Uniform Driver Interface

UDI Core Specification Version 1.0

Volume II (Chapters 20-34)

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

UDI Core Specification

Abstract

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

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

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

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

Status of This Document

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

Copyright Notice

Acknowledgements

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

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

Thanks to the following folks who contributed significant amounts of time, ideas, or authoring in support of the development of this specification:

> Richard Arndt (IBM), Bob Barned (Lockheed Martin), Mark Bradley (Adaptec), Darren Busing (Adaptec), Thomas Clark (Sun), Mark Evenson (HP), Barry Feild (SCO), Kurt Gollhardt (SCO), Jim Heidbrink (Lockheed Martin), Chris Ilnicki (HP), Bret Indrelee (SBS Technologies), David Kahn (Sun), John Lee (Sun), Robert Lipe (SCO), Lynne McCue (IBM), Mark Parenti (DEC), James Partridge (IBM), Kevin Quick (Interphase), Larry Robinson (Adaptec), James Smart (Compaq), Pete Smoot (HP), David Stoft (HP), Wolfgang Thaler (Sun), Linda Wang (Sun), Mike Wenzel (HP).

In addition, thanks to everyone who worked on the prototype implementations that helped us validate the specification, including:

> Betty Dall (HP), Don Dugger (Intel), Bob Goudreau (Data General), James Hall (SCO), Matt Kaufman (SCO), Mike Lyons (IBM), Alex Malone (DEC), Guru Pangal (Starcom), Scott Popp (SCO), Richard Schall (Intel), Sam Shteingart (HP), Ajmer Singh (SCO), Ramaswamy Tummala (Starcom).

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

Table of Contents

Volume I

Section 1: Overview

Section 2: Architecture

Volume II

Section 4: Core Utility Functions

Section 5: Core Metalanguages

Section 9: Appendices

List of Reference Pages by Chapter

Volume I

Chapter 9 Fundamental Types

Chapter 10 Initialization

Chapter 11 Control Block Management

Chapter 12 Memory Management

Chapter 13 Constraints Management

Chapter 14 Buffer Management

Chapter 15 Time Management

Chapter 16 Instance Attribute Management

Chapter 17 Inter-Module Communication

Chapter 18 Tracing and Logging

Chapter 19 Debugging Services

Volume II

Chapter 21 String/Memory Utility Functions

Chapter 22 Queue Management Utility Functions

Chapter 23 Endianness Management Utility Functions

Chapter 25 Management Metalanguage

Chapter 26 Generic I/O Metalanguage

Chapter 27 Diagnostics Support

Chapter 29 Metalanguage-to-Environment Interface

UDI Core Specification

Section 4: Core Utility Functions

Introduction to Utility Functions 20

20.1 Overview

This section defines general utility functions (library functions) and macros available to UDI drivers. UDI utility functions, whether defined in this section or elsewhere, 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.

Because UDI utilities have no platform dependencies (they may be implemented differently in different environments, but not in a way that affects cross-platform portability), the UDI utility functions may be implemented as macros without affecting binary portability. In other words, the utility can be implemented in various ways in the C language, but the functionality provided by the utility is in no way platform dependent and will therefore not break binary portability if implemented as a macro.

Note – Unless otherwise stated, the results of passing a NULL or other invalid pointer to a utility function are unspecified.

The utility functions in this section are divided into three categories:

1. String/Memory Utility Functions
2. Queue Management Utility Functions
3. Endianness Management Utility Functions

NEED TO MOVE THIS:

Non-utility functions can have platform dependencies and therefore must generally be implemented as external function calls. However, an ABI may specify that some functional interfaces may be partially implemented as macros. Such macros would in turn call ABI-specified external functions to perfom any environment-specific functionality that would not be portable across UDI environments that support this ABI. For example, ABIs might specify `udi_assert` as a macro that perfoms the assertion check and calls an ABI-specified function if the assertion fails.

See Section 34.4, "Binary Bindings to the Source-Level Specifications", for additional information.

String/Memory Utility Functions 21

21.1 Overview

This chapter defines string and memory utility functions. The first section lists general string/memory functions. The second section lists `udi_snprintf` and related formatting functions.

21.2 General 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).

NAME **udi_strlen** *Determine string length*

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_strcat, udi_strncat *String concatenation*

SYNOPSIS

```
#include <udi.h>

char *udi_strcat (
     char *s1,
     const char *s2 );

char *udi_strncat (
     char *s1,
     const char *s2,
     udi_size_t n );
```

ARGUMENTS

s1 is a pointer to the destination string.

s2 is a pointer to the source string.

n is the destination string maximum length (in bytes).

DESCRIPTION

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.

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.

RETURN VALUES

The `udi_strcat` and `udi_strncat` functions return a pointer to the resulting null-terminated string ***s1***.

NAME

udi_strcmp, udi_strncmp, udi_memcmp *String/memory comparison*

SYNOPSIS

```
#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 );
```

ARGUMENTS

s1 is a pointer to the first character string or memory area.

s2 is a pointer to the second character string or memory area.

n is the maximum size to be compared (in bytes).

DESCRIPTION

The `udi_strcmp` and `udi_strncmp` 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.

For the `udi_strncmp` 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 `udi_strncmp` function return value indicates that the strings are equal, regardless of any remaining content.

The `udi_memcmp` function operates in a similar manner as `udi_strncmp` 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.

RETURN VALUES

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.

NAME

udi_strcpy, udi_strncpy, udi_memcpy, udi_memmove *String/memory copy*

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 must 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_strncpy_rtrim *Copy char array to string, removing trailing blanks*

SYNOPSIS

```
#include <udi.h>

char *udi_strncpy_rtrim (
     char *s1,
     const char *s2,
     udi_size_t n );
```

ARGUMENTS

s1 is a pointer to the destination character array string.

s2 is a pointer to the source character array, which must be at least ***n***-bytes in size.

n is the number of bytes in ***s2*** to be operated on (i.e., trailing blanks removed and the remaining bytes copied).

DESCRIPTION

The `udi_strncpy_rtrim` function copies the first ***n*** bytes of the character array pointed to by ***s2***, with trailing blanks removed, to the character array string pointed to by ***s1***. The resulting ***s1*** will be null terminated, with a null character ('\0') appended if necessary. The character arrays pointed to by ***s1*** and ***s2*** must not overlap and the destination array ***s1*** must be large enough to receive the copy (i.e., up to ***n***+1 bytes in size).

The source array pointed to by ***s2*** is operated on as an ***n***-byte character array; any embedded null characters will be copied along with other non-trailing blank characters. For example, if ***s2*** is a 10-byte array containing the 4 characters "abcd", followed by '\0', followed by "efghi", the resulting ***s1*** will contain the same 10 characters with an additional '\0' appended at the end. Similarly, if ***s2*** is a 10-byte array containing the 9 characters "abcdefghi" followed by '\0', the resulting ***s1*** will contain these 10 characters followed by an additional '\0'.

NAME

udi_strchr, udi_strrchr, udi_memchr *String/memory searching*

SYNOPSIS

```
#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 );
```

ARGUMENTS

s is a pointer to the string or memory area to be searched.

c is the character to search for.

n is the maximum number of bytes to search.

DESCRIPTION

The `udi_strchr` function returns a pointer to the first occurrence of the character ***c*** in the null-terminated string ***s***.

The `udi_strrchr` function return a pointer to the last occurrence of the character ***c*** in the null-terminated string ***s***.

The `udi_memchr` function returns a pointer to the first occurrence of the (unsigned character) byte ***c*** in the specified memory region, regardless of null bytes.

RETURN VALUES

These functions return a pointer to the matched character or NULL if the character is not found.

NAME **udi_memset** *Memory initialization*

SYNOPSIS

```
#include <udi.h>

void *udi_memset (
     void *s,
     udi_ubit8_t c,
     udi_size_t n );
```

ARGUMENTS

s is a pointer to the memory area to be initialized.

c is the unsigned 8-bit value to use for initialization.

n is the size of the memory area (in bytes).

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***.

NAME

udi_strtou32 *Convert string to unsigned 32-bit value*

SYNOPSIS

```
#include <udi.h>

udi_ubit32_t udi_strtou32 (
     const char *s,
     char **endptr,
     int base );
```

ARGUMENTS

s is a pointer to the null-terminated string to be converted.

endptr optionally points to a character pointer in which a pointer to the first unconverted character is stored.

base is the base radix for interpretation of the numeric string.

DESCRIPTION

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.

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.

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.

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.

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.)

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.)

RETURN VALUE

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.

If the numeric string would cause a 32-bit integer overflow then the resulting value is unspecified.

21.3 String 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** *Format printable string*

SYNOPSIS

```
#include <udi.h>

udi_size_t udi_snprintf (
     char *s,
     udi_size_t max_bytes,
     const char *format,
     ... );
```

ARGUMENTS

s is a pointer to the target buffer for the formatted output, which is a null-terminated printable string.

max_bytes is the maximum number of bytes to be written to ***s***, including the null terminator.

format is the format string, which controls the formatting of the output string.

... are the remaining arguments, which provide the values used for the formatting codes.

DESCRIPTION

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.

