT	<i>Insert bytes into a logical buffer</i>
SYNOPSIS	<pre>#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, NULL_CONSTRAINTS)</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>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.</p> <p>size is the number of bytes to insert into dst_buf.</p> <p>dst_buf is a handle to the logical buffer into which to insert bytes.</p> <p>dst_off is the logical offset from the first valid data byte in the buffer to the start of the insertion, in bytes.</p>	
DESCRIPTION	<p>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.</p> <p>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:</p> <pre>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);</pre> <pre>typedef void udi_buf_write_call_t (udi_cb_t *gcb, udi_buf_t new_dst_buf);</pre>	
REFERENCES	udi_buf_write	

NAME	UDI_BUF_DELETE	<i>Delete bytes from a logical buffer</i>
SYNOPSIS	<pre>#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, NULL_CONSTRAINTS)</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>size is the number of bytes to delete from dst_buf.</p> <p>dst_buf is a handle to the logical buffer from which to delete bytes.</p> <p>dst_off is the logical offset from the first valid data byte in the buffer to the start of the deletion, in bytes.</p>	
DESCRIPTION	<p>UDI_BUF_DELETE deletes size bytes from dst_buf starting at offset dst_off, logically moving any additional data “up” to fill the gap.</p> <p>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:</p> <pre>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);</pre>	
REFERENCES	<p>udi_buf_write</p>	

NAME	UDI_BUF_DUP	<i>Copy a logical buffer in its entirety</i>
SYNOPSIS	<pre>#include <udi.h> #define \ UDI_BUF_DUP(\ callback, gcb, src_buf, constraints) \ udi_buf_copy(callback, gcb, src_buf, \ 0, 0, NULL_BUF, 0, 0, \ constraints)</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>src_buf is a handle to the logical buffer to copy.</p> <p>constraints is a non-null constraints handle used to ensure that the allocated buffer memory meets the needs of devices to which it may be passed.</p>	
DESCRIPTION	<p>UDI_BUF_DUP makes a logical copy of src_buf and passes the new buffer to the driver with a callback.</p> <p>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:</p> <pre>void UDI_BUF_DUP (udi_buf_copy_call_t *callback, udi_cb_t *gcb, udi_buf_t src_buf, udi_constraints_t constraints);</pre> <pre>typedef void udi_buf_copy_call_t (udi_cb_t *gcb, udi_buf_t new_dst_buf);</pre>	
REFERENCES	<p>udi_buf_copy</p>	

13.4 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, to determine the current logical size of 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).

13.4.1 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 implementations 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 up the UDI chain with or without the buffer depending on the operation's success or failure. If the operation was a write operation and was successful, the UDI driver itself will typically deallocate the buffer directly to avoid any overhead associated with passing the buffer back to the upper level modules. However, on failures, the UDI driver will typically pass the buffer back to the

requester along with the failure indication; the requester can then choose to retransmit the buffer or perform other operations with it depending on the implementation of that requesting module.

- 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 wished to implement a retransmit algorithm based on timers or remote acknowledgements, it created 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 acceptable 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.

13.4.2 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 by the region cleanup code and an associated UDI event indication on the associated channel (see Section 20.5, “Channel Event Indication Operation”). In this situation, the UDI environment will return any buffers held by the parent region to the child as part of the region cleanup operations. The metalanguage specification will indicate which buffers are handled in this manner.

NAME	udi_buf_copy	<i>Copy data from one logical buffer to another</i>
SYNOPSIS	<pre>#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_constraints_t constraints);</pre> <pre>typedef void udi_buf_copy_call_t (udi_cb_t *gcb, udi_buf_t new_dst_buf);</pre>	
ARGUMENTS	<p>callback, gcb are the standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”</p> <p>src_buf is a handle to buffer containing data to be copied. This must not be set to NULL_BUF.</p> <p>src_off is the offset, in bytes, from the first valid data byte to the start of the copy area in the source buffer:</p> $0 \leq \text{src_off} < \text{valid source buffer data length}$ <p>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 end of the valid data.</p> $0 < \text{src_len} \leq (\text{valid source buffer data length} - \text{src_off})$ <p>For src_len of zero, use udi_buf_write instead.</p> <p>dst_buf is a handle to the buffer that is the target of the data copy. If set to NULL_BUF, a new, empty buffer will be allocated before copying the source data.</p> <p>dst_off is the offset, in bytes, from first valid data byte to the start of the copy area in the destination buffer. The buffer will be extended if necessary to accommodate the data.</p> $0 \leq \text{dst_off} \leq \text{valid destination buffer data length}$ <p>dst_len is the number of bytes in dst_buf to be replaced with data copied from the source buffer.</p> $0 \leq \text{dst_len} \leq (\text{valid destination buffer data length} - \text{dst_off})$ <p>Note that if dst_buf is NULL_BUF both dst_off and dst_len must be zero.</p>	

	<p>constraints is a handle to the transfer constraints object to be used if a new buffer must be allocated. If dst_buf is not NULL_BUF on entry, its existing constraints will continue to be used and this parameter must be set to NULL_CONSTRAINTS; otherwise it must be non-null. See also Section 8.6.1.3, “Constraints Handle,” on page 8-8.</p> <p>new_dst_buf New handle to modified destination buffer.</p>																														
DESCRIPTION	<p>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.</p>																														
	<p>Table 13-2 Common actions for udi_buf_copy/udi_buf_write arguments</p> <table border="1"> <thead> <tr> <th>Action</th><th>src_buf/src_mem</th><th>src_len</th><th>dst_buf</th><th>dst_len</th></tr> </thead> <tbody> <tr> <td>Allocate/initialize</td><td>non-null</td><td>N</td><td>NULL_BUF</td><td>0</td></tr> <tr> <td>Overwrite</td><td>non-null</td><td>N</td><td>non-null</td><td>N</td></tr> <tr> <td>Delete</td><td>NULL_BUF/NULL</td><td>0</td><td>non-null</td><td>N</td></tr> <tr> <td>Insert</td><td>non-null</td><td>N</td><td>non-null</td><td>0</td></tr> <tr> <td>Ensure space</td><td>NULL_BUF/NULL</td><td>N</td><td>NULL_BUF</td><td>0</td></tr> </tbody> </table> <p>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.)</p> <p>This routine is very similar to udi_buf_write, except that the data source is another buffer, rather than a virtually-contiguous data structure.</p> <p>If dst_buf is NULL, a new buffer will be allocated to hold the data. In this case, the UDI_BUF_HEADER_SIZE constraints attribute supplies a hint for the number of bytes of extra space to allocate at the beginning of the buffer outside of the valid data (since additional data is expected to be inserted at the front of the buffer at a later point). Similarly, UDI_BUF_TRAILER_SIZE supplies a hint for extra space at the end of the buffer.</p> <p>If there is insufficient space in the destination buffer for copying the data, this routine will allocate sufficient new or additional buffer memory to complete the request.</p> <p>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.</p> <p>WARNINGS</p> <p>Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p>	Action	src_buf/src_mem	src_len	dst_buf	dst_len	Allocate/initialize	non-null	N	NULL_BUF	0	Overwrite	non-null	N	non-null	N	Delete	NULL_BUF/NULL	0	non-null	N	Insert	non-null	N	non-null	0	Ensure space	NULL_BUF/NULL	N	NULL_BUF	0
Action	src_buf/src_mem	src_len	dst_buf	dst_len																											
Allocate/initialize	non-null	N	NULL_BUF	0																											
Overwrite	non-null	N	non-null	N																											
Delete	NULL_BUF/NULL	0	non-null	N																											
Insert	non-null	N	non-null	0																											
Ensure space	NULL_BUF/NULL	N	NULL_BUF	0																											

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_BUF; ***new_dst_buf*** may return the same handle value as the input value of ***dst_buf***.

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_read	<i>Read data bytes from a logical buffer</i>
SYNOPSIS	<pre>#include <udi.h> udi_size_t udi_buf_read (udi_buf_t src_buf, udi_size_t src_off, udi_size_t src_len, void *dst_mem);</pre>	
ARGUMENTS	<p>src_buf is a handle to a buffer containing data to be read.</p> <p>src_off is the offset, in bytes, into the valid data of src_buf at which to start reading data. src_off must be \leq src_buf's valid data size.</p> <p>src_len The number of bytes to be read from src_buf. If src_off+src_len extends beyond src_buf's valid data, only the bytes up to the end of valid data will be read and copied into dst_mem.</p> <p>dst_mem pointer to caller's memory where data is to be copied.</p>	
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.	
RETURN VALUES	The number of bytes actually read from src_buf is returned to the caller.	
REFERENCES	udi_buf_get_size	

NAME	udi_buf_write	<i>Write data bytes into a logical buffer</i>
SYNOPSIS	<pre>#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_constraints_t constraints);</pre> <pre>typedef void udi_buf_write_call_t (udi_cb_t *gcb, udi_buf_t new_dst_buf);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>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. Once src_mem is passed into udi_buf_write the data area it points to cannot be written into by the driver until the corresponding callback is received. The memory pointed to by src_mem must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, src_mem must point to the beginning of the memory block.</p> <p>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.</p> <p>dst_buf are the same arguments as used in udi_buf_copy.</p> <p>dst_off</p> <p>dst_len</p> <p>constraints</p> <p>new_dst_buf</p>	
DESCRIPTION	<p>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.</p>	

WARNINGS

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.

REFERENCES

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 “*Standard Calling Sequences*”.

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).

udi_buf_copy, **udi_cancel**

NAME	udi_buf_free	<i>Free a logical buffer</i>
SYNOPSIS	#include <udi.h> void udi_buf_free (udi_buf_t buf);	
ARGUMENTS	buf is a handle for the buffer being deallocated. If the handle is NULL_BUF 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_BUF, 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	

NAME	udi_buf_get_size	<i>Get the valid data size of a UDI buffer</i>
SYNOPSIS	#include <udi.h> udi_size_t udi_buf_get_size (udi_buf_t buf);	
ARGUMENTS	buf	Handle to buffer being examined.
DESCRIPTION	udi_buf_get_size is used to obtain the number of valid data bytes in buf . As described in the udi_buf_copy routine (and elsewhere), data bytes that have been deleted from the logical buffer may or may not be still present in the internal data structures of the buffer. This function does <i>not</i> include such deleted portions of data in the return value, nor does it include pre-reserved header or trailer space that is not yet part of the valid data.	
RETURN VALUES	The number of valid data bytes in buf is returned to the caller.	

13.5 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.

13.5.1 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	<i>Buffer tag type</i>
SYNOPSIS	<pre>#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 (1<<0) /* Update Category Tag Types */ #define UDI_BUFTAG_SET_iBE16_CHECKSUM (1<<8) #define UDI_BUFTAG_SET_TCP_CHECKSUM (1<<9) #define UDI_BUFTAG_SET_UDP_CHECKSUM (1<<10) /* Status Category Tag Types */ #define UDI_BUFTAG_TCP_CKSUM_GOOD (1<<17) #define UDI_BUFTAG_UDP_CKSUM_GOOD (1<<18) #define UDI_BUFTAG_IP_CKSUM_GOOD (1<<19) #define UDI_BUFTAG_TCP_CKSUM_BAD (1<<21) #define UDI_BUFTAG_UDP_CKSUM_BAD (1<<22) #define UDI_BUFTAG_IP_CKSUM_BAD (1<<23) /* Drivers Category Tag Types */ #define UDI_BUFTAG_DRIVER1 (1<<24) #define UDI_BUFTAG_DRIVER2 (1<<25) #define UDI_BUFTAG_DRIVER3 (1<<26) #define UDI_BUFTAG_DRIVER4 (1<<27) #define UDI_BUFTAG_DRIVER5 (1<<28) #define UDI_BUFTAG_DRIVER6 (1<<29) #define UDI_BUFTAG_DRIVER7 (1<<30) #define UDI_BUFTAG_DRIVER8 (1<<31)</pre>	
DESCRIPTION	<p>The <i>udi_tagtype_t</i> 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.</p> <p>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 should be stored in driver-endianness regardless of the endianness of the buffer data.</p>	

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 should 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.

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.

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.

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.

REFERENCES

udi_buf_tag_t

NAME	udi_buf_tag_t	<i>Buffer tag structure</i>					
SYNOPSIS	<pre>#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;</pre>						
MEMBERS	<p>tag_type is the type of tag represented by this tag structure. Although <code>udi_tagtype_t</code> is a bitmask type only one tag type may be specified in the <code>udi_buf_tag_t</code> structure (i.e. only one bit may be set).</p> <p>tag_value is the value associated with this tag.</p> <p>tag_off is the starting buffer data offset for which the tag applies.</p> <p>tag_len is the length of data (in bytes) for which the tag applies.</p>						
DESCRIPTION	<p>The <code>udi_buf_tag_t</code> 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.</p>						
Table 13-3 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 = $(\text{tag_len}+1)/2$)				
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				
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				

Table 13-3 Tag structure field usage

tag_type UDI_BUFTAG_xxx	tag_value	tag_off	tag_len
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	unused	start of TCP header	total byte length of TCP header and TCP data
UDP_CKSUM_GOOD	unused	start of UDP header	total byte length of UDP header and UDP data
IP_CKSUM_GOOD	unused	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...DRIVER9	driver-defined	driver-defined	driver-defined

REFERENCES*udi_buf_tag_set, udi_buf_tag_get*

NAME	udi_buf_tag_set	<i>Sets a tag for a portion of buffer data</i>
SYNOPSIS	<pre>#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_t tag);</pre> <pre>typedef void udi_buf_tag_set_call_t (udi_cb_t *gcb, udi_buf_t new_buf);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>buf is the buffer for which the new tag is to be set.</p> <p>tag specifies the tag information being set.</p> <p>new_buf is the handle to the buffer with the new tag value set.</p>	
DESCRIPTION	<p>The udi_buf_tag_set operation is used to set a tag for the associated buffer. The tag information is specified in the tag structure argument. If a tag of the same type, offset, and length already exists, its value will be replaced with the new tag.tag_value. Otherwise, a new tag will be created for the buffer with information from the tag structure.</p> <p>The range specified by the tag offset and length must consist entirely of valid data.</p>	
WARNINGS	<p>Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>On successful completion, buf will no longer be valid and new_buf must be used instead.</p>	
REFERENCES	udi_buf_tag_t, udi_buf_tag_get	

NAME	udi_buf_tag_get	<i>Gets one or more tags from a buffer</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS	<p>buf is the buffer for which the tag information is to be returned</p> <p>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.</p> <p>tag_array is a pointer to an array of <code>udi_buf_tag_t</code> structures that are to be filled in with the obtained tag information.</p> <p>tag_array_length is the number of entries that may be written to tag_array.</p> <p>tag_start_idx is the number of tags of the specified type to skip before returning tag information.</p>	
DESCRIPTION	<p>The <code>udi_buf_tag_get</code> 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.</p> <p>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.</p>	
REFERENCES	udi_buf_tag_t, udi_buf_tag_set	

13.5.2 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	<i>Compute values from tagged buffer data</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS	<p>buf is the buffer for which the tag value is to be computed.</p> <p>off is the offset into the buffer at which the computation is to begin. The offset specified must point to valid buffer data.</p> <p>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.</p> <p>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).</p>	
DESCRIPTION	<p>The <i>udi_buf_tag_compute</i> 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.</p> <p>The buffer range specified must consist entirely of valid data bytes.</p> <p>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.</p> <p>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.</p>	
RETURN VALUE	The computed tag value.	
REFERENCES	<i>udi_buf_tag_apply</i> , <i>udi_buf_tag_get</i>	

NAME	udi_buf_tag_apply	<i>Apply modifications to tagged buffer data</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>buf is the buffer for which tag values are to be computed and set.</p> <p>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.</p> <p>new_buf is the buffer returned to the caller after tags have been computed and have been written into the buffer.</p>	
DESCRIPTION	<p>The <code>udi_buf_tag_apply</code> 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).</p> <p>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 <code>udi_buf_tag_compute</code>) 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 <code>udi_buf_tag_apply</code> should be made with the required sequence of tag types.</p> <p>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.</p>	
WARNINGS	Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.	

On successful completion, **buf** will no longer be valid and **new_buf** is substituted, even if **buf** was not specified as NULL_BUF; **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.

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	<i>Time value structure</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_ubit32_t seconds; udi_ubit32_t nanoseconds; } udi_time_t;</pre>	
MEMBERS	<p>seconds is the number of seconds of time.</p> <p>nanoseconds is the number of nanoseconds into the current second. Thus nanoseconds ranges from zero to one less than one thousand million (10^9) nanoseconds.</p>	
DESCRIPTION	<p>The <i>udi_time_t</i> 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 <i>udi_limits_t</i> values should be consulted to determine the actual granularity of the environment's timers, as all specified <i>udi_time_t</i> values will be rounded up to integral multiples of the minimum system timer resolution.</p> <p>This structure is not used to represent absolute ("wall-clock") times. UDI provides no facility to determine absolute time.</p>	
REFERENCES	udi_limits_t	

NAME	udi_timer_start	<i>Start a callback timer</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>interval is the desired minimum interval that should elapse between the time the event is initiated with <code>udi_timer_start</code> 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.</p>	
DESCRIPTION	<p><code>udi_timer_start</code> 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.</p> <p>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 <code>udi_cb_alloc</code> in order to be able to dedicate it to this purpose.</p> <p>A <code>udi_timer_start</code> request may be cancelled at any time by calling the <code>udi_cancel</code> routine with the original control block. For more information regarding cancelling asynchronous service calls, see the definition of <code>udi_cancel</code> on page 10-9.</p>	
WARNINGS	Control block usage must follow the rules described in the “Asynchronous Service Calls” section of “ <i>Standard Calling Sequences</i> ”.	
REFERENCES	<code>udi_time_t</code> , <code>udi_limits_t</code> , <code>udi_cb_alloc</code> , <code>udi_cancel</code>	

NAME	udi_timer_start_repeating	<i>Start a repeating timer</i>
SYNOPSIS	<pre>#include <udi.h> 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 (udi_cb_t *gcb, udi_ubit32_t nmissed);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>interval is the repeating period for this timer (see <i>udi_timer_start</i>).</p> <p>nmissed is the number of timeout callbacks missed.</p>	
DESCRIPTION	<p><i>udi_timer_start_repeating</i> behaves just like <i>udi_timer_start</i> except that the callback routine is called repeatedly at each successive occurrence of interval. The interval argument specifies the period of this repeating timer.</p> <p>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.</p> <p>The repeating timer can be stopped by calling <i>udi_cancel</i> from either the callback timeout routine or from other code within the region that started the timer with the original control block. For more information regarding cancel operations, please see the description of the <i>udi_cancel</i> routine.</p>	
WARNINGS	<p>Using a control block with a repeating interval timer is an exception to the normal rules of control block usage (wherein control block ownership is transferred out of the driver when used in an interface operation and returned to the driver in the callback). In the repeating timer case, the control block is passed to the environment on the <i>udi_timer_start_repeating</i> call. Each time the <i>udi_timer_tick_call_t</i> function is invoked the ownership of the control block is explicitly <i>not</i> transferred back to the driver. The only valid operations that the <i>udi_timer_tick_call_t</i> function can perform on the gcb are (1) read or write accesses to the visible parts of the control block, and (2) passing the control block as the argument to <i>udi_cancel</i>. These operations are guaranteed to be serialized since they are executing in the driver’s region.</p>	
REFERENCES	<p><i>udi_time_t</i>, <i>udi_limits_t</i>, <i>udi_cb_alloc</i>, <i>udi_cancel</i></p>	

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 8.6.2.1, “Timestamp Type,” on page 8-9.

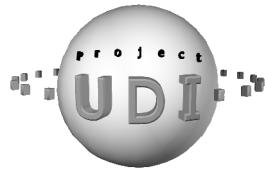
NAME	udi_time_current	<i>Return indication of the current relative time</i>
SYNOPSIS	<pre>#include <udi.h> udi_timestamp_t udi_time_current (void);</pre>	
DESCRIPTION		<p>udi_time_current returns the current time (relative to some arbitrary starting point), in implementation-specific units. The system time resolution can be determined from the min_curtme_res field in the udi_limits_t structure.</p> <p>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.</p> <p>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.</p>
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.
REFERENCES		<p>udi_time_t, udi_limits_t, udi_time_since, udi_time_between</p> <p>See also Section 8.6.2.1, “Timestamp Type,” on page 8-9.</p>

NAME	udi_time_between	<i>Return time interval between two points</i>
SYNOPSIS	<pre>#include <udi.h> udi_time_t udi_time_between (udi_timestamp_t start_time, udi_timestamp_t end_time);</pre>	
ARGUMENTS	<p>start_time is a timestamp value marking the starting point of the interval.</p> <p>end_time is a timestamp value marking the ending point of the interval.</p>	
DESCRIPTION	<p><code>udi_time_between</code> returns the time delta between two previously recorded times, in start_time and end_time. The previously recorded times must have been obtained via <code>udi_time_current</code>.</p> <p>start_time must reflect a time that occurred no later than end_time.</p> <p>The system time resolution can be determined from the min_curtime_res field in the <code>udi_limits_t</code> structure.</p>	
RETURN VALUES	The time interval, in seconds and nanoseconds, is returned to the caller.	
REFERENCES	<code>udi_time_t</code> , <code>udi_limits_t</code> , <code>udi_time_current</code> , <code>udi_time_since</code>	

NAME	udi_time_since	<i>Return time interval since a starting point</i>
SYNOPSIS	<pre>#include <udi.h> udi_time_t udi_time_since (udi_timestamp_t start_time);</pre>	
ARGUMENTS		<i>start_time</i> 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 <i>start_time</i> , 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 <i>min_curtime_res</i> field in the udi_limits_t structure.
		udi_time_since is equivalent to:
		udi_time_between(<i>start_time</i> , udi_time_current())
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

Time Mgmt



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. 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 63 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 dollar-sign ('\$'), a caret ('^'), an at sign ('@'), an exclamation point ('!'), or a hash character ('#'); these 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.

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 five principle classes of driver instance attributes:

1. Instance-private attributes
2. Enumeration attributes
3. Sibling group attributes
4. Parent visible attributes
5. Message string 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. This is the most common class of attributes and can be used for any driver-related information.

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 and are identified by the exclamation point ('!') prefix; the enumeration attributes are set atomically (i.e. none of the attributes can be read or changed until all of the enumeration attributes 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.

15.4.2.1 Locator Enumeration Attribute

All device instances enumerated must have a *!locator* attribute. This attribute provides a textual representation that can be used to refer to that driver instance in the current environment for system administration purposes. The *!locator* attributes of driver instances in an I/O path can be concatenated together to provide an indication to the system administrator or users of the hierarchical location of the final driver instance in the device node tree. Each metalanguage defines the method for constructing the *!locator* attribute value for child nodes of that metalanguage.

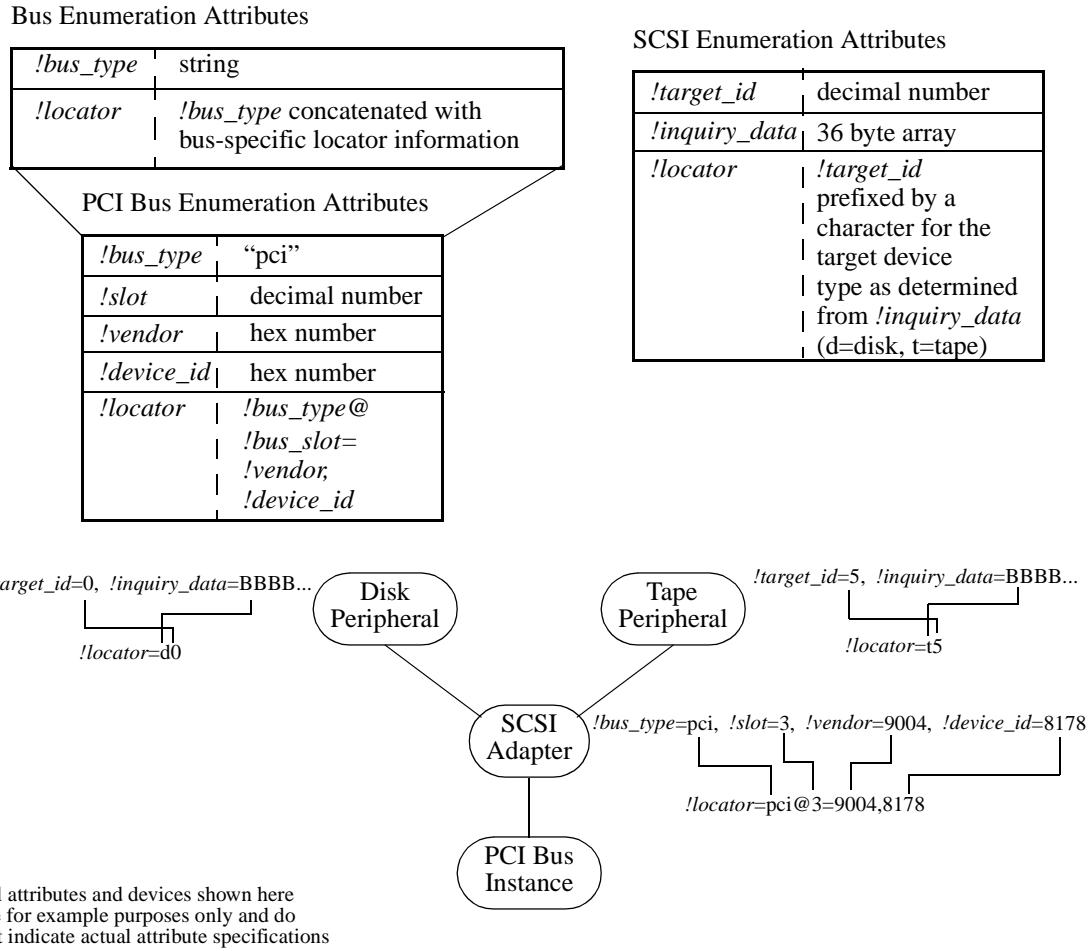
The information used to build a *!locator* attribute is usually a string-format combination of other enumeration attributes; the *!locator* attribute value is not required to be a strict string conversion of those values but there must not be information in the locator attribute that is not also provided in a specific enumeration attribute.

15.4.2.1.1 Locator Enumeration Attribute Example

Using the example device configuration and enumeration attributes shown in Figure 15-1 on page 15-3, a system might represent the devices shown to the System Administrator in the following fashion:

- Disk Device: /pci@3=9004,8178/d0
- Tape Device: /pci@3=9004,8178/t5

Figure 15-1 Example Enumeration Attributes and Locators



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 the caret 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 and written only by that instance's parent (and 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 attributes are identified by attribute names that begin with the at-sign character (@).

Parent visible attributes are persistent.

15.4.5 Message String Attributes

Message string attributes are special attributes of type UDI_ATTR_STRING that are used to obtain a locale-specific string from the driver's static driver properties or message files (see Section 28.6.9, "Message Declaration," on page 28-17). The locale setting for the current system determines which version of the corresponding message is used. These attributes are read-only for the driver and are provided for convenience in formatting tracing or logging entries or for other udi_snprintf-related operations.

The attribute name for a message string attribute consists of the hash character (#) followed by the message number as an ASCII-encoded decimal string. Any leading zeros in the number are ignored when comparing against message numbers in "message" property declarations.

Parent visible attributes are persistent.

15.4.6 Attribute Classification

Table 15-1 Instance Attribute Classification Table

Attribute Type	Prefix	Persistent?	Read-only?	Visible to Self?	Visible to Siblings?	Visible to Parent?
Private Persistent		✓		✓		
Private Volatile	\$			✓		
Enumeration	!		✓	✓		Specified by but not visible to.
Sibling Group	^			✓	✓	
Parent Visible	@	✓				✓
Message String	#	✓	✓	✓		

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	<i>Instance attribute data-type type</i>
SYNOPSIS	<pre>#include <udi.h> typedef udi_ubit8_t udi_instance_attr_type_t; /* Instance Attribute Types */ #define UDI_ATTR_NONE 0x0 #define UDI_ATTR_STRING 0x1 #define UDI_ATTR_ARRAY8 0x2 #define UDI_ATTR_UBIT32 0x3 #define UDI_ATTR_BOOLEAN 0x4 #define UDI_ATTR_FILE 0x5</pre>	
DESCRIPTION	<p>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.</p> <p>A list of supported instance attribute data type codes is given below, along with a description of each attribute.</p> <p>UDI_ATTR_NONE indicates that an attribute has no current value. This type is only legal with an attribute length of zero.</p> <p>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.</p> <p>UDI_ATTR_ARRAY8 identifies a sequence of <code>udi_ubit8_t</code> values.</p> <p>UDI_ATTR_UBIT32 identifies a single <code>udi_ubit32_t</code> value. The attribute length for attributes of this type must be exactly <code>sizeof(udi_ubit32_t)</code>.</p> <p>UDI_ATTR_BOOLEAN identifies a single <code>udi_boolean_t</code> value. The attribute length for attributes of this type must be exactly <code>sizeof(udi_boolean_t)</code>.</p> <p>UDI_ATTR_FILE identifies a read-only attribute whose value is contained in a driver-provided external file. The attribute name must match a “<code>readable_file</code>” entry in the driver’s persistent configuration information, optionally suffixed with a colon (‘<code>:</code>’) 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.</p>	

Instance Attributes *udi_instance_attr_get*

NAME	udi_instance_attr_get	<i>Read an attribute value for a driver instance</i>
SYNOPSIS	<pre>#include <udi.h> void udi_instance_attr_get (udi_instance_attr_get_call_t *callback, udi_cb_t *gcb, const char *attr_name, const void *enumeration_context, 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);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>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. Once attr_name is passed into udi_instance_attr_get the data area it points to must not be written into by the driver until the corresponding callback is received. The memory pointed to by attr_name must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, attr_name must point to the beginning of the memory block.</p> <p>enumeration_context is the enumeration context associated with the specific child instance for which this attribute has been set if it is a parent visible attribute (prefix character ‘@’).</p> <p>This argument must be non-null for parent visible attributes; it is ignored for other types of attributes.</p> <p>attr_value is a pointer to a memory area to receive the attribute value. Once attr_value is passed into udi_instance_attr_get the data area it points to must not be read or written by the driver until the corresponding callback is received. The memory pointed to by attr_value must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, attr_value must point to the beginning of the memory block.</p>	

DESCRIPTION

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 15-6 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.

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.

The **attr_name** read may be either persistent or volatile depending on the type of attribute (as indicated by the prefix character for the attribute).

WARNINGS

Multiple *udi_instance_attr_get* calls using UDI_ATTR_FILE to get the contents of a readable file are not guaranteed to be atomic: the file contents may change between operations.

REFERENCES

udi_instance_attr_type_t, *udi_instance_attr_set*

Instance Attributes

udi_instance_attr_set

NAME	udi_instance_attr_set	<i>Set a driver instance attribute value</i>
SYNOPSIS	<pre>#include <udi.h> void udi_instance_attr_set { udi_instance_attr_set_call_t *callback, udi_cb_t *gcb, const char *attr_name, const void *enumeration_context, 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);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>attr_name is the name of the attribute whose value is to be set. Once attr_name is passed into udi_instance_attr_get the data area it points to cannot be written into by the driver until the corresponding callback is received. The memory pointed to by attr_name must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, attr_name must point to the beginning of the memory block.</p> <p>enumeration_context is the enumeration context associated with the specific child instance for which this attribute is to be set if it is a parent visible attribute (prefix character ‘@’).</p> <p>This argument must be non-null for parent visible attributes; it is ignored for other types of attributes.</p> <p>attr_value is a pointer to the attribute value to set. attr_value must be NULL if and only if attr_length is 0; otherwise, once attr_value is passed into udi_instance_attr_get the data area it points to cannot be written into by the driver until the corresponding callback is received. The memory pointed to by attr_value must be either movable memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the gcb’s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, attr_value must point to the beginning of the memory block.</p> <p>attr_length is the length of the value pointed to by attr_value.</p>	

DESCRIPTION

attr_type is the type specifier for the attribute value. See *udi_instance_attr_type_t* on page 15-6 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.

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 may not be used with the UDI_ATTR_FILE attribute type.

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_set, *UDI_INSTANCE_ATTR_DELETE*,
udi_limits_t

Instance Attributes **UDI_INSTANCE_ATTR_DELETE**

NAME	UDI_INSTANCE_ATTR_DELETE	<i>Driver instance attribute delete macro</i>
SYNOPSIS	<pre>#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);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>attr_name is the name of the attribute to delete.</p>	
DESCRIPTION	<p>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 <code>udi_instance_attr_set</code> service call and sets the attr_length parameter to zero to effect the deletion of the corresponding attribute.</p> <p>The callback function specified for this macro should be of the <code>udi_instance_attr_set_call_t</code> type.</p>	
REFERENCES	<code>udi_instance_attr_set</code>	



16.1 Overview

The Inter-Module Communication (IMC) services allow drivers to create *regions* and *channels*, to anchor channels within a region, to dynamically set the channel context, and to close a channel. Regions and channels are defined in Chapter 4, “*Execution Model*”. See that chapter for a general understanding of these concepts.

16.2 Service Calls

This section defines a service call which allows the driver to dynamically create a region, service calls which allow the driver to create and anchor channels, a call to dynamically set the channel’s context, and a call to close a channel.

Note that the initial 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 from its `init_module` routine. Additionally, secondary regions which have been prepared via calls to `udi_secondary_region_init` are automatically created by the Management Agent for each driver instance, along with an initial channel between the primary region and each such secondary region.

Channels and regions are finite system resources. It is a responsibility of a device driver to allocate, track and return these objects back to the environment in a responsible manner. Only those channels and regions explicitly created by the driver, via calls to the services in this chapter, can be freed by the driver.

NAME	udi_secondary_region_create	<i>Create a secondary region and its first channel</i>
SYNOPSIS	<pre>#include <udi.h> void udi_secondary_region_create (udi_secondary_region_create_call_t *callback, udi_cb_t *gcb, udi_index_t region_idx); typedef void udi_secondary_region_create_call_t (udi_cb_t *gcb, udi_channel_t channel_to_secondary);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”.</p> <p>region_idx is a non-zero driver-dependent index value that indirectly identifies a late-bound set of platform-dependent properties that will be attached to the new region (address and capability domains, memory residence, priority, etc.) for the region being created. These properties are derived from the driver’s region attributes (see Section 28.6.6, “Region Declaration,” on page 28-14). 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). A region index value of zero is reserved for the primary region of a driver instance.</p> <p>channel_to_secondary is a channel handle to the loose end of a channel to the new secondary region. The new region gets a handle to its (anchored) end of this channel via its <code>udi_init_context_t</code> structure.</p>	
DESCRIPTION	<p><code>udi_secondary_region_create</code> allocates and initializes the resources necessary to create a new region and its first channel, based upon the region_idx parameter. Note that the new region may be in a different address space or capability domain than the region that makes this call to create the new region. Therefore, all region-related resources must be allocated in a bootstrap fashion within the new region itself.</p> <p>The region_idx must match a region index passed to a prior call to <code>udi_secondary_region_prepare</code> from the caller module’s <code>init_module</code> routine. The ops index and other parameters passed to <code>udi_secondary_region_prepare</code> will be used to create the new region and channel. UDI initializes the new region’s first channel’s ops vector with the set of entry points identified by the ops index. Typically this will be for an ops vector type from a custom metalanguage used to communicate between regions internal to a driver.</p>	

When the new region has been fully initialized, UDI passes a handle for the original region's end of the new channel to the caller of `udi_secondary_region_create` via the **callback**. This channel endpoint is initially loose, and must be anchored using `udi_channel_anchor`.

The new region's region data will contain a `udi_init_context_t` structure, followed by as much extra space as was requested by `udi_secondary_region_prepare`. The bytes following the context structure will all be zero. The **first_channel**'s context will point to this `udi_init_context_t`.

As a recommended and expected (but not required) convention in the driver-internal interface definition, the entity that created the region should make some form of initialization call to the new region's first channel. This call is intended to pass parameters that will help the region choose structure sizes, initialize fields, etc., so that it can become ready to be "open for business".

REFERENCES

`udi_cancel`, `udi_secondary_region_prepare`,
`udi_init_context_t`

NAME	udi_channel_anchor	<i>Anchor a channel to the current region</i>
SYNOPSIS	<pre>#include <udi.h> 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);</pre> <pre>typedef void udi_channel_anchor_call_t (udi_cb_t *gcb, udi_channel_t anchored_channel);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”.</p> <p>channel is the channel handle for the loose end to be anchored. Once <code>udi_channel_anchor</code> is called, the driver may no longer use this handle.</p> <p>ops_idx is an ops index for the ops vector that the driver wants to associate with the specified channel, as passed to the appropriate <code><<meta>>_<<role>>_ops_init</code> at <code>init_module</code> time. ops_idx must be non-zero.</p> <p>channel_context is a channel context pointer to be associated with the anchored channel endpoint.</p> <p>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 <code>udi_channel_anchor</code>.</p>	
DESCRIPTION	<p><code>udi_channel_anchor</code> 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 <code>udi_channel_spawn</code> request. A primary region’s bind channel may also be loose, if the driver did not specify parent or child binding ops when calling <code>udi_primary_region_init</code>.</p> <p>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.</p> <p>Loose ends may be passed between regions as parameters to channel operations. Anchored ends may not.</p> <p>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.</p>	

Once both ends of the channel are anchored, the channel may be used for communication, by invoking channel operations. It is the drivers' responsibility to 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_cancel`, `udi_channel_spawn`,
`udi_primary_region_init`

NAME	udi_channel_spawn	<i>Spawn a new channel with loose ends.</i>
SYNOPSIS	<pre>#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);</pre> <pre>typedef void udi_channel_spawn_call_t (udi_cb_t *gcb, udi_channel_t new_channel);</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “<i>Standard Calling Sequences</i>”</p> <p>channel is the channel handle for an existing anchored channel. The new channel will be spawned relative to this channel.</p> <p>spawn_idx is a small integer which allows the environment to match two spawn requests (one from each end of the channel) together.</p> <p>ops_idx is an ops index for the ops vector that the driver wants to associate with the specified channel, as passed to the appropriate <<meta>>_<<role>>_ops_init at init_module time, or zero.</p> <p>channel_context is a channel context pointer to be associated with the new anchored channel endpoint.</p> <p>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.</p>	
DESCRIPTION	<p>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.</p> <p>The pair of the original channel handle and the spawn index uniquely identify an in-progress spawn operation. Once both regions have rendezvoused inside a spawn request, the new channel is created and each driver is passed its handle for the new channel via the callback.</p> <p>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.</p>	

REFERENCES

It is the drivers' responsibility to 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 the original channel.