All format specifications are of the form *%mf* where *m* is an optional modifier of the form [[0, -] *nn*] and *f* is one or more format codes.

The following format modifiers are defined:

Format Modifier	Output Control
nn	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 will 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 udi_ubit32_t. 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 udi_sbit32_t and udi_ubit32_t
%hx, %hX	unsigned hexadecimal udi_ubit16_t
%hd, %hu	signed and unsigned decimal udi_sbit16_t and udi_ubit16_t
%bx, %bX	unsigned hexadecimal udi_ubit8_t
%bd, %bu	signed and unsigned decimal udi_sbit8_t and udi_ubit8_t
%p, %P	hexadecimal pointer value, size as appropriate for the host machine
%a, %A	64-bit bus address value (DMA address type udi_busaddr64_t; see the description of the udi_scgth_t 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 udi_ubit32_t value; the bitmask interpretation of that value is based on the *istring* information as described here, where bit number 0 is the least-significant bit in the value: *istring* := [,] *bitspec* {, *bitspec*} [,] *bitspec* := [~] *bitnum* = <string> | *bitnum* - *bitnum* = <string> [*fieldspec* ...] *bitnum* := <decimal number 0-31> *fieldspec* := : <unsigned decimal number> = <string> The output is delimited by '<' and '>' characters. If bitspec is specified as a single *bitnum* (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 bitspec 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 fieldspec values, the fieldspec 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** *Format printable string with varargs*

SYNOPSIS

```
#include <udi.h>

udi_size_t udi_vsnprintf (
     char *s,
     udi_size_t max_bytes,
     const char *format,
     va_list ap );
```

ARGUMENTS

s is a pointer to the target buffer for the formatted output, which is a null-terminated printable string.

max_bytes is the maximum number of bytes to be written to ***s***, including the null terminator.

format is the format string, which controls the formatting of the output string.

ap is the varags argument list. (The `va_list` type definition is provided by the `udi.h` header file.)

DESCRIPTION

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.

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.

EXAMPLE

```
#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);
```

REFERENCES `udi_snprintf`

Queue Management Utility Functions 22

22.1 Overview

This chapter defines queue management utility functions.

22.2 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.

22.2.1 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 22-1.

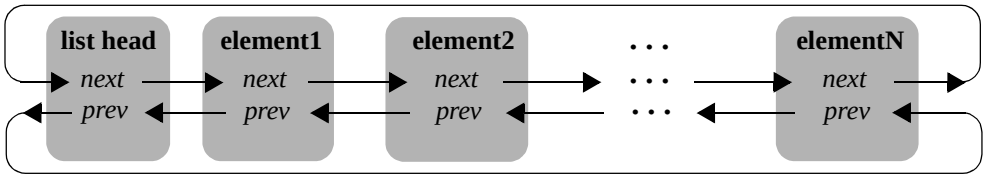


Figure 22-1

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 *Queue element structure*

SYNOPSIS

```
#include <udi.h>

typedef struct udi_queue {
        struct udi_queue *next;
        struct udi_queue *prev;
} udi_queue_t;
```

MEMBERS

next is a pointer to the next element in the queue.

prev is a pointer to the previous element in the queue.

DESCRIPTION

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.

22.2.2 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 *Insert a queue element into a queue*

SYNOPSIS

```
#include <udi.h>

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

ARGUMENTS

new_el is a pointer to a queue element.

old_el is a pointer to the queue's list head or an element already on the queue.

DESCRIPTION

udi_enqueue inserts ***new_el*** after ***old_el*** in the queue to which ***old_el*** belongs.

udi_enqueue shall be equivalent to the following implementation:

```
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;
}
```

REFERENCES

udi_dequeue, udi_queue_t

NAME **udi_dequeue** *Dequeue a queue element*

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:

```
udi_queue_t *udi_dequeue(
     udi_queue_t *element)
{
     element->next->prev = element->prev;
     element->prev->next = element->next;
     return element;
}
```

RETURN VALUES

The ***element*** passed in is returned.

REFERENCES

udi_enqueue, udi_queue_t

22.2.3 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.

NAME

UDI_QUEUE_INIT,
UDI_QUEUE_EMPTY *Initialize queue; check if it's empty*

SYNOPSIS

```
#include <udi.h>

#define UDI_QUEUE_INIT(listhead) \
     ((listhead)->next = (listhead)->prev = listhead)

#define UDI_QUEUE_EMPTY(listhead) \
     ((listhead)->next == (listhead))
```

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:

```
void UDI_QUEUE_INIT (
          udi_queue_t *listhead );

udi_boolean_t UDI_QUEUE_EMPTY (
          udi_queue_t *listhead );
```

REFERENCES

udi_queue_t

NAME

UDI_ENQUEUE_XXX,
UDI_QUEUE_INSERT_XXX *Insert an element into a queue*

SYNOPSIS

```
#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)
```

ARGUMENTS

listhead is a pointer to a list head element.

element is a pointer to a queue element.

old_el is a pointer to a queue element that is currently linked into a UDI queue.

new_el is a pointer to a queue element that is not currently linked into a UDI queue.

DESCRIPTION

UDI_ENQUEUE_HEAD inserts ***element*** at the head of the queue specified by ***listhead***. This macro is equivalent to the udi_enqueue function.

UDI_ENQUEUE_TAIL appends ***element*** to the tail of the queue specified by ***listhead***.

UDI_QUEUE_INSERT_AFTER inserts the queue element ***new_el*** after the element ***old_el***.

UDI_QUEUE_INSERT_BEFORE inserts the queue element ***new_el*** in front of the element ***old_el***.

These macros must be called as if they, respectively, had the following functional interfaces:

```
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 );
```

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

REFERENCES udi_enqueue

NAME

UDI_DEQUEUE_XXX,
UDI_QUEUE_REMOVE *Remove an element from a queue*

SYNOPSIS

```
#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))
```

ARGUMENTS

listhead is a pointer to a list head element.

element is a pointer to a queue element.

DESCRIPTION

UDI_DEQUEUE_HEAD removes the ***element*** at the head of the queue specified by ***listhead***, and returns it to the caller.

UDI_DEQUEUE_TAIL removes the ***element*** at the tail of the queue specified by ***listhead***, and returns it to the caller.

UDI_QUEUE_REMOVE removes ***element*** from its queue. With the exception that there is no return value, this macro is equivalent to the udi_dequeue 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.

These macros must be called as if they, respectively, had the following functional interfaces:

```
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 );
```

REFERENCES

udi_dequeue

NAME

UDI_FIRST/ LAST/ NEXT/ PREV_ELEMENT — *Get first/last/next/previous element in queue*

SYNOPSIS

```
#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)
```

ARGUMENTS

listhead is a pointer to a list head element.

element is a pointer to a queue element.

DESCRIPTION

UDI_FIRST_ELEMENT returns a pointer to the first element in the queue specified by ***listhead***.

UDI_LAST_ELEMENT returns a pointer to the last element in the queue specified by ***listhead***.

UDI_NEXT_ELEMENT returns a pointer to the next queue element immediately after ***element***.

UDI_PREV_ELEMENT returns a pointer to the queue element immediately preceding ***element***.

These macros must be called as if they, respectively, had the following functional interface:

```
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 );
```

REFERENCES

udi_queue_t

NAME

UDI_QUEUE_FOREACH *Safe mechanism to walk a queue*

SYNOPSIS

```
#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)))
```

ARGUMENTS

listhead is a pointer to a list head element.

element is a queue element pointer variable that may be uninitialized on entry, and is set successively to each element in the queue.

tmp is a queue element pointer variable for temporary storage in the loop.

DESCRIPTION

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.

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:

```
UDI_QUEUE_FOREACH (
          udi_queue_t *listhead,
          udi_queue_t *element,
          udi_queue_t *tmp );
```

EXAMPLES

The following code reverses the elements in a queue:

```
{
   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))
   }
}
```

REFERENCES

udi_queue_t

NAME

UDI_BASE_STRUCT *Find base of structure from pointer to member*

SYNOPSIS

```
#include <udi.h>

#define UDI_BASE_STRUCT( \
          memberp, struct_type, member_name) \
               ((struct_type *)((char *)(memberp) - \
                  offsetof(struct_type, member_name)))
```

ARGUMENTS

memberp is a pointer to a member of a structure.

struct_type is the ISO C data type of the structure.

member_name is the name of the member within the structure.

DESCRIPTION

`UDI_BASE_STRUCT` takes a pointer to a structure member, ***memberp***, and returns a point to the base of the structure.

This has general utility beyond queueing, but is particularly useful with the queueing utilities, when a `udi_queue_t` is embedded beyond the first member of a structure.

RETURN VALUE

This macro returns a pointer to the base of the structure, of type (***struct_type*** *).

Endianness Management Utility Functions *23*

23.1 Overview

This chapter defines endianness management utility functions.

23.2 Endianness 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 which is 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 characteristcs 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.

23.2.1 Rules for C Structure Definitions

As specified in Section 9.8, "Structures Requiring a Fixed Binary Representation," on page 9-11, 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 9.8 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.

23.2.1.1 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 23.2.2 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,
                              lun,\ dpo, fua, reladr) \
   {                                    \
       (cdb)->c10_opcode = opcode;   \
       (cdb)->c10_flags = ((lun)<<5) | (((dpo)<,4) | ((fua)<<3) | \
                           (reladr));      \
       UDI_MBSET(4, cdb, c10_lba, block_num);\
       UDI_MBSET(2, cdb, c10_len, data_len);\
       (cdb)->c10_ctrl = 0;         \
   }
```

23.2.2 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 9.8.