`udi_cancel`, `udi_channel_anchor`

NAME	udi_channel_set_context	<i>Attach a new context to a channel endpoint</i>
SYNOPSIS	<pre>#include <udi.h> void udi_channel_set_context (udi_channel_t target_channel, void *channel_context);</pre>	
ARGUMENTS	<p>target_channel is a channel handle for the channel endpoint to be modified.</p> <p>channel_context is a generic pointer that will be returned as-is by UDI in any channel operations related to this channel.</p>	
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.	
WARNINGS	udi_channel_set_context must be called from the region containing the channel endpoint. This endpoint must already be anchored.	

NAME	udi_channel_close	<i>Close a channel</i>
SYNOPSIS	<pre>#include <udi.h> void udi_channel_close (udi_channel_t channel);</pre>	
ARGUMENTS		channel is a channel handle for the channel endpoint being closed.
DESCRIPTION		<p>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 <code>channel_event_ind</code> operation of type <code>UDI_CHANNEL_CLOSED</code>, to close its end of the channel.</p> <p>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.</p> <p>When one end of a channel is closed, either by the driver explicitly calling <code>udi_channel_close</code> or by the environment if a driver is killed, the other end receives a <code>udi_channel_event_ind</code> operation of type <code>UDI_CHANNEL_CLOSED</code>. This tells the driver at the other end that one of its neighbors has gone away unexpectedly. (See page 20-7 for the definition of the <code>udi_channel_event_ind</code> operation).</p> <p><code>udi_channel_close</code> may be used on loose ends, as well as anchored channel endpoints.</p> <p>If a driver calls <code>udi_channel_close</code> on a channel whose other end is loose, the <code>udi_channel_event_ind</code> operation will be delivered if and when that other end is anchored.</p> <p>If a driver invokes operations on a channel whose other end is closed, those operations are ignored, and the associated control blocks and data objects are freed.</p> <p>Once both ends of a channel are closed, all environment resources associated with the channel are released. Calling <code>udi_channel_close</code> on the single end of a half-spawned channel has this effect as well.</p> <p>Closing the last channel anchored in a secondary region also destroys that region. It is implementation specific whether the current operation or callback in progress, if any, is aborted, or whether the region destruction is deferred until the operation completes.</p>

It is valid to pass a zeroed handle (*i.e.* a handle whose value has been zeroed via a `udi_memset` operation) to the `udi_channel_close` routine; it will recognize that the handle is unassigned and return immediately (similar to calling `udi_channel_close` on a channel that is already closed).

WARNINGS

`udi_channel_close` must not be used to close a channel that is not attached to the caller's region. `udi_channel_close` must not be used on management channels.

REFERENCES

`udi_channel_event_ind`, `udi_channel_anchor`,
`udi_channel_spawn`



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. These two types of data are processed by separate services in the environment: a Tracing Facility and a Logging Facility. Providing log data is required of a driver; providing trace data is optional.

This chapter defines the tracing and logging service calls, and an associated module-global entry point for trace/log formatting. Operations to specify the level of tracing generated are defined in the Management Metalanguage (see “Tracing Control Operations” on page 21-5). Service calls are provided that a driver can use to record both trace data and log data. The driver may also provide a formatting entry point to convert previously recorded trace or log data into printable/readable form.

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 21.4.1, “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 Tracing Facility itself 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 classes as tracing is (though `udi_log_write` may be used to simultaneously log and trace data). As with tracing, however, the Logging Facility may choose to filter final output on its own.

Although drivers can function without logging any data, making calls to the Logging Facility when appropriate should not be considered optional. Such logging is particularly important because the Logging Facility tags udi_status_t values with a correlation code (see the *Fundamental Types* Chapter), 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 21.4.1) 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.

The trace events are divided into three different classes as defined in this section:

- 1) *common trace event codes*, which apply to all drivers and metalanguages
- 2) *driver-defined trace event codes* (UDI_TREVENT_INTERNAL_n) are available for tracing events specific to a single driver implementation, and
- 3) *metalanguage-defined trace event codes* (UDI_TREVENT_META_SPECIFIC_n).

Note that drivers and metalanguages can define the use of their own event codes 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	<i>Trace event type definition</i>
SYNOPSIS	<pre>#include <udi.h> typedef udi_ubit32_t udi_trevent_t; /* Common Trace Events */ #define UDI_TREVENT_LOCAL_PROC_ENTRY(1<<0) #define UDI_TREVENT_LOCAL_PROC_EXIT (1<<1) #define UDI_TREVENT_EXTERNAL_ERROR (1<<2) /* Metalanguage Trace Events */ #define UDI_TREVENT_IO_SCHEDULED (1<<6) #define UDI_TREVENT_IO_COMPLETED (1<<7) #define UDI_TREVENT_STATE_CHANGE (1<<8) #define UDI_TREVENT_META_SPECIFIC_1 (1<<11) #define UDI_TREVENT_META_SPECIFIC_2 (1<<12) #define UDI_TREVENT_META_SPECIFIC_3 (1<<13) #define UDI_TREVENT_META_SPECIFIC_4 (1<<14) #define UDI_TREVENT_META_SPECIFIC_5 (1<<15) /* Driver-Specific Trace Events */ #define UDI_TREVENT_INTERNAL_1 (1<<16) #define UDI_TREVENT_INTERNAL_2 (1<<17) #define UDI_TREVENT_INTERNAL_3 (1<<18) #define UDI_TREVENT_INTERNAL_4 (1<<19) #define UDI_TREVENT_INTERNAL_5 (1<<20) #define UDI_TREVENT_INTERNAL_6 (1<<21) #define UDI_TREVENT_INTERNAL_7 (1<<22) #define UDI_TREVENT_INTERNAL_8 (1<<23) #define UDI_TREVENT_INTERNAL_9 (1<<24) #define UDI_TREVENT_INTERNAL_10 (1<<25) #define UDI_TREVENT_INTERNAL_11 (1<<26) #define UDI_TREVENT_INTERNAL_12 (1<<27) #define UDI_TREVENT_INTERNAL_13 (1<<28) #define UDI_TREVENT_INTERNAL_14 (1<<29) #define UDI_TREVENT_INTERNAL_15 (1<<30) /* Logging Event */ #define UDI_TREVENT_LOG (1<<31)</pre>	
DESCRIPTION	<p>The <code>udi_trevent_t</code> 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.</p> <p>Note that drivers and metalanguages can define the use of their own event codes 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.</p> <p>The following common trace event codes are defined independently of any metalanguage.</p>	

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 classes are designed to trace metalanguage-specific information or operations, and can therefore be selectively enabled and disabled on a per-metalanguage basis. For these classes, tracing is enabled or disabled only for the metalanguages indicated by *meta_idx* of the trace usage operation (see “Tracing Control Operations” on page 21-5). Each metalanguage defines its own rules and conventions for the use of these classes; therefore, the metalanguage specifications should be consulted before using these classes.

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 active handling/initiation of a requested I/O operation. (Example: Start of a SCSI command on the SCSI bus.)

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.)

UDI_TREVENT_STATE_CHANGE – Trace metalanguage-defined driver state transitions.

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 – 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	<i>Record trace data</i>
SYNOPSIS	<pre>#include <udi.h> void udi_trace_write (udi_init_context_t *init_context, udi_trevent_t trace_event, udi_index_t meta_idx, const void *trace_data, udi_size_t trace_data_length, udi_size_t fmt_data_length);</pre>	
ARGUMENTS	<p><i>init_context</i> is a pointer to the front of the driver’s region data area.</p> <p><i>trace_event</i> is the type of trace event being reported.</p> <p><i>meta_idx</i> is a metalanguage index number that identifies the metalanguage to which <i>trace_event</i> 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 28), or 0 for the Management Metalanguage. If the event is not metalanguage-selectable, <i>meta_idx</i> is ignored.</p> <p><i>trace_data</i> is a pointer to the data buffer associated with the trace. The format of the data is driver-specific, since the trace data will later be sent to a separate driver-provided sub-formatter routine (<i>ddd_trace_log_format</i>) for print formatting.</p> <p><i>trace_data_length</i> is the length, in bytes, of the <i>trace_data</i>.</p> <p><i>fmt_data_length</i> is the expected maximum length of the formatted trace information, and will be provided to the driver’s <i>ddd_trace_log_format</i> routine for later formatting of this trace entry. This argument is ignored if the driver passed a NULL <i>trace_log_format_func</i> value to <i>udi_primary_region_init</i> (see page 9-8). This size must be less than <i>max_format_data_len</i> in <i>udi_limits_t</i>.</p>	
DESCRIPTION	<p>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.</p> <p>To simplify usage, <i>udi_trace_write</i> does not involve a callback. The environment will immediately copy the <i>trace_data</i> buffer into its own buffers for later 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.</p> <p>If the driver wishes to log and trace the same event, the <i>udi_log_write</i> operation should be used instead.</p>	
REFERENCES	udi_log_write, ddd_trace_log_format	

NAME	udi_log_write	<i>Record log data</i>
SYNOPSIS	<pre>#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, const void *log_data, udi_size_t log_data_length, udi_size_t fmt_data_length, udi_status_t original_status); 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</pre>	
ARGUMENTS	<p>callback, gcb are standard arguments described in the “Asynchronous Service Calls” section of “Standard Calling Sequences”. Note that no init context pointer is required (unlike with <code>udi_trace_write</code>), since region identity is established through the gcb.</p> <p>trace_event is the type of trace event to be logged. For log data that is not associated with trace events, use <code>UDI_TREVENT_LOG</code>.</p> <p>severity specifies the severity level of the log data.</p> <p>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 <code><meta_idx></code> in the corresponding “child_meta”, “parent_meta”, or “internal_meta” declaration of the driver’s Static Driver Properties (see Chapter 28), or 0 for the Management Metalanguage. If the event is not metalanguage-selectable, meta_idx is ignored.</p> <p>log_data is a pointer to the data buffer associated with the log. The format of the data is driver-specific, since it will later be sent to a separate driver-provided formatter routine (<code>ddd_trace_log_format</code>) for final formatting to printable/readable form. Once log_data is passed into <code>udi_log_write</code> the data area it points to cannot be written into by the driver until the corresponding callback is received. The memory pointed to by log_data must be either movable</p>	

memory (allocated by udi_mem_alloc with the UDI_MEM_MOVABLE flag), part of the **gcb**'s scratch space, or part of a driver-global static (read-only) variable. If movable memory is used, **log_data** must point to the beginning of the memory block.

log_data_length is the length of **log_data**, in bytes.

fmt_data_length is the expected maximum length of the formatted trace information, and will be provided to the driver's ddd_trace_log_format routine for later formatting of this trace entry. This argument is ignored if the driver passed a NULL **trace_log_format_func** value to udi_primary_region_init (see page 9-8). This size must be less than the **max_format_data_len** value set in udi_limits_t.

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 provide supplemental information in the **log_data** and ddd_trace_log_format output if desired.

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.

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_INFORMATIVE 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.

The Logging Facility is responsible for associating related events in different drivers to each other. To this end, `udi_log_write` receives any UDI status value associated with the log event. It may choose to modify this status to include a correlation value, which is used to associate any errors generated by other drivers receiving unusual status.

When the callback routine is invoked, the Logging Facility returns the log data to the driver for reuse. Note the difference from `udi_trace_write`, in which the **trace_data** buffer can be reused immediately.

REFERENCES

`udi_trace_write`, `ddd_trace_log_format`

17.3 Tracing and Logging Entry Points

17.3.1 Formatting of Trace and Log Entries

UDI provides a mechanism for drivers to format their trace and log entries in two stages: the initial reporting stage and the final delivery stage. This allows the majority of processing to take place when trace and log files are requested (the final delivery stage), rather than at driver runtime (the initial reporting stage). Driver writers can chose to completely format entries at runtime, in which case the final delivery stage is unneeded. At the opposite extreme, the driver can construct runtime entries with only a trace-entry code and arguments, letting the formatter construct the string and insert the arguments in their correct places.

The format of data buffers passed to `udi_log_write` or `udi_trace_write` are driver-specific. The driver writer provides a separate routine, `ddd_trace_log_format`, to interpret these entries at the final delivery stage. The `ddd_trace_log_format` function is called directly by the UDI environment code and does not operate in the context of a region (unlike the original `udi_log_write` or `udi_trace_write` call). Because the formatter does not run in region context it cannot access any region data; all information needed to construct the output string must be passed in the initial log or trace call.

UDI utilities include `udi_snprintf`, a variant of the ISO C `snprintf`, to simplify construction of entries at both the initial reporting stage and the final delivery stage.

It is recommended that driver writers keep their data buffers to `udi_trace_write` as short as possible. For simplicity, `udi_trace_write` does not have a callback function, and therefore the environment cannot guarantee that some trace data is not lost before it can be put into permanent storage. Providing shorter trace entries minimizes the chance of lost trace data. This risk of lost data does not apply to logging, nor to final formatting of trace and log entries.

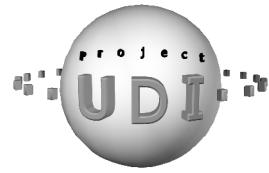
NAME	ddd_trace_log_format	<i>Format log or trace data to printable form</i>
SYNOPSIS	<pre>#include <udi.h> void ddd_trace_log_format (udi_trevent_t event, udi_index_t meta_idx, const void *input_data, udi_size_t input_data_length, char *final_data, udi_size_t final_data_length);</pre>	
ARGUMENTS	<p>event is the event code for the corresponding input data.</p> <p>meta_idx is the metalanguage index number that was passed in the corresponding <code>udi_trace_write</code> or <code>udi_log_write</code> call.</p> <p>input_data is a pointer to a memory area which contains a copy of the original data buffer passed to <code>udi_log_write</code> or <code>udi_trace_write</code>.</p> <p>input_data_length is the number of valid bytes in input_data.</p> <p>final_data is the fully-formatted, printable string to be placed in the trace or log output. This is an municipalized memory region which is allocated by the environment prior to calling the <code>ddd_trace_log_format</code> entry point; the driver should use various facilities (e.g. <code>udi_snprintf</code>) to format the trace information contained in input_data. Upon return, the Tracing and Logging Facility will add its own information to the entry (timestamp, etc.).</p> <p>final_data_length is the number of bytes of space available in final_data. This value will be greater than or equal to the fmt_data_length specified on the corresponding <code>udi_trace_write</code> or <code>udi_log_write</code> call.</p>	
DESCRIPTION	<p>The <code>ddd_trace_log_format</code> function is a driver-supplied routine used to format the tracing or logging information previously supplied by the driver in a corresponding <code>udi_trace_write</code> or <code>udi_log_write</code> call. Driver developers are encouraged to specify trace or log data in an abbreviated format to minimize the impact on performance and, in the tracing case, to minimize the risk of data loss. This routine is then called by the environment to translate the abbreviated information into user-readable trace or log information.</p> <p>This routine is called outside of any region context. The UDI environment calls this function directly passing only the information originally obtained from the trace or log service call; this routine must format the output based solely on the information provided.</p>	

This routine is registered with the UDI environment by passing its address as a parameter to `udi_primary_region_init` (see page 9-8). It is recommended but not necessary to implement this routine: all data may be completely formatted when passed to `udi_trace_write` or `udi_log_write`, and this function pointer may be passed as NULL to the `udi_primary_region_init` call.

WARNING

This routine does not execute in the context of a region. As a result it is very restricted in the services it may use: no external functions besides the debugging and utility functions defined in Chapters 18 and 19 respectively must be called from this routine.

NAME	UDI_HANDLE_ID	<i>Get identification value for specified handle</i>
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 the handle specified.	
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.	
RETURN VALUES	This macro is useful in conjunction with udi_snprintf.	
	This macro returns a void * value that can be formatted using the %p format code for udi_snprintf calls.	
WARNINGS	This macro may only be used from within a region, not from ddd_trace_log_format.	
EXAMPLES	udi_snprintf("Passed buffer %p to channel %p\n", UDI_HANDLE_ID(chan, udi_channel_t), UDI_HANDLE_ID(buf, udi_buf_t));	
REFERENCES	udi_snprintf	



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	<i>Perform driver internal consistency check</i>
SYNOPSIS	<pre>#include <udi.h> void udi_assert (expr);</pre>	
ARGUMENTS	expr	expression to evaluate for truthfulness
DESCRIPTION	<p>The <code>udi_assert</code> function is used by the driver to perform an internal consistency check. The supplied expression <code>expr</code> 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.</p> <p>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 (<i>e.g.</i> free <i>vs.</i> checked) where the failed assertions have different results depending on the mode.</p> <p>While it is not actually guaranteed that <code>udi_assert</code> will not return to the driver if <code>expr</code> is false, it is expected that drivers will be coded as if that were the case.</p> <p>As an esoteric note, an environment may choose not to directly handle the <code>udi_assert</code> 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 <code>udi_assert</code> call will not return.</p>	

NAME	udi_debug_break	<i>Request a debug breakpoint at the current location</i>
SYNOPSIS	<pre>#include <udi.h> void udi_debug_break (udi_init_context_t *init_context, const char *message);</pre>	
ARGUMENTS	<p>init_context is the initial context supplied to the driver on the primary region's management channel.</p> <p>message is a string used to indicate the cause of the debug break.</p>	
DESCRIPTION	<p>The <code>udi_debug_break</code> function is used for driver debugging purposes. In a debug configuration, calling this routine indicates that the system debugger should be entered at the current time for developer debugging operations.</p> <p>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. If <code>udi_debug_break</code> is called from the <code>init_module</code> or <code>ddd_trace_log_format</code> routines, NULL should be specified for the init_context argument.</p> <p>The implementation of this function is environment dependent and the actions taken may be defined by an operational mode of that UDI environment (i.e. debug mode v.s. non-debug mode).</p> <p>In debugging mode, it is expected that the message string is output to the debug console and that 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.</p> <p>In a non-debugging mode, the environment may completely ignore this request and simply return immediately to the UDI driver code.</p>	



19.1 Overview

This chapter defines general utility functions (library functions) available to UDI drivers. These are functions that are not in any way platform or environment implementation dependent, and therefore could have been coded in the driver itself, but are provided by the environment for driver writers' convenience. Placing these functions in the environment instead of each individual device driver also improves the degree of code sharing.

19.2 String/Memory Functions

UDI defines several string and memory operator functions. These functions parallel their ISO 9899 (ISO C) counterparts but are specifically designed to be used from a UDI driver perspective. Most of these routines are chosen and optimized for processing speed and are fully reentrant (i.e. no global writable storage is involved).

Unless otherwise stated, any utility function passed a NULL or other invalid pointer produces indeterminate results.

NAME	udi_strcat, udi_strncat	<i>String concatenation</i>
SYNOPSIS	<pre>#include <udi.h> char *udi_strcat (char *s1, const char *s2); char *udi_strncat (char *s1, const char *s2, udi_size_t n);</pre>	
ARGUMENTS	<p>s1 is a pointer to the destination string.</p> <p>s2 is a pointer to the source string.</p> <p>n is the destination string maximum length (in bytes).</p>	
DESCRIPTION	<p>The udi_strcat and udi_strncat functions are used to append the contents of string s2 to the end of the existing string s1, overwriting the null-terminator character at the end of s1 and ending with a new null-terminator character. The strings must not overlap and the s1 string must have enough space for the result.</p> <p>The udi_strncat form may be used to limit the size of the result: this function will stop copying bytes from s2 to s1 once the length of s1 has reached n-1 bytes; a null-terminator will be supplied as the n'th byte if the end of s2 has not been reached.</p>	
RETURN VALUES	The udi_strcat and udi_strncat functions return a pointer to the resulting null-terminated string s1 .	

Utilities *udi_strcmp, udi_strncmp, udi_memcmp*

NAME	udi_strcmp, udi_strncmp, udi_memcmp	<i>String/memory comparison</i>
SYNOPSIS	<pre>#include <udi.h> udi_sbit8_t udi_strcmp (const char *s1, const char *s2); udi_sbit8_t udi_strncmp (const char *s1, const char *s2, udi_size_t n); udi_sbit8_t udi_memcmp (const void *s1, const void *s2, udi_size_t n);</pre>	
ARGUMENTS	<p>s1 is a pointer to the first character string or memory area.</p> <p>s2 is a pointer to the second character string or memory area.</p> <p>n is the maximum size to be compared (in bytes).</p>	
DESCRIPTION	<p>The <code>udi_strcmp</code> and <code>udi_strncmp</code> functions are used to compare the contents of two null-terminated character strings. The strings are compared on a byte-by-byte basis and a comparison value is returned when the first differing character or the end of the strings is reached.</p> <p>For the <code>udi_strncmp</code> function, comparison halts after comparing n characters unless the end of either string has already been reached. If both strings are identical throughout the first n characters, the <code>udi_strncmp</code> function return value indicates that the strings are equal, regardless of any remaining content.</p> <p>The <code>udi_memcmp</code> function operates in a similar manner as <code>udi_strncmp</code> except that null characters do not terminate the comparison, which will always continue until the specified n bytes have been compared or a difference has been reached.</p>	
RETURN VALUES	<p>These functions return an integer value less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is lexicographically less than, equal, to, or greater than s2. This comparison is made by comparing each unsigned byte value until there is a mismatch or all bytes compare equal.</p>	

udi_strcpy, udi_strncpy, udi_memcpy, udi_memmove

NAME	udi_strcpy, udi_strncpy, udi_memcpy, udi_memmove <i>String/memory copy</i>
SYNOPSIS	#include <udi.h> char *udi_strcpy (char *s1, const char *s2); char *udi_strncpy (char *s1, const char *s2, udi_size_t n); void *udi_memcpy (void *s1, const void *s2, udi_size_t n); void *udi_memmove (void *s1, const void *s2, udi_size_t n);
ARGUMENTS	s1 is a pointer to the destination string or memory area. s2 is a pointer to the source string or memory area. n is the number of bytes to copy from s2 to s1 .
DESCRIPTION	The udi_strcpy and udi_strncpy functions copy the character array string pointed to by s2 (including the null-terminator character) to the character array string pointed to by s1 . The strings may not overlap and the destination string s1 must be large enough to receive the copy. The udi_strncpy function will stop copying once the specified n number of bytes has been copied or when a null terminator is encountered in the s2 string, whichever comes first. If there is no null byte encountered among the first n bytes of the s2 string, the result in s1 will not be null terminated. In the case where the length of s2 is less than n , the remainder of s1 will be padded with null characters. The udi_memcpy function will operate in the same manner as the udi_strncpy function except that null characters ('\0') will be ignored and n bytes will always be copied into the s1 string. The memory areas must not overlap. The udi_memmove function is similar to udi_memcpy but allows overlapping regions; it operates as if the contents of the s2 area were first copied to a temporary area and then copied back to the s1 area.
RETURN VALUES	These functions return a pointer to the destination string s1 .

NAME	udi_strchr, udi_strrchr, udi_memchr	<i>String/memory searching</i>
SYNOPSIS	<pre>#include <udi.h> char *udi_strchr (const char *s, char c); char *udi_strrchr (const char *s, char c); void *udi_memchr (const void *s, udi_ubit8_t c, udi_size_t n);</pre>	
ARGUMENTS	<p>s is a pointer to the string or memory area to be searched.</p> <p>c is the character to search for.</p> <p>n is the maximum number of bytes to search.</p>	
DESCRIPTION	<p>The <code>udi_strchr</code> function returns a pointer to the first occurrence of the character c in the null-terminated string s.</p> <p>The <code>udi_strrchr</code> function return a pointer to the last occurrence of the character c in the null-terminated string s.</p> <p>The <code>udi_memchr</code> function returns a pointer to the first occurrence of the (unsigned character) byte c in the specified memory region, regardless of null bytes.</p>	
RETURN VALUES	These functions return a pointer to the matched character or NULL if the character is not found.	

NAME	udi_strtou32	<i>Convert string to unsigned 32-bit value</i>
SYNOPSIS	<pre>#include <udi.h> udi_ubit32_t udi_strtou32 (const char *s, char **endptr, int base);</pre>	
ARGUMENTS	<p>s is a pointer to the null-terminated string to be converted.</p> <p>endptr optionally points to a character pointer in which a pointer to the first unconverted character is stored.</p> <p>base is the base radix for interpretation of the numeric string.</p>	
DESCRIPTION	<p>The udi_strtou32 function is similar to its ISO C strtoul counterpart in that it converts the string pointed to by s to an unsigned 32-bit integer value according to the given base radix which must be between 2 and 36 inclusive, or be the special value of 0.</p> <p>The string must begin with an arbitrary amount of whitespace (zero or more space, tab, carriage-return, or line-feed characters in any order) followed by a single optional '+' or '-' sign.</p> <p>If base is between 2 and 36, inclusive, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if base is 16.</p> <p>If base is zero, the string itself determines the base as follows: After an optional leading sign, one or more leading zeros indicates octal conversion, and a leading "0x" or "0X" indicates hexadecimal conversion. Otherwise, decimal conversion is used.</p> <p>The remainder of the string is converted to an unsigned 32-bit integer value in the obvious manner, stopping at the first character that is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)</p> <p>If endptr is not NULL, udi_strtoul stores a pointer to the first invalid character into *endptr. (Thus, if *s is not '\0' but **endptr is '\0' on return, the entire string was a valid numeric.)</p>	
RETURN VALUE	<p>The udi_strtoul function returns the result of the conversion as an unsigned 32-bit value. If the input string contained a leading minus sign ('-'), the result will be as the appropriate negative number cast to a udi_ubit32_t unsigned type.</p> <p>If the numeric string would cause a 32-bit integer overflow then the resulting value is unspecified.</p>	

NAME	udi_strlen	<i>Determine string length</i>
SYNOPSIS	#include <udi.h> udi_size_t udi_strlen (const char * s);	
ARGUMENTS	s	is a pointer to a null-terminated string.
DESCRIPTION		The udi_strlen function scans the specified string to locate the null-terminator and returns the number of bytes in the string (not including the terminator).
RETURN VALUES		The udi_strlen function returns the number of bytes in the string s .

NAME	udi_memset	<i>Memory initialization</i>
SYNOPSIS	<pre>#include <udi.h> void *udi_memset (void *s, udi_ubit8_t c, udi_size_t n);</pre>	
ARGUMENTS	<p>s is a pointer to the memory area to be initialized.</p> <p>c is the unsigned 8-bit value to use for initialization.</p> <p>n is the size of the memory area (in bytes).</p>	
DESCRIPTION	The udi_memset function is used to fill in the first n bytes of the memory area pointed to by the s argument with the unsigned byte value of c .	
RETURN VALUES	This function returns a pointer to the memory area s .	

19.3 Output Formatting Functions

The functions described in this section assist in formatting strings with embedded parameters. The functions described here parallel their ISO C counterparts, but are not identical.

NAME	udi_snprintf	<i>Format printable string</i>												
SYNOPSIS	<pre>#include <udi.h> udi_size_t udi_snprintf (char *s, udi_size_t max_bytes, const char *format, ...);</pre>													
ARGUMENTS	<p>s is a pointer to the target buffer for the formatted output, which is a null-terminated printable string.</p> <p>max_bytes is the maximum number of bytes to be written to s, including the null terminator.</p> <p>format is the format string, which controls the formatting of the output string.</p> <p>... are the remaining arguments, which provide the values used for the formatting codes.</p>													
DESCRIPTION	<p>The udi_snprintf routine is used to generate a formatted string from a set of input arguments and values. The operation of this utility is comparable to the ISO snprintf function with the exceptions noted and supporting only the format codes and modifiers documented below.</p> <p>All format specifications are of the form <i>%mf</i> where <i>m</i> is an optional modifier of the form [[0, -] <i>nn</i>] and <i>f</i> is one or more format codes.</p> <p>The following format modifiers are defined:</p>													
	<hr/> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding-bottom: 2px;">Format</th> <th style="text-align: left; padding-bottom: 2px;">Modifier</th> <th style="text-align: left; padding-bottom: 2px;">Output Control</th> </tr> </thead> <tbody> <tr> <td style="padding-top: 2px;"><i>nn</i></td><td></td><td>minimum field width as an unsigned decimal number. (e.g. %4x prints a hexadecimal number taking up 4 or more digits). By default, the output is preceded by spaces to meet the minimum field width unless changed by other format modifiers. This modifier applies to the %c format code as well; in this case the single character is preceded by the necessary number of spaces (or otherwise adjusted according to any other modifiers present).</td></tr> <tr> <td style="padding-top: 2px;">0</td><td></td><td>leading characters needed for minimum field width compliance should be padded with zeros instead of spaces when used with numeric formats (e.g. %04x for a value of 12 will output “000c”).</td></tr> <tr> <td style="padding-top: 2px;">-</td><td></td><td>left-justify within field width (e.g. %-4x for a value of 12 will output ‘c’ followed by 3 spaces. It is not valid to use the ‘-’ and ‘0’ modifiers together.</td></tr> </tbody> </table> <hr/>	Format	Modifier	Output Control	<i>nn</i>		minimum field width as an unsigned decimal number. (e.g. %4x prints a hexadecimal number taking up 4 or more digits). By default, the output is preceded by spaces to meet the minimum field width unless changed by other format modifiers. This modifier applies to the %c format code as well; in this case the single character is preceded by the necessary number of spaces (or otherwise adjusted according to any other modifiers present).	0		leading characters needed for minimum field width compliance should be padded with zeros instead of spaces when used with numeric formats (e.g. %04x for a value of 12 will output “000c”).	-		left-justify within field width (e.g. %-4x for a value of 12 will output ‘c’ followed by 3 spaces. It is not valid to use the ‘-’ and ‘0’ modifiers together.	
Format	Modifier	Output Control												
<i>nn</i>		minimum field width as an unsigned decimal number. (e.g. %4x prints a hexadecimal number taking up 4 or more digits). By default, the output is preceded by spaces to meet the minimum field width unless changed by other format modifiers. This modifier applies to the %c format code as well; in this case the single character is preceded by the necessary number of spaces (or otherwise adjusted according to any other modifiers present).												
0		leading characters needed for minimum field width compliance should be padded with zeros instead of spaces when used with numeric formats (e.g. %04x for a value of 12 will output “000c”).												
-		left-justify within field width (e.g. %-4x for a value of 12 will output ‘c’ followed by 3 spaces. It is not valid to use the ‘-’ and ‘0’ modifiers together.												

The *udi_snprintf* function supports the following format codes chosen to provide fast execution and common utility:

Format Code	Output generated
%x, %X	unsigned hexadecimal <i>udi_ubit32_t</i> . The alphanumeric characters output as a result of this format will be shown in either lower or upper case as specified by the case of the format code.
%d, %u	signed and unsigned decimal <i>udi_sbit32_t</i> and <i>udi_ubit32_t</i>
%hx, %hX	unsigned hexadecimal <i>udi_ubit16_t</i>
%hd, %hu	signed and unsigned decimal <i>udi_sbit16_t</i> and <i>udi_ubit16_t</i>
%bx, %bX	unsigned hexadecimal <i>udi_ubit8_t</i>
%bd, %bu	signed and unsigned decimal <i>udi_sbit8_t</i> and <i>udi_ubit8_t</i>
%p, %P	hexadecimal pointer value, size as appropriate for the host machine
%a, %A	64-bit bus address value (DMA address type <i>udi_busaddr64_t</i> ; see the description of the <i>udi_scgth_t</i> structure in the UDI Physical I/O Specification) printed as a hexadecimal value in lower or upper case, respectively. Not supported in environments that do not support the Physical I/O Services.
%c	single printable character
%s	null-terminated string
%<istring>	Value bitmask formatter. This format code outputs a formatted string interpretation of a bitmask. The argument for this format code is a <i>udi_ubit32_t</i> value; the bitmask interpretation of that value is based on the <i>istring</i> information as described here, where bit number 0 is the least-significant bit in the value: <i>istring</i> := [,] <i>bitspec</i> {, <i>bitspec</i> } [,] <i>bitspec</i> := [~] <i>bitnum</i> = < <i>string</i> > <i>bitnum</i> - <i>bitnum</i> = < <i>string</i> > [<i>fieldspec</i> ...] <i>bitnum</i> := <decimal number 0-31> <i>fieldspec</i> := : <unsigned decimal number> = < <i>string</i> > The output is delimited by ‘<’ and ‘>’ characters. If <i>bitspec</i> is specified as a single <i>bitnum</i> (the first form) then the associated string is printed if the specified bit is set (or printed if the bit is not set if ~ is specified); otherwise nothing is printed. If <i>bitspec</i> is specified as a range of bits (the second form) then the associated string will be output followed by ‘=’ and then the hexadecimal value of the specified range of bits; if the value also matches any of the optional <i>fieldspec</i> values, the <i>fieldspec</i> string is printed instead of the value.

Once formatting has reached the specified **max_bytes** output length for **s**,

this function returns without processing the remainder of the **format** or other arguments. A null terminator character will always be placed at the end of the generated string in **s**.

RETURN VALUES

The number of bytes in **s**, not including the null terminator.

EXAMPLES

The following code segment:

```
udi_snprintf(s, 256, "%s %-2c %d: 0x%08x "
             "%<15=Active,14=DMA Ready,13=XMIT,"
             "~13=RCV,0-2=Mode:0=HDX:1=FDX"
             ":2=Sync HDX:3=Sync FDX:4=Coded,"
             "3-6=TX Threshold,7-10=RX Threshold>",
             "Register", '#', 0, 0xc093, 0xc093);
```

would result in the following contents of string **s** (without line breaks):

```
"Register # 0: 0x0000c093 <Active, DMA Ready, RCV,
Mode=Sync FDX, TX Threshold=2, RX Threshold=1>"
```

The following code segment will produce different output based on the architecture on which it is run:

```
udi_snprintf(s, 256, "Stored at 0x%p", &var);
```

will produce one of the following output strings, depending on pointer size:

16-bit (e.g. 8086): "Stored at 0x801c"

32-bit (e.g. PA-RISC): "Stored at 0x505d806f"

64-bit (e.g. Alpha): "Stored at 0x300040cc6069f0c0"

NAME	udi_vsnprintf	<i>Format printable string with varargs</i>
SYNOPSIS	<pre>#include <udi.h> udi_size_t udi_vsnprintf (char *s, udi_size_t max_bytes, const char *format, va_list ap);</pre>	
ARGUMENTS	<p>s is a pointer to the target buffer for the formatted output, which is a null-terminated printable string.</p> <p>max_bytes is the maximum number of bytes to be written to s, including the null terminator.</p> <p>format is the format string, which controls the formatting of the output string.</p> <p>ap is the varags argument list. (The va_list type definition is provided by the udi.h header file.)</p>	
DESCRIPTION	<p>The udi_vsnprintf routine is used to generate a formatted string from a set of input varargs arguments. This utility operates in the same manner as the udi_snprintf utility routine except that it uses a previously obtained varargs argument list instead of a sequence of actual arguments as parameters to this routine.</p> <p>This routine is useful if the string formatting is to be done from within a routine that has already been passed the actual sequence of arguments and can only refer to those arguments via the varargs functionality.</p>	
EXAMPLE	<pre>#include <udi.h> udi_size_t mydriver_snprintf(char *s, udi_size_t max_bytes, const char *format, ...) { static char prefix_str[] = "From MYDRIVER: "; #define PREFIX_LEN (sizeof(prefix_str)-1) va_list arglist; udi_size_t retval; va_start(arglist, format); udi_assert(max_bytes > PREFIX_LEN); udi_strcpy(s, prefix_str); retval = udi_vsnprintf(s + PREFIX_LEN, max_bytes - PREFIX_LEN, format, arglist); va_end(arglist); return retval; } l = mydriver_snprintf("Byte 1 = %02bx, " "pktlen = %hu\n", *pkt->data, pkt->len);</pre>	
REFERENCES	udi_snprintf	

19.4 Queue Management

The queuing interfaces are designed using macros built on top of two basic functional interfaces in order to provide a high-performance and fully functional set of queue management interfaces. The interfaces are provided for the convenience of the UDI driver writer for driver-internal queuing. The driver may design its own internal queuing routines and algorithms, but it is recommended that, where applicable, the driver use these interfaces as a high-performance standard queuing interface. It is expected that the interfaces provided here will serve several common queuing needs in UDI drivers, helping ease driver development effort.

19.4.1 Queue Element Structure

The *queues* defined in UDI are circular doubly-linked lists that are linked together via `udi_queue_t` structures (known as *queue elements*) as depicted in Figure 19-1.

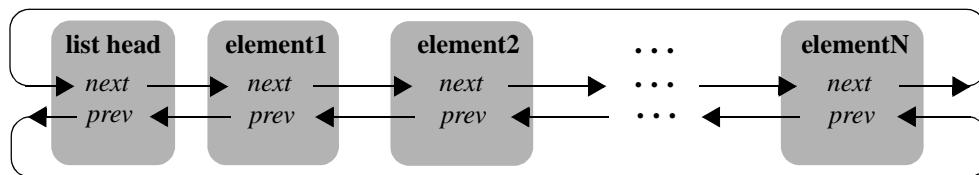


Figure 19-1

A UDI *queue* is composed of a *list head element* and zero or more *queue elements*. The queue is referred to via the list head, which is the only permanent piece of memory associated with a given queue: an empty queue is composed of a list head linked to itself. UDI drivers will typically instantiate internal queues in their region data area or channel contexts by placing `udi_queue_t` list head structures in their region contexts, and calling `UDI_QUEUE_INIT` on each queue before operating on it.

Additional UDI queue terminology: “head of queue” and “first element in queue” are equivalent and refer to the element immediately following the list head. Similarly for “tail of queue” and “last element in queue,” which refer to the element immediately preceding the list head.

NAME	udi_queue_t	<i>Queue element structure</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct udi_queue { struct udi_queue *next; struct udi_queue *prev; } udi_queue_t;</pre>	
MEMBERS	<p>next is a pointer to the next element in the queue.</p> <p>prev is a pointer to the previous element in the queue.</p>	
DESCRIPTION	<p>The udi_queue_t structure is the queue element structure. UDI queues are linked together via these structures. The list head used to reference a particular queue is also a structure of this type, and is the only permanent piece of memory associated with a given queue.</p>	

19.4.2 Queuing Functions

There are two queuing functions defined in UDI that provide high-performance basic queuing and dequeuing functionality: `udi_enqueue`, which inserts a specified element at the head of the specified queue, and `udi_dequeue`, which removes the specified element from the queue. The macros that follow build on top of these two functions to provide a rich set of queue management utilities.

NAME	udi_enqueue	<i>Insert a queue element into a queue</i>
SYNOPSIS	<pre>#include <udi.h> void udi_enqueue (udi_queue_t *new_el, udi_queue_t *old_el);</pre>	
ARGUMENTS	<p>new_el is a pointer to a queue element.</p> <p>old_el is a pointer to the queue's list head or an element already on the queue.</p>	
DESCRIPTION	<p>udi_enqueue inserts new_el after old_el in the queue to which old_el belongs.</p> <p>udi_enqueue shall be equivalent to the following implementation:</p> <pre>void udi_enqueue(udi_queue_t *new_el, udi_queue_t *old_el) { new_el->next = old_el->next; new_el->prev = old_el; old_el->next->prev = new_el; old_el->next = new_el; }</pre>	
REFERENCES	udi_dequeue, udi_queue_t	

NAME	udi_dequeue	<i>Dequeue a queue element</i>
SYNOPSIS	#include <udi.h> udi_queue_t *udi_dequeue (udi_queue_t *element);	
ARGUMENTS	element is a pointer to a queue element	
DESCRIPTION	udi_dequeue removes element from its queue and returns it in the function return. The specified element must be linked into a UDI queue at the time udi_dequeue is called, and must not be the list head. The next and prev fields of element are not modified.	
	udi_dequeue shall be equivalent to the following implementation:	
	<pre>udi_queue_t *udi_dequeue(udi_queue_t *element) { element->next->prev = element->prev; element->prev->next = element->next; return element; }</pre>	
RETURN VALUES	The element passed in is returned.	
REFERENCES	udi_enqueue, udi_queue_t	

19.4.3 Queuing Macros

The macros defined in this section are general queue management macros used for the queuing of arbitrary structures linked together via embedded `udi_queue_t` structures. The behavior is indeterminate if the `listhead` passed to any of these macros other than UDI_QUEUE_INIT has not been previously initialized with UDI_QUEUE_INIT, or if the `element` or `old_el` parameter passed to any of the macros other than UDI_ENQUEUE_HEAD/TAIL and UDI_QUEUE_FOREACH is not currently linked into a UDI queue.

UDI_QUEUE_INIT, UDI_QUEUE_EMPTY Utilities

NAME	UDI_QUEUE_INIT, UDI_QUEUE_EMPTY	<i>Initialize queue; check if it's empty</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_QUEUE_INIT(listhead) \ ((listhead)->next = (listhead)->prev = listhead) #define UDI_QUEUE_EMPTY(listhead) \ ((listhead)->next == (listhead))</pre>	
ARGUMENTS	listhead	is a pointer to a list head element.
DESCRIPTION	UDI_QUEUE_INIT initializes the queue's list head and must be called before any other operations are performed on the queue.	UDI_QUEUE_EMPTY is used to determine if the queue specified by listhead is empty, based on the boolean return value (non-zero if empty; zero if non-empty).
	These macros must be called as if they, respectively, had the following functional interfaces:	
	<pre>void UDI_QUEUE_INIT (udi_queue_t *listhead);</pre> <pre>udi_boolean_t UDI_QUEUE_EMPTY (udi_queue_t *listhead);</pre>	
REFERENCES	udi_queue_t	

NAME	UDI_ENQUEUE_XXX, UDI_QUEUE_INSERT_XXX <i>Insert an element into a queue</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_ENQUEUE_HEAD(listhead, element) \ udi_enqueue(element, listhead) #define UDI_ENQUEUE_TAIL(listhead, element) \ udi_enqueue(element, (listhead)->prev) #define UDI_QUEUE_INSERT_AFTER(old_el, new_el) \ udi_enqueue(new_el, old_el) #define UDI_QUEUE_INSERT_BEFORE(old_el, new_el) \ udi_enqueue(new_el, (old_el)->prev)</pre>
ARGUMENTS	<p>listhead is a pointer to a list head element.</p> <p>element is a pointer to a queue element.</p> <p>old_el is a pointer to a queue element that is currently linked into a UDI queue.</p> <p>new_el is a pointer to a queue element that is not currently linked into a UDI queue.</p>
DESCRIPTION	<p>UDI_ENQUEUE_HEAD inserts element at the head of the queue specified by listhead. This macro is equivalent to the <code>udi_enqueue</code> function.</p> <p>UDI_ENQUEUE_TAIL appends element to the tail of the queue specified by listhead.</p> <p>UDI_QUEUE_INSERT_AFTER inserts the queue element new_el after the element old_el.</p> <p>UDI_QUEUE_INSERT_BEFORE inserts the queue element new_el in front of the element old_el.</p> <p>These macros must be called as if they, respectively, had the following functional interfaces:</p> <pre>void UDI_ENQUEUE_HEAD (udi_queue_t *listhead, udi_queue_t *element); void UDI_ENQUEUE_TAIL (udi_queue_t *listhead, udi_queue_t *element); void UDI_QUEUE_INSERT_AFTER (udi_queue_t *old_el, udi_queue_t *new_el);</pre>

UDI_ENQUEUE_XXX, UDI_QUEUE_INSERT_XXX

REFERENCES

```
void UDI_QUEUE_INSERT_BEFORE (
    udi_queue_t *old_el,
    udi_queue_t *new_el );
```

udi_enqueue

NAME	UDI_DEQUEUE_XXX, UDI_QUEUE_REMOVE	<i>Remove an element from a queue</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_DEQUEUE_HEAD(listhead) \ udi_dequeue((listhead)->next) #define UDI_DEQUEUE_TAIL(listhead) \ udi_dequeue((listhead)->prev) #define UDI_QUEUE_REMOVE(element) \ ((void)udi_dequeue(element))</pre>	
ARGUMENTS	listhead is a pointer to a list head element.	
	element is a pointer to a queue element.	
DESCRIPTION	<p>UDI_DEQUEUE_HEAD removes the element at the head of the queue specified by listhead, and returns it to the caller.</p> <p>UDI_DEQUEUE_TAIL removes the element at the tail of the queue specified by listhead, and returns it to the caller.</p> <p>UDI_QUEUE_REMOVE removes element from its queue. With the exception that there is no return value, this macro is equivalent to the <code>udi_dequeue</code> function. Since the caller is specifying the element to remove, it is expected that normal usage would be to call this macro without expecting a return value, so the function return is voided out.</p> <p>These macros must be called as if they, respectively, had the following functional interfaces:</p>	
REFERENCES	<pre>udi_queue_t *UDI_DEQUEUE_HEAD (udi_queue_t *listhead); udi_queue_t *UDI_DEQUEUE_TAIL (udi_queue_t *listhead); void UDI_QUEUE_REMOVE (udi_queue_t *element);</pre> <p>udi_dequeue</p>	

UDI_FIRST/LAST/NEXT/PREV_ELEMENT Utilities

NAME	UDI_FIRST/ LAST/ NEXT/ PREV_ELEMENT	<i>Get first/last/next/previous element in queue</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_FIRST_ELEMENT(listhead) \ ((listhead)->next) #define UDI_LAST_ELEMENT(listhead) \ ((listhead)->prev) #define UDI_NEXT_ELEMENT(element) \ ((element)->next) #define UDI_PREV_ELEMENT(element) \ ((element)->prev)</pre>	
ARGUMENTS	<p>listhead is a pointer to a list head element.</p> <p>element is a pointer to a queue element.</p>	
DESCRIPTION	<p>UDI_FIRST_ELEMENT returns a pointer to the first element in the queue specified by listhead.</p> <p>UDI_LAST_ELEMENT returns a pointer to the last element in the queue specified by listhead.</p> <p>UDI_NEXT_ELEMENT returns a pointer to the next queue element immediately after element.</p> <p>UDI_PREV_ELEMENT returns a pointer to the queue element immediately preceding element.</p> <p>These macros must be called as if they, respectively, had the following functional interface:</p> <pre>udi_queue_t *UDI_FIRST_ELEMENT (udi_queue_t *listhead); udi_queue_t *UDI_LAST_ELEMENT (udi_queue_t *listhead); udi_queue_t *UDI_NEXT_ELEMENT (udi_queue_t *element); udi_queue_t *UDI_PREV_ELEMENT (udi_queue_t *element);</pre>	
REFERENCES	udi_queue_t	

NAME	UDI_QUEUE_FOREACH	<i>Safe mechanism to walk a queue</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_QUEUE_FOREACH(listhead, element, tmp) \ for ((element) = UDI_FIRST_ELEMENT(listhead)); \ ((tmp) = UDI_NEXT_ELEMENT(element)), \ ((element) != (listhead)); \ (element) = (tmp))</pre>	
ARGUMENTS	<p>listhead is a pointer to a list head element.</p> <p>element is a queue element pointer variable that may be uninitialized on entry, and is set successively to each element in the queue.</p> <p>tmp is a queue element pointer variable for temporary storage in the loop.</p>	
DESCRIPTION	<p>UDI_QUEUE_FOREACH walks through the elements in the queue specified by listhead, setting element successively to each element in the queue, beginning at the head and continuing to the tail of the queue. This provides a safe mechanism to walk through each element in a queue, and do operations on each element (including removing it from the queue and re-queuing it in another queue) without affecting the traversal to the next element in the queue.</p> <p>The UDI_QUEUE_FOREACH macro produces an iteration (loop) statement and hence must be followed by a C statement which is the action to take for each iteration through the loop (e.g., an action on each element in the queue). The parameters to this macro must have the following type definitions:</p> <pre>UDI_QUEUE_FOREACH (udi_queue_t *listhead, udi_queue_t *element, udi_queue_t *tmp);</pre>	
EXAMPLES	<p>The following code reverses the elements in a queue:</p> <pre>{ udi_queue_t *elem, *tmp; udi_queue_t *head = &region_data->my_queue; UDI_QUEUE_FOREACH(head, elem, tmp) { UDI_ENQUEUE_HEAD(udi_dequeue(elem)) } }</pre>	
REFERENCES	<p>udi_queue_t</p>	

19.5 Endian Management

UDI drivers are portable across big and little endian platforms. Endianness of data must be managed in any structure that requires a specific endianness potentially different from the driver's endianness. This includes hardware protocol-defined structures such as SCSI commands (which are big endian) or networking headers (which are generally big endian), shared control structures (DMA-able structures which are shared between the driver and the device, are typically in system memory and in the endianness of the device), or device memory (registers, card RAM, etc.).

UDI physical I/O drivers access device memory using an access handle with which the driver associates its device endianness, allowing the environment to implicitly take care of any needed endian conversions (see the Physical I/O Specification). The other two cases (protocol structures and shared control structures) are directly accessed by the driver using a pointer and therefore require direct involvement of the driver in the managing of the structure endianness.

Protocol structures and shared control structures have different characteristics with respect to endianness handling in UDI. First, protocol structures have a required endianness that is known at compile time, whereas shared control structures have a required endianness which in general is only known at runtime (due to the impacts of intervening bus bridges). Second, the relationship between driver endianness and protocol structure endianness is not indicated in any UDI-defined manner, whereas a shared control structure has a `must_swap` flag associated with it when it is allocated.

These characteristics lead to different approaches in the construction of the corresponding C structure definitions, as described below.

19.5.1 Rules for C Structure Definitions

As specified in Section 8.9, “Structures Requiring a Fixed Binary Representation,” on page 8-18, 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 within (offsets from the beginning of) the structure. Bit fields are not allowed (see the warning in Section 8.9 for details).
2. Every byte in the structure must be accounted for.

These rules should generally be sufficient for the needs of shared control structures, combined with appropriate byte-swapping based on the `must_swap` flag. However, since protocol structures have no associated `must_swap` flag or other UDI-defined swapping indication, protocol structures should be handled by the driver using the further rules defined in the byte-by-byte layout method below.

Note – The compile-time endianness of a shared control structure (i.e., when there are non-naturally-aligned quantities that need to be split up and treated as having a given endianness at compile time) must be the same as the "device endianness" specified in the `flags` parameter to `udi_dma_mem_alloc` if the driver wants the sense of the `must_swap` flag to indicate byte swapping in the expected manner. Also note that drivers that can change their device's endianness at runtime may choose to optimize away any runtime byte-swapping in the driver by allocating the shared control memory once specifying the device's default endianness, and then freeing and re-allocating using the opposite endianness if `must_swap` had come back true.

19.5.1.1 Byte-by-byte structure layout

In this method, the C structure definition for a hardware-defined structure is laid out byte-by-byte, splitting up multi-byte integer quantities and combining adjoining integer quantities that share a byte. This means that each field in the structure must be either exactly one byte in size or an array of byte-sized elements.

To help the driver deal with these structures, bit-field, multi-byte, and other helper utilities are provided in Section 19.5.2 on page 19-29 below. The multi-byte helper utilities impose a rule on the naming of fields making up a multi-byte quantity in these structures: the fields must all be named with a common name followed by a digit 0..N, where myfield0 is the least significant byte of the multi-byte quantity, myfield1 the next most significant byte, and myfieldN is the most significant byte. See “Multi-Byte Macros” below for details.

For example, a 10-byte SCSI command might look like the following when defined naturally in SCSI’s big endian format, using non-portable bit fields for a particular compiler. This structure would only work correctly on a big-endian platform, with ISO C compilers that make appropriate assumptions about bit field order, and on platforms that don’t require natural alignment and in which an **int** is 32 bits.

```
typedef struct {
    udi_ubit32_t c10_opcode:8; /* command code */
    udi_ubit32_t c10_lun:3;   /* LUN */
    udi_ubit32_t c10_dpo:1;   /* Disable Page Out */
    udi_ubit32_t c10_fua:1;   /* Force Unit Access */
    udi_ubit32_t c10_rsvd1:2; /* Reserved: must be zero */
    udi_ubit32_t c10_reladr:1; /* Addr Relative to prev linked cmd */
    udi_ubit32_t c10_lba3:8;  /* LBA - high order byte */
    udi_ubit32_t c10_lba2:8;  /* LBA - next order byte */
    udi_ubit16_t c10_lba0;    /* LBA - low order two bytes */
    udi_ubit8_t c10_rsvd2;   /* Reserved: must be zero */
    udi_ubit16_t c10_len;    /* transfer length */
    udi_ubit8_t c10_ctrl;    /* control byte */
} udi_scsi_cdb_10_t;
```

When converted using the byte-by-byte layout rules this becomes

```
typedef struct {
    udi_ubit8_t c10_opcode; /* Command code */
    udi_ubit8_t c10_flags;  /* Flags */
    udi_ubit8_t c10_lba3;  /* LBA - high order byte */
    udi_ubit8_t c10_lba2;  /* LBA - next order byte */
    udi_ubit8_t c10_lba1;  /* LBA - next order byte */
    udi_ubit8_t c10_lba0;  /* LBA - low order byte */
    udi_ubit8_t c10_rsvd1; /* Reserved: must be zero */
    udi_ubit8_t c10_len1;  /* Transfer length - high order byte */
    udi_ubit8_t c10_len0;  /* Transfer length - low order byte */
    udi_ubit8_t c10_ctrl;  /* Control byte */
} udi_scsi_cdb_10_t;
```

which is completely portable, across any platform, big or little endian, with or without natural alignment, with any ISO C compiler, etc. Using the helper macros, the following “build cdb” macro could be defined:

```
#define UDI_SCSI_BUILD_CDB_10(cdb, opcode, data_len, block_num,flags) \
{ \
    (cdb)->c10_opcode = opcode; \
    (cdb)->c10_flags = flags; \
}
```

```
UDI_MDEP(4, cdb, c10_lba, block_num);\\
UDI_MDEP(2, cdb, c10_len, data_len);\\
(cdb)->c10_ctrl = 0;           \\
}
```

See the UDI SCSI Driver Specification for additional examples of the construction of such structures and the use of the helper macros.

19.5.2 Helper Macros

These macros are provided for use in the handling of hardware structures built using the byte-by-byte structure layout, which typically is only used with hardware protocol structures. However these macros should be useful in general with any hardware structures that have non-naturally-aligned multi-byte quantities (or in general any multi-bit quantity that isn't naturally aligned on a power-of-two byte boundary) because such quantities will need to be split up to meet the general requirements on the definition of hardware structures as given in Section 8.9.

19.5.2.1 Bit-field Macros

Bit-fields in these macros refer to sub-divisions of a byte and are defined by a 2-tuple (p,len) where p is the bit position in the byte of the least significant bit in the bit field, and len is the size in bits of the bit field. The bit position p is 0 for the least significant bit in the byte, 7 for the most significant bit. Note that $0 \leq p \leq 7$ and $1 \leq \text{len} \leq 8-p$.

The following bit-field macros are defined for use on udi_ubit8_t variables:

```
/* Create a p,len bit mask (all 1's in the bit field, zeroes elsewhere) */
#define UDI_BFMASK(p, len) \
    (((1<<(len))-1) << (p))

/* Extract an unsigned p,len bit-field from val */
#define UDI_BFGET(val, p, len) \
    (((val) >> (p)) & ((1<<(len))-1))

/* Deposit val into the p,len bit-field in dst */
#define UDI_BFSET(val, p, len, dst) \
    ((dst) = ((dst) & ~UDI_BFMASK(p,len)) | \
     ((val) << (p)) & UDI_BFMASK(p,len)))
```

19.5.2.2 Multi-Byte Macros

These macros have arguments called “structp” and “field”. The structp argument is a pointer to a structure that contains N single-byte (and byte-aligned) members whose names are “field”0, “field”1, ... “field”N, which together represent a multi-byte quantity in the structure. “field”0 is the least significant byte; “field”N is the most significant.

For example, a structure that has a 3-byte “sum” field might have field names of sum0, sum1, and sum2, and the complete 24-bit field could be extracted from the structure into a variable, my_sum, using the UDI_MBGET macro as follows:

```
my_sum = UDI_MBGET(3, &my_struct, sum);
```

To write a value into this 3-byte field, use the UDI_MBDEP macro as follows:

```
UDI_MBSET(3, &my_struct, sum, my_sum);
```

The following multi-byte macros are defined:

```
/* Extract a multi-byte value from a structure. */
#define UDI_MBGET(N, structp, field) \
    UDI_MBGET_##N (structp, field)
#define UDI_MBGET_2(structp, field) \
    ((structp)->field##0 | ((structp)->field##1 << 8))
```

```
#define UDI_MBGET_3(structp, field) \
    (((structp)->field##0 | ((structp)->field##1 << 8) | \
     ((structp)->field##2 << 16)) \
#define UDI_MBGET_4(structp, field) \
    (((structp)->field##0 | ((structp)->field##1 << 8) | \
     ((structp)->field##2 << 16) | ((structp)->field##3 << 24)) \
/* Deposit "val" into a multi-byte field in a structure. */ \
#define UDI_MBSET(N, structp, field, val) \
    UDI_MBSET_##N (structp, field, val) \
#define UDI_MBSET_2(structp, field, val) \
    (((structp)->field##0 = (val) & 0xff, \
      (structp)->field##1 = ((val) >> 8) & 0xff) \
#define UDI_MBSET_3(structp, field, val) \
    (((structp)->field##0 = (val) & 0xff, \
      (structp)->field##1 = ((val) >> 8) & 0xff, \
      (structp)->field##2 = ((val) >> 16) & 0xff) \
#define UDI_MBSET_4(structp, field, val) \
    (((structp)->field##0 = (val) & 0xff, \
      (structp)->field##1 = ((val) >> 8) & 0xff, \
      (structp)->field##2 = ((val) >> 16) & 0xff, \
      (structp)->field##3 = ((val) >> 24) & 0xff)
```

19.5.3 Endian-Swapping Utilities

UDI provides two basic macros for endian translation of a single 16-bit or 32-bit integer: UDI_ENDIAN_SWAP16 and UDI_ENDIAN_SWAP32, respectively. The utility functions udi_endian_swap and udi_endian_swap_copy provide for the endian conversion of greater than 32-bit integers. These two functions have **rep_count** and **stride** parameters, allowing for multiple byte-swaps of a given size in a single call. These can be used to byte-swap each element in an array (as shown by the UDI_ENDIAN_SWAP_ARRAY macro), or to do more complex sequences of byte-swaps across sub-elements of an array of structures.

Note – Except for use with DMA-able shared control structures (described in the UDI Physical I/O Specification), drivers should use the byte-by-byte structure layout rather than using these utilities, since the relationship between driver and device endianness is not indicated in any UDI-specified fashion, either at compile time or at run time.

NAME	UDI_ENDIAN_SWAP16/32	<i>Byte-swap 16 or 32-bit integers</i>
SYNOPSIS	<pre>#include <udi.h> #define UDI_ENDIAN_SWAP_16(data16) \ ((((data16) & 0x00ff) << 8) \ (((data16) >> 8) & 0x00ff)) #define UDI_ENDIAN_SWAP_32(data32) \ ((((data32) & 0x000000ff) << 24) \ (((data32) & 0x0000ff00) << 8) \ (((data32) >> 8) & 0x0000ff00) \ (((data32) >> 24) & 0x000000ff))</pre>	
ARGUMENTS	<p>data16 is a 16-bit data value.</p> <p>data32 is a 32-bit data value.</p>	
DESCRIPTION	<p>UDI_ENDIAN_SWAP_16 byte-swaps a single 16-bit data value; UDI_ENDIAN_SWAP_32 byte-swaps a single 32-bit data value. These two macros provide basic endian translation of an individual data item. See udi_endian_swap and udi_endian_swap_copy for endian utilities which provide for larger than 32-bit endian translation and which allow for multiple byte-swaps in a single call.</p> <p>These macros must be called as if they, respectively, had the following functional interface:</p> <pre>void UDI_ENDIAN_SWAP_16 (udi_ubit16_t data16); void UDI_ENDIAN_SWAP_32 (udi_ubit32_t data32);</pre>	
EXAMPLES		
REFERENCES	udi_endian_swap, udi_endian_swap_copy	

NAME	udi_endian_swap	<i>Byte-swap data in place</i>
SYNOPSIS	<pre>#include <udi.h> void udi_endian_swap (void *src, udi_ubit8_t swap_size, udi_ubit8_t stride, udi_ubit16_t rep_count);</pre>	
ARGUMENTS	<p>src points to a memory area across which byte swapping will be performed, as defined below.</p> <p>swap_size is the size, in bytes, of each element to be byte-swapped.</p> <p>swap_size must be a power-of-two which is less than or equal to stride.</p> <p>stride is the number of bytes by which to increment src between repetitions.</p> <p>rep_count is the number of swap_size byte-swaps to perform.</p>	
DESCRIPTION	The udi_endian_swap function generates rep_count byte-swaps, of swap_size bytes each, of data elements in the memory area pointed to by src . Between each repetition src is incremented by stride to obtain the next element to be swapped. The results are written in place in the src memory; only src memory actually byte-swapped will be changed as a result of this call.	
	If swap_size is 1 this function acts as a no-op.	
EXAMPLES	If the driver has an array of structures that need to be endian translated, it can call udi_endian_swap once for each type of multi-byte field in the structure, setting src to point to the address of the field in the zeroth element of the array (i.e., <code>&array[0]->my_field</code>), swap_size to the size of the field, stride to the size of the structure, and rep_count to the number of elements in the array.	
REFERENCES	udi_endian_swap_copy	

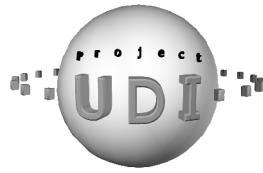
NAME	udi_endian_swap_copy	<i>Byte-swap data and copy the results</i>
SYNOPSIS	<pre>#include <udi.h> void udi_endian_swap_copy (const void *src, void *dst, udi_ubit8_t swap_size, udi_ubit8_t stride, udi_ubit16_t rep_count);</pre>	
ARGUMENTS	<p>src points to a memory area across which byte swapping will be performed.</p> <p>dst points to a memory area into which the results of the byte-swap will be placed. This memory area must be at least the size of the memory area pointed to by src.</p> <p>swap_size is the size, in bytes, of each element to be byte-swapped. swap_size must be a power-of-two which is less than or equal to stride.</p> <p>stride is the number of bytes by which to increment src between repetitions.</p> <p>rep_count is the number of swap_size byte-swaps to perform.</p>	
DESCRIPTION	The udi_endian_swap_copy function operates exactly like udi_endian_swap except that the results are placed in the memory pointed to by dst , in the same relative positions as the swap_size elements exist in the src memory area. Only these swap_size elements in dst will be updated as a result of this call; other portions of the dst memory will not be affected.	
	If swap_size is 1 this function acts as a no-op.	
EXAMPLES	If the driver has an array of structures that need to be endian translated, it can call udi_endian_swap once for each type of multi-byte field in the structure, setting src to point to the address of the field in the zeroth element of the array (i.e., <code>&array[0]->my_field</code>), swap_size to the size of the field, stride to the size of the structure, and rep_count to the number of elements in the array.	
REFERENCES	udi_endian_swap	

NAME	UDI_ENDIAN_SWAP_ARRAY	<i>Byte-swap each element in an array</i>
SYNOPSIS	<pre>#include <udi.h> #define \ UDI_ENDIAN_SWAP_ARRAY(src, element_size, count) \ udi_endian_swap(src, element_size, \ element_size, count)</pre>	
ARGUMENTS	<p>src points to an array of count elements, each of which are element_size bytes in size.</p> <p>element_size is the size, in bytes, of each element to be byte-swapped. element_size must be a power-of-two.</p> <p>count the number of elements in the src array to be byte-swapped.</p>	
DESCRIPTION	<p>UDI_ENDIAN_SWAP_ARRAY byte-swaps count elements in the array pointed to by src. The results are written in place in the src array.</p> <p>The macro UDI_ENDIAN_SWAP_ARRAY 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_endian_swap:</p> <pre>void UDI_ENDIAN_SWAP_ARRAY (void *src, udi_ubit8_t element_size, udi_ubit16_t count);</pre>	
EXAMPLES		
REFERENCES	udi_endian_swap	



UDI Core Specification

Section 4: Core Metalanguages



20.1 Overview

A metalanguage defines a communication protocol for use over a UDI channel. Metalanguages allow two drivers or a driver and the environment to communicate with each other in a strongly-typed manner. Driver writers may define their own metalanguages, for intra-driver communication (between multiple regions within a driver instance) or between layers in a driver stack, but a number of metalanguages are defined within the UDI specifications. These latter metalanguages are called *standard metalanguages*, while other metalanguages are referred to as *custom metalanguages*. Note that for complete driver portability a custom metalanguage that is used for driver-internal communication must be implemented using the interfaces defined in Section 4: MEI Services.

Only a few generally applicable metalanguages are defined in the Core Specification: the Management Metalanguage, which is needed by all UDI drivers for configuration and system management purposes; the Internal Management Metalanguage, which is available for use by any multi-region driver; and the Generic I/O Metalanguage, which is available for use as a generic pass-through metalanguage. Other metalanguages are defined in separate UDI specification books; e.g., the Bridge Metalanguage in the “UDI Physical I/O Specification”, the SCSI Metalanguage in the “UDI SCSI Driver Specification”, etc.

The remainder of this chapter describes conventions and requirements common to all metalanguages.

20.2 Standard Metalanguage Functions and Parameters

Required metalanguage functions (i.e., the “ops_init” and “cb_init” functions) are defined in Section 6.3, “Module-Global Initialization Functions”. Standard parameters and calling conventions of channel operations are defined in Section 6.4, “Channel Operations”. Note that typically UDI’s metalanguage specifications only document the caller-side interfaces of channel operations; the corresponding callee-side entry point interface can be derived by removing the first parameter.

Default routines are provided for events which will never occur for some drivers. For example, if a driver creates no secondary regions, either statically or dynamically, it will never receive the `udi_region_attach_ind` channel operation, and so may set its `udi_region_attach_ind_op_t` to `udi_region_attach_unused` instead of providing a routine that always just responded immediately with `udi_region_attach_res`.

20.3 Channel Operation Suffixes

Channel operations generally operate in pairs: e.g., a request and a corresponding acknowledgment, or an event and a corresponding response. The corresponding handshake operation tends to be generally useful in the UDI model to cycle the control block back to the requestor or event initiator or to cycle back buffers and other transferable objects. As a result, metalanguage operations tend to fall into the following categories. (The term “driver” below may also refer to the Management Agent in the case of the Management Metalanguage).

Table 20-1 Channel Operation Categories

Suffix	Category	Description
_req	Request	Operations sent by a driver to request service from another driver
_ack	Acknowledgement	Operations returned to handshake a request and indicate results and status (normal completion, warning, or failure)
_nak	Negative Acknowledgement	Operations returned to handshake a request and specifically indicate non-normal status; used in performance-critical cases to keep status checks out of the normal path
_ind	Event Indication	Operations sent by a driver to notify another driver of an event
_res	Event Response	Operations returned in response to event indications

The suffixes listed in the above table are conventionally used in names of channel operation that belong to the corresponding categories.

20.4 General Rules for Handling Channel Operations

This section provides general information on the handling of channel operations performed on a driver.

Upon receipt of a channel operation, a driver may handle it in one of the following ways, depending on the relationship of the request to the driver's current state:

- Normal: the driver knows how to handle the operation, and can handle it in its present state.
- Not understood: the driver doesn't understand how to handle the operation, given its parameters.
- Unsupported: the driver understands the operation, but chooses not to support it.
- Invalid state: the driver knows how to handle the operation, but is not in a state such that it can process the request.

20.4.1 Normal Operation Handling

Under normal circumstances, a driver is called to handle a request, does some processing, and performs an acknowledge operation. (Processing an operation may or may not involve performing operations on other drivers.) If the operation was an indication that came from a parent driver, the driver may have to interrogate its hardware about the interrupt and/or perform an operation on its child (or parent) driver.

20.4.2 Not Understood Operations

When a driver receives an operation that is not understood because the parameters or associated data are not appropriate, the driver should reply back with a status value of UDI_STAT_NOT_UNDERSTOOD indicating that the operation is invalid.

20.4.3 Unsupported Operations

When a driver receives an operation that it recognizes as being validly defined, but for which it has not implemented the specified function, the driver should reply back with a status value of UDI_STAT_NOT_SUPPORTED indicating that the function is not supported.

20.4.4 Invalid State Operations

If a driver is called to handle an operation that it would normally accept, but cannot process at this time because of its current state, the action is metalanguage-dependent. Some metalanguages will elect to have the driver reject the operation to the originator with a status value of UDI_STAT_INVALID_STATE. Others will require the driver to internally queue the operation in order to try it at a later time. In either case, if the operation is properly timed with respect to the metalanguage, *no error should be logged*. If the operation violates the sequence rules of the metalanguage, an error log to this effect should be generated. It is important that all drivers for a given metalanguage function the same way, in order to provide consistent behavior to client modules.

20.5 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:

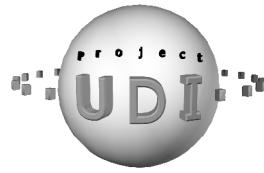
1. A `udi_channel_event_ind` operation of type `UDI_CHANNEL_CLOSED` is passed to the other end of a channel whenever a channel is closed. This can occur as a result of an explicit `udi_channel_close` (see page 16-9) or as a result of the region being prematurely terminated by the environment.
2. A `udi_channel_event_ind` operation of type `UDI_CONSTRAINTS_CHANGED` is passed to the other end of a channel whenever a driver (or the environment) makes a constraints propagation request via a call to `udi_constraints_propagate` (see page 12-14).
3. A `udi_channel_event_ind` operation of type `UDI_BUFFER_RECOVERED` is passed to one end of a channel whenever a buffer is being recovered from a prematurely-terminated neighboring driver instance on the other end of the channel. The buffer must have been previously passed over this channel via a channel operation that marks buffers for recovery return (as specified by the metalanguage definition for that channel op). Normal error events within a region will be handled within that region and the buffer will be returned to the original owner via a metalanguage operation, but in the case of an abrupt region kill situation, the region being killed is not entered and therefore does not have an opportunity to return these buffers to the original owner, so the buffers must be returned using this recovery mechanism. For more information on buffer recovery see Section 13.4.2, “Buffer Recovery Mechanism,” on page 13-11.

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	<i>Control block for channel_event_ind</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; udi_ubit8_t event; udi_constraints_t new_constraints; void *orig_tr_context; udi_buf_t buf; } udi_channel_event_cb_t; /* Channel event types */ #define UDI_CHANNEL_CLOSED 0 #define UDI_CONSTRAINTS_CHANGED 1 #define UDI_BUFFER_RECOVERED 2</pre>	
MEMBERS	<p>gcb is the standard control block section providing scratch space and context information.</p> <p>event is the type of event that is being indicated:</p> <p>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 <code>udi_channel_close</code> on its end of the channel.</p> <p>UDI_CONSTRAINTS_CHANGED indicates that the new_constraints field of the control block contains a valid constraints handle provided by the driver at the other end of the channel via a call to <code>udi_constraints_propagate</code>. The new constraints should be kept or merged with existing constraints.</p> <p>UDI_BUFFER_RECOVERED indicates that a buffer is being returned to this region as a result of the premature termination of a region at the other end of the associated channel. The orig_tr_context field is set according to the metalanguage specification for the channel operation last used to send the buffer across the channel; the buf field contains a handle to the buffer being returned.</p> <p>new_constraints is valid when the event is UDI_CONSTRAINTS_CHANGED, in which case it contains a constraints handle representing the new constraints attributes being passed by the initiating module. Otherwise it will be set to NULL_CONSTRAINTS.</p> <p>orig_tr_context is the transaction context associated with the buffer (if any) being recovered for a UDI_BUFFER_RECOVERED channel event. The metalanguage specifies which buffers are to be</p>	

	<p>recovered with this method and the corresponding transaction ID field to be used (if any). This value is indeterminate and must not be used for all other types of channel events.</p>
	<p>buf is the handle for the buffer being returned for UDI_BUFFER_RECOVERED channel events. The metalanguage specification defines which buffers (if any) are to be recovered with this mechanism for each request. This field will be set to NULL_BUF for all other types of channel events.</p>
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 is returned to the environment by calling udi_cb_free.
REFERENCES	udi_channel_close, udi_constraints_propagate, UDI_DL_BUF_RETURN, UDI_DL_BUF_CONTEXT

NAME	udi_channel_event_ind	<i>Channel event notification (env-to-driver)</i>
SYNOPSIS	<pre>#include <udi.h> void udi_channel_event_ind (udi_channel_t target_channel, udi_channel_event_cb_t *cb);</pre>	
ARGUMENTS		target_channel , cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.
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 occurred on the other end of the channel. The type of event that has occurred, and additional parameters for the event, are contained in the <i>udi_channel_event_cb_t</i> 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 21.6, “Unbinding Operations,” on page 21-22. If a driver closes its end of the channel itself (with <i>udi_channel_close</i>) before the other end is closed or before a <i>udi_channel_event_ind</i> 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.
WARNING		Drivers may not invoke this operation.



21.1 Overview

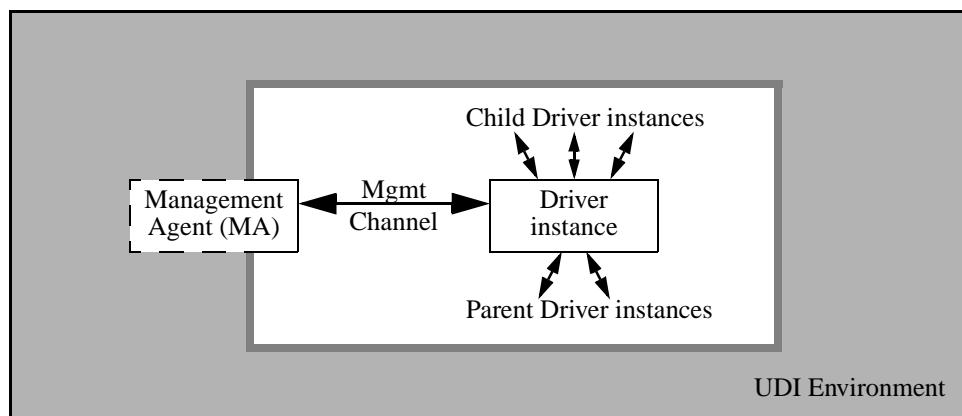
This chapter defines the channel operations and associated service calls of the Management Metalanguage, which is used by the environment's *Management Agent* (MA) to communicate with the primary region of each driver instance for configuration and other system management purposes. The MA uses the Management Metalanguage to communicate binding information between driver instances including initial channels between related driver instances, according to system configuration information. The MA may also participate in cleanup operations by indicating that various channels are to be closed via indirect requests or direct operations.