23.2.2.1 Bit-field Macros

The bit-field macros, `UDI_BFMASK`, `UDI_BFGET`, and `UDI_BFSET`, are used to perform basic operations on bit-fields within a `udi_ubit8_t` variable or value. `UDI_BFMASK` is used to create a bit mask (all 1's in the bit field, zeroes elsewhere); `UDI_BFGET` extracts a bit field; and `UDI_BFGET` deposits a value into a bit field.

NAME

UDI_BFMASK,
UDI_BFGET, UDI_BFSET *Bit-field helper macros*

SYNOPSIS

```
#include <udi.h>

#define UDI_BFMASK(p, len) \
     (((1U<<(len))-1) << (p))

#define UDI_BFGET(val, p, len) \
     (((udi_ubit8_t)(val) >> (p)) & ((1U<<(len))-1))

#define UDI_BFSET(val, p, len, dst) \
     ((dst) = ((dst) & ~UDI_BFMASK(p,len)) | \
               (((udi_ubit8_t)(val) << (p)) & \
                   UDI_BFMASK(p,len)))
```

ARGUMENTS

p is the bit position in the byte of the least significant bit in the bit field. The bit position p is 0 for the least significant bit in the byte, 7 for the most significant bit. $0 \le p \le 7$.

len is the size in bits of the bit field. $1 \le len \le 8-p$.

val is a `udi_ubit8_t` variable or value.

dst is a `udi_ubit8_t` variable into which a value will be deposited.

DESCRIPTION

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 $0 \le p \le 7$ and $1 \le len \le 8-p$.

`UDI_BFMASK` creates a ***p***, ***len*** bit mask containing all 1's in the corresponding bit field, zeroes elsewhere.

`UDI_BFGET` extracts an unsigned ***p***, ***len*** bit-field from ***val***.

`UDI_BFSET` deposits val into the ***p***, ***len*** bit-field in ***dst***.

These macros must be called as if they, respectively, had the following functional interfaces:

```
udi_ubit8_t UDI_BFMASK (
          udi_ubit8_t p,
          udi_ubit8_t len);

udi_ubit8_t UDI_BFGET (
          udi_ubit8_t val,
          udi_ubit8_t p,
          udi_ubit8_t len);
```

```
void UDI_BFSET (
          udi_ubit8_t val,
          udi_ubit8_t p,
          udi_ubit8_t len,
          udi_ubit8_t dst);
```

Note that UDI_BFSET modifies ***dst***, and ***dst*** must be an lvalue (assignable on the left side of an assignment statement).

23.2.2.2 Multi-Byte Macros

The multi-byte helper macros, `UDI_MBGET` and `UDI_MBSET` and their variants, are used to extract and deposit multi-byte quantities from a structure which has been constructed according to the byte-by-byte layout rules given in Section 23.2.1.1 on page 23-2.

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_MBSET` macro as follows:

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

NAME

UDI_MBGET, UDI_MBGET_2/3/4 *Multi-byte extract helper macros*

SYNOPSIS

```
#include <udi.h>

#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))
```

ARGUMENTS

N is the number of bytes in the corresponding multi-byte quantity. ***N*** must be 2, 3, or 4.

structp is a pointer to a structure that contains ***N*** single-byte (and byte-aligned) members whose names are ***field***0, ***field***1, ... ***field***n (n=***N***-1), which together represent an ***N***-byte quantity in the structure.

field is the base name of a sequence of members in ***structp***. The name of each member in the sequence is ***field*** followed by a decimal number in the range 0..***N***-1; this number represents the byte number in a multi-byte quantity, with byte 0 being the least significant byte.

DESCRIPTION

These macros are used to extract multi-byte quantities from a structure which has been constructed according to the byte-by-byte layout rules given in Section 23.2.1.1 on page 23-2. The structure is pointed to by the ***structp*** argument, and the multi-byte quantities are represented by a sequence of fields in the structure whose names are based on the ***field*** argument.

As described above, 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 (n=***N***-1), which together represent an ***N***-byte quantity in the structure. ***field***0 is the least significant byte; ***field***n is the most significant.

UDI_MBGET extracts an ***N***-byte quantity from the structure pointed to by ***structp***. UDI_MBGET_2, UDI_MBGET_3, and UDI_MBGET_4, extract 2, 3, and 4-byte quantities, respectively.

These macros don't translate well into functional interfaces, so no corresponding functional interfaces are given.

NAME

UDI_MBSET, UDI_MBSET_2/3/4 *Multi-byte deposit helper macros*

SYNOPSIS

```
#include <udi.h>

#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)
```

ARGUMENTS

N is the number of bytes in the corresponding multi-byte quantity.

structp is a pointer to a structure that contains ***N*** single-byte (and byte-aligned) members whose names are ***field***0, ***field***1, ... ***field***n (n=***N***-1), which together represent an ***N***-byte quantity in the structure.

field is the base name of a sequence of members in ***structp***. The name of each member in the sequence is ***field*** followed by a decimal number in the range 0..***N***-1; this number represents the byte number in a multi-byte quantity, with byte 0 being the least significant byte.

val is the value to deposit into the multi-byte quantity in the structure pointed to by ***structp*** and corresponding to ***field***.

DESCRIPTION

These macros are used to deposit a value into a multi-byte quantity in a structure which has been constructed according to the byte-by-byte layout rules given in Section 23.2.1.1 on page 23-2. The structure is pointed to by the ***structp*** argument, and the multi-byte quantities are represented by a sequence of fields in the structure whose names are based on the ***field*** argument.

As described above, 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 (n=***N***-1), which together represent an ***N***-byte quantity in the structure. ***field***0 is the least significant byte; ***field***n is the most significant.

UDI_MBSET deposits val into the an ***N***-byte quantity represented by ***field*** in the structure pointed to by ***structp***. UDI_MBSET_2, UDI_MBSET_3, and UDI_MBSET_4, deposit into 2, 3, and 4-byte quantities, respectively.

These macros don't translate well into functional interfaces, so no corresponding functional interfaces are given.

23.2.3 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 function `udi_endian_swap` provides for the endian conversion of greater than 32-bit integers; this function has a ***`rep_count`*** and ***`stride`*** parameter, allowing for multiple byte-swaps of a given size in a single call. This can be used to byte-swap each element in an array (as shown by the `UDI_ENDIAN_SWAP_ARRAY` macro), or to perform 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_SWAP_16/32** *Byte-swap 16 or 32-bit integers*

SYNOPSIS

```
#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) )
```

ARGUMENTS

data16 is a 16-bit data value.

data32 is a 32-bit data value.

DESCRIPTION

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 on page 23-13 for information on an endian utility that provides for larger than 32-bit endian translation and that allows for multiple byte-swaps in a single call.

These macros must be called as if they, respectively, had the following functional interface:

```
udi_ubit16_t UDI_ENDIAN_SWAP_16 (
     udi_ubit16_t data16 );

udi_ubit32_t UDI_ENDIAN_SWAP_32 (
     udi_ubit32_t data32 );
```

REFERENCES udi_endian_swap

NAME

udi_endian_swap *Byte-swap multiple data items*

SYNOPSIS

```
#include <udi.h>

void udi_endian_swap (
     const void *src,
     void *dst,
     udi_ubit8_t swap_size,
     udi_ubit8_t stride,
     udi_ubit16_t rep_count );
```

ARGUMENTS

src points to a memory area across which byte swapping will be performed, as defined below.

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***. ***src*** and ***dst*** may be the same, but if not, the memory areas must be non-overlapping.

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***.

stride is the number of bytes by which to increment ***src*** between repetitions.

rep_count is the number of ***swap_size*** byte-swaps to perform.

DESCRIPTION

The udi_endian_swap function generates ***rep_count*** byte-swapping copies from ***src*** to ***dst***, of ***swap_size*** bytes each. Between each repetition ***src*** is incremented by ***stride*** to obtain the next element to be swapped. Only the memory actually byte-swapped into ***dst*** will be changed as a result of this call.

If ***swap_size*** is 1, and ***src*** is equal to ***dst***, this function acts as a no-op. If ***swap_size*** is 1, and ***src*** is not equal to ***dst***, this function performs a copy.

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 (e.g. &array[0]->my_field), ***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_ARRAY

NAME **UDI_ENDIAN_SWAP_ARRAY** *Byte-swap each element in an array*

SYNOPSIS

```
#include <udi.h>

#define \
   UDI_ENDIAN_SWAP_ARRAY(src, element_size, count) \
         udi_endian_swap(src, src, element_size, \
                         element_size, count)
```

ARGUMENTS

src points to an array of ***count*** elements, each of which are ***element_size*** bytes in size.

element_size is the size, in bytes, of each element to be byte-swapped. ***element_size*** must be a power-of-two.

count the number of ***elements*** in the ***src*** array to be byte-swapped.

DESCRIPTION

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.

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:

```
void UDI_ENDIAN_SWAP_ARRAY (
     void *src,
     udi_ubit8_t element_size,
     udi_ubit16_t count );
```

REFERENCES

udi_endian_swap

UDI Core Specification

Section 5: Core Metalanguages

Introduction to UDI Metalanguages *24*

24.1 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 portability a custom metalanguage must be implemented using the interfaces defined in Section 4: MEI Services.

Two 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; 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 Bus 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.

24.2 Standard Metalanguage Functions and Parameters

See Chapter 7, *"Calling Sequence and Naming Conventions"* for requirements and conventions on metalanguage interfaces and their names.

In addition to the required channel operation functions, metalanguages may also provide *proxy* routines for events that will never occur for some drivers, or for operations that require no action by the driver. For example, if a driver has no children to enumerate, it may set its `udi_enumerate_req_op_t` to `udi_enumerate_req_unused` instead of providing its own routine that always just responds immediately with `udi_enumerate_ack`.

24.3 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 24-1 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
`_rdy`	Ready for Event	Operation submitted “against the flow” to prime the initiator with a set of control blocks for it to use, in order to have flow control.

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

24.4 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.
- Not supported: 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 correct state for such a request.
- Mistaken identity: the driver understands the operation, but its parameters are out of range for the associated object.

For all but the normal case, the driver must respond with an appropriate status code and also log the error using `udi_log_write`. (See udi_log_write on page 18-7.)

24.4.1 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.

24.4.2 Operations That Are Not Understood

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

24.4.3 Operations That Are Not Supported

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

24.4.4 Operations Received In An Invalid State

When a driver receives an operation that is understood and implemented, but is not valid in the driver's current state with respect to the sequence rules of the metalanguage, the driver must reply back with a status value of `UDI_STAT_INVALID_STATE`.

24.4.5 Operations With Mistaken Identity

When a driver receives an operation that is understood, implemented, and received in the correct state, but with a range of values not appropriate for the corresponding device or other object, the driver must reply back with a status value of `UDI_STAT_MISTAKEN_IDENTITY`.

Management Metalanguage 25

25.1 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 uses the Management Metalanguage to enumerate device presence or loss, as well as to manage changes in state for hot plugging and power management. The MA may also participate in cleanup operations by indicating that various channels are to be closed via indirect requests or direct operations.

All UDI drivers must support the Management Metalanguage.

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.

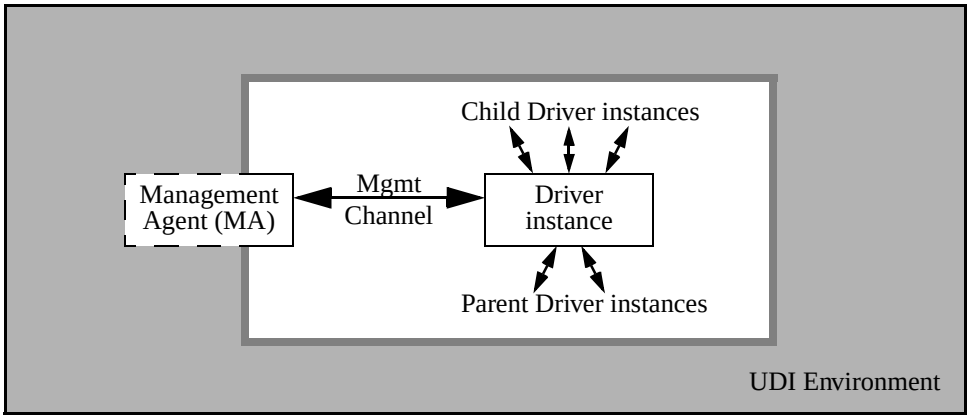
Management Metalanguage operations are not abortable with `udi_channel_op_abort`.

25.2 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.



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. Asynchronous event indication in the Management Metalanguage is handled by *posting* a request to the driver; that is, the MA sends a request to the driver asking for future event notification, and the driver holds on to the control block indefinitely, sending a reply only when the (first such) asynchronous event occurs.

25.2.1 Driver Instantiation

The MA is responsible for controlling the creation of new driver instances, and determining when to do so. The MA combines information obtained from the enumeration responses of a parent driver instance with the information provided by candidate child drivers' static driver properties to make this decision. The environment then uses information from the selected child driver's `udi_init_info` to set up appropriate linkages between the driver and the rest of the system. The process of creating new driver instances is referred to as *driver instantiation.*

The following model describes the typical sequence of events surrounding the instantiation of a driver instance. 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:

0. ... the parent driver instance has previously been instantiated and has a management channel established between its primary region and the MA.

1. The MA issues an *enumeration request* (`udi_enumerate_req`) to the parent driver instance.

2. The parent driver returns an *enumeration response* (`udi_enumerate_ack`) with information describing an instance of a "*child*"[1] to the MA.

3. The MA locates a driver that corresponds to the instance attribute information provided by the parent in the enumeration response.

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.

4. 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 `udi_init_info` structure will be processed.

5. 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, *internal bind channels* connecting secondary regions to the primary region, and a *bind channel* between the parent and child driver instances to be used for initial parent/child communications. All of these channels are pre-anchored. The MA also creates internal configuration information and prepares any necessary resources.

6. The MA issues a *usage indication* (`udi_usage_ind`) to the newly created driver to allow it to perform preliminary initialization of internal data structures and adjust resource usage and trace levels. This is guaranteed to be the first operation the driver instance receives on any of its channels. Further, no additional operations will be received until the driver calls `udi_usage_res` for this first operation.

7. If the driver has requested any static secondary regions or any dynamic regions for this binding, it will receive a UDI_CHANNEL_BOUND event indication (see **udi_channel_event_cb_t** on page 17-10) on one end of the internal bind channel for each such region. No external bind operations or additional management operations will be delivered until the driver has processed all the internal bind events and called `udi_channel_event_complete` for each one.

8. The MA then begins the child/parent bind sequence by generating `UDI_CHANNEL_BOUND` channel event indication on the new child's end of the bind channel to its parent. This event may be delivered to the primary region or to a secondary region, depending on the region index value in the corresponding "parent_bind_ops" declaration (see Chapter 31, *"Static Driver Properties"*).

9. The child now performs internal initialization, examining the enumeration and configuration attributes of its device if necessary (using `udi_instance_attr_get`), and then issues a metalanguage-specific bind request to the parent instance via the bind channel.

10. The parent instance processes the child's bind request, propagates constraints to the child (using `udi_constraints_propagate`) over the bind channel and then returns a bind response via the same bind channel.

11. The child completes initialization and then issues a `udi_channel_event_complete` corresponding to the parent bind event indication, to let the MA know that the parent/child initialization process has completed (successfully or unsuccessfully). On receipt of a successful completion notification, the parent and child are considered "open for business" and should expect to perform normal activities.

12. The MA may then repeat this sequence with the new child taking on the parent role, and issue an enumeration request to this new child instance to determine if it has children of its own. If this driver has no children of its own, the enumeration response to the MA indicates this fact and the configuration of this driver instance is complete.

A multi-parent (multiplexer) driver may receive additional parent bindings after the above sequence has completed. In this case, the bind sequence begins with Step 7.

25.3 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.
- Management control blocks must always be sent back to the channel over which they were received, since there is only one management channel per driver. Therefore, drivers must not modify or even reference the ***channel*** member of a management control block.
- 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 ***<<meta>>_<<role>>*_OPS_NUM** for management operations since the single management ops vector is supplied via `udi_primary_init_t` (see page 10-5).
- There are no ***<<meta>>_<<cbgroup>>*_CB_NUM** values 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).
- The ***mgmt_scratch_size*** value in `udi_primary_init_t` 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, no constraints will be propagated across the management channel, and the MA will never be terminated.

25.4 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.

25.4.1 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.

25.4.2 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 *Management Meta channel ops vector*

SYNOPSIS

```
#include <udi.h>

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

DESCRIPTION

`udi_mgmt_ops_t` is an *ops vector* structure that contains function pointers for all of the Management Metalanguage entry point routines for the driver. It is used in the driver's `udi_init_info` to register these entry points with the environment. For additional information on ops vectors, see "Channel Operations Vectors" on page 7-6.

For the `udi_usage_ind` and `udi_enumerate_req` entry points, the driver can specify corresponding environment-provided proxy functions. See the corresponding reference pages for information on when these proxy functions may be used.

Note – The `udi_mgmt_ops_t` deviates from other channel ops vectors, in that it does not have a `udi_channel_event_ind` operation.

REFERENCES

`udi_init_info, udi_primary_init_t`

NAME **udi_mgmt_cb_t** *Common Management Control Block*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
} udi_mgmt_cb_t;
```

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.

Scratch space size for all Management Metalanguage control blocks is determined by the setting of ***mgmt_scratch_requirement*** in the driver's `udi_primary_init_t` in its `udi_init_info`. Management Metalanguage control blocks are allocated only by the MA, not by drivers.