Each subsection defines the channel operations, associated control blocks and service calls, constraints and guidelines for the use of each operation, and any error conditions that can occur.

21.2 Management Agent

The MA is an abstract entity within the UDI environment; it represents the environment's control and configuration mechanisms. All device configuration, driver instantiation, and initial channel creation is driven and controlled by the MA. The MA is an integral part of the environment and is always present in any UDI environment implementation.

The Management Metalanguage defines the communication between a driver instance and the MA and is used for the *management channel* (sometimes abbreviated “mgmt channel”), which is a channel that is always present between the MA and the driver's primary region. When the MA wishes to instruct a driver instance to perform a management-related operation it will generate a Management Metalanguage request to that driver instance over its management channel. The driver will respond via a Management Metalanguage acknowledgement.



Additionally the driver can indicate system-level events by generating responses to corresponding Management Metalanguage inquiries, but the driver will never initiate an operation to the MA.

21.2.1 Driver Instantiation

The MA is the entity within the UDI environment that determines the need for and subsequently drives the instantiation of a driver instance. It does this by combining the information obtained from the enumeration responses of the parent driver instance with the information provided by the driver's static driver properties and its `init_module` routine.

The following model describes the typical sequence of events surrounding the instantiation of a driver region. The actual sequence of operations and MA functionality may differ but the events described by this model will be valid from the driver's perspective for any environment:

1. ... the parent driver instance has previously been instantiated and has a management channel established between its primary region and the MA.
2. The MA issues an *enumeration request* to the parent driver instance.
3. The parent driver returns an *enumeration response* with information describing an instance of a "child",¹ to the MA.
4. The MA locates a driver that corresponds to the instance attribute information provided by the parent in the enumeration response.
5. The selected driver may have been previously loaded or it may be dynamically loaded at this time. In either case, once it is loaded, its `init_module` routine will be run.
6. The MA creates a new driver instance for the child, including a primary region, optional secondary regions, a management channel connecting the MA to the child's primary region, and a *bind channel* between the parent and child driver instances to be used for initial parent/child communications. The MA also creates internal configuration information and prepares any necessary resources.
7. The MA issues a `udi_usage_ind` operation to the newly created driver to allow it to perform preliminary initialization of internal data structures and adjust resource usage and trace levels.
8. If the MA has auto-created secondary regions as a result of `udi_secondary_region_init` calls from the child driver's `init_module` routine, it now issues a *region-attach indication* to the child's primary region for every secondary region created, along with a handle to the *internal management channel* that the MA created for communication between the driver's primary and secondary regions.
9. The MA then issues a *prep-for-child request* to the parent driver instance over its management channel. This instructs the parent to prepare for binding to a new child. The parent's end of the bind channel is passed to the parent as part of this operation.

1. The child represents a physical or logical entity that can be accessed via the parent device instance and for which a corresponding driver should be instantiated. Examples of parent/child relationships are: (1) a PCI Bus/PCI Card, a SCSI HBA/SCSI Disk, etc.

10. After performing necessary internal initialization (which may include instantiating secondary regions and passing the new bind channel to one of those secondary regions, to be anchored there) the parent driver issues a *prep-for-child acknowledgement* to the MA via the management channel.
11. The MA then issues a *bind-to-parent request* to the new child instance via the corresponding operation over the management channel to the child's primary region. The child's end of the new bind channel is passed to the child as part of this operation. The child may anchor the bind channel in its primary region or pass it to a secondary region to be anchored there.
12. The child now performs internal initialization and then issues a metalanguage-specific bind request to the parent region via the bind channel, which by this point is guaranteed to be fully anchored between the parent and child instances.
13. The parent instance processes the child's bind request and returns a bind response via the same bind channel.
14. The child completes initialization and then issues a *bind-to-parent acknowledgement* back to the MA via the management channel, signifying the completion of the parent/child initialization process. On receipt of this acknowledgement, the parent and child are considered "open for business" and should expect to perform normal activities.
15. The MA may then issue an enumerate request to the new child instance to determine if it has children of its own. If it does, this model is repeated for the new parent/child group starting at Step 3 above. If the current child has no children of its own, the enumeration response to the MA indicates this fact and the configuration of this driver instance has completed.

21.3 Management Metalanguage Considerations

Because of its unique nature, the Management Metalanguage differs in a number of ways from other UDI metalanguages:

- The management channel is always created by the MA and exists prior to invocation of the driver's primary region.
- There is only one management channel per driver instance, regardless of the number of regions for that instance, and it is automatically anchored in the driver's primary region.
- There is no way for the driver to create a management channel.
- There is only one role the driver may play for the Management Metalanguage since the other end is always handled by the MA; therefore only one channel ops vector is defined for this metalanguage.
- There is no `ops_init` function for the management channel since the operations vector is supplied via `udi_primary_region_init` (see **udi_primary_region_init** on page 9-8).
- There are no `cb_init` or `cb_select` functions for management agent control blocks since all management control blocks are allocated by the MA and passed to the driver via the single Management Channel (*i.e.* all Management Metalanguage requests are initiated by the MA).

Management Metalanguage Considerations *Mgmt*

- The ***mgmt_scratch_size*** parameter to `udi_primary_region_init` determines the scratch space size for *all* control blocks used on the management channel.
- There is no `channel_event_ind_op` in the Management Metalanguage ops vector, since the MA will never close the management channel while the driver instance exists.

21.4 Initialization

This section describes the system calls that are performed to register this module for communications with the Management Agent and the operations that are used on that channel to initialize the driver.

21.4.1 Tracing Control Operations

One of the functions of the Management Metalanguage is to implement control over the tracing operations performed by a driver. When the system (or user) wishes to obtain specific classes of tracing information from a driver a management operation is issued to the driver over the Management Metalanguage channel. The driver updates its internal tracing operations accordingly and acknowledges the update back to the Management Agent.

For more information on generating trace information please consult the *UDI Tracing and Logging* Chapter of this specification.

21.4.2 Resource Management

The UDI environment handles resource management (passively) through the `udi_limits_t` information and actively through the selective control and completion of individual driver resource allocation requests. Both of these activities primarily focus on the ability of the UDI environment to restrict the initial and ongoing supply of new resources to a driver but do not have any effect on the amount of resources that the driver is currently maintaining.

As an ultimate measure, the UDI environment may choose to kill and unload driver instances in an attempt to deal with critical resource availability conditions. However, it is frequently desirable to manage resources in a more graceful manner that will allow the driver to voluntarily release resources back to the UDI environment while continuing to operate.

To support this, the UDI Management Metalanguage provides the ability for the UDI environment to indicate the current level of environment-supplied resource availability to a driver instance for resources.

NAME	udi_mgmt_ops_t	<i>Management Meta channel ops vector</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_usage_ind_op_t *usage_ind_op; udi_region_attach_ind_op_t *region_attach_ind_op; udi_prep_for_child_req_op_t *prep_for_child_req_op; udi_bind_to_parent_req_op_t *bind_to_parent_req_op; udi_unbind_parent_req_op_t *unbind_parent_req_op; udi_enumerate_req_op_t *enumerate_req_op; udi_child_release_ind_op_t *child_release_ind_op; udi_devmgmt_req_op_t *devmgmt_req_op; udi_final_cleanup_req_op_t *final_cleanup_req_op; } udi_mgmt_ops_t;</pre>	
DESCRIPTION	<p>udi_mgmt_ops_t is an <i>ops vector</i> structure that contains function pointers for all of the Management Metalanguage entry point routines for the driver. It is passed to udi_primary_region_init to register these entry points with the environment. For additional information on ops vectors, see “Channel Operations Vectors” on page 6-2.</p> <p>For the usage_ind, region_attach_ind, and enumerate_req entry points, the driver can specify a corresponding environment-provided default entry point. See the corresponding reference page for details under which the default entry point may be used.</p>	
REFERENCES	<hr/> <p>Note – The udi_mgmt_ops_t deviates from other channel ops vectors, in that it does not have a channel_event_ind operation.</p> <hr/> <p>udi_primary_region_init</p>	

NAME	udi_mgmt_cb_t	<i>Common Management Control Block</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; } udi_mgmt_cb_t;</pre>	
MEMBERS	gcb	is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.
DESCRIPTION		The udi_mgmt_cb_t defines a control block which is used in Management Metalanguage operations that do not require any additional parameters in the control block. This control block is used for requests or indications from the Management Agent to the target driver and is passed back to the Management Agent in the acknowledgement or response operation.
REFERENCES		udi_cb_t

NAME	udi_usage_cb_t	<i>Resource indication and trace level control block</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; udi_trevent_t trace_mask; udi_index_t meta_idx; } udi_usage_cb_t;</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>trace_mask is a bitmask describing the types of trace events that the driver is to report. Setting one of the trace mask bits enables tracing for all events of that type. Some event types are selectable on a metalanguage basis using the meta_idx field.</p> <p>For the definition of the udi_trevent_t type and the corresponding trace events that may be enabled, see udi_trevent_t on page 17-3 of the <i>UDI Tracing and Logging</i> Chapter.</p> <p>meta_idx is a metalanguage index that indicates to which metalanguage the metalanguage-selectable trace_mask bits apply. 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 28), or 0 for the Management Metalanguage.</p>	
DESCRIPTION	The trace modification control block is used with the udi_usage_ind and udi_usage_res operations to modify the set of trace classes for which trace events should be reported.	
REFERENCES	udi_trevent_t, udi_usage_ind, udi_usage_res	

NAME	udi_usage_ind	<i>Indicate desired resource usage and trace levels</i>
SYNOPSIS	<pre>#include <udi.h> void udi_usage_ind (udi_channel_t target_channel, udi_usage_cb_t *cb, udi_ubit8_t resource_level); udi_usage_ind_op_t udi_static_usage; /* values for resource_level */ #define UDI_RESOURCES_CRITICAL 1 #define UDI_RESOURCES_LOW 2 #define UDI_RESOURCES_NORMAL 3 #define UDI_RESOURCES_PLENTIFUL 4</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>resource_level is an indication of the current UDI environment resource capabilities.</p>	
TARGET CHANNEL	The affected driver instance’s management channel.	
DESCRIPTION	<p>The udi_usage_ind operation is used to provide advisory information to the target driver regarding the current level of system resources and the desired level of tracing information to be reported by the driver.</p> <p>This is the first operation that will be issued over the management channel to a newly created target region. The Management Agent may also call this operation any time it wishes to change the level of trace output from the driver or if the system resource levels change significantly.</p> <p>This operation is used to activate and deactivate tracing of particular types of events. The Management Agent issues this request to the driver when it is to activate or deactivate tracing according to the values set in the trace_mask and meta_idx fields of the cb, as described for udi_usage_cb_t.</p> <p>The resource_level argument is used by the UDI Management Agent to indicate the UDI environment’s current resource levels to the UDI driver. The UDI driver may use this information to reduce or increase its resource allocation and utilization as appropriate to assist in overall UDI resource management. Driver may treat some or all of the resource levels as the same.</p> <p>Resource levels that may be indicated by the resource_level argument are:</p> <p>UDI_RESOURCES_CRITICAL - Indicates that UDI resource levels are critically low and that the driver should release any and all resources that are not absolutely essential to its operation. It is expected that drivers that receive and respond to this indication will operate in reduced capability and/or reduced performance modes.</p>	

	<p>UDI_RESOURCES_LOW - Indicates that UDI resource levels are below a system threshold level. This is typically an indication that resource allocation requests will begin to be delayed and that system performance may be beginning to be affected due to the resource restrictions. The driver should return any extra resources that it is currently holding but it is not expected to release any resources that would adversely affect its operational state to any great degree.</p> <p>UDI_RESOURCES_NORMAL - Indicates that UDI resource levels are normal and system resources are still readily available. This is the default state that drivers may assume on startup until indicated otherwise. This indication is most frequently used to undo the effects of a previous UDI_RESOURCES_LOW or UDI_RESOURCES_CRITICAL indication.</p> <p>UDI_RESOURCES_PLENTIFUL - Indicates that UDI resource usage levels are low relative to the amount of total available resources. Drivers receiving this indication should operate in an “extravagant” mode to achieve absolute peak performance and functionality levels, acquiring additional resources if necessary.</p> <p>The MA is not required to notify the driver of resource level changes nor is it restricted to indicating updates levels in any particular sequence. The UDI environment may unload UDI drivers whether or not they have reduced their resource allocations levels if the environment determines that this driver should be unloaded to retrieve needed resources.</p> <p>Drivers that will not adjust their resource utilization levels or their level of tracing output may specify <code>udi_static_usage</code> as the entry point for this operation.</p>
WARNING	Drivers must not invoke this operation.
REFERENCES	<code>udi_usage_cb_t</code> , <code>udi_usage_res</code>

NAME	udi_usage_res	<i>Resource usage and trace level response operation</i>
SYNOPSIS	<pre>#include <udi.h> void udi_usage_res (udi_channel_t target_channel, udi_usage_cb_t *cb);</pre>	
ARGUMENTS		target_channel , cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.
TARGET CHANNEL		The affected driver instance’s management channel.
DESCRIPTION		<p>The udi_usage_res operation is used by the driver to respond to update information provided by the Management Agent from a udi_usage_ind operation.</p> <p>If the driver does not support one or more of the requested trace event types, it must clear the corresponding bits in the trace_mask field of the control block in the acknowledgement. For example, if tracing is entirely unsupported a trace_mask of zero will always be returned.</p> <p>This operation is also used to acknowledge the receipt of the resource indication; it does <i>not</i> indicate to the Management Agent that the driver has completed any internal resource adjustments as a result of the resource indication, but merely indicates that the driver is aware of the new resource levels.</p> <p>The control block must be the same one passed to udi_usage_ind.</p>
REFERENCES		udi_usage_cb_t , udi_usage_ind

NAME	udi_region_attach_cb_t	<i>Control block for region attachments</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; udi_index_t region_idx; udi_channel_t channel1_to_secondary; } udi_region_attach_cb_t;</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>region_idx is an index that indicates the type of region being attached. This matches a region_idx value previously passed to <i>udi_secondary_region_init</i> by the driver's <i>init_module</i> routine.</p> <p>channel1_to_secondary is a channel handle to the anchored end of a channel to the new secondary region. The new region gets a handle to its (anchored) end of this channel via its <i>udi_init_context_t</i> structure.</p>	
DESCRIPTION	A <i>udi_region_attach_cb_t</i> control block is used with <i>udi_region_attach_ind</i> to complete the attachment of channels between primary and secondary regions in a driver instance.	
REFERENCES	<i>init_module</i> , <i>udi_secondary_region_init</i>	

NAME	udi_region_attach_ind	<i>Secondary region attachment event indication</i>
SYNOPSIS	<pre>#include <udi.h> void udi_region_attach_ind (udi_channel_t target_channel, udi_region_attach_cb_t *cb); </pre> <pre>udi_region_attach_ind_op_t udi_region_attach_unused;</pre>	
ARGUMENTS		target_channel , cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.
TARGET CHANNEL		The target channel for this operation is the management channel for the primary region to which the new secondary region is being attached.
DESCRIPTION		<p><code>udi_region_attach_ind</code> is called by the Management Agent to inform the driver that the secondary region which the driver asked to be created (via <code>udi_secondary_region_init</code>) has now been created and is being attached to the driver’s primary region. The driver must respond with <code>udi_region_attach_res</code>.</p> <p><code>udi_region_attach_ind</code> will be invoked in the primary region before any other channel operation is invoked; it is never invoked in a secondary region.</p> <p><code>udi_region_attach_ind</code> is not used for secondary regions created dynamically by <code>udi_secondary_region_create</code>.</p> <p><code>udi_region_attach_unused</code> may be used as a driver’s <code>udi_region_attach_ind</code> entry point if the driver does not expect any secondary region attachment indications; i.e. it is either a single-region driver or creates all of its secondary regions dynamically.</p>
WARNING		Drivers must not invoke this operation.
REFERENCES		<code>init_module</code> , <code>udi_secondary_region_init</code> , <code>udi_region_attach_res</code> , <code>udi_secondary_region_create</code>

NAME	udi_region_attach_res	<i>Secondary region attach event response</i>
SYNOPSIS	<pre>#include <udi.h> void udi_region_attach_res (udi_channel_t target_channel, udi_region_attach_cb_t *cb);</pre>	
ARGUMENTS	target_channel , cb	are standard arguments described in the “Channel Operations” section of “ <i>Standard Calling Sequences</i> ”.
TARGET CHANNEL		The target channel is the management channel for the driver’s primary region.
DESCRIPTION	udi_region_attach_res	is called by the driver in response to a udi_region_attach_ind call from the Management Agent. The control block must be the same one passed to udi_region_attach_ind .
REFERENCES		udi_region_attach_ind

21.5 Binding Operations

The environment's Management Agent (MA) establishes a connection between two driver instances through a series of Management Metalanguage operations that take the connection through the following sequence of states:

1. Unbound
2. Parent Preparation In Progress (`udi_prep_for_child_req` sent)
3. Parent Prepped (`udi_prep_for_child_ack` received)
4. Bind In Progress (`udi_bind_to_parent_req` sent)
 - a. Child Unbound
 - b. Parent Bound (metalanguage-specific bind request sent from child to parent)
 - c. Constraints Propagated (parent has called `udi_constraints_propagate` on the child's channel)
 - d. Child Bound (metalanguage-specific bind ack received by child from parent)
5. Bound (`udi_bind_to_parent_ack` received)

While in the Bind In Progress state the child driver initiates a metalanguage-specific binding operation over the newly created parent/child channel and the parent acknowledges the binding. The child then progresses to the Bound state, whereupon the drivers can begin normal communication over the established channel. In some cases, the MA may need to abort the connection before it reaches the Bound state; special operations allow it to return to the Unbound state from the intermediate Prepped and Bind In Progress states.

Additionally, the binding operations may be used to suspend and resume an established connection to facilitate online replacement, addition, or deletion of a device.

NAME	udi_bind_cb_t	<i>Prep_for_child/bind_to_parent control block</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; udi_channel_t bind_channel; udi_index_t meta_idx; } udi_bind_cb_t;</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>bind_channel is a handle for the called driver's end of the communication channel between the two drivers.</p> <p>meta_idx is a metalanguage index number that identifies which metalanguage is being used (and therefore the type of child driver) for this bind channel. It will be set by the MA to 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 28).</p>	
DESCRIPTION	<p>The udi_bind_cb_t control block is a structure used between MA and driver instances for bind-related operations. When passed to the target driver, this control block itself provides the context for the binding and is returned to the MA when the binding operation is complete.</p> <p>udi_bind_cb_t control blocks are only allocated by the environment, not by drivers.</p>	

NAME	udi_prep_for_child_req	<i>Prepare for a child bind request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_prep_for_child_req (udi_channel_t target_channel, udi_bind_cb_t *cb, void *child_context);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>child_context is a driver-private context pointer previously associated with the child device when it was discovered (enumerated) by the parent.</p>	
TARGET CHANNEL	The affected driver instance’s management channel	
DESCRIPTION	<p>The udi_prep_for_child_req channel operation is used to prepare a driver for handling a new child driver instance prior to the activation of that child and subsequent receipt of a metalanguage-specific bind request from that child. This gives the parent driver a chance to associate device attributes cached in the child_context with the channel context for the new bind channel. The parent driver will have previously set child_context through a udi_enumerate_ack response operation.</p> <p>The bind_channel field in the control block is a handle to the parent driver’s end of the new bind channel. The parent driver’s end of the bind channel may be loose or anchored, depending upon the value of child_bind_ops_idx passed to udi_primary_region_init from the parent’s init_module routine.</p> <p>If child_bind_ops_idx was non-zero, the bind channel will automatically be anchored to the driver instance’s primary region, using the designated child-binding channel ops, and will have its context set to the initial region data for this region. Otherwise, the parent driver’s end of the bind channel will be a loose end and must be anchored, by calling udi_channel_anchor, before the udi_prep_for_child_req request is acknowledged.</p> <p>Once the parent driver has completed this operation, it must acknowledge the request by calling udi_prep_for_child_ack.</p> <p>The parent driver must not invoke any operations on the bind channel until it receives an operation from the child driver (that is, the child driver “goes first”). UDI guarantees that no operations will be received from the child driver until after udi_prep_for_child_ack is called.</p>	
WARNING	Drivers must not invoke this operation.	
REFERENCES	udi_primary_region_init , udi_prep_for_child_ack , udi_enumerate_req , udi_enumerate_ack	

NAME	udi_prep_for_child_ack	<i>Acknowledge prep_for_child operation</i>
SYNOPSIS	<pre>#include <udi.h> void udi_prep_for_child_ack (udi_channel_t target_channel, udi_bind_cb_t *cb, udi_status_t status);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>status indicates the success or failure of this operation.</p>	
TARGET CHANNEL	The Management Agent’s management channel	
DESCRIPTION	<p>Acknowledge a completed udi_prep_for_child_req request.</p> <p>The control block must be the same one passed to udi_prep_for_child_req.</p>	
STATUS VALUES	<p>UDI_OK operation succeeded; driver is ready to be bound.</p> <p>UDI_STAT_NOT_RESPONDING The specified child is no longer present and is being unenumerated, so no binding should be established.</p> <p>UDI_STAT_RESOURCE_UNAVAIL There are no resources available to maintain a binding to this child. This most commonly occurs when this is a new parent device hot-replaced into an existing device tree and this parent is unable to handle all of the previous parent’s bindings.</p>	
REFERENCES	udi_prep_for_child_req	

NAME	udi_bind_to_parent_req	<i>Request to bind to a parent driver</i>
SYNOPSIS	<pre>#include <udi.h> void udi_bind_to_parent_req (udi_channel_t target_channel, udi_bind_cb_t *cb, void *parent_context);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>parent_context is a context pointer chosen by the driver to specify a given parent. If NULL this is an initial binding of this parent; otherwise, this is a re-binding of this driver to this parent. Re-binding can occur, for example, after an abrupt termination of a parent.</p>	
TARGET CHANNEL	The affected driver instance’s management channel	
DESCRIPTION	<p>The udi_bind_to_parent_req channel operation is used to request that a driver bind with the specific instance of a parent driver represented by bind_channel in the cb.</p> <p>The bind_channel field in the control block is a handle to the child’s end of the new bind channel. The child’s end of the bind channel may be loose or anchored, depending upon the value of parent_bind_ops_idx passed to udi_primary_region_init from the child’s init_module routine. If parent_bind_ops_idx was non-zero, the bind channel will automatically be anchored to the driver instance’s primary region, using the designated parent-binding channel ops, and will have its context set to the initial region data for this region.</p> <p>Otherwise, the child driver’s end of the bind channel will be a loose end and must be anchored, by calling udi_channel_anchor, before the udi_bind_to_parent_req request is acknowledged, and before using the channel to communicate with the parent driver.</p> <p>As part of the udi_bind_to_parent_req operation, the child driver must establish communication with the parent driver by performing some metalanguage-specific “bind” operation(s) on the bind channel. This initialization of the bind channel must complete before the udi_bind_to_parent_req can be complete.</p> <p>The child driver may examine the driver instance attributes initially established for this child instance by the parent’s udi_enumerate_ack operation and make changes or additions to these device attributes during udi_bind_to_parent_req.</p> <p>Once the child driver has completed the udi_bind_to_parent_req operation, it must acknowledge the request by calling udi_bind_to_parent_ack.</p>	

Note that `udi_bind_to_parent_req` may be called multiple times for the same driver instance, all with `parent_context` null, if it supports multiple parents.

If a re-bind is requested, as indicated by a non-NULL `parent_context`, then all state related to the parent device must be re-initialized, as the device may have been replaced and may not be identical. This includes, but is not limited to, re-propagation of constraints.

WARNING

Drivers must not invoke this operation.

REFERENCES

`udi_primary_region_init`, `udi_bind_to_parent_ack`,
`udi_enumerate_ack`

NAME	udi_bind_to_parent_ack	<i>Acknowledge completion of parent/child binding</i>
SYNOPSIS	<pre>#include <udi.h> void udi_bind_to_parent_ack (udi_channel_t target_channel, udi_bind_cb_t *cb, void *parent_context, udi_status_t status);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>parent_context is a context pointer chosen by the driver to represent this binding. This value will be used by UDI in future references to this binding (e.g., in a udi_unbind_parent_req call).</p> <p>status indicates the success or failure of the operation</p>	
TARGET CHANNEL	The Management Agent’s management channel	
DESCRIPTION	Acknowledge a completed udi_bind_to_parent_req request.	
	Even if the parent_context passed to udi_bind_to_parent_req was non-NUL (indicating that this parent is being re-bound), then the parent_context value returned with udi_bind_to_parent_ack becomes the new context value for this parent. In the case where the parent is rebound to the child, the actual parent instance may not be the same parent as was originally bound; the current driver must re-establish any parent-related resources, including DMA bindings and interrupt registration.	
STATUS VALUES	<p>The control block must be the same one passed to udi_bind_to_parent_req.</p> <p>UDI_OK operation succeeded, driver is bound.</p> <p>UDI_STAT_CANNOT_BIND Driver couldn’t bind to parent driver.</p> <p>UDI_STAT_TOO_MANY_PARENTS The udi_bind_to_parent_req cannot be supported because this driver instance is already bound to the maximum number of parents that it can support.</p> <p>UDI_STAT_BAD_PARENT_TYPE The udi_bind_to_parent_req 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.</p>	
REFERENCES	udi_bind_to_parent_req	

21.6 Unbinding Operations

To tear down an already established connection, the MA simply needs to take the connection through the following states:

1. Bound
2. Unbind In Progress (udi_unbind_parent_req sent)
3. Unbound (udi_unbind_parent_ack received)

NAME	udi_unbind_parent_req	<i>Request driver to unbind from h/w or parent</i>
SYNOPSIS	<pre>#include <udi.h> void udi_unbind_parent_req (udi_channel_t target_channel, udi_mgmt_cb_t *cb, void *parent_context);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>parent_context is the context pointer for the parent binding to be released. This value was given to the MA by the driver in the <code>udi_bind_to_parent_ack</code> for the same binding.</p>	
TARGET CHANNEL	The affected driver instance’s management channel	
DESCRIPTION	<p>The MA issues this operation to request that the driver unbind from a specific instance of a parent driver identified by the parent_context. If bound to the specified parent driver, the receiving driver must perform metalanguage-dependent unbind operations on the bind channel to the parent (if such unbind operations are defined in the metalanguage).</p> <p>As much as possible, the device should be shut down, as if it might be removed or powered off after this operation completes if this is the last parent. Storage device write-back caches should be flushed to permanent storage, for example.</p> <p>The driver must then free all resources associated with this driver instance relative to the servicing of this parent, including destroying any secondary regions, whether created statically with <code>udi_secondary_region_init</code> or dynamically with <code>udi_secondary_region_create</code>, by having each secondary region call <code>udi_channel_close</code> on all of its channels.</p> <p>When the binding is released, the driver must perform a <code>udi_unbind_parent_ack</code> operation, passing it the same control block as was passed to <code>udi_unbind_parent_req</code>.</p>	
WARNING	Drivers must not invoke this operation.	

NAME	udi_unbind_parent_ack	<i>Acknowledge completion of an unbind</i>
SYNOPSIS	<pre>#include <udi.h> void udi_unbind_parent_ack (udi_channel_t target_channel, udi_mgmt_cb_t *cb);</pre>	
ARGUMENTS	target_channel , cb	are standard arguments described in the “Channel Operations” section of “ <i>Standard Calling Sequences</i> ”.
TARGET CHANNEL		The affected driver instance’s management channel
DESCRIPTION		Handshakes a <i>udi_unbind_parent_req</i> operation. This indicates to the MA that the unbind operation has completed. The driver must first do any required metalanguage-specific unbinding on the parent channel, followed by the clean up of any parent-related resources. If the parent channel has been closed or becomes closed during this sequence the driver is effectively unbound from the parent and must clean up any parent-related resources and call <i>udi_unbind_parent_ack</i> .
		The driver is not allowed to fail an unbind operation.
		The control block must be the same one passed to <i>udi_unbind_parent_req</i> .
REFERENCES		<i>udi_unbind_parent_req</i>

NAME	udi_final_cleanup_req	<i>Request driver to release all resources and context prior to instance unload.</i>
SYNOPSIS	<pre>#include <udi.h> void udi_final_cleanup_req (udi_channel_t target_channel, udi_mgmt_cb_t *cb);</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>child_context is the context pointer for the parent binding to be released. This value was given to the MA by the driver in the <code>udi_bind_to_parent_ack</code> for the same binding.</p>	
TARGET CHANNEL	The affected driver instance’s management channel	
DESCRIPTION	<p>The MA issues this operation to request that the driver fully remove all resources and region instance context. The MA will typically invoke this request after all parents and children have been unbound from the instance and the instance is now to fully be removed from the system. The driver must fully return any resources allocated on behalf of the instance. The driver must also destroy any secondary regions, whether created statically with <code>udi_secondary_region_init</code> or dynamically with <code>udi_secondary_region_create</code>, by having each secondary region call <code>udi_channel_close</code> on all of its channels.</p> <p>Upon completion of this request, the driver must perform a <code>udi_final_cleanup_ack</code> operation, passing it the same control block as was passed to <code>udi_final_cleanup_req</code>. After sending the ack, it should logically appear as if the driver instance has not appeared in the system.</p>	
WARNING	Drivers must not invoke this operation.	
REFERENCES	<code>udi_final_cleanup_ack</code>	

NAME	udi_final_cleanup_ack	<i>Acknowledge completion of a final cleanup request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_final_cleanup_ack (udi_channel_t target_channel, udi_mgmt_cb_t *cb);</pre>	
ARGUMENTS	target_channel , cb	are standard arguments described in the “Channel Operations” section of “ <i>Standard Calling Sequences</i> ”.
TARGET CHANNEL		The affected driver instance’s management channel
DESCRIPTION		Handshakes a <i>udi_final_cleanup_req</i> operation. This indicates to the MA that the removal operation has completed. The driver is not allowed to fail a removal operation. The control block must be the same one passed to <i>udi_final_cleanup_req</i> .
REFERENCES		<i>udi_final_cleanup_req</i>

21.7 Enumeration Operations

This subsection of the Management Metalanguage defines child enumeration operations. As described in “Driver Instantiation” on page 21-2, each driver instance is given the opportunity to enumerate the presence, number, and initial instance attributes of any children that are detected by the current driver instance. The typical response of the MA to an enumerated child is to create a driver instance and initiate a bind operation between that child and the current driver instance (which becomes the parent of that new child).

The MA issues an enumeration request to the driver to instruct it to respond with information regarding the child devices present. The enumeration request and its response, along with associated data objects, are described in this subsection.

21.7.1 Enumeration Attributes

Each child device instance is described by a set of *enumeration attributes*. These attributes are a set of driver instance attributes that serve to identify the specific child instance. The set of attributes needed to identify a child instance are defined by the associated metalanguage and represent the set of attributes initially attached to the child instance when it is instantiated.

21.7.2 Child Context

Each child device instance is also identified by an *child context*. The child context is a unique value associated with the child device instance from the point of view of the parent driver. The child context value is assigned by the parent driver instance and the value itself is interpreted only by that parent driver instance. The child context value is expected to be unique to a specific child of that parent and to be valid for the duration of the existence of that child instance.

When a new child instance is enumerated, both the child context and the associated set of enumeration attributes for that child are specified by the parent driver.

21.7.3 Enumeration Filters

In cases where the Management Agent wishes to query the target driver for information about a specific range of child devices instead of obtaining information about any and all children, it may use an *enumeration filter* to specify this restricted level of interest to the target driver. The target driver may use this enumeration filter as a hint to indicate which child or children the Management Agent is interested in. The enumeration filter specification is only a hint, however, and the target driver is free to ignore it and return information about any or all known children that make up the same set or a superset of those specified by the filter.

Enumeration filters are especially useful in situations where the child enumeration can be quite large or when enumeration can take significant amounts of time to probe for each child. If the Management Agent knows about these characteristics it can provide appropriate enumeration filter hints to limit the scope of the enumeration query. Environment implementations that are not concerned with the size of the child instance space or the amount of time taken to enumerate that space will typically not provide any filter hints to the target driver.

An enumeration filter is specified as one or more enumeration attributes and the desired range and granularity of values for those attributes. The associated metalanguage is responsible for providing information about the interpretation of enumeration attribute filters as well as defining the enumeration attributes. Any enumeration attribute not specified in the filter is “unfiltered” and will match any value.

21.7.4 Parent Context

In cases where the Management Agent wishes to query the target driver instance for information about which of its children would be affected by the unbinding of a particular parent of the driver instance, it may specify a *parent context*. The *parent context* identifies which parent may be unbound. The *parent context* is a driver-specific handle that was given to the MA by the driver instance when it originally bound to the parent. *Parent Context* can logically be thought of as an enumeration filter where the filter is the identified parent. However, **this is not a hint**. The driver must enumerate all children that would become dysfunctional if the instances’ parent were to be unbound. The driver must not enumerate children that can continue unabated once the parent binding is lost. It is acceptable for both *parent context* and *enumeration filters* to be specified. If so, the driver is to enumerate only those children that would be affected by the unbinding of the indicated parent, but is allowed to further reduce this set based on application of the enumeration filters.

Enumeration based on parent context will normally be used during instance unbinding and hardware hot swap scenarios. It is used to determine the portion of the topology tree that will be unbound or affected by the hot swapping of a component or set of components.

Since most drivers have a direct relationship between the loss of a parent binding and the ability to service I/O requests from its children, the MA will, by default, assume that any parent unbinding affects all children of the instance. As such, the MA does not need to query the driver instance to enumerate the affected children. If a driver can insulate some or all of its children from the unbinding of a parent, the driver should include an “active_router” declaration in its static driver properties. This declaration will override the default behavior, causing the MA to query the driver instance for the set of affected children. (See Section 28.6.7, “Active_router Declaration,” on page 28-16.)

21.7.5 Unenumeration

The converse to the enumeration operation is the case where a child has “gone away” from the perspective of the parent driver. In UDI, this is handled by an *unenumeration* operation. This operation is performed as a variation of the enumeration operation: the child context value indicates which child has gone away and there are no enumeration attributes specified. This response to the enumeration request indicates to the Management Agent that the corresponding child instance is no longer valid and that the Management Agent should initiate cleanup operations for that child driver instance.

21.7.6 Notification of Topology Changes

As described above, the MA issues enumeration requests to the driver when it wishes to obtain information regarding child devices that are present. The same mechanism is used by the MA to obtain information about any subsequent topology changes. Once the known devices have been enumerated to the MA the MA will typically issue an enumeration request to the driver instructing the driver to tell it about any new enumeration events.

The driver will hold this enumeration request indefinitely until one of the following events occurs:

1. A child device is added or removed.
2. The enumeration request is overridden with a new enumeration request.
3. The driver instance is shutdown and removed.

In this manner, the MA keeps an enumeration request “posted” to the driver that the driver can use to inform the MA about any changes in the child topology of the driver.

The driver is only expected to maintain one posted enumeration request from the MA. If the MA issues another enumeration request the enumeration request being held should be acknowledged with a “no child events” status and the new enumeration request should be processed and (if necessary) held to use for future enumeration notifications to the MA. The MA will typically override a posted enumeration request when it wishes to rescan all known children or when it wishes to change the enumeration filter of the posted request.

21.7.7 Directed Enumeration

The children of a particular driver may not always be determined by performing a physical scan or other specific determination. In some cases the children are determined by the presence of other modules in the system, whereas in other cases the children are determined by system configuration, possibly resulting from input from the system administrator. This is typically true of pseudo-drivers and especially *orphan* drivers which have no parent regions.

To accomodate this type of configuration, UDI implements the concept of *directed enumeration* wherein the target driver is “directed” to enumerate a specific child instance by the Management Agent. The Management Agent will use the operations described in this section to request the target driver to enumerate a specific child; the resulting actions are identical to those that would be performed for a physically enumerated child: the target parent driver prepares internal management structures and then generates an enumeration acknowledgement for the new child. Subsequent child instance creation and binding operations then proceed normally.