REFERENCES `udi_cb_t, udi_primary_init_t, udi_init_info`

NAME

udi_usage_cb_t *Resource indication and trace level control block*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
     udi_trevent_t trace_mask;
     udi_index_t meta_idx;
} udi_usage_cb_t;
```

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.

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.

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 *UDI Tracing and Logging* Chapter.

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 "meta" declaration of the driver's static driver properties (see Chapter 31), or be zero for the Management Metalanguage.

DESCRIPTION

This control block is used with the udi_usage_ind and udi_usage_res operations to indicate the types of events to be traced by the driver and the resource level to which it should comply.

Scratch space size for all Management Metalanguage control blocks is determined by the setting of ***mgmt_scratch_requirement*** in the driver's udi_primary_init_t in its udi_init_info. Management Metalanguage control blocks are allocated only by the MA, not by drivers.

REFERENCES

udi_cb_t, udi_primary_init_t, udi_init_info, udi_trevent_t, udi_usage_ind, udi_usage_res

NAME

udi_usage_ind *Indicate desired resource usage and trace levels*

SYNOPSIS

```
#include <udi.h>

void udi_usage_ind (
     udi_usage_cb_t *cb,
     udi_ubit8_t resource_level );

/* Values for resource_level */
#define UDI_RESOURCES_CRITICAL          1
#define UDI_RESOURCES_LOW               2
#define UDI_RESOURCES_NORMAL            3
#define UDI_RESOURCES_PLENTIFUL         4
```

ARGUMENTS

cb is a pointer to a Management Metalanguage usage control block.

resource_level is an indication of the current resource level to which the driver should adhere.

TARGET CHANNEL

The affected driver instance's management channel.

PROXIES

udi_static_usage *Proxy for udi_usage_ind*

```
udi_usage_ind_op_t udi_static_usage;
```

Drivers that do not adjust their resource utilization levels and do not support tracing may specify `udi_static_usage` as the entry point for this operation.

DESCRIPTION

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.

This is the first operation that will be issued to a newly instantiated driver instance. 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.

When used as the driver's first operation, the environment guarantees that no additional channel operations will be delivered to the driver instance on any channel until the driver calls `udi_usage_res` for this first operation.

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`.

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.

Resource levels that may be indicated by the ***resource_level*** argument are:

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.

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.

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.

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.

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.

WARNING

Drivers must not invoke this operation.

Resource levels and corresponding low and plentiful thresholds are somewhat arbitrary and may be set or modified by the UDI environment via automatic means or by the system administrator. Driver notification is purely optional.

REFERENCES

`udi_usage_cb_t, udi_usage_res`

NAME

udi_usage_res *Resource usage and trace level response operation*

SYNOPSIS

```
#include <udi.h>

void udi_usage_res (
     udi_usage_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a Management Metalanguage usage control block.

TARGET CHANNEL

The affected driver instance's management channel.

DESCRIPTION

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.

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. In particular, if tracing is entirely unsupported, ***trace_mask*** must be set to zero.

This operation is also used to acknowledge the receipt of the resource indication; it does *not* indicate to the Management Agent that the driver has completed any internal resource adjustements as a result of the resource indication, but merely indicates that the driver is aware of the new resource levels.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_usage_ind` operation.

REFERENCES

`udi_usage_cb_t, udi_usage_ind`

25.5 Enumeration Operations

This subsection of the Management Metalanguage defines child enumeration operations. As described in "Driver Instantiation" on page 25-2, each driver instance is given the opportunity to enumerate the presence, number, and initial instance attributes of any child devices (actual or pseudo) associated with the current driver instance. The typical response of the MA to an enumerated child device is to create one or more driver instances and initiate a bind operation between each child instance and the current driver instance (which becomes the parent of that new child).

This process begins with the MA issuing a sequence of enumeration requests to the driver, to instruct it to respond with information regarding each child device present. The enumeration request and acknowledgement operation, along with associated data objects, are described in this subsection.

25.5.1 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. If multiple child driver instances are associated with the same device instance, the same enumerated attributes will be used for all of those instances.

25.5.2 Child ID

Each child device instance is also identified by a *child ID*. The child ID is a value that uniquely referss to a given child device instance with respect to its parent. This allows communications between the environment and the parent driver to refers to specific child devices.

When a new child instance is enumerated, the parent driver specifies the child ID, the enumeration attributes for the child device, and a channel ops index indicating the metalanguage and entry points to use for the bind channel between this child and its parent. If the parent driver supports multiple metalanguages for a given child device, it can enumerate the same ***child_ID*** multiple times with different ops index values. The ***child_ID*** value must not be reused for a new device until the MA tells the parent to release it.

25.5.3 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.

25.5.4 Parent ID

When a driver instance is bound to (one of) its parent(s), the MA provides a parent ID, which is a number that can later be used to identify this particular parent, relative to the target driver instance. The same parent ID value is passed back to the driver to request it to unbind from a specific parent. Drivers that don’t support multiple parents can ignore the parent ID.

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 ID*. In this case, the *parent ID* 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 ID* 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 ID will normally be used during instance unbinding and hardware hot plug scenarios. It is used to determine the portion of the topology tree that will be unbound or affected by the hot plugging of a component or set of components.

Drivers that have only a single parent per instance have a direct relationship between the loss of a parent binding and the ability to service I/O requests from their children: all children will be affected. As such, the MA does not need to query such a driver to enumerate its affected children. On the other hand, if a driver has multiple parents, the MA cannot know the relationship between that driver’s parents and its children, so it will query the driver instance for the set of affected child devices.

25.5.5 Dynamic Enumeration (Hot Plug)

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 instance is only expected to maintain one posted enumeration request from the MA per ***parent_ID***. If the MA issues another enumeration request the enumeration request being held must be acknowledged with a "no child events" status and the new enumeration request must 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.

25.5.6 Unenumeration

The converse to the enumeration operation is the case where a child has "gone away" from the perspective of the parent driver, such as when a device is unplugged or turned off. In UDI, this is handled by an *unenumeration* operation. This operation is performed as a variation of the enumeration operation: the child ID 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.

25.5.7 Directed Enumeration

The children of a particular driver instance 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, and 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_filter_element_t** *Enumeration filter element structure*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     char attr_name[UDI_MAX_ATTR_NAMELEN];
     udi_ubit8_t attr_min[UDI_MAX_ATTR_SIZE];
     udi_ubit8_t attr_min_len;
     udi_ubit8_t attr_max[UDI_MAX_ATTR_SIZE];
     udi_ubit8_t attr_max_len;
     udi_instance_attr_type_t attr_type;
     udi_ubit32_t attr_stride;
} udi_filter_element_t;
```

MEMBERS

attr_name is the name of the attribute to be filtered.

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.

attr_min_len specifies the valid length (in bytes) of the ***attr_min*** value. Must not be zero.

attr_max is the maximum acceptable value for the attribute in this filter.

attr_max_len specifies the valid length (in bytes) of the ***attr_max*** value. Must not be zero.

attr_type is the attribute type as specified for **udi_instance_attr_type_t** on page 16-7. Must not be UDI_ATTR_NONE or UDI_ATTR_FILE.

attr_stride specifies the periodicity of the filter match values starting at ***attr_min*** and ending at or above ***attr_max***.

DESCRIPTION

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. This can be used to reduce the amount of work a driver needs to go through to scan for child devices.

The interpretation of ***attr_stride*** is unique to each ***attr_name*** attribute and is specified by the metalanguage when describing that attribute.

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.

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_GET (page 16-14) must be used to extract values from ***attr_min*** and ***attr_max***.

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 (as initialized by the MA):

```
udi_filter_element_t f = {
     "scsi_target", UDI_ATTR32_INIT(2), 1,
                    UDI_ATTR32_INIT(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_ATTR32_GET, UDI_ATTR32_INIT, udi_enumerate_req`

NAME **udi_enumerate_cb_t** *Enumeration operation control block*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
     udi_ubit32_t child_ID;
     void *child_data;
     udi_instance_attr_list_t *attr_list;
     udi_ubit8_t attr_valid_length;
     const udi_filter_element_t *filter_list;
     udi_ubit8_t filter_list_length;
     udi_ubit8_t parent_ID;
} udi_enumerate_cb_t;