NAME	udi_instance_attr_list_t	<i>Enumeration instance attribute list</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { char attr_name[64]; udi_ubit8_t attr_value[64]; udi_ubit8_t attr_length; udi_instance_attr_type_t attr_type; } udi_instance_attr_list_t; #define UDI_ATTR32_SET(v) \ { (v) & 0xff, ((v) >> 8) & 0xff, \ ((v) >> 16) & 0xff, ((v) >> 24) & 0xff } #define UDI_ATTR32_GET(va) \ ((va)[0] + ((va)[1] << 8) + \ ((va)[2] << 16) + ((va)[3] << 24))</pre>	
MEMBERS	<p>attr_name is the name of the attribute. This name must start with the ‘!’ character indicating that this is an enumeration instance attribute.</p> <p>attr_value is the value of this attribute for the child instance. Must not be zero.</p> <p>attr_length is the valid length (in bytes) of the attr_value.</p> <p>attr_type is the attribute type as specified for udi_instance_attr_type_t on page 15-6. Must not be UDI_ATTR_NONE or UDI_ATTR_FILE.</p>	
DESCRIPTION	<p>The udi_instance_attr_list_t structure is used to describe a single enumeration instance attribute. The parent driver allocates space for a contiguous array of these structures as a movable memory block in order to provide information describing the child instance for the udi_enumerate_ack operation.</p> <p>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.</p>	
REFERENCES	<p>udi_mem_alloc, udi_enumerate_req, udi_enumerate_ack, udi_instance_attr_type_t</p>	

NAME	udi_filter_element_t	<i>Enumeration filter element structure</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { char attr_name[64]; udi_ubit8_t attr_min[64]; udi_ubit8_t attr_min_len; udi_ubit8_t attr_max[64]; udi_ubit8_t attr_max_len; udi_instance_attr_type_t attr_type; udi_ubit32_t attr_stride; } udi_filter_element_t;</pre>	
MEMBERS	<p>attr_name is the name of the attribute to be filtered.</p> <p>attr_min is the minimum acceptable value for the attribute in this filter. When combined with the attr_max value an inclusive range of valid attribute values for the filter is specified.</p> <p>attr_min_len specifies the valid length (in bytes) of the attr_min value. Must not be zero.</p> <p>attr_max is the maximum acceptable value for the attribute in this filter.</p> <p>attr_max_len specifies the valid length (in bytes) of the attr_max value. Must not be zero.</p> <p>attr_type is the attribute type as specified for udi_instance_attr_type_t on page 15-6. Must not be UDI_ATTR_NONE or UDI_ATTR_FILE.</p> <p>attr_stride specifies the periodicity of the filter match values starting at attr_min and ending at or above attr_max.</p>	
DESCRIPTION	<p>The udi_filter_element_t structure is used to specify an attribute being filtered and the valid range and periodicity of the values for that filter. A child instance's attributes should match one of these values to be validated against this filter.</p> <p>The interpretation of attr_stride is unique to each attr_name attribute and is specified by the metalanguage when describing that attribute.</p> <p>The interpretation of attr_min and attr_max values will be determined by the attr_type specified for that attr_name attribute when the attribute is being enumerated.</p> <p>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_min and attr_max, and attr_min_len and attr_max_len must be 4. In this case, UDI_ATTR32_SET and UDI_ATTR32_GET must be used to access attr_min and attr_max.</p>	

EXAMPLE If the current target driver instance is a SCSI HD that is enumerating SCSI Metalanguage children (SCSI peripheral devices) for the MA, the following filter specification:

```
udi_filter_element_t f = {  
    "scsi_target", UDI_ATTR32_SET(2), 1,  
    UDI_ATTR32_SET(15), 1,  
    UDI_ATTR_UBIT32, 3 };
```

indicates that the MA only cares about SCSI targets with one of the following SCSI Target ID values: 2, 5, 8, 11, 14.

REFERENCES *udi_instance_attr_type_t*, *udi_instance_attr_list_t*,
udi_enumerate_req

NAME	udi_enumerate_cb_t	<i>Enumeration operation control block</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; void *parent_context; udi_instance_attr_list_t *attr_list; udi_ubit8_t attr_list_length; const udi_filter_element_t *filter_list; udi_ubit8_t filter_list_length; } udi_enumerate_cb_t;</pre>	
MEMBERS	<p>gcb is the standard control block information structure.</p> <p>parent_context is the parent context that the enumerated children are to be related to. This value was given to the MA by the driver in the <code>udi_bind_to_parent_ack</code> for the same binding. If no parent context is specified, the value shall be NULL.</p> <p>attr_list is a pointer to movable memory containing a contiguous array of <code>udi_instance_attr_list_t</code> structures. This attribute list is used to describe the initial set of instance attributes being enumerated by this driver; additionally, it specifies the desired set of child attributes for a directed enumeration operation.</p> <p>attr_list_length is the number of entries in attr_list.</p> <p>filter_list is a pointer to an array of <code>udi_filter_element_t</code> structures used to specify the attributes filter to be applied to this enumeration request by the target driver.</p> <p>filter_list_length is the number of elements in the filter_list array.</p>	
DESCRIPTION	<p>The <code>udi_enumerate_cb_t</code> is the control block used for the <code>udi_enumerate_req</code> and <code>udi_enumerate_ack</code> channel operations. This control block is allocated by the MA when the MA issues the <code>udi_enumerate_req</code> and is to be returned to the MA by the driver in the <code>udi_enumerate_ack</code> response.</p> <p>The filter_list passed to the driver in this control block on the enumeration request should be returned to the MA in the acknowledgement operation and should not be used after that acknowledgement has been issued.</p> <p>The driver fills in the attribute passed in this control block with the attributes of the child node for the <code>udi_enumerate_ack</code> operation. If this is used for a directed enumeration operation, the attribute list is also used to specify the minimum set of enumeration attributes for the child that the driver is being directed to enumerate; these initial attributes must be preserved for the acknowledgement although additional attributes may be added.</p>	
REFERENCES	<code>udi_enumerate_req</code> , <code>udi_enumerate_ack</code>	

NAME	udi_enumerate_req	<i>Request information regarding a child instance</i>
SYNOPSIS	<pre>#include <udi.h> void udi_enumerate_req (udi_channel_t target_channel, udi_enumerate_cb_t *cb, udi_ubit8_t enumeration_level); /* values for enumeration_level */ #define UDI_ENUMERATE_START 1 #define UDI_ENUMERATE_START_RESCAN 2 #define UDI_ENUMERATE_NEXT 3 #define UDI_ENUMERATE_NEW 4 #define UDI_ENUMERATE_DIRECTED 5 udi_enumerate_req_op_t udi_enumerate_no_children;</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>enumeration_level is a value describing the relationship of this enumeration request to previous enumeration requests (see below).</p>	
TARGET CHANNEL	The affected driver instance’s management channel.	
DESCRIPTION	<p>The Management Agent issues this request to obtain enumeration information about child instances of the current driver. If there is information that may be returned for child instances, the receiving driver will allocate movable memory to initialize as an array of udi_instance_attr_list_t structures to describe that child instance.</p> <p>The enumeration_level argument indicates the relationship of this enumeration request to any previous enumeration requests based on the following values:</p> <p>UDI_ENUMERATE_START - The Management Agent is starting a new enumeration cycle. The target driver should provide information about all child instances regardless of whether those children were previously enumerated (or are currently actively bound). Subsequent enumeration requests are expected to have an enumeration_level of UDI_ENUMERATE_NEXT to obtain information about more child instances; receipt of another UDI_ENUMERATE_START will restart the enumeration back at the beginning.</p>	

UDI_ENUMERATE_START_RESCAN - This enumeration level is the same as the UDI_ENUMERATE_START enumeration level except that with this level the driver should not use any previously obtained or cached information to report child instances and should perform physical verification as appropriate to obtain the filtered list of children.

UDI_ENUMERATE_NEXT - The target driver should return information about the next child instance relative to the one described in the previous enumerate_req/enumerate_ack operation. When all children have been enumerated the udi_enumerate_ack operation will return an enumeration_flag of UDI_ENUMERATE_DONE. The driver is expected to maintain the context (in its region data) to implement the UDI_ENUMERATE_NEXT operations properly.

Although the enumeration *filter_list* is passed to the driver each time the udi_enumerate_req is called, the MA will not change the filter specification from the previous udi_enumerate_req call if the UDI_ENUMERATE_NEXT enumeration level is specified.

UDI_ENUMERATE_NEW - The target driver should return information about any child instances changes (*i.e.* new or removed children) that have been detected since the completion of any previous enumeration cycles. The MA will typically issue a udi_enumerate_req of this type to the target driver that will be held by the target driver for an indefinite period of time until such a child event occurs, at which point the target driver will indicate the event by completing this request with a udi_enumerate_ack operation.

UDI_ENUMERATE_DIRECTED - This enumeration level is an indication to the target driver that it should create a child with the specific attributes indicated in the associated attribute list. This is used when the Management Agent needs to instantiate children as a result of external configuration information rather than hardware probed configuration.

If a UDI_ENUMERATE_NEW is received after a UDI_ENUMERATE_START (and zero or more UDI_ENUMERATE_NEXT's), but before the driver has provided information regarding all child instances, it will be treated as if it were a UDI_ENUMERATE_NEXT.

The *filter_list* specified in the enumeration control block specifies the attribute filter hints to be applied to the enumeration by the target driver. The target driver may use these hints to select which child instances to return information about, or it may ignore these hints and return a superset of the filtered children.

If the *parent_context* in the enumeration control block is non-null, it identifies the potential parent unbinding that the enumeration is to be in respect to.

`udi_enumerate_no_children` may be used as a driver's `udi_enumerate_req` entry point if the driver never needs to enumerate any child instances. It will simply acknowledge the request with appropriate values set to indicate that there are and never will be any children of this device.

WARNING

Drivers must not invoke this operation.

REFERENCES

`udi_enumerate_ack`, `udi_instance_attr_list_t`,
`udi_enumerate_cb_t`

NAME	udi_enumerate_ack	<i>Provide child instance information</i>
SYNOPSIS	<pre>#include <udi.h> void udi_enumerate_ack (udi_channel_t target_channel, udi_enumerate_cb_t *cb, udi_ubit8_t enumeration_result, void *child_context, udi_ubit8_t attr_valid_length);</pre> <p><i>/* values for enumeration_result */</i></p> <pre>#define UDI_ENUMERATE_OK 0 #define UDI_ENUMERATE_LEAF 1 #define UDI_ENUMERATE_DONE 2 #define UDI_ENUMERATE_RESCAN 3 #define UDI_ENUMERATE_REMOVED 4 #define UDI_ENUMERATE_MORE_ATTRS 5 #define UDI_ENUMERATE_FAILED 6</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “<i>Standard Calling Sequences</i>”.</p> <p>enumeration_result is a value indicating what type of enumeration acknowledgement is being generated.</p> <p>child_context is a driver-private context pointer supplied to identify the child instance being enumerated to a subsequent <code>udi_prep_for_child_req</code> operation. This argument is not used or changed by the MA or the environment.</p> <p>attr_valid_length indicates the number of attributes that are being returned as valid in the control block’s attr_list (or the number desired if the enumeration_result is <code>UDI_ENUMERATE_MORE_ATTRS</code>).</p>	
TARGET CHANNEL	The affected driver instance’s management channel.	
DESCRIPTION	<p>The <code>udi_enumerate_ack</code> channel operation is used in response to a <code>udi_enumerate_req</code> channel operation and provides information about a particular child device instance.</p> <p>The attr_list in the control block specifies the volatile enumeration instance attributes that will be present for the child driver instance when it is created. These enumeration attributes may be obtained by the child driver itself via calls to <code>udi_instance_attr_get</code> in its <code>udi_bind_to_parent_req</code> routine to obtain additional information about the instance being created. All child instances that match the <code>filter_list</code> in the enumeration request control block must be included in an enumerate acknowledgement. Other children may also be included.</p>	

The **child_context** is preserved by the environment and passed back to this driver as part of the udi_prep_for_child_req operation to provide context information to this driver and identify an explicit child enumeration. This value will not be changed by the environment.

The **enumeration_result** argument must be one of the following values and indicates the type of enumeration response being generated by the driver and how the associated **child_context** and **attr_list** arguments should be interpreted:

UDI_ENUMERATE_OK - Indicates that the enumeration is returning a valid child instance enumeration and that no special cases apply. The **child_context**, and **attr_valid_length** values are set accordingly.

UDI_ENUMERATE_LEAF - Indicates that there are and never will be any children of this device.

UDI_ENUMERATE_DONE - Indicates that all children have been enumerated and no more children exist.

UDI_ENUMERATE_RESCAN - Indicates that the Management Agent should re-scan the entire set of children via UDI_ENUMERATE_START and UDI_ENUMERATE_NEXT operations. This can be returned in situations where there are multiple changes to the child instances, child instances have changed during the enumeration process, or when the target driver is unable to determine the exact change that has occurred.

This value must only be returned for UDI_ENUMERATE_NEW and UDI_ENUMERATE_NEXT requests.

UDI_ENUMERATE_REMOVED - Indicates that the specified child instance has been removed (as opposed to added) and that the Management Agent should initiate handling of a device removal. The **child_context** indicates which child has been removed by passing the same value that the child was originally enumerated with. If **child_context** is NULL then this operation indicates that the current (target) device instance has detected the abrupt removal of its own hardware and that it itself should be removed by the Management Agent.

This value must only be returned for UDI_ENUMERATE_NEW requests.

It is important to note that the use of UDI_ENUMERATE_REMOVED for a UDI_ENUMERATE_NEW request is an *explicit* unenumeration of a child device. A child device may also be *implicitly* unenumerated by not listing it as part of a UDI_ENUMERATE_START / UDI_ENUMERATE_NEXT scan.

Whether explicitly or implicitly unenumerated, the driver must

maintain the validity of the **child_context** associated with this child until the Management Agent acknowledges the unenumeration with a udi_child_release_ind operation.

UDI_ENUMERATE_MORE_ATTRS - Indicates that the target driver wishes to enumerate a child but that the attribute list supplied in the control block is not large enough to contain the set of attributes for the child. The Management Agent is expected to allocate a larger attribute list and then re-issue the enumeration request to this driver with the larger sized attribute list. The desired size of the attribute list is specified in the **attr_valid_length** argument; no attributes are actually returned with this response, however.

UDI_ENUMERATE_FAILED - Indicates that the directed enumeration cannot be satisfied by this target driver. This may only be returned for UDI_ENUMERATE_DIRECTED requests.

In all cases except UDI_ENUMERATE_OK, the contents of **attr_list** is ignored and returned unchanged. The **attr_valid_length** argument is always zero except for UDI_ENUMERATE_OK where it indicates the number of child enumeration attributes or UDI_ENUMERATE_MORE_ATTRS where it indicates the number of entries needed in the **attr_list** to enumerate the child. In all cases except UDI_ENUMERATE_OK and UDI_ENUMERATE_REMOVED, the **child_context** value is ignored.

Table 21-1 **enumeration_flag** value usage

enumeration_flag	valid for	child_context	attr_valid_length
UDI_ENUMERATE_xxx			
OK	all	new child instance value	number of child instance enumeration attributes specified
LEAF	all	ignored	0
DONE	all	ignored	0
RESCAN	NEXT NEW	ignored	0
REMOVED	NEW	child instance to be removed (NULL for self)	0
MORE_ATTRS	all	ignored	desired number of entries in the control block's attr_list
FAILED	DIRECTED	ignored	0

The control block must be the same one passed to udi_enumerate_req.

REFERENCES

udi_enumerate_req, udi_instance_attr_list_t,
udi_enumerate_cb_t, udi_child_release_ind

NAME	udi_child_release_ind	<i>Request to release child context</i>
SYNOPSIS	<pre>#include <udi.h> void udi_child_release_ind (udi_channel_t target_channel, udi_mgmt_cb_t *cb, void *child_context); udi_child_release_ind_op_t udi_child_release_ind_unused;</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “<i>Standard Calling Sequences</i>”.</p> <p>child_context is a child context value previously provided by the driver in a <i>udi_enumerate_ack</i> operation.</p>	
TARGET CHANNEL	The Management Agent’s management channel to the parent driver.	
DESCRIPTION	<p>The Management Agent issues this indication to the parent driver to confirm that the child associated with this child_context has been removed and that the child context may now be released. The parent driver may indicate that the child instance has gone away via a UDI_ENUMERATE_REMOVED response in the <i>udi_enumerate_ack</i> operation but must maintain the child context until the removal is confirmed via this <i>udi_child_release_ind</i> operation.</p> <p><i>udi_child_release_ind_unused</i> may be used as a driver’s <i>udi_child_release_ind</i> entry point if the driver uses <i>udi_enumerate_no_children</i> or never indicates UDI_ENUMERATE_REMOVED via <i>udi_enumerate_ack</i>.</p>	
WARNING	<p>Drivers must not invoke this operation.</p> <p>Drivers must maintain the validity of any child context provided to the MA until this indication has been received.</p>	
REFERENCES	<i>udi_mgmt_cb_t</i> , <i>udi_child_release_res</i> , <i>udi_enumerate_req</i> , <i>udi_enumerate_ack</i>	

NAME	udi_child_release_res	<i>Acknowledge child release request.</i>
SYNOPSIS	<pre>#include <udi.h> void udi_child_release_res (udi_channel_t target_channel, udi_mgmt_cb_t *cb);</pre>	
ARGUMENTS		<i>target_channel, cb</i> are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.
TARGET_CHANNEL		The parent driver’s primary region management channel.
DESCRIPTION		This operation is used by the parent driver to acknowledge receipt and handling of the <i>udi_child_release_ind</i> operation and must be called after the latter operation has been handled internally.
		The control block must be the same one passed to <i>udi_child_release_ind</i> .
REFERENCES		<i>udi_child_release_cb_t, udi_child_release_ind</i>

21.8 Device Management Operations

This subsection of the Management Metalanguage is used to manage the flow of I/O operations within a driver instance during hot swap scenarios. In addition to controlling further I/O operations, the Management Metalanguage also allows the driver instance to communicate its operational state to the MA. A driver instance may indicate that it cannot currently support the suspension of activity. The MA could then decide to discontinue the hot swap operation, or forcibly continue. The driver instance may indicate that it can suspend internally and will queue all new I/O requests. The MA could then decide to no longer propagate the hot swap operation to the driver instance's children, leaving them unaffected.

The following model describes the typical sequence of events surrounding a hot swap operation. The actual sequence of operations and MA functionality may differ but the events described by this model will be valid from the driver's perspective for any environment:

1. ... a hot swap event occurs and the driver instance is determined to be in the set of affected driver instances.
2. The MA issues a *Prepare_To_Suspend* operation to the driver instance. The driver instance takes appropriate action (see 20.8.1) and acknowledges the operation. Note: if the MA were to subsequently cancel the hot swap operation, it would issue a *Resume* operation to the driver instance.
3. The MA issues a *Suspend* operation to the driver instance. The driver instance takes appropriate action (see 20.8.2) and acknowledges the operation. Note: if the MA were to subsequently cancel the hot swap operation, it would issue a *Resume* operation to the driver instance.
4. If the instance is to be unbound, the MA will cause all children to unbind from the driver instance. Note: if the MA were to subsequently cancel the hot swap operation, it would cause the children to rebind to the driver instance.
5. If parent instance(s) are to be unbound, the MA will request the driver instance to unbind from the respective parent(s).
6. If the instance is to be removed from the system, the MA will invoke `udi_remove_instance_req()` to cause the instance to be fully removed.
7. ... the affected hardware is powered down, swapped, and re-enabled. The MA starts normal attachment. The hardware is identified as belonging to the driver.
8. If the driver was removed from the system, the instance is recreated and re-bound to its parent(s).
9. The MA enumerates the driver instance's children. If the children were unbound, the MA will initiate binding to each of the children. If the children were not unbound, the MA will issue a *Resume* operation to the driver instance.

21.8.1 Prepare To Suspend

This device management operation serves as an informational notice that a *Suspend* operation is about to be performed relative to the indicated parent. It serves the following purposes:

Relative to the driver instance:

1. If the instance cannot support suspending operation and/or unbinding, it shall return the proper error code in the acknowledgment.
2. As a configuration change is about to take place, changes to the instance's configuration, state, etc that may conflict with a configuration change must be avoided or kept track of.
3. To minimize the generation of new I/O traffic based on the receipt of unsolicited inbound requests, the instance should take action, if possible, to turn off unsolicited inbound traffic (for example, a network driver should turn off the reception of new packets).
4. To minimize the length of time that the *Suspend* operation will take, the instance should avoid, if possible, issuing new I/O requests to its parent.

Relative to the MA and the environment:

1. Based on the response of the driver instance, the MA is given an indication as to whether the hot swap operation can succeed. This allows the MA to determine if it should cancel the operation or whether it must forcibly remove this portion of the tree. If the MA is to cancel the operation, it can simply *Resume* operation on the device instances previously sent a *Prepare To Suspend*. This early failure notification allows the MA to avoid the costly unbinding and rebinding process on the portion of the topology that was traversed prior to the failure.

21.8.2 Suspend

This management operation instructs the driver instance to suspend all activity via the indicated parent. The instance is to no longer initiate transactions to the indicated parent. In addition, prior to acknowledging the *Suspend* operation, it is to wait for all transactions outstanding with the indicated parent to complete (successfully or otherwise).

If the instance determines that it is in a state that cannot be suspended, it shall return a proper error status in the acknowledgment.

If the instance receives new requests (from its children) that are targeted for the indicated parent, the instance can either queue the requests or discard the requests as appropriate for the instance's device model. If the instance is queuing requests, it must continue to process them in as much as it is capable relative to the metalanguage definition. In any case, the driver must ensure that the suspension is not directly apparent to its children, though there may be indirect effects, such as extended delays or additional retry requirements.

21.8.3 Shutdown

This management operation is identical to a *Suspend* operation with the addition that it also instructs the driver instance to shutdown and detach as much as possible with its associated hardware. All communication connections (pio handles, dma handles, etc) should be terminated. Upon a subsequent *Resume* operation, the driver will reattach and reinitialize its hardware as appropriate and resume operation. The *Shutdown* operation is commonly used when no further device activity is desired for some period of time (e.g. power may be removed from the device; the hardware is a PCMCIA card, which may be temporarily removed from the system; etc).

21.8.4 Parent Suspended

This management operation is a notification that is used by the MA to affect some level of flow control over resources and requests that are issued by instances that are descendants of a suspended or shutdown instance. Typically, this will be used when the MA determines it no longer needs to propagate the *Suspend* operation because it has encountered an instance that sufficiently queues new requests such that its children are no longer affected by the hot swap operation.

An instance that receives this operation should throttle its operation. The MA will send this instance a *Resume* operation once the ancestor has been resumed.

21.8.5 Resume

This management operation is used by the MA to cancel any *Prepare To Suspend*, *Suspend*, *Shutdown*, and *Parent Suspended* states in driver instances. It may be issued when the MA encounters a scenario in which the MA needs to abort the hot swap operation, or when sufficient hardware has been rebound such that I/O should resume.

If resuming from a suspend, a different parent device may have been re-bound, and this driver must adapt to all device property changes, such as those indicated by a constraints propagation. If the driver cannot, or chooses not to, maintain sufficient state to reprogram the (replacement) device when it is resumed, then it must respond to *Prepare To Suspend* with an indication that it does not support transparent resume.

NAME	udi_devmgmt_req	<i>Device Management request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_devmgmt_req (udi_channel_t target_channel, udi_mgmt_cb_t *cb, udi_ubit8_t mgmt_op, void *parent_context);</pre> <p>/* values for mgmt_op */</p> <pre>#define UDI_DMGMT_PREPARE_TO_SUSPEND 1 #define UDI_DMGMT_SUSPEND 2 #define UDI_DMGMT_SHUTDOWN 3 #define UDI_DMGMT_PARENT_SUSPENDED 4 #define UDI_DMGMT_RESUME 5</pre>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>mgmt_op is a value describing the operation type.</p> <p>parent_context is the parent context that indicates the parent for which the operation is to take place. This value was given to the MA by the driver in the <code>udi_bind_to_parent_ack</code> for the same binding.</p>	
TARGET CHANNEL	The Management Agent’s management channel to the parent driver.	
DESCRIPTION	<p>The Management Agent issues this request to manage I/O requests within a driver instance during hot swap operations.</p> <p>The mgmt_op argument must be one of the following values and indicates the type of management operation being requested:</p> <p>UDI_DMGMT_PREPARE_TO_SUSPEND - Indicates that a Suspend operation is about to take place.</p> <p>UDI_DMGMT_SUSPEND - Requests the instance to suspend all operation relative to the indicated parent, and queue or fail new requests that are received. The instance must not acknowledge the request until all outstanding requests to the indicated parent are complete. The device must be put in a state that is prepared for the possibility of having power removed (for example, disk caches must be flushed), but device state and communications connections should not be completely shutdown.</p> <p>UDI_DMGMT_SHUTDOWN - Treated as UDI_DMGMT_SUSPEND, with the addition that the device must be completely shutdown (in particular, all communications connections should be terminated).</p> <p>UDI_DMGMT_PARENT_SUSPENDED - Indicates that outbound traffic via the indicated parent has been suspended.</p>	

	<p>UDI_DMGMT_RESUME - Indicates that the instance is to cancel any suspended or throttled state and is to resume full operation.</p>
WARNING	Drivers must not invoke this operation.
REFERENCES	udi_devmgmt_ack, udi_mgmt_cb_t

NAME	udi_devmgmt_ack	<i>Acknowledge a device management request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_devmgmt_ack { udi_channel_t target_channel, udi_mgmt_cb_t *cb, udi_ubit8_t flags, udi_status_t status }</pre> <p><i>/* values for flags */</i> <i>#define UDI_DMGMT_NONTRANSPARENT (1 << 0)</i></p> <p><i>/* meta-specific status codes */</i> <i>#define UDI_DMGMT_STAT_ROUTING_CHANGE \ (UDI_STAT_META_SPECIFIC 1)</i></p>	
ARGUMENTS	<p>target_channel, cb are standard arguments described in the “Channel Operations” section of “Standard Calling Sequences”.</p> <p>status indicates the success or failure of the operation.</p>	
TARGET CHANNEL	The parent driver’s primary region management channel.	
DESCRIPTION	<p>The udi_devmgmt_ack channel operation is used in response to a udi_devmgmt_req channel operation and provides information about a device management function requested of an instance.</p> <p>The flags argument may include:</p> <p>UDI_DMGMT_NONTRANSPARENT - Indicates that the requested UDI_DMGMT_PREPARE_TO_SUSPEND or UDI_DMGMT_SUSPEND operation has been complied with. The instance is also indicating that it does not support transparent resume.</p> <p>The control block must be the same one passed to udi_devmgmt_req.</p> <p>UDI_STAT_NOT_SUPPORTED - Indicates that the instance has failed the UDI_DMGMT_PREPARE_TO_SUSPEND, UDI_DMGMT_SUSPEND, or UDI_DMGMT_SHUTDOWN request, because it does not maintain sufficient state to be able to suspend.</p> <p>UDI_STAT_INVALID_STATE - Indicates that the instance has failed the UDI_DMGMT_PREPARE_TO_SUSPEND, UDI_DMGMT_SUSPEND, or UDI_DMGMT_SHUTDOWN request because its hardware, configuration state, coding level, etc do not allow it to be suspended at this time.</p> <p>UDI_DMGMT_STAT_ROUTING_CHANGE - Indicates that the instance has failed the UDI_DMGMT_SUSPEND or UDI_DMGMT_SHUTDOWN request. The instance is indicating that the set of children related to the indicated parent has changed since it was last enumerated. The MA is to re-enumerate and resume the operation. Drivers that do not</p>	
STATUS VALUES		

REFERENCES

udi_devmgmt_req, udi_devmgmt_cb_t

declare the “active_router” property (see Section 28.6.7, “Active_router Declaration,” on page 28-16) must not use this status code.

21.9 Management Metalanguage States

The following states, along with the state diagram shown in Figure 21-1, define the valid states for a UDI driver relative to the Management Metalanguage and the allowed operations in each of the states.

Operations or events which cause a state change are indicated by a character label on the associated state change path in the state diagram; the character labels refer to events as shown in Table 21-2 below. If the operation is a success or failure indication, the success path is indicated by the single-character label and the failure path is indicated by a hash mark ('#') following the single-character label. Operations and events which are not listed in the state diagram do not cause state changes to occur.

Figure 21-1 Management Metalanguage State Diagrams

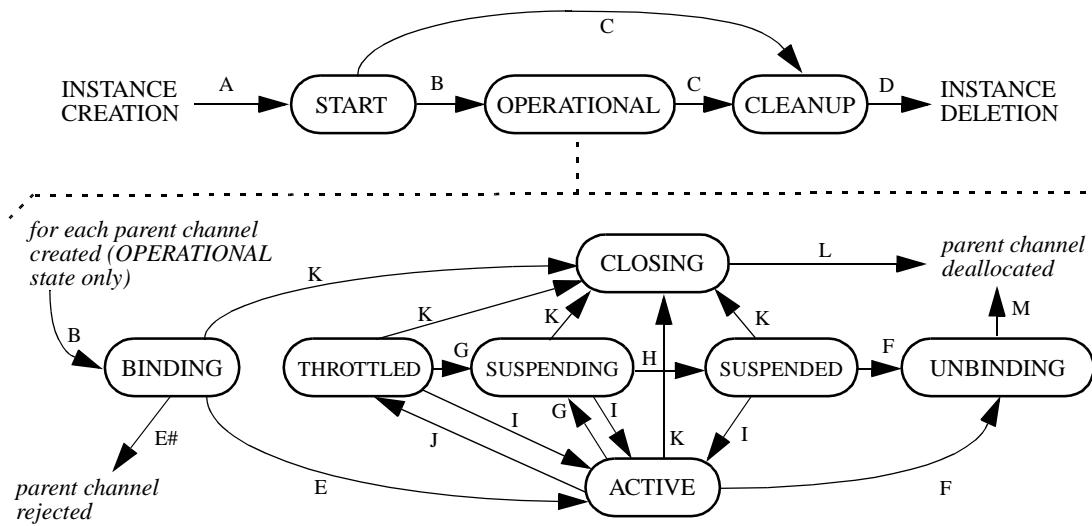


Table 21-2 Management Metalanguage Events

Event	Operation
A	udi_usage_ind
B	udi_bind_to_parent_req
C	udi_final_cleanup_req
D	udi_final_cleanup_ack
E	udi_bind_to_parent_ack
F	udi_unbind_parent_req
G	udi_devmgmt_req (Prepare to Suspend)
H	udi_devmgmt_req (Suspend or Shutdown)
I	udi_devmgmt_req (Resume)
J	udi_devmgmt_req (Parent Suspended)
K	udi_channel_event_ind (UDI_CHANNEL_CLOSED) on parent channel
L	udi_channel_close on parent channel
M	udi_unbind_parent_ack

21.9.1 Management Metalanguage States

START	This is the initial state for any newly instantiated region. A region in this state has been newly created along with a management channel and is being prepared for I/O operations, but is not yet bound to any parents or children. This is the only state wherein secondary regions will be automatically instantiated and all secondary regions will be instantiated before leaving this state.
OPERATIONAL	This is the primary state for a region instance. In this state, parent and child channels may be bound to the region and normal I/O channel operations and functionality may occur.
CLEANUP	This is the final state for a region and is entered from any state when <code>udi_final_cleanup_req</code> is received on that region's management channel. Any remaining resources held by the driver should be released, preparatory to this region instance being removed from the system.

21.9.1.1 Operational Sub-States

BINDING	A driver in the BINDING state is in the process of satisfying the MA's <code>udi_bind_to_parent_req</code> for a newly created parent bind channel. This is a transition state into the ACTIVE state.
ACTIVE	This is the normal functional state for the region. When in this state, the region is bound to one or more parents and may also be bound to one or more child regions. The region is expected to be able to handle I/O traffic and any associated device activity.
UNBINDING	A region enters this state when it has received a <code>udi_unbind_parent_req</code> request for the last parent region. In this state the driver is expected to complete any pending I/O to that parent and clean up any resources associated with that parent. Upon completion of this state (signalled by a <code>udi_unbind_parent_ack</code> operation) the parent channel will be deallocated.
	Note: the MA may later re-use the parent context to enter the BINDING state but other than possible re-use internal resources, the target driver should treat this binding as if it were a completely new binding.
THROTTLED	This state is entered when the Management Agent has suspended (or is in the process of suspending) a parent region of the current region; the region should throttle operations and generate as little traffic as possible to the parent channel(s).
SUSPENDING	This state is entered when the Management Agent is preparing to handle a device shutdown/suspension in order to replace the device or perform some other device management operation.
SUSPENDED	This state is entered when the Management Agent issues a device management operation instructing the region to temporarily halt I/O activities.
CLOSING	This state is entered when a UDI_CHANNEL_CLOSED event indication has been received on the parent's channel. This indicates that the parent was abruptly removed as part of a region kill or other catastrophic event. When in the CLOSING state the target driver must cleanup all resources relative to the parent device immediately without exchanging further channel operatoins with the parent region.

Table 21-3 Management Metalanguage: Valid Operations by State

Operation	START	OPERATIONAL							CLEANUP
		BINDING	ACTIVE	UNBINDING	THROTTLED	SUSPENDING	SUSPENDED	CLOSING	
udi_bind_to_parent_req	YES	no	no	no	no	no	no	no	no
udi_bind_to_parent_ack	no	YES	no	no	no	no	no	YES	YES
udi_unbind_parent_req	no	no	YES	no	no	no	YES	no	no
udi_unbind_parent_ack	no	no	no	YES	no	no	no	no	YES
udi_region_attach_ind	YES	no	no	no	no	no	no	no	no
udi_region_attach_res	YES	YES	no	no	no	no	no	no	YES
udi_prep_for_child_req	no	no	YES	no	YES	YES	YES	no	no
udi_prep_for_child_ack	no	no	YES	YES	YES	YES	YES	YES	YES
udi_enumerate_req	no	no	YES	no	YES	YES	YES	no	no
udi_enumerate_ack	no	no	YES	YES	YES	YES	YES	YES	YES
udi_child_release_ind	no	no	YES	no	YES	YES	YES	no	no
udi_child_release_res	no	no	YES	YES	YES	YES	YES	YES	YES
udi_devmgmt_req (Prepare to Suspend)	no	no	YES	no	YES	no	no	no	no
udi_devmgmt_req (Suspend or Shutdown)	no	no	no	no	no	YES	no	no	no
udi_devmgmt_req (Resume)	no	no	no	no	YES	YES	YES	no	no
udi_devmgmt_req (Parent Suspended)	no	no	YES	no	no	no	no	no	no
udi_devmgmt_ack	no	no	YES	YES	YES	YES	no	no	YES
udi_usage_ind	YES	YES	YES	YES	YES	YES	YES	YES	no
udi_usage_res	YES	YES	YES	YES	YES	YES	YES	YES	YES
udi_final_cleanup_req	YES	YES	YES	YES	YES	YES	YES	YES	YES
udi_final_cleanup_ack	no	no	no	no	no	no	no	no	YES
udi_channel_event_ind (UDI_CHANNEL_CLOSED) on parent channel	no	YES	YES	no	YES	YES	YES	no	no



22.1 Overview

This chapter defines the interface operations and associated service calls for the Internal Management Metalanguage, which may be used by drivers to communicate internally between primary and secondary regions of the same driver instance.

Each subsection defines the interface operations, associated control blocks, the rationale for the operation's existence, constraints and guidelines for the use of each operation, and error conditions that can occur.

The Internal Management Metalanguage can be used as a driver-internal metalanguage in multi-region drivers to communicate between the primary region and a secondary region, facilitating the development of multi-region drivers. Secondary regions start out with a channel to the primary region. Thus, one can think of multi-region driver instances as being initially created in a star topology, with the primary region at the center and the secondary regions each attached outward, like spokes on a wheel, via their initial channels. Of course, during this configuration process or subsequently, the driver may spawn new channels and cause them to be attached between different secondary driver regions, creating any desired driver-internal topology.

The Internal Management Metalanguage is expected to be particularly well-suited for many simple two-region designs in which there is a primary region and single secondary region such as an interrupt region.

The Internal Management Metalanguage provides the following functionality:

- the ability to do bind/unbind operations between the primary and secondary region;
- the ability to send and receive data buffers between the primary and secondary region;
- the ability to send control operations from the primary to the secondary region;
- and the ability to send event notifications from the secondary to the primary region.

22.1.1 Roles

There are two roles to the Internal Management Metalanguage: the “primary” and the “secondary.”

To keep things simple, the Internal Management Metalanguage defines a single channel between the primary and the secondary. Additional channels can be created between the primary and a given secondary using control operations with appropriate driver-internal definitions of the control information being passed.

22.2 Initialization

[TO BE SUPPLIED]

22.3 Interface Operations



23.1 Overview

This chapter defines the channel operations and associated service calls for the Generic I/O Metalanguage, which is available for use as a generic pass-through metalanguage.

Each subsection defines the channel operations, associated control blocks, the rationale for the operation's existence, constraints and guidelines for the use of each operation, and error conditions that can occur.

The Generic I/O Metalanguage can be used as a “top-side” metalanguage for drivers when a more specific metalanguage does not (yet) exist. Some of the ways this might be used are:

1. as a prototyping vehicle, until a more specific metalanguage can be constructed;
2. as a “super pass-through”, for vendor-specific applications to talk to their own drivers;
3. as a way for diagnostic applications to invoke diagnostic operations in the driver;
4. and as a top-side metalanguage for pseudo-drivers that are so specialized, and possibly even OS-specific, that they don't deserve the investment in a custom metalanguage.

The Generic I/O Metalanguage provides the following functionality:

- the ability to send and receive data buffers to/from the driver;
- the ability to send control operations to the driver;
- the ability to abort and/or timeout outstanding data and control operations;
- and the ability to send event notifications from the driver “upward.”

23.1.1 Roles

There are two roles to the Generic I/O Metalanguage: the “client” and the “provider.” The client is always a child of the provider.

To keep things simple, the initial bind channel is also used for all I/O operations. As a result, there is just one channel between the two drivers.

23.2 Generic I/O Initialization

NAME	udi_gio_provider_ops_init	<i>Register client-to-provider ops</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_channel_event_ind_op_t *channel_event_ind_op; udi_gio_bind_req_op_t *gio_bind_req_op; udi_gio_unbind_req_op_t *gio_unbind_req_op; udi_gio_xfer_req_op_t *gio_xfer_req_op; udi_gio_abort_req_op_t *gio_abort_req_op; udi_gio_event_res_op_t *gio_event_res_op; } udi_gio_provider_ops_t; void udi_gio_provider_ops_init (udi_index_t ops_idx, udi_gio_provider_ops_t *ops);</pre>	
ARGUMENTS	<p>ops_idx is a non-zero channel ops index number, assigned by the driver to uniquely identify this ops vector to subsequent service calls. See “Ops Index” on page 8-6 for additional details.</p> <p>ops is a pointer to the related vector of driver entry points. For additional details, see the definition of the ops argument in section 6.3.1 and Section 6.3.1.1, “Channel Operations Vectors,” on page 6-2.</p>	
DESCRIPTION	udi_gio_provider_ops_init is called by a driver acting as a Generic I/O “provider” (as opposed to a “client”) from the driver’s init_module routine to register its entry points for receiving generic I/O bind, transfer and abort requests, and event responses.	
WARNINGS	This call shall only be made from the init_module routine of one of the driver’s modules.	
EXAMPLE	The driver’s init_module routine might include the following:	
	<pre>#define MY_GIO_OPS 1 /* Ops for my child GIO client */ #define MY_OTHER_OPS 2 /* Some other ops */ static const udi_gio_provider_ops_t ddd_gio_provider_ops = { ddd_gio_channel_event_ind, ddd_gio_bind_req, ddd_gio_unbind_req, ddd_gio_xfer_req, ddd_gio_abort_req, ddd_gio_event_res }; ... }</pre>	

```
void init_module(void)
{
    ...
    udi_gio_provider_ops_init(
        MY_GIO_OPS,
        &ddd_gio_provider_ops);
    ...
}
```

NAME	udi_gio_client_ops_init	<i>Register provider-to-client ops</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_channel_event_ind_op_t *channel_event_ind_op; udi_gio_bind_ack_op_t *gio_bind_ack_op; udi_gio_unbind_ack_op_t *gio_unbind_ack_op; udi_gio_xfer_ack_op_t *gio_xfer_ack_op; udi_gio_xfer_nak_op_t *gio_xfer_nak_op; udi_gio_abort_ack_op_t *gio_abort_ack_op; udi_gio_event_ind_op_t *gio_event_ind_op; } udi_gio_client_ops_t; void udi_gio_client_ops_init (udi_index_t ops_idx, udi_gio_client_ops_t *ops);</pre>	
ARGUMENTS	<p>ops_idx is a non-zero channel ops index number, assigned by the driver to uniquely identify this ops vector to subsequent service calls. See “Ops Index” on page 8-6 for additional details.</p> <p>ops is a pointer to the related vector of driver entry points. For additional details, see the definition of the ops argument in section 6.3.1 and Section 6.3.1.1, “Channel Operations Vectors,” on page 6-2.</p>	
DESCRIPTION	udi_gio_client_ops_init is called by a driver acting as a Generic I/O “client” (as opposed to a “provider”) from the driver’s init_module routine to register its entry points for receiving generic I/O bind, transfer and abort acknowledgements, and event indications.	
WARNINGS	This call shall only be made from the init_module routine of one of the driver’s modules.	
EXAMPLE	<p>The driver’s init_module routine might include the following:</p> <pre>#define MY_GIO_OPS 1 /* Ops for my parent GIO provider */ #define MY_OTHER_OPS 2 /* Some other ops */ static const udi_gio_client_ops_t ddd_gio_client_ops = { ddd_gio_channel_event_ind, ddd_gio_bind_ack, ddd_gio_unbind_ack, ddd_gio_xfer_ack, ddd_gio_xfer_nak, ddd_gio_abort_ack, ddd_gio_event_ind }; ... }</pre>	

```
void init_module(void)
{
    ...
    udi_gio_client_ops_init(
        MY_GIO_OPS,
        &ddd_gio_client_ops);
    ...
}
```

NAME	udi_gio_bind_cb_init	<i>Register GIO bind control block properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_bind_cb_init (udi_index_t cb_idx, udi_size_t scratch_requirement);</pre>	
ARGUMENTS	<p>cb_idx is a control block index number, assigned by the driver to uniquely identify this set of control block properties (scratch requirements).</p> <p>scratch_requirement is the number of bytes of scratch space the driver requires to be associated with bind control blocks that will be allocated using cb_idx.</p>	
DESCRIPTION	A device driver calls <code>udi_gio_bind_cb_init</code> to register its scratch requirements for Generic I/O bind control blocks.	
WARNINGS	This call shall only be made from the <code>init_module</code> routine of one of the driver's modules.	
REFERENCES	<code>udi_gio_bind_cb_t</code>	

NAME	udi_gio_xfer_cb_init	<i>Register GIO transfer control block properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_xfer_cb_init (udi_index_t cb_idx, udi_size_t scratch_requirement, udi_ubit8_t params_size, udi_layout_t params_layout); /* maximum value for params_size */ #define UDI_GIO_MAX_PARAMS_SIZE 255</pre>	
ARGUMENTS	<p>cb_idx is a control block index number, assigned by the driver to uniquely identify this set of control block properties (scratch requirements).</p> <p>scratch_requirement is the number of bytes of scratch space the driver requires to be associated with transfer control blocks that will be allocated using cb_idx.</p> <p>params_size is the number of bytes of inline array space the driver requires for the tr_params member of transfer control blocks that will be allocated using cb_idx. params_size must be at most UDI_GIO_MAX_PARAMS_SIZE (255 bytes).</p> <p>params_typespec is a type specifier describing the layout of the tr_params inline structure.</p>	
DESCRIPTION	A device driver calls udi_gio_xfer_cb_init to register its scratch requirements for Generic I/O transfer control blocks.	
WARNINGS	This call shall only be made from the init_module routine of one of the driver's modules.	
REFERENCES	udi_gio_xfer_cb_t	

NAME	udi_gio_abort_cb_init	<i>Register GIO abort control block properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_abort_cb_init (udi_index_t cb_idx, udi_size_t scratch_requirement);</pre>	
ARGUMENTS	<p>cb_idx is a control block index number, assigned by the driver to uniquely identify this set of control block properties (scratch requirements).</p> <p>scratch_requirement is the number of bytes of scratch space the driver requires to be associated with abort control blocks that will be allocated using cb_idx.</p>	
DESCRIPTION	A device driver calls <code>udi_gio_abort_cb_init</code> to register its scratch requirements for Generic I/O abort control blocks.	
WARNINGS	This call shall only be made from the <code>init_module</code> routine of one of the driver's modules.	
REFERENCES	<code>udi_gio_abort_cb_t</code>	

NAME	udi_gio_event_cb_init	<i>Register GIO event control block properties</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_event_cb_init (udi_index_t cb_idx, udi_size_t scratch_requirement);</pre>	
ARGUMENTS	<p>cb_idx is a control block index number, assigned by the driver to uniquely identify this set of control block properties (scratch requirements).</p> <p>scratch_requirement is the number of bytes of scratch space the driver requires to be associated with event control blocks that will be allocated using cb_idx.</p>	
DESCRIPTION	A device driver calls <code>udi_gio_event_cb_init</code> to register its scratch requirements for Generic I/O event control blocks.	
WARNINGS	This call shall only be made from the <code>init_module</code> routine of one of the driver's modules.	
REFERENCES	<code>udi_gio_event_cb_t</code>	

23.3 Generic I/O Operations and Control Blocks

NAME	udi_gio_bind_cb_t	<i>Control block for GIO binding operations</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; } udi_gio_bind_cb_t;</pre>	
MEMBERS	gcb	is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.
DESCRIPTION		The Generic I/O bind control block is used between the client and provider drivers to complete initial binding over the bind channel.
REFERENCES		udi_gio_bind_cb_init , udi_cb_alloc

NAME	udi_gio_bind_req	<i>Request a binding to a GIO provider driver</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_bind_req (udi_channel_t target_channel, udi_gio_bind_cb_t *gio_bind_cb);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_bind_cb is a pointer to a GIO bind control block.</p>	
TARGET CHANNEL		The target channel for this operation is the bind channel connecting a GIO client driver with its parent provider driver.
DESCRIPTION	<p>A Generic I/O client driver uses this operation to bind to its parent Generic I/O provider driver.</p> <p>The client driver must prepare for the <code>udi_gio_bind_req</code> operation by allocating a GIO bind control block (calling <code>udi_cb_alloc</code> with a cb_idx that was previously passed to <code>udi_gio_bind_cb_init</code>).</p> <p>Next, the client driver sends the GIO bind control block to the provider driver with a <code>udi_gio_bind_req</code> operation.</p> <p>The <code>udi_gio_bind_req</code> operation must be the first channel operation sent on the bind channel. The client driver must not send any further operations on the bind channel until it receives the corresponding <code>udi_gio_bind_ack</code> from the provider driver.</p>	
REFERENCES		<code>udi_gio_bind_cb_t</code> , <code>udi_gio_bind_ack</code>

NAME	udi_gio_bind_ack	Acknowledge a GIO binding
SYNOPSIS	<pre>#include <udi.h> void udi_gio_bind_ack (udi_channel_t target_channel, udi_gio_bind_cb_t *gio_bind_cb, udi_ubit32_t device_size_lo, udi_ubit32_t device_size_hi, udi_status_t status);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_bind_cb is a pointer to a GIO bind control block.</p> <p>device_size_lo is the least-significant 32 bits of the (logical) device size, in bytes. This affects the behavior of standard read/write operations; its effect on custom operations, if any, is defined by the provider driver.</p> <p>device_size_hi is the next-most-significant 32 bits of the (logical) device size, in bytes. This affects the behavior of standard read/write operations; its effect on custom operations, if any, is defined by the provider driver.</p> <p>status indicates whether or not the binding was successful.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.	
DESCRIPTION	<p>The <code>udi_gio_bind_ack</code> operation is used by a Generic I/O provider driver to acknowledge binding with a child Generic I/O client driver (or failure to do so, as indicated by status), as requested by a <code>udi_gio_bind_req</code> operation.</p> <p>If device_size_lo or device_size_hi are non-zero, the standard read/write operations, UDI_GIO_OP_READ and UDI_GIO_OP_WRITE, are treated as random-access operations; that is, the offset_lo and offset_hi members of <code>udi_gio_rw_params_t</code> indicate the starting device offset for each transfer and transfers may be sent to the provider in any order. The client driver must not send any such requests that would extend beyond the end of the device as indicated by device_size_lo and device_size_hi.</p> <p>If device_size_lo and device_size_hi are both zero, and the client uses standard read/write operations, it must send them to the provider in device order.</p>	
STATUS VALUES	UDI_OK UDI_STAT_CANNOT_BIND	
REFERENCES	<code>udi_gio_bind_cb_t</code> , <code>udi_gio_bind_req</code>	

NAME	udi_gio_unbind_req	<i>Request to unbind from a GIO provider driver</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_unbind_req (udi_channel_t target_channel, udi_gio_bind_cb_t *gio_bind_cb);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_bind_cb is a pointer to a GIO bind control block.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO client driver with its parent provider driver.	
DESCRIPTION	<p>A Generic I/O client driver uses this operation to unbind from its parent Generic I/O provider driver.</p> <p>The client driver must prepare for the <code>udi_gio_unbind_req</code> operation by allocating a GIO bind control block (calling <code>udi_cb_alloc</code> with a cb_idx that was previously passed to <code>udi_gio_bind_cb_init</code>).</p> <p>Next, the client driver sends the GIO bind control block to the provider driver with a <code>udi_gio_unbind_req</code> operation.</p>	
REFERENCES	<code>udi_gio_bind_cb_t</code> , <code>udi_gio_unbind_ack</code>	

NAME	udi_gio_unbind_ack	<i>Acknowledge a GIO unbind request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_unbind_ack (udi_channel_t target_channel, udi_gio_bind_cb_t *gio_bind_cb);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_bind_cb is a pointer to a GIO bind control block.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.	
DESCRIPTION	The <code>udi_gio_unbind_ack</code> operation is used by a Generic I/O provider driver to acknowledge unbinding from a child Generic I/O client driver as requested by a <code>udi_gio_unbind_req</code> operation.	
REFERENCES	<code>udi_gio_bind_cb_t</code> , <code>udi_gio_unbind_req</code>	

NAME	udi_gio_xfer_cb_t	<i>Control block for GIO transfer operations</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; void *tr_context; udi_buf_t data_buf; udi_size_t data_len; udi_gio_op_t op; void *tr_params; } udi_gio_xfer_cb_t; /* direction flag values for op */ #define UDI_GIO_DIR_READ (1 << 8) #define UDI_GIO_DIR_WRITE (1 << 9)</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>tr_context is a transaction context pointer passed by the client driver to the provider in the request, and must be passed back unchanged by the provider driver in the corresponding acknowledgement.</p> <p>data_buf is a buffer handle for a buffer used to carry the data portion of a transfer. See <i>udi_gio_xfer_req</i> and <i>udi_gio_xfer_ack</i> for details on buffer usage.</p> <p>data_len is the number of data bytes to be transferred. Depending on the direction(s) of transfer, this length may apply to the data buffer in the request, the acknowledgement, or both.</p> <p>op is a code designating the specific operation to be performed. Operation codes 0 through 255 may be used to indicate different operation semantics. Custom operations are supported, as well as standard operations. See <i>udi_gio_xfer_req</i> for a description of these operations.</p> <p>The op argument also includes a bitmask of zero, one, or both direction flags from the following list:</p> <ul style="list-style-type: none"> UDI_GIO_DIR_READ - from provider to client UDI_GIO_DIR_WRITE - from client to provider <p>These indicate the direction of data flow for the data_buf buffer. They do not imply any particular operation semantics.</p> <p>tr_params is a pointer to an inline array that is used to hold operation-specific parameters. Its size is determined by the params_size argument to <i>udi_gio_xfer_cb_init</i>.</p>	

DESCRIPTION The Generic I/O transfer control block is used between the client and provider drivers to process a data or control transfer.

REFERENCES *udi_gio_op_t*, *udi_gio_xfer_cb_init*, *udi_cb_alloc*

NAME	udi_gio_op_t	<i>GIO operation type</i>
SYNOPSIS	<pre>#include <udi.h> typedef udi_ubit16_t udi_gio_op_t; /* limit values for udi_gio_op_t */ #define UDI_GIO_OP_CUSTOM 32 #define UDI_GIO_OP_MAX 255</pre>	
DESCRIPTION	<p>This type is used to hold an operation code, including direction flags, for a Generic I/O transfer control block.</p> <p>The following standard operation codes are defined:</p> <pre>#define UDI_GIO_OP_READ UDI_GIO_DIR_READ #define UDI_GIO_OP_WRITE UDI_GIO_DIR_WRITE</pre> <p>Additional, optional, standard operation codes are defined for device diagnostics in Chapter 24, “<i>Diagnostics Support</i>”. The provider driver may define additional custom operations, whose semantics and parameters are completely defined by the driver. However, the basic rules for use of data_buf, data_len, and the direction flags must be followed in all cases. Custom operations must use op values of UDI_GIO_OP_CUSTOM or greater.</p> <p>The read and write operations use a udi_gio_rw_params_t structure for tr_params in the transfer control block.</p> <p>The UDI_GIO_OP_READ operation reads data from the device at the offset indicated by offset_lo and offset_hi (if applicable). If there are fewer than data_len bytes remaining on the device at the time the request is processed, then those bytes that are present should be returned and data_len adjusted accordingly. If there are no data bytes available, and the possibility exists of more data arriving eventually, the provider driver must wait until at least one byte becomes available before responding.</p> <p>The UDI_GIO_OP_WRITE operation writes data to the device at the offset indicated by offset_lo and offset_hi (if applicable). If the device cannot hold data_len additional bytes at the time the request is processed, then those bytes that fit should be sent to the device and data_len should be adjusted accordingly. Note that if the device is just temporarily unable to accept more data (for example, due to flow control), and can reasonably be expected to be eventually able to accept more data without external action, then the provider driver must continue to process the write operation once the device is no longer busy, and must not respond early with a short count.</p> <p>Transfer constraints (see Chapter 12, “<i>Constraints Management</i>”) apply to the standard read/write operations, but not to any other standard or custom operations.</p>	
REFERENCES	udi_gio_xfer_cb_t , udi_gio_rw_params_t	

NAME	udi_gio_rw_params_t	<i>Parameters for standard GIO read/write ops</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_ubit32_t offset_lo; udi_ubit32_t offset_hi; udi_ubit16_t timeout; } udi_gio_rw_params_t;</pre>	
MEMBERS		<p>offset_lo is the least-significant 32 bits of an offset in bytes from the beginning of the (logical) device. This value is ignored if device_size_lo and device_size_hi were set to zero in the call to udi_gio_bind_ack.</p> <p>offset_hi is the next-most-significant 32 bits of an offset in bytes from the beginning of the (logical) device. This value is ignored if device_size_lo and device_size_hi were set to zero in the call to udi_gio_bind_ack.</p> <p>timeout is the time-out value for this request. If non-zero, it specifies the number of seconds after which the request should be automatically aborted if not yet complete. Timed-out requests must return the status code, UDI_STAT_TIMEOUT.</p>
DESCRIPTION		<p>This structure is used to hold additional parameters for the standard GIO read/write operations: UDI_GIO_OP_READ and UDI_GIO_OP_WRITE. It is passed to a udi_gio_xfer_req operation using the tr_params inline array of the udi_gio_xfer_cb_t.</p> <p>The tr_params pointer itself must not be changed; instead it should be cast to (udi_gio_rw_params_t *) and then the structure may be read or written through the resulting pointer.</p> <p>Any control block allocated for use with udi_gio_rw_params_t must result from a udi_gio_xfer_cb_init call with params_size set to at least sizeof(udi_gio_rw_params_t).</p>
REFERENCES		<p>udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_ack, udi_gio_xfer_cb_init</p>

NAME	udi_gio_xfer_req	<i>Request a Generic I/O transfer</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_xfer_req (udi_channel_t target_channel, udi_gio_xfer_cb_t *gio_xfer_cb);</pre>	
ARGUMENTS		target_channel See Standard Arguments . gio_xfer_cb is a pointer to a GIO transfer control block.
TARGET CHANNEL		The target channel for this operation is the bind channel connecting a GIO client driver with its parent provider driver.
DESCRIPTION		<p>A Generic I/O client driver uses this operation to send a transfer request to its parent Generic I/O provider driver.</p> <p>The client driver must prepare for the udi_gio_xfer_req operation by allocating a GIO transfer control block (calling udi_cb_alloc with a cb_idx that was previously passed to udi_gio_xfer_cb_init) and filling in all of its members.</p> <p>If op includes neither UDI_GIO_DIR_READ nor UDI_GIO_DIR_WRITE, data_len is ignored and data_buf must be set to NULL_BUF. Otherwise data_buf must be set to NULL_BUF if data_len is zero, or to a valid buffer handle if data_len is not zero.</p> <p>If data_buf is not NULL_BUF, it must contain exactly data_len bytes of valid data if op includes UDI_GIO_DIR_WRITE, or zero valid data bytes if it does not.</p> <p>Next, the client driver sends the GIO transfer control block to the provider driver with a udi_gio_xfer_req operation.</p> <p>The particular semantics and parameters for the request depend on the op value in the udi_gio_xfer_cb_t transfer control block. See udi_gio_op_t on page 23-19 for descriptions of valid operation codes.</p>
REFERENCES		udi_gio_xfer_cb_t , udi_gio_op_t , udi_gio_xfer_ack , udi_gio_xfer_cb_init

NAME	udi_gio_xfer_ack	<i>Acknowledge a GIO transfer request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_xfer_ack (udi_channel_t target_channel, udi_gio_xfer_cb_t *gio_xfer_cb);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_xfer_cb is a pointer to a GIO transfer control block.</p> <p>status indicates whether or not the transfer was successful.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.	
DESCRIPTION	<p>The <code>udi_gio_xfer_ack</code> operation is used by a Generic I/O provider driver to acknowledge a transfer request back to a child Generic I/O client driver (indicating success), as requested by a <code>udi_gio_xfer_req</code> operation. The <code>udi_gio_xfer_nak</code> operation is used to indicate failure or other exceptional conditions.</p> <p>The tr_context and op members of the control block must have the same values they had at the time of the <code>udi_gio_xfer_req</code> operation. The contents of the tr_params inline array are ignored for <code>udi_gio_xfer_ack</code>, but the array itself must be big enough for the needs of another request with the same op value, in case the client driver recycles the control block.</p> <p>If op includes either UDI_GIO_DIR_READ or UDI_GIO_DIR_WRITE, data_len must be set to the amount of data actually transferred; otherwise, data_len is ignored. Whether or not data_len must be the same as passed in to <code>udi_gio_xfer_req</code> is defined by the semantics of the particular op used.</p> <p>If op includes UDI_GIO_DIR_READ and (the original) data_buf is not NULL_BUF, data_buf must contain exactly data_len bytes of valid data, and the data_buf member must have the same value as in the request, or must be the result of a series of one or more <code>udi_buf_copy</code> or <code>udi_buf_write</code> calls with data_buf as the destination buffer.</p> <p>If op does not include UDI_GIO_DIR_READ, data_buf must be set to NULL_BUF (any original buffer must be freed).</p>	
REFERENCES	<p><code>udi_gio_xfer_cb_t</code>, <code>udi_gio_xfer_req</code>, <code>udi_gio_xfer_nak</code>, <code>udi_buf_copy</code></p>	

NAME	udi_gio_xfer_nak	<i>Abnormal completion of a GIO transfer request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_xfer_nak (udi_channel_t target_channel, udi_gio_xfer_cb_t *gio_xfer_cb, udi_status_t status);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_xfer_cb is a pointer to a GIO transfer control block.</p> <p>status indicates whether or not the transfer was successful.</p>	
TARGET CHANNEL		The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.
DESCRIPTION		<p>The <code>udi_gio_xfer_nak</code> operation is used by a Generic I/O provider driver to send a negative acknowledgement of a transfer request back to the child Generic I/O client driver (indicating failure) that requested the transfer using a <code>udi_gio_xfer_req</code> operation.</p> <p>The tr_context and op members of the control block must have the same values they had at the time of the <code>udi_gio_xfer_req</code> operation. The contents of the tr_params inline array are ignored for <code>udi_gio_xfer_nak</code>, but the array itself must be big enough for the needs of another request with the same op value, in case the client driver recycles the control block.</p> <p>If op includes either UDI_GIO_DIR_READ or UDI_GIO_DIR_WRITE, data_len must be set to the amount of data actually transferred; otherwise, data_len is ignored. Whether or not data_len must be the same as passed in to <code>udi_gio_xfer_req</code> is defined by the semantics of the particular op used.</p> <p>If op includes UDI_GIO_DIR_READ and (the original) data_buf is not NULL_BUF, data_buf must contain exactly data_len bytes of valid data, and the data_buf member must have the same value as in the request, or must be the result of a series of one or more <code>udi_buf_copy</code> or <code>udi_buf_write</code> calls with data_buf as the destination buffer.</p> <p>If op includes UDI_GIO_DIR_WRITE, data_buf and the valid data and tags therein must be the same as in the corresponding <code>udi_gio_xfer_req</code>.</p>
STATUS VALUES		<i>many</i>
REFERENCES		<code>udi_gio_xfer_cb_t</code> , <code>udi_gio_xfer_req</code> , <code>udi_gio_xfer_ack</code> , <code>udi_buf_copy</code>

NAME	udi_gio_abort_cb_t	<i>Control block for GIO abort operations</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; void *tr_context; void *orig_tr_context; } udi_gio_abort_cb_t;</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>tr_context is a transaction context pointer passed by the client driver to the provider in the request, and must be passed back unchanged by the provider driver in the corresponding acknowledgement.</p> <p>orig_tr_context is a transaction context pointer that was previously passed by the client driver to the provider. This is used to identify the pending request that is to be aborted.</p>	
DESCRIPTION	The Generic I/O abort control block is used between the client and provider drivers to abort a previous data or control transfer.	
REFERENCES	udi_gio_abort_cb_init , udi_cb_alloc	

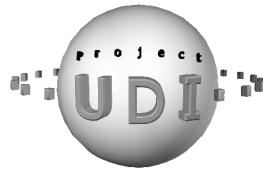
NAME	udi_gio_abort_req	<i>Abort a pending transfer request</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_abort_req (udi_channel_t target_channel, udi_gio_abort_cb_t *gio_abort_cb);</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_abort_cb is a pointer to a GIO abort control block.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO client driver with its parent provider driver.	
DESCRIPTION	<p>A Generic I/O client driver uses this operation to abort a previously sent transfer request (<i>udi_gio_xfer_req</i>).</p> <p>The client driver must prepare for the <i>udi_gio_abort_req</i> operation by allocating a GIO abort control block (calling <i>udi_cb_alloc</i> with a cb_idx that was previously passed to <i>udi_gio_abort_cb_init</i>) and setting its orig_tr_context to the tr_context value of the original request.</p> <p>Next, the client driver sends the GIO abort control block to the provider driver with a <i>udi_gio_abort_req</i> operation.</p>	
REFERENCES	<i>udi_gio_abort_cb_t</i> , <i>udi_gio_abort_ack</i> , <i>udi_gio_xfer_req</i>	

NAME	udi_gio_abort_ack	<i>Acknowledge a GIO abort operation</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_abort_ack (udi_channel_t target_channel, udi_gio_abort_cb_t *gio_abort_cb); </pre>	
	<pre>udi_gio_abort_ack_op_t udi_gio_abort_ack_unused;</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_abort_cb is a pointer to a GIO abort control block.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.	
DESCRIPTION	<p>The <code>udi_gio_abort_ack</code> operation is used by a Generic I/O provider driver to acknowledge an abort request from a child Generic I/O client driver, as requested by a <code>udi_gio_abort_req</code> operation. The original request, if still pending at the time the abort was processed, will be returned (via <code>udi_gio_xfer_ack</code>) with a status of UDI_STAT_ABORTED.</p> <p><code>udi_gio_abort_ack_unused</code> may be used as a client driver's <code>udi_gio_abort_ack</code> entry point if the driver never makes any abort requests (and therefore expects to receive no acknowledgements).</p>	
REFERENCES	<code>udi_gio_abort_cb_t</code> , <code>udi_gio_abort_req</code> , <code>udi_gio_xfer_ack</code>	

NAME	udi_gio_event_cb_t	<i>Control block for GIO event operations</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_cb_t gcb; udi_ubit8_t event_code; } udi_gio_event_cb_t;</pre>	
MEMBERS	<p>gcb is a generic control block header, which includes a pointer to the scratch space associated with this control block. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.</p> <p>event_code is a driver-specific code that indicates the type of event which occurred. If additional information is required, the client must request it from the provider by using a control operation.</p>	
DESCRIPTION	The Generic I/O event control block is used between the client and provider drivers to notify the client of an asynchronous event.	
REFERENCES	udi_gio_event_cb_init, udi_cb_alloc	

NAME	udi_gio_event_ind	<i>GIO event indication</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_event_ind (udi_channel_t target_channel, udi_gio_event_cb_t *gio_event_cb) ; udi_gio_event_ind_op_t udi_gio_event_ind_unused;</pre>	
ARGUMENTS	<p>target_channel See Standard Arguments.</p> <p>gio_event_cb is a pointer to a GIO event control block.</p>	
TARGET CHANNEL	The target channel for this operation is the bind channel connecting a GIO provider driver with its child GIO client driver.	
DESCRIPTION	<p>A Generic I/O provider driver uses this operation to send an event notification to its child Generic I/O client driver.</p> <p>The provider driver must prepare for the <code>udi_gio_event_ind</code> operation by allocating a GIO event control block (calling <code>udi_cb_alloc</code> with a cb_idx that was previously passed to <code>udi_gio_event_cb_init</code>).</p> <p>Next, the provider driver sends the GIO event control block to the client driver with a <code>udi_gio_event_ind</code> operation. The provider driver does not need to wait to receive a response before sending another <code>udi_gio_event_ind</code>; multiple indications may be pending at once.</p> <p>Whether or not a provider driver supports event notification, and whether or not the client driver must enable events explicitly (via custom operations), is defined by the provider driver. There are no standard events.</p> <p><code>udi_gio_event_ind_unused</code> may be used as a client driver's <code>udi_gio_event_ind</code> entry point if the driver expects that the provider driver will never send it any event indications.</p>	
REFERENCES	<code>udi_gio_event_cb_t</code> , <code>udi_gio_event_res</code>	

NAME	udi_gio_event_res	<i>GIO event response</i>
SYNOPSIS	<pre>#include <udi.h> void udi_gio_event_res (udi_channel_t target_channel, udi_gio_event_cb_t *gio_event_cb);</pre> <p>udi_gio_event_res_op_t udi_gio_event_res_unused;</p>	
ARGUMENTS		target_channel See Standard Arguments . gio_event_cb is a pointer to a GIO event control block.
TARGET CHANNEL		The target channel for this operation is the bind channel connecting a GIO client driver with its parent provider driver.
DESCRIPTION		The <code>udi_gio_event_res</code> operation is used by a Generic I/O client driver to acknowledge an event indication from its parent Generic I/O provider driver, as delivered by a <code>udi_gio_event_ind</code> operation. <code>udi_gio_event_res_unused</code> may be used as a provider driver's <code>udi_gio_event_res</code> entry point if the driver never sends any event indications (and therefore expects no responses).
REFERENCES		<code>udi_gio_event_cb_t</code> , <code>udi_gio_event_ind</code>



It is recommended, but not required, that UDI drivers support some level of diagnostics capability. The following recommendations provide a framework for executing diagnostics tests and reporting the results, but the semantics and descriptions of the tests are necessarily specific to the driver and adapter being tested. In usage, these tests will probably be executed from an application that will assist in directing the user to configure the hardware, run the tests, and interpret the results.

24.1 Diagnostics State

Since diagnostics tests may be destructive to the state of the device, and normal device operation may cause a diagnostic to report an erroneous failure on a functional device, the diagnostic test sessions are bracketed by `UDI_GIO_OP_DIAG_ENABLE` and `UDI_GIO_OP_DIAG_DISABLE` requests. The test session must start by the driver receiving a `UDI_GIO_OP_DIAG_ENABLE` request. If the driver is in a state such that either running diagnostics tests might cause the device to lose state or data, or continued normal operation of the device might cause a spurious failure of a diagnostic test, the driver should reject the request with a status of `UDI_STAT_BUSY` until such time as it is in a state to run the diagnostics. For example, a network device driver may reject an attempt to run diagnostics while any network interface is enabled. Other devices may need to be taken completely offline or be unbound from child devices. In most cases, external action beyond the control of the driver needs to be taken before diagnostics can be run. The driver is only responsible for answering the question “is it safe to run diagnostics?”.

If the device does not support any diagnostics capability, it must return a status of `UDI_STAT_NOT_SUPPORTED` to the `UDI_GIO_OP_DIAG_ENABLE` request.

When a driver receives the `UDI_GIO_OP_DIAG_ENABLE` request and it is in a state to run its defined set of diagnostics tests safely, it will acknowledge the request with a status of `UDI_OK` and set its internal state to prevent the initiation of any activities that might not complete successfully due to the execution of diagnostics tests or that might interfere with the results of the diagnostics tests. For example, a network driver currently running diagnostics might refuse to allow any network interfaces to be enabled until the tests are concluded. Other devices may acknowledge a `udi_prep_for_child_req()` with a status of `UDI_STAT_BUSY`. Once the driver has agreed to allow the diagnostics session to begin, it should not allow normal activities to resume until it has received the `UDI_GIO_OP_DIAG_DISABLE` request.

If a driver receives an `UDI_GIO_OP_DIAG_RUN_TEST` request without first having responded to a `UDI_GIO_OP_DIAG_ENABLE` request with `UDI_OK`, the driver is not in the proper state to run diagnostics and must respond with a status of `UDI_STAT_INVALID_STATE`.

Diagnostics State

Diagnostics Support

If a driver receives another UDI_GIO_OP_DIAG_ENABLE request after having responded to a UDI_GIO_OP_DIAG_ENABLE request with UDI_OK but without an intervening UDI_GIO_OP_DIAG_DISABLE, the driver is not in the proper state to enter diagnostics mode and must respond to the UDI_GIO_OP_DIAG_ENABLE with a status of UDI_STAT_INVALID_STATE.

When the driver receives a UDI_GIO_OP_DIAG_DISABLE request, it must clear the internal state set by UDI_GIO_OP_DIAG_ENABLE, terminate any tests running with a status of UDI_STAT_ABORTED, and prepare the device to resume normal operations. The driver must always return a status of UDI_OK to a UDI_GIO_OP_DIAG_DISABLE request, regardless of driver state.

NAME	udi_gio_op_t	<i>Diagnostics control operations</i>
SYNOPSIS	<pre>#include <udi.h> typedef udi_ubit16_t udi_gio_op_t; /* values for udi_gio_op_t */ #define UDI_GIO_OP_DIAG_ENABLE 1 #define UDI_GIO_OP_DIAG_DISABLE 2 #define UDI_GIO_OP_DIAG_RUN_TEST \ (3 + UDI_GIO_DIR_READ)</pre>	
DESCRIPTION	<p>The following optional Generic I/O standard operations are defined to support diagnostics operations on drivers. These supplement the operations defined for <i>udi_gio_op_t</i> on page 23-19.</p> <p>UDI_GIO_OP_DIAG_ENABLE is used to enable diagnostics mode for a particular device. If the device is in a state where it can safely run diagnostics, the driver shall return a status of UDI_OK and set its state to reject any attempts at normal device usage with a status of UDI_STAT_BUSY until the driver receives a UDI_GIO_OP_DIAG_DISABLE request. If the device is not in a state where it is safe to run diagnostics, the driver must respond to the UDI_GIO_OP_DIAG_ENABLE request with a status of UDI_STAT_BUSY. If the device does not support any diagnostics capability, it must return a status of UDI_STAT_NOT_SUPPORTED. If the device is currently in its internal diagnostics mode, it must reject any subsequent UDI_GIO_OP_DIAG_ENABLE requests with a status of UDI_STAT_INVALID_STATE.</p> <p>No data payload is used with this operation, so data_len must be zero and data_buf must be NULL_BUF.</p> <p>UDI_GIO_OP_DIAG_DISABLE clears the driver internal state set by a previous UDI_GIO_OP_DIAG_ENABLE operation and terminates any diagnostics tests that may be running. The only status returned is UDI_OK. At the conclusion of this operation the device is assumed to be ready for normal usage.</p> <p>No data payload is used with this operation, so data_len must be zero and data_buf must be NULL_BUF.</p> <p>UDI_GIO_OP_DIAG_RUN_TEST causes the driver to execute a selected diagnostics test. Drivers that support diagnostics must support at least one test. Test numbers start from zero. By convention, the lower-numbered tests are usually device self-tests which require no intervention, while the higher-numbered tests are more complicated tests which may require operator intervention to prepare for or recover from the test. Test number zero must be a self-test.</p>	

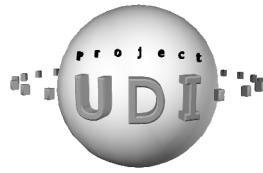
	<p>In the <code>udi_gio_xfer_ack</code> returned in response to the <code>UDI_GIO_OP_DIAG_RUN_TEST</code> operation, the status may be <code>UDI_OK</code> if the test passed, <code>UDI_STAT_HW_PROBLEM</code> if the test failed due to a hardware problem, <code>UDI_STAT_NOT_SUPPORTED</code> if the specified test is not supported on this device, <code>UDI_STAT_ABORTED</code> if the test was aborted, or <code>UDI_STAT_INVALID_STATE</code> if the driver is not in the proper state for running diagnostics.</p> <p>In the case of <code>UDI_STAT_HW_PROBLEM</code>, the buffer pointed to by <code>data_buf</code> is filled in with a message string containing additional information to help isolate the failure to a specific field replaceable unit. The <code>data_len</code> field must be set to the length of that string (without any null terminator). If the original value of <code>data_len</code> is not large enough to hold the string, then the first <code>data_len</code> bytes of the message are stored in the buffer and the rest of the message is lost; the driver must not increase <code>data_len</code>.</p>
STATUS VALUES	<p><code>UDI_OK</code> – The operation completed successfully.</p> <p><code>UDI_STAT_BUSY</code> – The device or driver is not in a safe state for diagnostics.</p> <p><code>UDI_STAT_NOT_SUPPORTED</code> – The specified test number, or diagnostics in general, are not supported by this driver.</p> <p><code>UDI_STAT_INVALID_STATE</code> – The driver is not in diagnostics mode when requests to run tests or disable diagnostics, or is already in diagnostics mode when requested to enable diagnostics.</p> <p><code>UDI_STAT_HW_PROBLEM</code> – The requested diagnostics test failed.</p> <p><code>UDI_STAT_ABORTED</code> – A test in progress was aborted.</p>

NAME	udi_gio_diag_params_t	<i>Parameters for standard GIO diagnostic ops</i>
SYNOPSIS	<pre>#include <udi.h> typedef struct { udi_ubit8_t test_num; udi_ubit8_t test_params_size; } udi_gio_diag_params_t;</pre>	
MEMBERS	<p>test_num is the number of the diagnostic test to run. This value is ignored if the op member of the gio_xfer_cb is not set to UDI_GIO_OP_DIAG_RUN_TEST.</p> <p>test_params_size is the number of bytes of additional parameters, if any, for the test specified by test_num. This number may be zero, and must not be greater than two less than the params_size specified in udi_gio_xfer_cb_init. The semantics and structure of these additional parameters are defined by each driver; the corresponding params_layout must include the structure of the additional parameters. This value is ignored if the op member of the gio_xfer_cb is not set to UDI_GIO_OP_DIAG_RUN_TEST.</p>	
DESCRIPTION	<p>This structure is used to hold additional parameters for the GIO device diagnostics operation UDI_GIO_OP_DIAG_RUN_TEST. It is passed to a udi_gio_xfer_req operation using the tr_params inline array of the udi_gio_xfer_cb_t.</p> <p>The tr_params pointer itself must not be changed; instead it should be cast to (udi_gio_diag_params_t *) and then the structure may be read or written through the resulting pointer.</p> <p>Any control block allocated for use with udi_gio_diag_params_t must result from a udi_gio_xfer_cb_init call with params_size set to at least sizeof(udi_gio_diag_params_t).</p>	
REFERENCES	<p>udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_ack, udi_gio_xfer_cb_init</p>	



UDI Core Specification

Section 5: MEI Services



25.1 Overview

This section defines the Metalanguage-to-Environment Interfaces (MEI) available to implementors of UDI metalanguage libraries. The use of these interfaces (as opposed to using system-specific interfaces) is necessary to create portable metalanguage libraries, to allow for the dynamic loading and unloading of metalanguage libraries (initialization interfaces), and to allow for multi-domain I/O environments distributed across heterogeneous nodes.