/* Special parent_ID filter values */
#define UDI_ANY_PARENT_ID                0
```

MEMBERS

gcb is the standard control block information structure.

parent_ID is the parent ID that identifies a specific parent of the current driver instance. When this value is not UDI_ANY_PARENT_ID then the current driver must only enumerate children that relate to this indicated parent. This is used most often for multiplexer drivers which have multiple parents and children and an enumeration event needs to determine which children will be affected by a change in a parent's status.

The ***parent_ID*** value will match a value previously passed to the driver via a udi_channel_event_ind operation of type UDI_CHANNEL_BOUND. Drivers must not change ***parent_ID***.

child_ID is the child ID that identifies a specific child instance. When the driver enumerates a new child device, it assigns a unique ***child_ID*** to identify that device in subsequent operation.

The ***child_ID*** field is only valid in certain enumeration operations:

1. The ***child_ID*** field is valid in the udi_enumerate_req operation only for the UDI_ENUMERATE_RELEASE enumeration level.

2. The ***child_ID*** field is valid in the udi_enumerate_ack operation only for the UDI_ENUMERATE_OK and UDI_ENUMERATE_REMOVED enumeration levels.

The ***child_ID*** value is unspecified and must be ignored in all other situations.

child_data is a pointer to pre-allocated movable memory for per-child data, if any. If ***child_data_size*** in the driver's udi_primary_init_t is non-zero, that many bytes of

memory will be allocated, as if by udi_mem_alloc with the UDI_MEM_NOZERO and UDI_MEM_MOVABLE flags set, for each call to udi_enumerate_req with any enumeration level except UDI_ENUMERATE_RELEASE. Otherwise, ***child_data*** will be NULL.

If the driver responds in udi_enumerate_ack with any result besides UDI_ENUMERATE_OK, it must leave ***child_data*** unmodified and the environment will free the memory automatically. Otherwise, the driver must free the ***child_data*** using udi_mem_free when it is through with it.

attr_list is a pointer to an array of udi_instance_attr_list_t structures (see page 16-13), allocated and owned by the MA. This attribute list is used to describe the initial set of instance attributes being enumerated by this driver or the desired set of child attributes for a directed enumeration operation. The length of this array shall be equal to the value of ***enumeration_attr_list_length*** from the driver's udi_primary_init_t.

attr_valid_length indicates the number of valid attribute values currently stored in the control block's ***attr_list***. This is initialized by the MA to zero, or the number of attributes specified by the MA for directed enumeration. Attribute list elements beyond the first ***attr_valid_length*** elements are ignored and their values are unspecified.

filter_list is a pointer to an array of udi_filter_element_t structures, allocated and owned by the MA, used to specify the attributes filter to be applied to this enumeration request by the target driver.

filter_list_length is the number of elements in the ***filter_list*** array.

DESCRIPTION

The udi_enumerate_cb_t is the control block used for the udi_enumerate_req and udi_enumerate_ack channel operations. This control block is allocated by the MA when the MA issues the udi_enumerate_req and is to be returned to the MA by the driver in the udi_enumerate_ack response.

The ***filter_list*** passed to the driver in this control block on the enumeration request must be returned to the MA in the acknowledgement operation and must not be used after that acknowledgement has been issued.

The driver fills in the attribute list passed in this control block with the attributes of the child node for the udi_enumerate_ack 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.

All attribute names in both ***attr_list*** and ***filter_list*** must be enumeration attribute names and so must not have any special prefix characters.

Scratch space size for all Management Metalanguage control blocks is determined by the setting of ***mgmt_scratch_requirement*** in the driver's `udi_primary_init_t` in its `udi_init_info`. Management Metalanguage control blocks are allocated only by the MA, not by drivers.

WARNINGS

Drivers must not use pointer values as child_ID values, since pointer values are larger than 32 bits on some platforms.

REFERENCES

`udi_cb_t, udi_instance_attr_list_t, udi_primary_init_t, udi_init_info, udi_enumerate_req, udi_enumerate_ack, udi_ops_init_t, udi_child_chan_context_t`

NAME

udi_enumerate_req *Request information regarding a child instance*

SYNOPSIS

```
#include <udi.h>

void udi_enumerate_req (
     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
#define UDI_ENUMERATE_RELEASE            6
```

ARGUMENTS

cb is a pointer to an enumeration control block.

enumeration_level is a value describing the relationship of this enumeration request to previous enumeration requests (see below).

TARGET CHANNEL

The affected driver instance's management channel.

PROXIES

udi_enumerate_no_children *Proxy for udi_enumerate_req*

```
udi_enumerate_req_op_t udi_enumerate_no_children;
```

`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 devices. It will simply acknowledge the request with `UDI_ENUMERATE_LEAF`, to indicate that there are and never will be any children of this device.

DESCRIPTION

The Management Agent issues this request to obtain enumeration information about child devices of the current driver instance's device. If there is information that may be returned for child devices, the receiving driver fills in an array of `udi_instance_attr_list_t` structures to describe that child device.

The ***enumeration_level*** argument indicates the relationship of this enumeration request to any previous enumeration requests based on the following values:

`UDI_ENUMERATE_START` - The Management Agent is starting a new enumeration cycle. The target driver must provide information about at least all child devices that match the filter, 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 devices; receipt of another UDI_ENUMERATE_START will restart the enumeration back at the beginning.

UDI_ENUMERATE_START_RESCAN - This enumeration level is the same as the UDI_ENUMERATE_START enumeration level except that with this level the driver must not use any previously obtained or cached information to report child devices and nust instead perform physical verification as appropriate to obtain the filtered list of children. The physical scan must only be sufficient to satisfy the specified filter; an exhaustive physical scan is not necessary unless indicated by the filter (or lack thereof).

UDI_ENUMERATE_NEXT - The target driver shall return information about the next child device relative to the one described in the previous successful udi_enumerate_ack operation with a UDI_ENUMERATE_START, UDI_ENUMERATE_RESCAN, or UDI_ENUMERATE_NEW enumeration level that was relative to the same parent ID. When all children have been enumerated the udi_enumerate_ack operation shall indicate an enumeration_flag of UDI_ENUMERATE_DONE. The driver is expected to maintain the context (in its region data and, if a multiple-parent driver instance, in its parent context data) necessary 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 shall return information about any child device 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 must 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.

UDI_ENUMERATE_RELEASE - This enumeration level is an indication to the target driver that it should release resources associate with the indicated child (as specified by the ***child_ID*** member of the control block). This is used when the Management Agent has terminated all child instances for this ***child_ID*** and is no longer

planning to use this child device. This may occur whether or not the target driver has unenumerated the device instance (with UDI_ENUMERATE_REMOVED). This signals the current driver that the ***child_ID*** value for the indicated child and any associated context information may now be deallocated or re-used for future enumerations.

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 devices, it will be treated as if it were a UDI_ENUMERATE_NEXT.

UDI_ENUMERATE_DIRECTED and UDI_ENUMERATE_RELEASE may be invoked independently of other enumeration sequences. They do not affect the behavior of UDI_ENUMERATE_NEXT.

If a previous UDI_ENUMERATE_NEW for the same ***parent_ID*** is still pending, the driver must call udi_enumerate_ack with UDI_ENUMERATE_FAILED to "cancel" the previous request and then process the new enumeration request.

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_ID*** in the enumeration control block is notUDI_ANY_PARENT_ID, then it identifies thespecific parent of the current driver instance with respext to which the enumeration is to be performed.

WARNING

Drivers must not invoke this operation.

REFERENCES

udi_enumerate_ack, udi_instance_attr_list_t, udi_enumerate_cb_t

NAME

udi_enumerate_ack *Provide child instance information*

SYNOPSIS

```
#include <udi.h>

void udi_enumerate_ack (
     udi_enumerate_cb_t *cb,
     udi_ubit8_t enumeration_result,
     udi_index_t ops_idx );