Metalanguage stubs are the pieces of code that implement metalanguage channel operations. In the UDI execution model channel operations require a front end stub and a back end stub. The caller of the channel operation calls directly into the front end stub. If the target of the channel operation (at the other end of the channel) cannot be run immediately then the operation is queued, and later when the operation can be scheduled to run it is taken from the region queue and passed to the back end stub, which unmarshalls parameters and calls the target driver's entry point.

25.2 Metalanguage Design Rules

TO BE ADDED.

25.3 Data Model

Transferability, object ownership transfer, layout specifiers, etc.

25.4 Execution Model

Direct vs. indirect calls, intra-domain vs. inter-domain calls, etc.



26.1 Overview

The Metalanguage-to-Environment Interface (MEI) is a set of interfaces designed to allow for the creation of portable metalanguage libraries.

Comment – This chapter needs a lot of work in formatting and descriptive text, but the technical details are up to date and complete.

26.2 Structures

The following structures are defined for use with the MEI routines.

```

typedef void (*udi_mei_meta_id_t)(void);

typedef void udi_mei_op_t(void);

typedef udi_mei_op_t **udi_mei_ops_t;

typedef void udi_mei_direct_stub_t (
    udi_mei_op_t *op,
    udi_cb_t *gcb,
    va_list arglist );

typedef void udi_mei_backend_stub_t (
    udi_mei_op_t *op,
    udi_cb_t *gcb,
    void *marshal_space );

typedef struct {
    const char *optr_name;
    udi_ubit8_t optr_category;
} udi_mei_op_trace_t;

/* values for optr_category */
#define UDI_MEI_OPCAT_REQ 1
#define UDI_MEI_OPCAT_ACK 2
#define UDI_MEI_OPCAT_NAK 3
#define UDI_MEI_OPCAT_IND 4
#define UDI_MEI_OPCAT_RES 5

typedef struct {
    udi_mei_direct_stub_t *direct_stub;
    udi_mei_backend_stub_t *backend_stub;
    udi_index_t cb_group;
    udi_layout_t cb_layout;
    udi_layout_t marshal_layout;
    udi_mei_op_trace_t trace_info;
} udi_mei_op_template_t;

/* maximum sizes for control block layouts */
#define UDI_MEI_MAX_VISIBLE_SIZE 2000
#define UDI_MEI_MAX_MARSHAL_SIZE 4000

extern udi_mei_op_template_t
    udi_mei_channel_event_ind_template;

typedef struct {
    udi_ubit8_t relationship;
    udi_ubit8_t n_ops;
    const udi_mei_op_template_t op_templates[];
} udi_mei_ops_template_t;

```

```
/* values for relationship */
#define UDI_MEI_RELATION_SYMMETRIC      0
#define UDI_MEI_RELATION_PARENT          1
#define UDI_MEI_RELATION_PARENT_MULTI    2
#define UDI_MEI_RELATION_CHILD           3
#define UDI_MEI_RELATION_SIDE_A          4
#define UDI_MEI_RELATION_SIDE_B          5
```

relationship defines the relationship of a region using a channel to the region on the other end of that channel. Both ends of a channel must either be both SYMMETRIC or paired appropriately: PARENT or PARENT_MULTI with CHILD; SIDE_A with SIDE_B. If PARENT/CHILD, the driver instance that specifies PARENT or PARENT_MULTI must be the parent of the driver instance that specifies CHILD. If and only if **relationship** for a driver's child bind channel is UDI_MEI_RELATION_PARENT_MULTI, then this driver is prepared to have multiple child instances bound to it for each enumerated child context.

The visible size of any control block, as indicated by **visible_layout**, including the udi_cb_t header, must not exceed UDI_MEI_MAX_VISIBLE_SIZE (2000 bytes).

The size, in bytes, needed to marshal call-dependent parameters for any operation, as indicated by **marshal_layout**, must not exceed UDI_MEI_MAX_MARSHAL_SIZE (4000 bytes).

26.3 Marshalling

In order for channel operations to be queued or transferred between domains, call-dependent parameters must be marshalled into marshalling space associated with the control block. Since the layout of these parameters is known only to the metalanguage, the metalanguage library stubs are responsible for marshalling and unmarshalling these parameters.

However, the content of the marshalling space must be laid out in a well-defined order, in case the marshalled control block is passed to another domain, and the metalanguage stubs on the other end are implemented by a different instance of the metalanguage library. Both ends need to agree on the layout. Therefore, this specification standardizes that layout.

Thus, each additional parameter after the control block pointer, for a given channel operation, in left-to-right order shall be marshalled into the marshalling space starting at offset zero and proceeding with successive offsets.

26.4 Initialization Interfaces

The following interfaces are called from a metalanguage library as a result of calls from a driver's `init_module` routine to initialize properties of channel operations and control blocks used in the metalanguage.

```
void udi_mei_ops_init (
    udi_mei_meta_id_t meta_ID,
    udi_index_t ops_num,
    udi_index_t ops_idx,           /* from driver */
    udi_mei_ops_t ops,           /* from driver */
    udi_index_t n_ops_templates,
    const udi_mei_ops_template_t ops_templates[] );
```

meta_ID is a unique internal identifier for this metalanguage. Every occurrence of **meta_ID** in a metalanguage library must use the same value, which must be a pointer to a function within the library.

ops_num is a number that identifies this ops vector type with respect to others in this metalanguage. It is also used as an index into the **ops_templates** array to extract the template information for this type of ops vector.

```
void udi_mei_cb_init (
    udi_mei_meta_id_t meta_ID,
    udi_index_t cb_group,
    udi_index_t cb_idx,           /* from driver */
    udi_size_t scratch_requirement, /* from driver */
    udi_size_t *inline_sizes,
    udi_layout_t *inline_layouts,
    udi_ubit8_t n_inline );
```

cb_group is a number that identifies this control block group with respect to others in this metalanguage. It must be greater than zero.

inline_sizes is NULL or a pointer to driver-specified values for the size of each inline piece of this control block type. The location of inline pieces will be determined by finding the appropriate op_template registered by `udi_mei_ops_init`. If **inline_sizes** is non-NUL, there must be exactly one op_template for this **meta_ID** and **cb_group**. If **inline_sizes** is NULL, every op_template for this **meta_ID** and **cb_group** must not have any inline pieces defined.

inline_layouts is NULL or a pointer to driver-specified layout descriptors for each variable-type inline piece of this control block type (i.e. described in the control block's layout with UDI_DL_INLINE_VARIABLE). The location of such inline pieces will be determined by finding the appropriate op_template registered by `udi_mei_ops_init`. If **inline_layouts** is non-NUL, there must be exactly one op_template for this **meta_ID** and **cb_group**. If **inline_layouts** is NULL, every op_template for this **meta_ID** and **cb_group** must not have any variable-type inline pieces defined.

n_inline is the number of inline pieces in this control block type.

The environment can compute the maximum visible and marshal sizes for a control block group by aggregating across all occurrences of the **cb_group** in the **ops_templates**.

26.5 Front-End Stub Interfaces

The following function is called from within a portable front-end stub to implement a channel operation. `udi_mei_call` prepares the control block for transfer to the target region, possibly reallocating the space for the control block and/or its scratch space. It also arranges for the corresponding entry point to be called in the target region, either “directly” (w/o queuing) or as a queued or cross-domain indirect operation.

```
void udi_mei_call (
    udi_channel_t tgt_channel,
    udi_cb_t *gcb,
    void *meta_ID,
    udi_index_t ops_num,
    udi_index_t op_idx,
    . . . );
```

If the environment chooses to (and is able to) make a direct call, `udi_mei_call` will make use of the corresponding direct-call stub in the metalanguage library to make the actual call to the target region with the appropriate parameters. This is the highest performance path and is thus specially optimized. The direct-call stub (of type `udi_mei_direct_stub_t`) simply takes the arguments pointed to by the var-args `arglist`, and calls the indicated function with these arguments.

There are many reasons why the environment might not use a direct call. Some of these include excess call depth, busy regions and domain crossings. All other things being equal, ownership transfer for transferable objects can be handled with the direct case.

If `udi_mei_call` does not make a direct call, it must first marshal any call-dependent parameters into the marshalling space of the control block. It can determine the number and type of these parameters from the `marshal_layout` in the ops template for this operation. The ops template can be located by a combination of either `tgt_channel` and `op_idx` or `meta_ID` and `ops_num`. (Providing both of these sets of values to `udi_mei_call` allows for a double-check that the correct types of channel and control block were passed to the channel operation.)

After the control block is queued, copied across domains, or subject to any further processing needed by the environment, it will eventually need to be passed to the target region with the appropriate call-dependent parameters. This is done by calling the appropriate back-end stub (of type `udi_mei_backend_stub_t`) in the metalanguage library. The back-end stub unmarsals the parameters from the marshalling space (pointed to by `marshal_space`) and calls the driver entry point with these parameters.

The direct-call and back-end stubs are very similar and differ only in how they fetch parameters via their third arguments.

The following sample pseudo-code illustrates the implementation of a portable front-end stub in a metalanguage library:

```

void
<<meta>>_<<op>> (
    udi_channel_t chan,
    <<meta>>_<cbtype>>_cb_t *cb,
    ...<<call-dependent parms>>... )
{
    udi_mei_call(chan, UDI_GCB(cb), XXX_META_ID, \
                 ZZZ_OPS_NUM, ZZZ_OP_IDX, \
                 ...<<call-dependent parms>>...);
}

```

Similarly, a direct-call stub would look like:

```

static void
<<meta>>_<<op>>_direct (
    udi_mei_op_t *op,
    udi_cb_t *gcb,
    va_list arglist )
{
    argl_type arg1 = va_arg(arglist, argl_type);
    ...
    (*(<<meta>>_<<op>>_op_t)op)
        (UDI_MCB(gcb, <<meta>>_<cbtype>>_cb_t),
         arg1... );
}

```

A back-end stub would look like:

```

static void
<<meta>>_<<op>>_backend (
    udi_mei_op_t *op,
    udi_cb_t *gcb,
    void *marshal_space )
{
    struct <<meta>>_<<op>>_marshal {
        argl_type arg1;
        ...
    } *mp = marshal_space;
    (*(<<meta>>_<<op>>_op_t)op)
        (UDI_MCB(gcb, <<meta>>_<cbtype>>_cb_t),
         mp->arg1... );
}

```

26.6 MEI Stub Implementation Macros

The following convenience macro may be used by metalanguage libraries to implement their stub functions. Each invocation of UDI_MEI_STUBS generates front-end, direct-call, and back-end stubs for a channel operation.

```
#define UDI_MEI_STUBS(op_name, cb_type, \
                     argc, args, arg_types, \
                     meta_ID, ops_num, op_idx) \
    void op_name ( udi_channel_t chan, cb_type *cb \
                  _UDI_ARG_LIST_##argc args ) { \
        udi_mei_call ( chan, UDI_GCB(cb), \
                        meta_ID, ops_num, op_idx \
                        _UDI_ARG_LIST_##argc args ); \
    } \
    static void op_name##_direct ( \
        udi_mei_op_t *op, udi_cb_t *gcb, \
        va_list arglist ) { \
            _UDI_VA_ARGS_##argc arg_types \
            (*(op_name##_op_t)op) ( \
                UDI_MCB(gcb, cb_type) \
                _UDI_ARG_VARS_##argc ); \
        } \
    static void op_name##_backend ( \
        udi_mei_op_t *op, udi_cb_t *gcb, \
        void *marshal_space ) { \
            struct op_name##_marshal { \
                _UDI_ARG_MEMBERS_##argc arg_types \
            } *mp = marshal_space; \
            (*(op_name##_op_t)op) ( \
                UDI_MCB(gcb, cb_type) \
                _UDI_MP_ARGS_##argc ); \
        } }
```

The following helper macros are used by UDI_MEI_STUBS but must not be invoked directly. Actual implementations of UDI_MEI_STUBS may vary, but must generate equivalent code.

```

#define UDI_ARG_LIST_0()          () ,0
#define UDI_ARG_LIST_1(a)         (a),1
#define UDI_ARG_LIST_2(a,b)       (a,b),2
#define UDI_ARG_LIST_3(a,b,c)     (a,b,c),3
#define UDI_ARG_LIST_4(a,b,c,d)   (a,b,c,d),4
#define UDI_ARG_LIST_5(a,b,c,d,e) (a,b,c,d,e),5
#define UDI_ARG_LIST_6(a,b,c,d,e,f) (a,b,c,d,e,f),6
#define UDI_ARG_LIST_7(a,b,c,d,e,f,g) \
    (a,b,c,d,e,f,g),7

#define _UDI_ARG_LIST_0()
#define _UDI_ARG_LIST_1(a)         ,a
#define _UDI_ARG_LIST_2(a,b)       ,a,b
#define _UDI_ARG_LIST_3(a,b,c)     ,a,b,c
#define _UDI_ARG_LIST_4(a,b,c,d)   ,a,b,c,d
#define _UDI_ARG_LIST_5(a,b,c,d,e) ,a,b,c,d,e
#define _UDI_ARG_LIST_6(a,b,c,d,e,f) ,a,b,c,d,e,f
#define _UDI_ARG_LIST_7(a,b,c,d,e,f,g) ,a,b,c,d,e,f,g

#define _UDI_VA_ARGS_0()
#define _UDI_VA_ARGS_1(a) \
    a arg1 = va_arg(arglist, a);
#define _UDI_VA_ARGS_2(a,b) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b);
#define _UDI_VA_ARGS_3(a,b,c) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b); \
    c arg3 = va_arg(arglist, c);
#define _UDI_VA_ARGS_4(a,b,c,d) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b); \
    c arg3 = va_arg(arglist, c); \
    d arg4 = va_arg(arglist, d);
#define _UDI_VA_ARGS_5(a,b,c,d,e) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b); \
    c arg3 = va_arg(arglist, c); \
    d arg4 = va_arg(arglist, d); \
    e arg5 = va_arg(arglist, e);
#define _UDI_VA_ARGS_6(a,b,c,d,e,f) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b); \
    c arg3 = va_arg(arglist, c); \
    d arg4 = va_arg(arglist, d); \
    e arg5 = va_arg(arglist, e); \
    f arg6 = va_arg(arglist, f);

```

```
#define _UDI_VA_ARGS_7(a,b,c,d,e,f,g) \
    a arg1 = va_arg(arglist, a); \
    b arg2 = va_arg(arglist, b); \
    c arg3 = va_arg(arglist, c); \
    d arg4 = va_arg(arglist, d); \
    e arg5 = va_arg(arglist, e); \
    f arg6 = va_arg(arglist, f); \
    g arg7 = va_arg(arglist, g);

#define _UDI_ARG_VARS_0
#define _UDI_ARG_VARS_1 ,arg1
#define _UDI_ARG_VARS_2 ,arg1,arg2
#define _UDI_ARG_VARS_3 ,arg1,arg2,arg3
#define _UDI_ARG_VARS_4 ,arg1,arg2,arg3,arg4
#define _UDI_ARG_VARS_5 ,arg1,arg2,arg3,arg4,arg5
#define _UDI_ARG_VARS_6 ,arg1,arg2,arg3,arg4,arg5, \
                        arg6
#define _UDI_ARG_VARS_7 ,arg1,arg2,arg3,arg4,arg5, \
                        arg6,arg7

#define _UDI_ARG_MEMBERS_0() \
    char dummy;
#define _UDI_ARG_MEMBERS_1(a) \
    a arg1;
#define _UDI_ARG_MEMBERS_2(a,b) \
    a arg1; \
    b arg2;
#define _UDI_ARG_MEMBERS_3(a,b,c) \
    a arg1; \
    b arg2; \
    c arg3;
#define _UDI_ARG_MEMBERS_4(a,b,c,d) \
    a arg1; \
    b arg2; \
    c arg3; \
    d arg4;
#define _UDI_ARG_MEMBERS_5(a,b,c,d,e) \
    a arg1; \
    b arg2; \
    c arg3; \
    d arg4; \
    e arg5;
#define _UDI_ARG_MEMBERS_6(a,b,c,d,e,f) \
    a arg1; \
    b arg2; \
    c arg3; \
    d arg4; \
    e arg5; \
    f arg6;
```

```
#define _UDI_ARG_MEMBERS_7(a,b,c,d,e,f,g) \
    a arg1; \
    b arg2; \
    c arg3; \
    d arg4; \
    e arg5; \
    f arg6; \
    g arg7;

#define _UDI_MP_ARGS_0
#define _UDI_MP_ARGS_1 ,mp->arg1
#define _UDI_MP_ARGS_2 ,mp->arg1,mp->arg2
#define _UDI_MP_ARGS_3 ,mp->arg1,mp->arg2,mp->arg3
#define _UDI_MP_ARGS_4 ,mp->arg1,mp->arg2, \
    mp->arg3,mp->arg4
#define _UDI_MP_ARGS_5 ,mp->arg1,mp->arg2, \
    mp->arg3,mp->arg4,mp->arg5
#define _UDI_MP_ARGS_6 ,mp->arg1,mp->arg2, \
    mp->arg3,mp->arg4,mp->arg5,mp->arg6
#define _UDI_MP_ARGS_7 ,mp->arg1,mp->arg2, \
    mp->arg3,mp->arg4,mp->arg5, \
    mp->arg6,mp->arg7
```

Examples:

The udi_bind_to_parent_ack channel operation has two extra parameters and could be implemented using the front-end stub macro as follows:

```
UDI_MEI_STUBS(udi_bind_to_parent_ack, udi_bind_cb_t,
                2, (parent_context, status),
                (void *, udi_status_t),
                UDI_MGMT_META_ID, UDI_MGMT_MA_OPS_NUM,
                UDI_BIND_TO_PARENT_REQ)
```

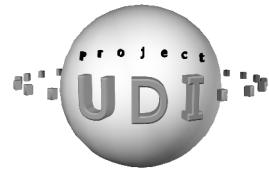
The udi_scsi_io_req channel operation has no extra parameters and could be implemented using the front-end stub macro as follows:

```
UDI_MEI_STUBS(udi_scsi_io_req, udi_scsi_io_cb_t,
                0, (), (),
                UDI_SCSI_META_ID, UDI_SCSI_HD_OPS_NUM,
                UDI_SCSI_IO_REQ)
```



UDI Core Specification

Section 6: Packaging and Distribution



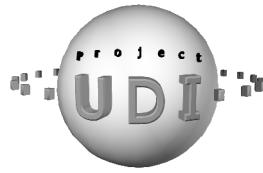
27.1 Introduction

This section specifies UDI packaging and distribution format requirements, as well as all external files and utilities used in conjunction with driver source or object code.

Chapter 28 defines the Static Driver Properties file that is used to provide global driver attributes.

Chapter 29 defines the packaging and distribution formats for UDI drivers.

Chapter 30 describes build and packaging utilities provided by UDI build environments.



28.1 Overview

This chapter defines *static driver properties* and how they are included with UDI drivers as an addition to the driver code itself.

Static driver properties are attributes of a driver or library that are known and fixed in advance of compiling its source code. These are generally properties that the environment into which a driver is being installed might need to know about prior to linking, loading and/or running the driver. For this reason, static driver properties are stored with the driver in such a way that they can be easily extracted without running the driver. The same is true for UDI libraries.

28.1.1 UDI Modules

UDI drivers and libraries are compiled and linked into binary object files called *modules*. A module is the basic unit of loadability. That is, each module can potentially be loaded at separate times or into separate domains, but all code and data in one module is loaded together. In this context, loading refers to any process through which an instance of the module code and data is made available for execution; this might involve dynamic loading into an already-running system or it might simply mean linking into a static image for use at a subsequent system reboot. Modules must not reference symbols in other modules (even within the same driver) or in the surrounding system except as included in explicitly exported/imported interfaces (see the “Requires Declaration” on page 28-6 and the “Provides Declaration” on page 28-9).

Three types of binary modules are supported:

1. primary driver module
2. secondary driver module
3. library module (including metalanguage libraries)

Each driver module contains the code to handle one or more region types that a driver supports (see Section 28.6.6, “Region Declaration,” on page 28-14 for more details on region types). No two modules for a driver may handle the same region type. The driver module that handles the driver’s primary region (region index zero) is called the primary driver module; all other driver modules, if any, are called secondary driver modules. Each driver module has its own `init_module` entry point routine (see `init_module` on page 9-4).

UDI libraries each consist of exactly one library module. Library modules do not have `init_module` routines and will not have any region data associated with them. UDI libraries provide functions that can be called by UDI drivers, but maintain no state of their own.

Each UDI driver or library shall include as part of its source code a static properties file, named “`udiprops.txt`”. At compile/build time, a special tool called “`udimkpkg`” attaches the property values from `udiprops.txt` to the binary file for the driver’s primary module or the library’s sole module, in a fashion appropriate to the particular binary file format used.

28.2 Basic Syntax

The following rules describe the basic structure of a static properties file.

- The file must consist entirely of a sequence of valid ISO 10646 (Unicode) characters encoded according to the Annex P (UTF-8) encoding scheme. The 7-bit ASCII character set, encoded in 8-bit bytes, is a subset of this encoding.
- Any sequence of zero or more CR characters (0x0D) followed by a single LF character (0x0A) is considered to be a “*line terminator*”.
- The file consists of multiple lines, each—except possibly the last line—ending in a line terminator. If the last line has no terminator, it is treated as if it did have a line terminator. In all cases, the line terminator is not counted as part of the line's contents.
- Each line, including the line terminator character(s), must be less than 512 bytes long.
- Any sequence of one or more consecutive SPACE characters (0x20) and/or HT characters (0x09) is considered “*whitespace*”.
- The DEL character (0x7F) and control characters (0x00 - 0x1F) besides HT, LF and CR are illegal.
- The “hash” character ('#') precedes comments. Any '#', and any subsequent characters up to the next line terminator are considered comments and will be completely ignored.
- Any whitespace at the end of a line (i.e. immediately preceding a comment or line terminator) is considered a comment and will be completely ignored.
- If the last non-comment character on a line is a backslash ('\') and is not immediately preceded by another backslash character, then the backslash and the line terminator are considered whitespace, and this line and the following line are treated as a single logical line. The total length of a logical line, including all backslashes and line terminators, must be less than 512 bytes long.
- Logical lines containing no non-comment characters are considered blank lines. Blank lines, including their line terminators, are considered comments and will be completely ignored.
- The non-comment portion of each non-blank logical line consists of a series of *tokens* delimited by whitespace. That is, a token is defined as any consecutive sequence of non-whitespace characters. Whitespace before the first token is optional and is ignored.

28.3 Property Declaration Syntax

Each non-blank logical line of a static properties file is interpreted as a *property declaration*. The first token on the line identifies the property or type of property that is being declared. Additional tokens provide values for the property. Definitions below describe the tokens required for each type of property declaration.

The first declaration in the file must be a “properties_version” declaration, which specifies the version of the static property syntax and semantics used for the file. The current version is “0x090”:

```
properties_version 0x090
```

Properties version 0x090 encompasses all of the rules and definitions in this chapter, including basic syntax and all property declaration definitions. Static properties files that specify this properties version must only include declarations defined for this version. Future versions of this specification may define additional properties versions, with their own set of definitions and rules. The two least-significant hexadecimal digits of the properties version represents 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).

Environments that support any particular properties version are also required to support all subsequent versions with the same major version number; if they do not specifically support the later version, they shall ignore all unrecognized declarations. Environments are required to refuse to install UDI modules that have static properties files with major version numbers that they do not support.

After the “properties_version” declaration, all remaining declarations may appear in any order, except as described for the “module” and “locale” declarations.

In the descriptions below, “<msgnum>” (or “<msgnum1>”, ...) is an ASCII-encoded decimal number used to select a (single-line) message string from a message declaration (described in the next section); leading zeros are ignored for purposes of comparing two message numbers. Message numbers are interpreted relative to each driver, so there is no need for the driver writer to generate numbers that are unique with respect to any other driver.

While drivers must provide the message strings that are specified to be required, environments that choose not to present messages to the user are free to ignore any or all message strings.

28.4 Common Property Declarations

This section lists those property declarations that apply to all types of modules.

28.4.1 Supplier Declaration

Exactly one “supplier” declaration must be included:

```
supplier <msgnum>
```

The supplier message string is used to display the verbose, human-readable name of the supplier of the driver or library. This name should be chosen to be as unique as possible, but the supplier is not required to guarantee that it is globally unique with respect to other suppliers.

28.4.2 Contact Declaration

One or more “contact” declarations must be included:

```
contact <msgnum>
```

The contact message string(s) supplement the “supplier” string with more detailed contact information in cases where verbose output is required. Each contact declaration corresponds to a separate line in the contact info listing. The contact info should generally include at least an e-mail address or URL.

28.4.3 Name Declaration

Exactly one “name” declaration must be included:

```
name <msgnum>
```

The name message string is used to display the verbose, human-readable name of the driver (as opposed to names for individual devices supported by the driver) or library. This name should be chosen to be as unique as possible, but the supplier is not required to guarantee that it is globally unique with respect to other drivers or libraries from the same supplier or from other suppliers.

28.4.4 Shortname Declaration

Exactly one “name” declaration must be included:

```
shortname <name_string>
```

The `<name_string>` string provides a recommended shorthand name for the driver or library. The environment may choose to use this name as is, modify it, or ignore it entirely. The string must be from 1 to 8 characters long and must consist only of upper and lower case letters, digits, and the underscore character (‘_’). This name should be chosen to be as unique as possible, but the supplier is not required to guarantee that it is globally unique with respect to other drivers or libraries from the same supplier or from other suppliers.

28.4.5 Release Declaration

Exactly one “release” declaration must be included:

```
release <sequence_number> <release_string>
```

The *<release_string>* string identifies a release of the driver or library, in “user-friendly” form, that may be presented to users to let them know which release of the driver or library that they are using. *<sequence_number>* is a number encoded as for UDI_ATTR_UBIT32 (see Table 28-2, “Enumeration Attribute Value Encoding,” on page 28-13) that may be used for automatic release comparisons; larger numbers represent more recent releases. Neither of these is related to the properties version or to any UDI interface version.

28.4.6 Requires Declaration

One or more “requires” declarations must be included:

```
requires <interface_name> <version_number> [ private ]
```

Each “requires” declaration specifies a set of programming interfaces (and the associated semantics) that the driver or library uses, and the version of those interfaces to which it conforms.

<interface_name> is a string, and *<version_number>* is a number encoded as a hexadecimal string preceded by “0x”. The combination of interface name and version number must match an interface version supported on the target system.

No two “requires” declarations for the same driver or library may have the same *<interface_name>*.

Specifying the module’s requirements allows the environment to provide support for the module that is specific to its needs. Environments may choose to support multiple versions of any given interface. Larger version numbers represent more recent versions for a given interface name.

All functions and structures defined in the UDI Core Specification are part of the “udi” interface, currently version “0x090”:

```
requires udi 0x090
```

The two least-significant hexadecimal digits of the version represents 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).

Interface names corresponding to other UDI Specifications are defined in those specifications. Library modules may also define and export their own interface names, as described in Section 28.5.1, “Provides Declaration,” on page 28-9.

If the “private” keyword is present, the required interface must match a “provides” declaration in the same package collection; that “provides” declaration must also specify “private”.

If a “requires” declaration precedes any “module” declarations, it applies to all modules of the driver or library. Otherwise, it applies only to the most recently declared module.

28.4.7 Module Declaration

One or more “module” declarations must be included:

```
module <filename> { <c_source_file> }
```

Each “module” declaration denotes a module that is part of this driver or library. Libraries must have only one module declaration. The <filename> string provides the name of a binary module file; it must be a local name, without any path separators, that must conform to “8.3” naming conventions (that is, it must consist of up to 8 characters, optionally followed by a dot (‘.’) and 1-3 additional characters) and must not use any upper-case letters. No two “module” declarations in the same file may use the same <filename> string.

For binary distributions, the module files are included in the distribution, having been previously built in a UDI build environment. Module files are not distributed with source-only distributions, but will instead be built when the driver source code is compiled and linked.

The list of <c_source_file> names specifies the list of C source files that must be compiled and linked in order to build this module. If this static properties file is for a binary-only distribution, no source files will be listed; otherwise, there must be at least one source file for each module. C source file names must be local names, without any path separators, must be less than 64 characters long, and must end in “.c”.

Several declaration types are sensitive to ordering relative to “module” declarations: “requires”, “compile_option”, and “region”. See each declaration section for details.

28.4.8 Compile_option Declaration

One or more optional “compile_option” declarations may be included:

```
compile_option <option> { <c_source_file> }
```

“Compile_option” declarations are used only when building (or re-building) a driver or library from source code. Binary-only distributions need not have any “compile_option” declarations.

If there are any <c_source_file> names listed, they must match one of the C source file names listed in the most recently declared “module” declaration. In this case, the compile option applies only to the listed files and only for that module.

If no C source file names are specified, this compile option applies to all C source files for the most recently declared module. If such a global “compile_option” declaration precedes the first “module” declaration, it applies to all C source files in all modules for this driver or library.

Multiple compile options may be applied to the same source file. If no compile options are applied to a source file, it will be compiled without any special compile options.

Valid compile options are listed in the following table.

Table 28-1 Compile Options

<option>	Description
-D<name>	Causes <name> to be defined as a macro to be replaced by “1”, as if by a #define directive.
-D<name>=<token>	Causes <name> to be defined as a macro to be replaced by <token>, as if by a #define directive.
-U<name>	Causes <name> to be undefined as a macro, as if by a #undef directive. Overrides any -D compile options.

Traditional compile options, such as -g or -O, are not supported, since they will be provided generically by the `udibuild` utility.

28.5 Property Declarations for Libraries

The property declarations in this section apply only to library modules.

28.5.1 Provides Declaration

One or more “provides” declarations must be included:

```
provides <iv_idx> <interface_name> <version_number> [ private ]
```

Each “provides” declaration specifies a set of programming interfaces (and the associated semantics) that the library provides for use by other libraries or drivers, along with the supported version of those interfaces. *<interface_name>* is a string, and *<version_number>* is a number encoded as a hexadecimal string preceded by “0x”. The combination of interface name and version number must be globally unique.

The two least-significant hexadecimal digits of the version represents 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).

The *<iv_idx>* token is an ASCII-encoded decimal number from 0 to 255 that is used to distinguish one “provides” declaration from another. This number must be unique with respect to all “provides” declarations in the same file.

By default, all global symbols exported by the library are available as part of the specified interface. Libraries that support more than one interface or version will need finer control. To do this, they can use the “symbols” declaration. Any library that has multiple “provides” declarations must include “symbols” declarations that correspond to all but at most one of the “provides” declarations.

If the “private” keyword is present, this interface is visible only to modules in the same package collection that also specify “private” with corresponding “requires” declarations.

28.5.2 Symbols Declaration

Zero or more “symbols” declarations may be included:

```
symbols <iv_idx> { [ <library_symbol> as ] <provided_symbol> }
```

Each “symbols” declaration specifies a set of symbols in the library that are associated with a particular interface version provided by the library. The *<iv_idx>* token must match a “provides” declaration in the same file. Multiple “symbols” declarations may be provided for the same *<iv_idx>*.

For any interface version with one or more corresponding “symbols” declarations, only the listed *<provided_symbol>* names will be available to other libraries or drivers that import these symbols via a “requires” declaration. If a listed symbol has a *<library_symbol>* associated with it (before the preceding “as” keyword), then the symbol named *<library_symbol>* in the library will be used to resolve references to the *<provided_symbol>* name; otherwise, the *<provided_symbol>* name will also be used as the library symbol name.

The ability to resolve references to one symbol as another symbol in the library allows a library to support multiple versions of an interface, even if the library’s implementation for some symbols is different for different versions.

Property Declarations for Libraries Static Properties

All symbol names in “symbols” declarations are spelled as they would be in a C language source file, regardless of how they might appear in a symbol table in an object file. Some language or object file conventions modify symbol names before placing them into a symbol table (for example, by prefixing with an underscore character).

28.6 Property Declarations for Drivers

The property declarations in this section apply only to drivers.

28.6.1 Category Declaration

One optional “category” declaration may be included:

```
category <msgnum>
```

The category message string is a human-readable brief (two or three word) description of the category of device supported by the driver. While the overall type of device can be inferred from the set of “requires” declarations, it may be desirable to supplement this categorization with a more specific description.

Each metalanguage that can be used as a child metalanguage specifies a recommended category name (in English) that can be used for the “category” declaration. If a driver wants to express a category that is closely equivalent to one of the recommended names, it should ensure that its specified message text for the POSIX (“C”) locale exactly matches the recommended category name, since some administrative user interfaces will group drivers by exact matches on their category message strings. Some environments may perform this match in the locale for which the strings will be presented; others may use the “C” locale for matching.

Examples of possible category names are listed below. Refer to metalanguage specifications for the official recommended names.

```
SCSI Host Bus Adapters  
Network Interface Cards  
Communications Cards  
Video Cards  
Sound Boards  
Miscellaneous
```

Note that category names are usually pluralized, since they may be used as headers for a group of drivers or devices that belong to the category.

28.6.2 Child_meta Declaration

One “child_meta” declaration must be included for each type of child metalanguage supported:

```
child_meta <meta_idx> <interface_name>
```

A “child_meta” declaration indicates a metalanguage that may be used to bind children to this driver. The <interface_name> string must be the same as the <interface_name> in a “requires” declaration for this driver. Some drivers may support multiple child metalanguages.

If a non-zero **child_bind_interface** is passed to udi_primary_region_init in the driver’s init_module routine, it must be for the parent role of the metalanguage specified by the one and only “child_meta” declaration for this driver. If a driver’s static properties include multiple “child_meta” declarations, **child_bind_interface** must be zero. (See udi_primary_region_init on page 9-8.)

The <meta_idx> token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another (0 is reserved for the Management Metalanguage). This number must be unique with respect to all “child_meta”, “parent_meta” and “internal_meta” declarations for this driver. It is passed to the driver’s udi_prep_for_child_req entry point via the **meta_idx** member of the udi_bind_cb_t control block, in order for the drivers that support multiple child metalanguages to know which one is being used for a given child.

Note – It is legal, though unusual, to have a driver with no “child_meta” declarations. Such a driver can have no children, and is thus really an application running as a UDI driver.

28.6.3 Parent_meta Declaration

One “parent_meta” declaration must be included for each type of parent metalanguage supported:

```
parent_meta <meta_idx> <interface_name>
```

A “parent_meta” declaration indicates a metalanguage that may be used to bind parents to this driver. The <interface_name> string must be the same as the <interface_name> in a “requires” declaration for this driver. Some drivers may support multiple parent metalanguages.

If a non-zero **parent_bind_interface** is passed to udi_primary_region_init in the driver’s init_module routine, it must be for the child role of the metalanguage specified by the one and only “parent_meta” declaration for this driver. If a driver’s static properties include multiple “parent_meta” declarations, **parent_bind_interface** must be zero. (See udi_primary_region_init on page 9-8.)

The <meta_idx> token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another (0 is reserved for the Management Metalanguage). This number must be unique with respect to all “child_meta”, “parent_meta” and “internal_meta” declarations for this driver. It is passed to the driver’s udi_bind_to_parent_req entry point via the **meta_idx** member of the udi_bind_cb_t control block, in order for the drivers that support multiple parent metalanguages to know which one is being used for a given parent.

Drivers with no “parent_meta” declarations can have no parents and are thus called *orphan drivers*. Orphan drivers control no actual devices, but still present the device model(s) appropriate to the child metalanguage(s) they support. (Sometimes the term pseudo-device driver or pseudo-driver is also used to refer to orphan drivers as well as other drivers that do not directly control actual devices.) Orphan drivers are treated specially in the following ways:

- During the normal bind process (see Section 21.5, “Binding Operations,” on page 21-15), the value of **bind_channel** is unspecified and must be ignored. During udi_bind_to_parent_req the orphan driver instance does not do a parent metalanguage-specific bind; it just initializes its own state and responds with udi_bind_to_parent_ack.
- Orphan driver instances are never rebound, so udi_bind_to_parent_req is called only once and therefore **bind_context** will be NULL.
- Orphan drivers have no parents in their device tree (each orphan driver instance forms the root of its own device tree), so must not use sibling group attributes. (See Section 15.4.3, “Sibling Group Attributes,” on page 15-3.)

28.6.4 Internal_meta Declaration

One “internal_meta” declaration must be included for each type of internal metalanguage supported:

```
internal_meta <meta_idx> <interface_name>
```

A “internal_meta” declaration indicates a metalanguage that may be used between regions internal to this driver. The <interface_name> string must be the same as the <interface_name> in a “requires” declaration for this driver.

The <meta_idx> token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another (0 is reserved for the Management Metalanguage). This number must be unique with respect to all “child_meta”, “parent_meta” and “internal_meta” declarations for this driver.

28.6.5 Device Declaration

One or more “device” declarations must be included:

```
device <msgnum> { <attr_name> <attr_value> }
```

The “device” declarations describe the device(s) that can be supported by this driver. There will be one declaration for each model of device. The message string is used to describe the particular device selected by the declaration; it is intended to be a verbose human-readable device name.

The attribute name and value pairs are matched against the enumeration attribute of devices that are possible candidates for being managed by this driver. The set of valid enumeration attribute names is specified by the instance attribute bindings of the driver’s parent metalanguage, as indicated by a “parent_meta” declaration. It is illegal to specify an attribute name that is not a parent metalanguage enumeration attribute or to specify an attribute value that is out of range. Orphan drivers must have exactly one “device” declaration and that declaration must have no attribute pairs; “device” declarations for other drivers must all have at least one attribute pair.

If any specified attributes do not match the corresponding enumeration attribute of a device instance, then this driver will not be used for that device instance. If multiple “device” declarations (from multiple drivers or from the same driver) match a given device instance, only those with the most attribute pairs specified are considered matches. It is environment implementation dependent what the behavior is when multiple candidates match the same device, but it shall not be considered a driver error.

The <attr_value> string must be a single token. Its encoding depends on the type of the enumeration instance attribute (see udi_instance_attr_type_t on page 15-6), according to the following table.

Table 28-2 Enumeration Attribute Value Encoding

Attribute Type	Type Name	Encoding
UDI_ATTR_STRING	string	Literal string value, except that whitespace and hash (#) characters cannot be included directly, so escape sequences from the following table are used to represent these characters. Matching is case-sensitive.
UDI_ATTR_UBIT32	ubit32	The numeric value may be encoded either as an ASCII-encoded decimal string, or as a hexadecimal string preceded by “0x”. Matching is case-insensitive.

Property Declarations for Drivers

Static Properties

Table 28-2 Enumeration Attribute Value Encoding

Attribute Type	Type Name	Encoding
UDI_ATTR_BOOLEAN	boolean	True values are encoded as the single character, “T”; false values are encoded as the single character, “F”. Matching is case-insensitive.
UDI_ATTR_ARRAY8	array	Each byte of the value is encoded as two ASCII-encoded hex digits, with no prefixes or punctuation. The first pair of digits corresponds to the first byte in the array, and so on. All digits must be specified, even if they are zero. Matching is case-insensitive.

Table 28-3 UDI_ATTR_STRING Escape Sequences

2-Character Escape Sequence	Interpretation
_	space
\H	hash character (#)
\\	backslash (/)
\p	Paragraph Break For “message” and “disaster_message” declarations only. May optionally be used by the environment when it formats a message to present to users. The manner in which a paragraph break is rendered is unspecified.
\m<msgnum>	Embedded Message For “message” and “disaster_message” declarations only. The text for the specified message number is recursively embedded into the message text that included this escape sequence. The resulting message text, after escape and whitespace processing must not exceed 2000 bytes. At most three (3) levels of nested embedding—not including the original message—may be used.

Values for enumeration attributes of other types not listed in Table 28-2 cannot be used in property declarations.

28.6.6 Region Declaration

One “region” declaration must be included for each type of region used by the driver:

```
region <region_idx> { <region_attribute> <value> }
```

Each “region” declaration describes a type of region for this driver. A declaration for region index zero is always required; this specifies attributes of the driver’s primary region. The region index is specified as an ASCII-encoded decimal number. No two “region” declarations in the same file may have the same region index.

“Region” declarations must not precede the first “module” declaration. The most recently declared module preceding any “region” declaration must be the module that handles this region index.

Valid values for <region_attribute> and <value> are shown in the following table. The same <region_attribute> must not be listed twice in the same “region” declaration.

Table 28-4 Region Attributes

<region_attribute>	<value>	Meaning
type	normal	A normal region. This is the default value for this attribute.
type	fp	Regions of this type may use floating point operations and data types.
priority	lo	Regions of this type should be scheduled, if possible, at a lower priority than other regions for this driver that have higher priority values.
priority	med	Regions of this type should be scheduled, if possible, at a priority between those for other regions for this driver that have lo priority values and those that have hi priority values. This is the default value for this attribute.
priority	hi	Regions of this type should be scheduled, if possible, at a higher priority than other regions for this driver that have lower priority values (lo or med).
latency	powerfail_warning	Regions of this type service devices that may deliver early warning of impending power failures. Some environments will consider this the most critical type of event to be serviced quickly.
latency	overrunable	Regions of this type service devices that are overrunable without possibility of retry. That is, if they are not serviced soon enough, they may permanently lose data.
latency	retryable	Regions of this type service devices that are overrunable but with the possibility of retry. That is, if they are not serviced soon enough, they may lose data but it can be recovered by retrying the operation.
latency	non_overrunable	Regions of this type service devices that are not overrunable. That is, they will maintain data associated with all outstanding operations until serviced, no matter how long it takes. This is the default value for this attribute.

Table 28-4 Region Attributes

<region_attribute>	<value>	Meaning
latency	non_critical	Regions of this type service devices that are not overrunable and are also considered non-critical relative to other devices. In other words, all other devices may be serviced in preference to a non-critical device if they both have service pending at the same time. Typically this is used for slow, infrequently-used devices like floppy disks.
overrun_time	<nanoseconds>	For regions with overrunnable or retryable latency, this attribute indicates the typical time to overrun, in nanoseconds.

28.6.7 Active_router Declaration

An “active_router” declaration must be included if and only if this driver (typically a multiplexer) actively participates in parent-child routing determination:

```
active_router
```

If an “active_router” declaration is included, the MA will send child routing enumeration requests to the driver when it needs to determine the driver’s parent-child routing topology. Otherwise, the MA will assume the default routing, as specified in Section 21.7.4, “Parent Context,” on page 21-28.

28.6.8 Locale Declaration

One or more optional “locale” declarations may be included:

```
locale <locale>
```

Each “locale” declaration changes the locale to which subsequent “message” and “disaster_message” declarations apply. Until the first “locale” declaration is encountered, the “C” locale will be used.

The locale specifier, <locale>, is in the following form, which is a subset of the POSIX locale specifier format described in ISO/IEC 9945-1:

```
language[_territory]
```

The language specifier is a two- or three-letter language code as defined by ISO 639-2/T, or the special “POSIX” locale designator, “C”. The territory code is an optional specifier, separated from the language specifier by an underscore, that indicates a particular territory or area in which the language is used differently from other areas. The territory code is a two- or three-letter country code as defined by ISO 3166.

At any given time, the environment will determine, in an environment-specific fashion (typically administrator driven), what is the current locale for a particular driver. As message strings are accessed (by driver request or by the environment), the environment will pick a message with the selected number that was associated with the current locale. If it can’t find one, it tries to find the same message number in the “C” locale. If it can’t find the message there either, it will construct a string, in either the “C” locale or the current locale, to the effect of:

[Unknown message number <msgnum>.]

28.6.9 Message Declaration

One or more optional “message” declarations may be included:

```
message <msgnum> { <text> }
```

Each “message” declaration provides text for a given message number, <msgnum>, for a particular locale (see the “locale” declaration). If multiple declarations are given for the same message number in the same locale, the environment may choose any one of the message texts.

The actual message string used will consist of each of the <text> tokens, along with any intervening whitespace, but not any preceding and trailing whitespace. Any whitespace between tokens is treated as a single space character when the message text is used. Each <text> token is encoded as for UDI_ATTR_STRING in Table 28-2. This encoding supports escape sequences that represent characters that can’t be included directly in a token.

Some messages are referenced by other declarations and may be used by the environment. Others may be used by the driver itself, typically for the purpose of tracing and logging.

The driver can retrieve the text of one of its messages by using udi_instance_attr_get with “#<msgnum>” as the attribute name (that is, a hash character followed by the decimal digits of the message number). This will yield an attribute of type UDI_ATTR_STRING that will have had all the escape sequence and whitespace processing already performed.

Environment implementations may choose to manage message strings in any number of ways. They may be accessed directly from the driver properties or the messages files, or they may first be copied into a central message database, possibly with a different format. They may be individually fetched as needed, or they may all be pre-loaded into memory when the corresponding driver is loaded. In fact, an extreme environment could even discard all messages. In any case, none of these environment implementation choices is visible to the driver.

28.6.10 Disaster_message Declaration

One or more optional “disaster_message” declarations may be included:

```
disaster_message <msgnum> { <text> }
```

Any “disaster_message” declaration is treated the same as a “message” declaration, except that it is intended specifically for messages that will be used to log messages with UDI_LOG_DISASTER severity. As such, some environments that don’t pre-load all messages may choose to pre-load just the disaster messages so they’re guaranteed to be available during system abort handling.

28.6.11 Message_file Declaration

One or more optional “message_file” declarations may be included:

```
message_file <filename>
```

Each “message_file” declaration denotes an external text file that includes additional message string definitions for this driver, besides any that may be included in the static driver properties file itself. The <filename> string must name a file that is included with the rest of the driver files, including the static driver properties file, in the same directory; it must be a local name, without any path separators.

Property Declarations for Drivers *Static Properties*

Message files are distributed as separate files from the main driver file(s), even for binary distributions.

Message files have the same format as `udiprops.txt`, and must also begin with a “`properties_version`” declaration. Aside from the “`properties_version`” declaration, however, the only declarations legal in a message file are “`message`”, “`disaster_message`”, and “`locale`”.

Message files must not be larger than 16 MB, and their filenames must conform to “8.3” naming conventions (that is, they must consist of up to 8 characters, optionally followed by a dot (‘.’) and 1-3 additional characters) and must not use any upper-case letters.

28.6.12 Readable_file Declaration

One or more optional “`readable_file`” declarations may be included:

```
readable_file <filename>
```

Each “`readable_file`” declaration denotes a file that may be read by the driver at run time. The `<filename>` string must name a file that is included with the rest of the driver files, including the static driver properties file, in the same directory; it must be a local name, without any path separators.

Readable files are distributed as separate files from the main driver file(s), even for binary distributions.

The driver can read the contents of readable files by using `udi_instance_attr_get` with “`<filename>`” as the attribute name. See Section 15.2, “Instance Attribute Names,” on page 15-1 for restrictions on attribute names. This will yield an attribute of type `UDI_ATTR_FILE`. Readable files are treated as raw binary files and are not in any way preprocessed by the environment.

The following files must not be used as readable files: `udiprops.txt`, `udibuild.txt`, or any file used as a message file (see the “`message_file`” declaration). Readable files must not be larger than 16 MB, and their filenames must conform to “8.3” naming conventions (that is, they must consist of up to 8 characters, optionally followed by a dot (‘.’) and 1-3 additional characters) and must not use any upper-case letters.

28.6.13 Custom Declaration

One or more optional “`custom`” declarations may be included:

```
custom <scope> <attr_name> <msgnum1> <msgnum2> <msgnum3> <choices>
```

Each “`custom`” declaration describes a custom configuration parameter for this driver. The environment will provide a way for the administrator or integrator to set values for each of these parameters. The selected parameter values are made available to the driver via its instance attributes.

`<scope>` determines the applicability of this parameter to the various device instances covered by this driver, according to the following table:

Table 28-5 Custom Parameter Scope

Value of <code><scope></code>	Meaning
<code>device</code>	The parameter applies to each device instance independently, and is required for all device instances.
<code>device_optional</code>	The parameter applies to each device instance independently, and is optional.

Table 28-5 Custom Parameter Scope

Value of <scope>	Meaning
driver	The parameter applies to all device instances covered by the driver and will be set to the same value for each one. The parameter is required for all device instances.
driver_optional	The parameter applies to all device instances covered by the driver and will be set to the same value for each one. The parameter is optional.

<attr_name> is the name of the instance attribute that will be used to represent the value of this parameter. The driver can access these values using instance attribute services (see Chapter 15, “*Instance Attribute Management*”).

<msgnum1> provides the “user-friendly” name for the parameter. It should concisely (about one to three words) elucidate the meaning of the parameter in a form that could be used as a table heading, a menu option or in prose such as “Would you like to change the <msgnum1> parameter?”

<msgnum2> provides a description of the parameter that can be presented to the user to help them understand what the parameter represents. It should be in the form of one or more complete sentences and may consist of multiple paragraphs (though environments are not required to display the message as multiple paragraphs). If the description text refers to any parameter name, it should use the name given by <msgnum1> rather than <attr_name>.

<msgnum3> indicates a sub-category of parameters to which this parameter belongs. Some environments may group parameters by sub-category when presenting lists of parameters to the user. If <msgnum3> is non-zero, the corresponding message string may be used as a heading for the sub-category. Examples of possible sub-categories include “Basic” vs “Advanced” and “Ethernet” vs “Token Ring”.

<choices> describes the set of valid values for the parameter, and is constructed as follows:

```
<attr_type> <default_value> \
( mutex { <value> } end | range <min_value> <max_value> <stride> | any )
```

<attr_type> is the type of the instance attribute that will be used to represent the value of this parameter. This must be one of the Type Names listed in Table 28-2, “*Enumeration Attribute Value Encoding*,” on page 28-13.

<default_value> provides the default value for a parameter, and is encoded as for <attr_value> in “device” declarations (see Table 28-2).

The remainder of the <choices> clause begins with a keyword identifying the type of choice available. The *mutex* keyword indicates a mutually-exclusive set of alternatives, terminated by the *end* keyword. The *range* keyword indicates a range of choices beginning with <min_value>, incrementing by <stride>, until the value exceeds <max_value>; if <attr_type> is not “ubit32”, <stride> is unused and must be set to “–”. The *any* keyword indicates that any value appropriate for the attribute type may be used.

<value>, <min_value>, <max_value>, and <stride>, if provided, are encoded as shown in Table 28-2.

28.6.14 Config_choices Declaration

One or more optional “config_choices” declarations may be included:

```
config_choices <msgnum> { <attr_name> <choices> }
```

These declarations are used for devices on buses that have device configuration attributes that can't be read by generic software (legacy ISA, for example). Such devices will generally require explicit user configuration. The “config_choices” declarations provide default choices for these configuration attributes. The <attr_name> string(s) must correspond to valid enumeration attributes, as described for “device” declarations. The <choices> clause describes a set of parameter value choices, as in the “custom” declaration.

The <msgnum> value must match a <msgnum> used in a “device” declaration. This associates one or more default settings with a given device declaration. If there are more than one for the same device, they represent alternate choices. It is environment implementation dependent how the choice between alternates is made. Environments may choose to ignore some or all “config_choices” declarations.

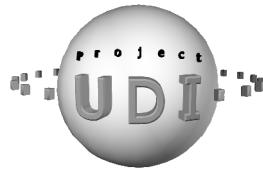
For devices that have factory default settings, the first “config_choices” declaration for such a device should represent the factory defaults.

28.7 Sample Static Driver Properties File

The following example shows what a static driver properties file for a network interface card driver from the XYZ Company might look like.

```
properties_version 0x090
supplier 1
contact 2
name 3
category 4
shortname xyznic
release 5 1.0b5
requires udi 0x090
requires udi_physio 0x090
requires udi_network 0x090
device 5 bus_type pci pci_vendor_id 1234 pci_device_id 19
custom media_type driver 10 11 0 string Ethernet \
    mutex Ethernet Token\Ring end
message 1 XYZ Corporation
message 2 support@xyz.com
message 3 XYZ 552x LAN Driver
message 4 Network Interface Card
message 5 xyz5524 10/100Base-T
message 10 Media Type
message 11 The Media Type parameter indicates the type of network
    to which the card is connected. This may be "Ethernet" or
    "Token Ring".
locale piglatin
message 10 Ediamay ypetay
message 11 Ethay Ediamay Ypetay arameterpay indicatesyay ethay
    ypetay ofyay etworknay otay ichway ethay ardcay isyay
    onnectedcay. Isthay aymay ebay "Ethernetyay" oryay
    "Okentay Ingray".
```

Sample Static Driver Properties File Static Properties



29.1 Overview

This chapter defines the UDI packaging and distribution format for both source and binary distributions of UDI drivers.

29.2 Packaging Format

The UDI packaging format specification describes a directory hierarchy that is used to contain the various components of a UDI driver “package”; i.e. all the files necessary to provide with a driver to make it usable. This directory structure is also used to install a driver once the package has been somehow transported to the target system.

Each environment implementation shall provide a utility program called `udisetup`, that converts the driver files into appropriate native form, installs copies of them into environment-specific locations, and performs any other operations necessary to make the driver available for use.

29.2.1 Directory Structure

In the directory structure shown below, all UDI files are contained under the top-level directory (`udi-dist.1`); if the directory structure were to change in a future version of this specification, a new top-level directory name would be used.

One or more completely independent driver packages may be included in this directory, each rooted at a sub-directory referred to below as `drv_xxx` (which may be any arbitrary name). For example, if a vendor ships multiple network interface controller drivers for different kinds of network adapters, each driver would be packaged under a different `drv_xxx` directory. Environments may choose to install all driver packages in a distribution or to present the user with a choice of driver package names.

One or more alternate versions of the same logical driver package may be included within the `drv_xxx` directory, each rooted at a sub-directory referred to below as `alt_yyy` (which may be any arbitrary name). Environments will install only one alternate version for any driver package. Alternates will be selected based on the UDI specification version on which they depend, as well as the versions of other interfaces, such as metalanguages. Only alternates that require only supported interface versions will be considered. If multiple alternates use supported interfaces, it is unspecified which one will be selected.

One or more components (individual UDI drivers or libraries) may be included with in the `alt_yyy` directory, each rooted at a sub-directory referred to below as `comp_zzz` (which may be any arbitrary name). Environments will install all components of a driver package for the selected alternate. This is useful for grouping internal metalanguage libraries with the drivers that use them, sets of cooperating drivers, etc.

One file named `udiprops.txt` must exist for each component. The contents of that file are specified in Chapter 28, “*Static Driver Properties*”. If this component is being distributed in source form, the driver source is contained in the `src` subdirectory. If this component is being distributed in binary form, the driver binaries are contained in the `bin` directory. Beneath the `bin` directory, the `<abi>` subdirectories contain the driver binaries built to conform to a particular Architected Binary Interface. For example `/bin/IA32` would contain the driver binaries for the 32-bit Intel Architecture, `/bin/IA64` for the 64-bit Intel Architecture, etc. The optional `aux` directory contains files readable by the driver. These are usually microcode files for downloading to a particular adapter or device. The optional `msg` directory contains text for messages used by the driver.

```
/udi-dist.1/<drv_xxx>/<alt_yyy>/<comp_zzz>/
    udiprops.txt
    src/<source files>...
    bin/<abi>/<primary module file>
        <secondary module files>...
    aux/<readable files>...
    msg/<message files>...
```

Each of the `src`, `bin`, `aux`, and `msg` subdirectories are optional. They are required only if one or more of the corresponding type of file is included in the package. At least one of `src` or `bin` must be present.

29.3 Distribution Format

The UDI distribution format specification defines how UDI driver packages may be placed on various distribution media. Environments are not required to support any of these media types. However, for any listed media type from which an environment supports installation of UDI drivers, the storage format and layout specified herein must be supported.

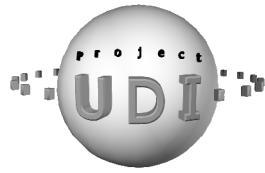
All UDI distribution formats simply use the UDI packaging format directory hierarchy stored on the media in a specified storage format.

29.3.1 Floppy Storage Format

For floppy disks, the storage format will be a DOS FAT12 filesystem. This filesystem supports directory hierarchies, but the directory and file names are limited to 8.3 format (up to 8 characters, optionally followed by a dot (‘.’) and 1-3 additional characters) and are case-insensitive but stored as upper case. The UDI package directory hierarchy must be placed at the root of this filesystem.

29.3.2 CD-ROM Storage Format

For CD-ROMs, the storage format shall be an ISO-9660 filesystem. The UDI package directory hierarchy must be placed at the root of the directory hierarchy identified by the Primary Volume Descriptor.



30.1 Overview

This chapter describes utilities that UDI build environments and UDI runtime environments must provide.

30.2 The *udimkpkg* Utility

All UDI build environments shall include a utility called *udimkpkg*. This utility takes the static driver properties file, *udiprops.txt*, and attaches it to the main module (typically a relocatable object file) of the driver. It also gathers the resulting modified object file and any auxiliary files that are part of the driver (including message files, readable files, and secondary module object files) and creates a UDI “package”, ready for distribution or installation.

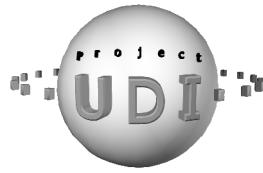
The output of *udimkpkg* is a package directory structure. Additional environment-specific tools may be required to copy this directory structure onto physical media or to upload it to a network server.

When run in the directory containing all of the input files, with no command-line arguments, the *udimkpkg* utility will create a distribution package rooted in a sub-directory named “*pkg*”.



UDI Core Specification

Section 7: ABI Bindings



31.1 Introduction

Most of the UDI specification documents (see Chapter 2, “*Document Organization*”) define programming interfaces and conventions which are applicable to any processor or platform architecture; these specifications, referred to as API-level specifications, provide source-level portability of UDI drivers from one platform or operating environment to another. There is a class of UDI specifications, called ABI Bindings, which define binary bindings to specific processor architectures; these specifications provide binary-level portability of UDI driver modules, allowing the driver to be compiled once for a target architecture and distributed to any platform or operating system which conforms to the ABI specification.

A UDI ABI binding consists of the following components:

1. A processor architecture, instruction set definition, and endianness. This defines the hardware datatypes, resources, instruction set, and endianness of the compiled UDI driver.
2. Runtime architecture. This defines the procedure calling conventions, register usage, stack conventions, data layouts, etc.
3. Binary bindings to the API-level UDI specifications. This specifies the sizes of fundamental UDI data types, and the binary requirements of implementation-dependent UDI macros.
4. Building the driver object. This specifies the object file format and the encapsulation of the static driver properties in the object files.

31.2 Processor Architecture

Each ABI binding specification must specify a processor architecture with its associated instruction set definition as well as supported endianness modes. For example, separate ABI bindings will be specified for IA-32 (little endian), IA-64 (both big and little endian), PowerPC (big endian), etc.

31.3 Runtime Architecture

Associated with a given processor architecture is a set of conventions for executing software on that processor called the runtime architecture. This includes such things as the procedure calling conventions, register usage, stack conventions, data layouts, etc. The runtime architecture definition is typically defined in a separate non-UDI specification for a particular processor architecture, and is simply referenced by the UDI ABI specification.

31.4 API-Level Bindings

A few aspects of the API-level specifications are implementation-dependent and thus require additional specification for binary portability. These fall into two general categories: the sizes of fundamental UDI data types, and the definitions of implementation-dependent UDI macros.

Note – The values for symbolic constants are all defined in the API-level specifications, and thus require no specification in the ABI bindings.

31.4.1 Sizes of UDI Data Types

As part of its C language binding, UDI defines a set of fundamental data types (see Chapter 8, “Fundamental Types”). The UDI interfaces, in any of the UDI specifications, use only these fundamental types or types derived therefrom. The fundamental types include only a few standard ISO C types, namely char, void, and the varargs types; all the other fundamental types are UDI-defined and include a set of specific-length types, abstract types, and opaque types. Of these fundamental types, only the abstract types and opaque types have implementation-dependent sizes. Additionally, the sizes of pointer types (“void *” or any other pointer type) are implementation-defined. Thus an ABI specification only needs to specify the sizes of UDI’s abstract types, opaque types, and pointer types, which are listed in the table below.

Table 31-1 UDI Data Types whose sizes need to be defined by the ABI

Data Type	Classification	Description
void *	Pointer type	Pointer type
udi_size_t	Abstract type	Size type
udi_index_t	Abstract type	Index type
udi_buf_t	Opaque type	Buffer Handle
udi_channel_t	Opaque type	Channel Handle
udi_constraints_t	Opaque type	Constraints Handle
udi_timestamp_t	Opaque type	Timestamp Type
udi_pio_handle_t	Opaque type	PIO Handle – defined in the Physical I/O Specification
udi_dma_handle_t	Opaque type	DMA Handle – defined in the Physical I/O Specification

All pointer types have the same size, represented by the “void *” row in the table. The only fundamental types not defined in the Core Specification are the handle types defined in the Physical I/O Specification. For convenience and completeness, the physical I/O fundamental types are included here.

Note – No specifications other than the Core or Physical I/O Specification are allowed to define UDI fundamental data types.

31.4.2 Implementation-Dependent Macros

There is one implementation-dependent macro in the UDI specifications - the UDI_HANDLE_ID macro on page 17-12. Each ABI must specify the binary requirements of the UDI_HANDLE_ID macro, while leaving implementation flexibility where appropriate. For example, if in a given ABI the handles all have size equal to the size of a pointer then the binary requirement might be simply that this macro returns the exact, untransformed contents of the corresponding handle without engendering any additional external symbol references. This would then allow for environment implementation flexibility with respect to the actual C definition of the macro; e.g., depending on the actual type definition of the handle and the target programming model (e.g., LP64 vs P64) it could be provided as either

```
#define UDI_HANDLE_ID(handle,handle_type) ((void *) (handle))
```

or

```
#define UDI_HANDLE_ID(handle,handle_type) (* (void **) (&(handle)))
```

and still conform to the ABI.

Note – Only the Core and Physical I/O Specifications are allowed to define macros which have implementation-dependent definitions.

31.5 Building the Driver Object

The Core Specification defines the generic aspects of building UDI driver object code. The ABI specification defines the binary bindings of building the driver object code: i.e., the object file format, and the encapsulation of the static driver properties in the driver's object files.

31.5.1 Object File Format

Each ABI specifies an applicable object file format. Typically this is defined in a non-UDI specification, and is simply referenced in the UDI ABI specification.

31.5.2 Static Driver Properties Encapsulation

Each ABI needs to specify how the static driver properties (provided by the driver in its udiprops.txt configuration file – see Chapter 28, “*Static Driver Properties*”) are attached to the driver's object files.



UDI Core Specification

Section 8: Appendices

Glossary

A

anchored channel	a channel end which has been permanently associated with a region, a set of channel operation entry points, and a channel context. Inter-module communication can only occur over a channel that has both ends anchored.
asynchronous service call	an environment service call that indicates its completion through an asynchronous callback. The service is not necessarily complete and/or the callback may not have been called upon return from the initial environment function call to the calling code.
bind channel	first communication channel between two driver instances, which results from the bind process.
bind process	the process of associating two driver instances in a child-parent (or client-server) relationship, typically reflecting the relationship of the associated hardware components. This process takes place as part of driver configuration, via a set of channel operations, and results in initialization of a communications channel between the two driver instances.
binding	see <i>bind process</i> .
callback	a driver-provided procedure that may be called immediately or at some later time when a requested service has been performed. The callback procedure is always invoked within the same driver region as the original service request, but possibly on a different thread. Example: Procedure <code>foo</code> requests a service, e.g., allocation of a block of memory. This service request specifies the address of procedure <code>foo_callback</code> , in addition to the amount of memory requested. If the resource is immediately available, the callback may be invoked on the requestor's thread before the request procedure returns. Otherwise, when the resource becomes available, <code>foo_callback</code> will be invoked by rescheduling the same region's processing on a (possibly different) thread.
channel	a bidirectional communication channel between two drivers, or between a driver and the environment. Channels allow code running in one region to invoke <i>channel operations</i> in another region. Example of channels include the <i>bind channel</i> between an adapter driver instance and its parent driver instance, and the <i>management channel</i> between a driver instance and the Management Agent.
channel context	a pointer to a driver-defined storage area, used by a driver to store state information, resources that it has allocated, etc. This pointer is associated with the end of a channel (each anchored channel end has a context associated with it), and

Glossary

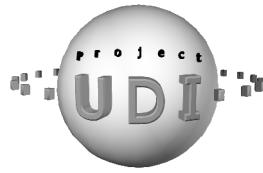
	<p>is passed to the driver as the first parameter for every channel operation which is invoked on that channel. A channel context pointer is local to the region containing the endpoint, and is not visible from the other end of the channel.</p>
channel handle	<p>an identifier used by a driver to refer to a channel. This handle is opaque and local to the driver region in which it is held. Channel handles are used in channel operation invocations to designate the destination of the operation.</p>
channel operation	<p>the unit of inter-module communication over a channel. This is a function call made from one driver region that invokes a similar function call in another region on the other end of the channel. Channel operations are strongly typed. Also known as “ops”.</p>
channel operations vector	<p>the set of metalanguage-defined driver entry points for a given channel or set of channels. When the driver’s init_module routine is called during initialization, it sets up at least one channel operations vector for each type of channel supported by the driver. Also known as “channel ops vector.”</p>
child driver instance	<p>of a pair of communicating driver instances, the one whose position in the device tree is farther from the root of the tree. For example, a SCSI disk is usually the child of a SCSI host bus adapter. The child driver instance is typically the <i>client</i> in this relationship.</p>
communications channel	<p>see <i>channel</i>.</p>
control block	<p>a semi-opaque object used by the UDI environment to store channel operation parameters in a region queue, when the region is busy or operating at a lower priority, or asynchronous service call parameters when the service call cannot be completed immediately. It can also be used by the driver to store channel operation parameters and other contextual information internal to the driver when the operation cannot be handled to completion in one invocation.</p>
destructive diagnostic request	<p>an operation which may have effects external to the driver to which the request was directed.</p>
external mapper	<p>an OS-specific software module having a native OS interface on one side and a UDI interface on the other. This module provides the interface between the native operating system and the “top-most” or “bottom-most” UDI driver. Since it straddles the UDI boundary, it must be viewed as two halves: half in and half out of the UDI environment. The half that is within the UDI environment must obey all rules for UDI drivers (e.g., non-blocking calls only), whereas the other half is not so restricted and may do whatever it must to satisfy the embedding system.</p>
IMC	<p>inter-module communication. The set of system services providing the complete data path for implementing channel operations between drivers, including all translations through metalanguage libraries, environment agents and metalanguage mappers. The process and path is totally transparent to both caller and callee</p>
implicit synchronization	<p>UDI guarantees that only a single call to one channel operation or callback will be activated at any one time for a given region. This causes each service routine to be a critical section; safe in uniprocessor and multiprocessor systems. The region instance inherently controls thread execution within a UDI driver, and so the granularity of UDI synchronization is defined by the granularity of the regions defined by the driver.</p>

Glossary

internal metalanguage	a driver-defined metalanguage used to communicate between multiple regions in a multi-region driver instance. Some drivers may choose to use the Internal Management Metalanguage as an internal metalanguage.
loose end	a channel end that has not yet been anchored. Channel handles for loose ends may be transferred between regions, but those for anchored channels may not.
MA	see <i>Management Agent</i> .
Management Agent	the agent or set of cooperating agents within the environment which is responsible for managing drivers, including creating driver instances and binding them together to reflect the system configuration topology, including the device tree.
management channel	the channel between the Management Agent and the primary region of a driver instance, used for management operations.
mapper module	a UDI software-only driver that maps one metalanguage to another, for example a Fibre Channel to SCSI mapper. A mapper module is 100% UDI code, unlike an <i>external mapper</i> . Also known simply as a <i>mapper</i> .
marshalling	see parameter marshalling.
MEI	Metalanguage-to-Environment Interface. Defines the interfaces needed to implement portable metalanguage libraries. See Chapter 25, “Introduction to MEI”.
metalanguage	an agreement between the respective channels of a driver and one or more clients; the operations that can be performed on each channel; the syntax and semantics of each operation; and the relationships among, and allowable sequences of, a set of channel operations. There are, for example, (in other UDI Specifications) a metalanguage defined for SCSI devices and one for Networking devices. Other metalanguages will be defined as more UDI drivers for other classes of devices are needed.
module	A set of ISO C routines which completely define some I/O-related functionality. The term is also used more specifically to refer to one of possibly several independently loadable parts of a UDI driver.
non-transferable handle	a handle that is not transferable. The environment is only guaranteed to understand such a handle (i.e. map it to the correct object) when used from the region for which it was originally allocated.
operation	a structured procedure call used to pass information between modules or to act upon an object. See also <i>channel operation</i> .
parameter marshalling	taking parameters of a channel operation and saving them in a storage area, such as in a <i>control block</i> . When performed within the same address domain, this merely involves copying information, but when performed in a domain-crossing operation, the parameters may need to be converted to a portable, storage-format-independent format; for example, ASN.1 or XDR.
parent driver instance	of a pair of communicating driver instances, the one whose position in the device tree is closer to the root of the tree. For example, a SCSI host bus adapter is usually the child of a SCSI disk. The parent driver instance is typically the <i>server</i> in this relationship.

Glossary

primary region	the region created by the Management Agent (MA) when it creates a driver instance. Drivers may create additional regions for a given driver instance, but this one comes for free. Only primary regions have management channels, to interact with the MA.
region	a UDI-internal data structure containing a synchronization queue to hold incoming channel operations and callbacks when they are delayed because a non-reentrant portion of the driver code is busy or operating at a lower priority. A region may be associated with, and hold (queue) channel operations for, one or more channels. The driver-writer specifies the grouping of channels to regions when channels are anchored. Region structures are not visible to UDI drivers or their clients.
region attribute	a driver region classification based on the general type of usage of a region and associated operational parameters. For example, there might be normal, interrupt, and low-priority regions. These properties (provided by the driver writer) are used by the platform to determine OS-dependent parameters like priority and capability privileges. These values may also be mapped to OS parameters configured by the system operator. The OS parameters that are indirectly determined by the region attribute are attached to the region at run-time and affect its handling by the UDI environment.
scratch space	a block of driver-private space associated with a control block.
semi-opaque object	a data structure which is shared between drivers and the environment but may contain environment-private data which is hidden from drivers. The UDI documents specify the “visible” fields a driver may access. The environment may store additional data before or after the visible fields. A <i>control block</i> is an example of a semi-opaque object.
service call	a call to the environment to perform a particular service. UDI has two types of service calls: synchronous and asynchronous.
synchronous service call	an environment service call that is complete upon return from the function call to the environment service and does not block.
system abort	an action causing a drastic, and immediate, termination of the OS and normally stalling or rebooting of the host CPU(s). No UDI device driver is allowed to directly (and intentionally) generate a system abort.
target channel	a channel handle used as the destination of an channel operation. The operation is sent to the other end of the target channel. Always the first argument to a channel operation from the caller’s point of view.
timeout distortion	the exact delay of a timeout is delimited only by the requested “floor” value provided by the original timeout call. Thus, a callback routine is subject to both the system event timing resolution of 10 mS (typical), the processing time for higher-priority actions preceding the timeout servicing, and the minimum delay requested.
transferable handle	a handle that can be passed from one region to another, and subsequently be used by the other region to access the object (via environment service calls). Transferable handles in UDI must only be transferred via strictly-typed channel operation parameters, so the environment has a chance to translate the handle as appropriate for the destination region. Handles must not be passed between regions without environment intervention, since the bit pattern representing a handle for a particular object may vary from region to region.



Index

A

Abstract Types 8-5
abstract types 8-1
asynchronous service calls 10-1, 10-3
attributes 15-1

B

Buffer Recovery 13-11, 20-4
bus bindings 7-1

C

callee side 4-4
caller side 4-4
channel
 channel endpoint 4-2
 definition 4-2
 ops vector 4-2
channel operation entry point 4-4
channel operation invocation 4-4
channel operations 4-4
channels 16-1
common trace event 17-2
control block index 8-5
custom metalanguages 20-1

D

device instance
 definition 4-1
Directed Enumeration 21-29
driver entry points 4-4
driver execution
 module-global 4-3
 per-instance 4-3
driver instance
 definition 4-1
 per-instance state 4-1

driver instance attributes 15-1

driver modules
 definition 4-1
 module property 4-1
 primary module 4-1
 relationship to regions 4-2
 secondary modules 4-1
driver-defined trace event 17-2

E

Enumeration Attribute
 Locator 15-2
Enumeration, Directed 21-29

F

Fundamental Types 8-1

G

generic pointers 8-2

I

instance 4-1
instance-independence 4-2
interrupt region 4-3
ISO C 8-2

L

line terminator 28-3
list head element 19-14
location-independence 4-2
loose ends 8-8

M

Management Agent 21-1
metalanguage 4-2, 7-1
metalanguage index 8-6

metalanguage-defined trace event 17-2
module-global service calls 4-4
modules 28-1
movable memory 5-2

N
NULL 8-2

O
opaque handles 8-7
Opaque Types 8-7
opaque types 8-1
ops index 8-6
orphan drivers 28-12

P
placeholder 8-3
property declaration 28-4

R
region
 context 4-3
 definition 4-1
 multi-region driver 4-2
 primary region 4-2
 secondary regions 4-2
 single-region driver 4-2
 sub-instance 4-1
region data area 9-9, 9-11
region index 8-6
region-context service calls 4-4
regions 16-1

S
semi-opaque types 8-1, 8-9
service calls 4-4
Specific-Length Types 8-3
specific-length types 8-1
standard metalanguages 20-1
static driver properties 28-1
structures
 fixed binary representation 8-18
 hardware-defined 8-18
synchronous service calls 10-1

T
token 28-3
Trace events 17-2

U
UDI environment
 implementations
 portability 1-1
udimkpkg 28-2
udiprops.txt 28-2

W
whitespace 28-3