/* Values for enumeration_result */
#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_REMOVED_SELF      5
#define UDI_ENUMERATE_FAILED            255
```

ARGUMENTS

cb is a pointer to an enumeration control block.

enumeration_result is a value indicating what type of enumeration acknowledgement is being generated.

ops_idx indicates (one of) the child binding ops index(es)—with associated metalanguage(s)—useable with this child device. If a driver calls `udi_enumerate_ack` multiple times with the same ***child_ID*** then each must have a different ***ops_idx***, corresponding to a different ***meta_idx***, and the environment may bind child drivers using any or all corresponding metalanguages. If ***enumeration_result*** is not `UDI_ENUMERATE_OK`, ***ops_idx*** is ignored and may be set to any value.

TARGET CHANNEL

The affected driver instance's management channel.

DESCRIPTION

The `udi_enumerate_ack` channel operation is used in response to a `udi_enumerate_req` channel operation and provides information about a particular child device instance.

The ***attr_list*** in the control block specifies the volatile enumeration instance attributes that will be present for corresponding child driver instance(s) when created. These enumeration attributes may be obtained by a child driver via calls to `udi_instance_attr_get`, to obtain additional information about the instance being created. All child devices that match the `filter_list` in the enumeration request control block must be included in an enumerate acknowledgement. Other children may also be included.

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_ID*** and ***attr_list*** values in the control block should be interpreted:

UDI_ENUMERATE_OK - Indicates that the enumeration is returning a valid child instance enumeration and that no special cases apply. The ***attr_valid_length*** value is adjusted accordingly and the ***child_ID*** field is filled in by the driver with a value that uniquely identifies this child. This value will be used to identify the bind channel to that child in the udi_child_chan_context_t structure (if the udi_ops_init_t ***chan_context_size*** is non-zero).

UDI_ENUMERATE_LEAF - Indicates that there are and never will be any child devices for the current device.

UDI_ENUMERATE_DONE - Indicates that all children have already been enumerated and no more child devicesexist. This response does not actually enumerate a child.

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 set of child devices, where the set has changed part way through the enumeration process, or when the target driver is unable to determine the exact change that has occurred.

This response does not actually enumerate a child.

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

UDI_ENUMERATE_REMOVED - Indicates that the specified child device has been removed (as opposed to added) and that the Management Agent should initiate handling of a device removal. The ***child_ID*** indicates which child has been removed by passing the same value that the child was originally enumerated with.

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_ID*** associated with this child until the Management Agent acknowledges the unenumeration with a udi_enumerate_req operation with an enumeration level of UDI_ENUMERATE_RELEASE.

UDI_ENUMERATE_REMOVED_SELF - Treated like UDI_ENUMERATE_REMOVED, except that this indicates that the enumerating device itself has been removed.

UDI_ENUMERATE_FAILED - Indicates that the dynamic or directed enumeration cannot be satisfied by this target driver. This may only be returned for UDI_ENUMERATE_NEW and 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*** member of the control block must always be zero except for the UDI_ENUMERATE_OK case, where it indicates the number of child enumeration attributes. In all cases except UDI_ENUMERATE_OK and UDI_ENUMERATE_REMOVED, the ***child_ID*** value is ignored.

Table 25-1 ***enumeration_result*** value usage

enumeration_result UDI_ENUMERATE_xxx	valid for	child_ID	attr_valid_length
OK	all	new child ID	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	0
REMOVED_SELF	NEW	ignored	0
FAILED	NEW, DIRECTED	ignored	0

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding udi_enumerate_req operation.

REFERENCES

udi_enumerate_req, udi_instance_attr_list_t, udi_enumerate_cb_t, udi_primary_init_t

25.6 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 plug 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 plug 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 plug operation to the driver instance's children, leaving them unaffected.

The following model describes the typical sequence of events surrounding a hot plug 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 plug 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 plug 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 plug 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 plug 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, after unbinding all parents and children, the MA will invoke `udi_final_cleanup_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 re-binding to each of the children. If the children were not unbound, the MA will issue a *Resume* operation to the driver instance.

25.6.1 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 plug 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.

25.6.2 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). The instance is to also take whatever actions necessary to prevent the delivery of unsolicited inbound requests. This may involve disabling the reception of new packets, disabling interrupts, exerting flow control, etc.

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.

25.6.3 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 should be terminated. The *Shutdown* operation is commonly used when no further device activity is desired but the device itself will remain powered on (e.g. when the operating system is to be rebooted).

Unlinke Suspend, Shutdown will not be followed by a Resume, but may be followed by an Unbind.

25.6.4 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 plug 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.

25.6.5 Resume

This management operation is used by the MA to resume normal operation after *Prepare To Suspend*, *Suspend*, *Shutdown*, or *Parent Suspended*. It may be issued when the MA encounters a scenario in which the MA needs to abort the hot plug 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. The MA may choose to abort the hot plug operation or continue with a non-transparent Suspend/Resume.

25.6.6 Abrupt Unbind

Instead of going through the normal device management unbind scenario, the MA may sometimes need to abruptly unbind a driver instance. This may happen as a result of an abrupt hot removal of a device (i.e. removing a device without informing the operating system). It may also happen as a result of “region-kill” as a result of a driver software failure. In either case, the event will propagate to neighboring driver instances as `udi_channel_event_ind` operations of type `UDI_CHANNEL_CLOSED`.

NAME

udi_devmgmt_req *Device Management request*

SYNOPSIS

```
#include <udi.h>

void udi_devmgmt_req (
     udi_mgmt_cb_t *cb,
     udi_ubit8_t mgmt_op,
     udi_ubit8_t parent_ID );

/* Values for mgmt_op */
#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
#define UDI_DMGMT_UNBIND                  6
```

ARGUMENTS

cb is a pointer to a miscellaneous Management Metalanguage control block.

mgmt_op is a value that selects the operation type.

parent_ID is the parent ID that indicates the parent for which the operation is to take place. This will match the value originally supplied by the MA when the parent was bound to the current driver via the `udi_channel_event_ind` operation of type `UDI_CHANNEL_BOUND`.

TARGET CHANNEL

The Management Agent's management channel to the parent driver.

DESCRIPTION

The Management Agent issues this request to manage I/O transfers within a driver instance during hot plug operations.

The ***mgmt_op*** argument must be one of the following values and indicates the type of management operation being requested:

UDI_DMGMT_PREPARE_TO_SUSPEND - Indicates that a Suspend operation is about to take place relative to the indicated parent.

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 shut down.

UDI_DMGMT_SHUTDOWN - Treated as `UDI_DMGMT_SUSPEND`, with the addition that the device must be completely shut down (in particular, all communications connections should be terminated).

UDI_DMGMT_PARENT_SUSPENDED - Indicates that outbound traffic via the indicated parent has been suspended.

UDI_DMGMT_RESUME - Indicates that the instance is to cancel any suspended or throttled state and is to resume full operation. I/O shall resume onto the then-active set of parents; if a multi-parent driver has parent-specific routing requirements, it must compare ***parent_ID*** against the set of currently-bound parents and fail if that parent is no longer (re-)bound.

UDI_DMGMT_UNBIND - Indicates that the driver must unbind from the indicated parent. The driver must first complete a metalanguage-specific unbind sequence with its parent and free resources related to that parent (it may choose to defer freeing some resources until it receives a `udi_final_cleanup_req`). 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. Communications connections should be terminated. Storage device write-back caches should be flushed to permanent storage, for example. When the unbinding is complete (and not before), the driver must respond to the `UDI_DMGMT_UNBIND` request with a corresponding `udi_devmgmt_ack`.

WARNING

Drivers must not invoke this operation.

REFERENCES

`udi_devmgmt_ack, udi_mgmt_cb_t`

NAME

udi_devmgmt_ack *Acknowledge a device management request*

SYNOPSIS

```
#include <udi.h>

void udi_devmgmt_ack {
     udi_mgmt_cb_t *cb,
     udi_ubit8_t flags,
     udi_status_t status }

/* Values for flags */
#define UDI_DMGMT_NONTRANSPARENT          (1U<<0)

/* Meta-Specific Status Codes */
#define UDI_DMGMT_STAT_ROUTING_CHANGE
                          (UDI_STAT_META_SPECIFIC|1)
```

ARGUMENTS

cb is a pointer to a miscellaneous Management Metalanguage control block.

status indicates the success or failure of the operation.

TARGET CHANNEL

The parent driver's primary region management channel.

DESCRIPTION

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.

The ***flags*** argument may include:

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.

STATUS VALUES

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.

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.

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 support multiple parents need not check for this condition and must not use this status code.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_devmgmt_req` operation.

REFERENCES

`udi_devmgmt_req, udi_devmgmt_cb_t`

NAME **udi_final_cleanup_req** *Release final resources prior to instance unload*

SYNOPSIS

```
#include <udi.h>

void udi_final_cleanup_req (
     udi_mgmt_cb_t *cb);
```

ARGUMENTS

cb is a pointer to a miscellaneous Management Metalanguage control block.

TARGET CHANNEL The affected driver instance's management channel

DESCRIPTION

The MA issues this operation to request that the driver fully remove all resources and region instance context. The MA will only 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, including closing any channels that the driver explicitly spawned.

Upon completion of this request, the driver must perform a `udi_final_cleanup_ack` operation, passing it the same control block as was passed to `udi_final_cleanup_req`. After sending the ack, it should logically appear as if the driver instance had not appeared in the system.

WARNING Drivers must not invoke this operation.

REFERENCES `udi_final_cleanup_ack`

NAME **udi_final_cleanup_ack** *Acknowledge completion of a final cleanup request*

SYNOPSIS

```
#include <udi.h>

void udi_final_cleanup_ack (
     udi_mgmt_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a miscellaneous Management Metalanguage control block.

TARGET CHANNEL The affected driver instance's management channel

DESCRIPTION Handshakes a `udi_final_cleanup_req` operation. This indicates to the MA that the removal operation has completed.

The driver must free all of its resources before calling `udi_final_cleanup_ack`, regardless of its current state.

WARNINGS The control block must be the same control block as passed to the driver in the corresponding `udi_final_cleanup_req` operation.

REFERENCES `udi_final_cleanup_req`

25.7 Metalanguage-Specific Trace Events

The following defines the rules and conventions in the Management Metalanguage for the use of the metalanguage-selectable trace events (see the “Metalanguage Trace Events” in the 17-3 of the UDI Core Specification).

- `UDI_TREVENT_IO_SCHEDULED`, `UDI_TREVENT_IO_COMPLETED`
  - These trace events are not applicable to the Management Metalanguage.
- `UDI_TREVENT_META_SPECIFIC_1`
  - This trace event is used to track the start and completion of scans for child devices (regardless of whether this causes any enumeration/denumeration operations with the management agent or whether it was triggered by a `udi_enumerate_req`). The driver may post trace events indicating the existence or non-existence of hardware as determined by the internal scan operations.
- `UDI_TREVENT_META_SPECIFIC_2` `UDI_TREVENT_META_SPECIFIC_3`, `UDI_TREVENT_META_SPECIFIC_4`, `UDI_TREVENT_META_SPECIFIC_5`
  - Reserved for future use.

Note – All returned status values other than `UDI_OK` that indicate exceptional conditions must be logged and, when enabled, may also be traced, even if such events are expected.

25.8 Management Metalanguage States

The following states, along with the state diagram shown in Figure 25-1, 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 that 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 25-2 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 that are not listed in the state diagram do not cause state changes to occur.

Figure 25-1 Management Metalanguage State Diagrams

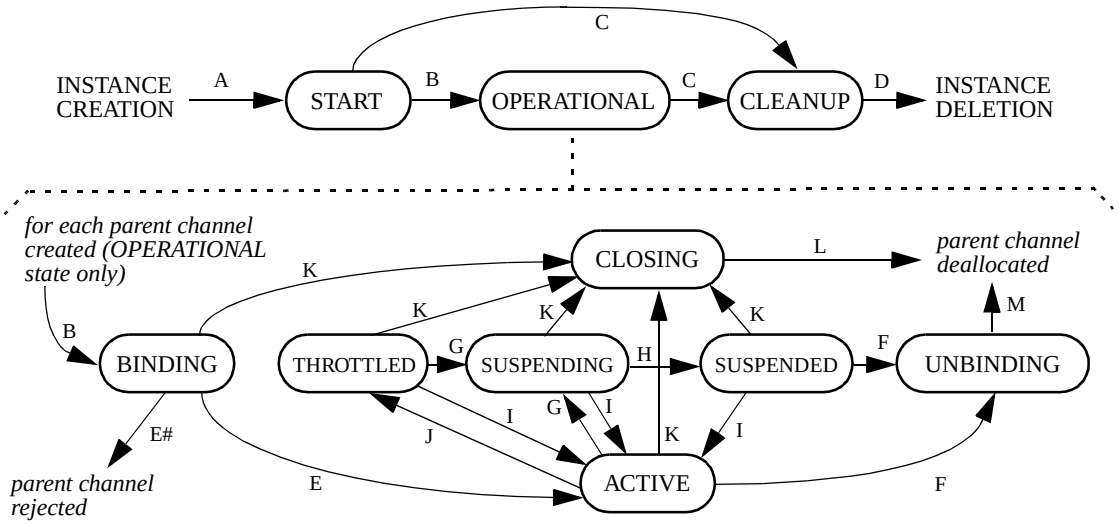


Table 25-2 Management Metalanguage Events

Event	Operation
A	`udi_usage_ind`
B	`udi_channel_event_ind (UDI_CHANNEL_BOUND)` *on parent channel*
C	`udi_final_cleanup_req`
D	`udi_final_cleanup_ack`
E	`udi_channel_event_complete (UDI_CHANNEL_BOUND)`
F	`udi_devmgmt_req` (Unbind)
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_devmgmt_ack` (Unbind)

25.8.1 Management Metalanguage States

START — This is the initial state for any newly instantiated driver instance. A driver instance 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 driver instance. In this state, parent and child channels may be bound to the instance and normal I/O channel operations and functionality may occur.

CLEANUP — This is the final state for a driver instance and is entered from any state that has no parents or children bound, when `udi_final_cleanup_req` is received on that driver's management channel. Any remaining resources held by the driver must be released, preparatory to this driver instance being removed from the system.

25.8.1.1 Operational Sub-States

BINDING — A driver in the BINDING state is in the process of satisfying a `UDI_CHANNEL_BOUND` event 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 driver instance. When in this state, the instance is bound to one or more parents and may also be bound to one or more child instances. The driver instance is expected to be able to handle I/O traffic and any associated device activity.

UNBINDING — A driver instance enters this state when it has received a `UDI_DMGMT_UNBIND` request for a particular parent. 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 corresponding `udi_devmgmt_ack` operation) the parent channel will be deallocated by the MA. The MA may later re-use the parent ID to enter the BINDING state but the target driver should treat this binding as a completely new binding.

THROTTLED — This state is entered when the MA has suspended (or is in the process of suspending) a parent of the current driver instance; the driver should throttle operations and generate as little traffic as possible to the parent channel(s).

SUSPENDING — This state is entered when the MA 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 MA issues a device management operation instructing the driver to temporarily halt I/O activities with respect to this parent.

CLOSING — This state is entered when a `UDI_CHANNEL_CLOSED` event indication has been received on the parent bind channel to this parent. This indicates that the parent was abruptly removed as part of a region kill or other catastrophic event. In CLOSING state the target driver must clean up all resources relative to the parent immediately without exchanging further channel operations with that parent driver instance.

UNBOUND — This state is reached when the last parent and child are unbound from this instance.

Table 25-3 Management Metalanguage: Valid Operations by State

Operation	START	OPERATIONAL								CLEANUP
		BINDING	ACTIVE	UNBINDING	THROTTLED	SUSPENDING	SUSPENDED	CLOSING	UNBOUND	
udi_usage_ind	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	no
udi_usage_res	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**no**
udi_channel_event_ind (UDI_CHANNEL_BOUND) *on parent channel*	**YES**	no	no	no	no	no	no	no	**YES**	no
udi_channel_event_complete (UDI_CHANNEL_BOUND)	no	**YES**	no	no	no	no	no	**no**	**no**	**no**
udi_enumerate_req	no	no	**YES**	no	**YES**	**YES**	**YES**	no	**no**	no
udi_enumerate_ack	no	no	**YES**	**YES**	**YES**	**YES**	**YES**	**YES**	**no**	**YES**
udi_devmgmt_req (Prepare to Suspend)	no	no	**YES**	no	**YES**	no	no	no	**no**	no
udi_devmgmt_req (Suspend or Shutdown)	no	no	no	no	no	**YES**	no	no	**no**	no
udi_devmgmt_req (Resume)	no	no	no	no	**YES**	**YES**	**YES**	no	**no**	no
udi_devmgmt_req (Parent Suspended)	no	no	**YES**	no	no	no	no	no	**no**	no
udi_devmgmt_req (Unbind)	no	no	**YES**	no	no	no	**YES**	no	**no**	no
udi_devmgmt_ack (Unbind)	no	no	no	**YES**	no	no	no	YES	**no**	**no**
udi_devmgmt_ack	no	no	**YES**	**YES**	**YES**	**YES**	no	YES	**no**	**no**
udi_final_cleanup_req	**YES**	no	no	no	no	no	no	no	**YES**	no
udi_final_cleanup_ack	no	no	no	no	no	no	no	no	**no**	**YES**
udi_channel_event_ind (UDI_CHANNEL_CLOSED) *on parent channel*	no	**YES**	**YES**	YES	**YES**	**YES**	**YES**	no	**no**	no

Generic I/O Metalanguage 26

26.1 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 can also be used as an internal metalanguage between a multi-region driver's primary and secondary regions or between secondary regions.

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 timeout outstanding data and control operations;
- and the ability to send event notifications from the driver "upward."

Note – The GIO client must use udi_channel_op_abort on page 17-7 to abort outstanding GIO transfer requests. Only the `udi_gio_xfer_req` operation is abortable in GIO; therefore for a GIO channel, the ***`orig_cb`*** parameter to `udi_channel_op_abort` must point to a `udi_gio_xfer_cb_t` control block that was previously passed on the GIO channel from the client to the provider.

26.1.1 Versioning

All functions and structures defined in this chapter are part of the "`udi_gio`" interface, currently at version "`0x100`". A driver that conforms to and uses the Generic I/O Metalanguage of the UDI Core Specification, Version 1.0, must include the following declaration in its `udiprops.txt` file (see Chapter 31, *"Static Driver Properties"*):

```
requires udi_gio 0x100
```

Compile-time versioning and header files for the Generic I/O Metalanguage are covered by the general requirements for the UDI Core Specification defined in Chapter 8, *"General Requirements"*.

A portable implementation of the Generic I/O Metalanguage must include a corresponding "provides" declaration in its `udiprops.txt` file and conform to the same compile-time versioning and header file requirements as for drivers.

26.1.2 Roles

There are two roles to the Generic I/O Metalanguage: the "client" and the "provider." When this metalanguage is used between drivers, the client is always a child of the provider. When used as an internal metalanguage, the client and provider are both in the same driver, but in different regions.

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 client and the provider.

26.2 Metalanguage Bindings

26.2.1 Bindings for Static Driver Properties

Some of the bindings for the static driver properties are defined in Section 26.1.1, "Versioning". This includes the definition of the relevant interface name(s) (i.e., the <interface_name> parameter on the "requires" and "provides" and other property declarations), and the definition of the interface version number for this version of this Specification.

The driver category to be used with the "category" declaration (see Section 31.5.3, "Category Declaration," on page 31-9) by a portable implementation of the GIO Metalangauge Library shall be "Miscellaneous".

26.2.2 Bindings for Transfer Constraints

The applicability and semantics of the UDI transfer constraints, defined in **udi_constraints_attr_t** on page 13-4, are metalanguage-specific. For GIO, these apply only to the `udi_gio_xfer_req` operation, and then only for standard read/write operations (see **udi_gio_op_t** on page 26-17).

26.2.3 Bindings for Instance Attributes

In each of the attribute tables below, the **ATTRIBUTE NAME** is a null-terminated string (see "Instance Attribute Names" on page 16-1); the **TYPE** column specifies an attribute data type as defined in udi_instance_attr_type_t on page 16-7; and the **SIZE** column specifies the valid sizes, in bytes, for each attribute.

26.2.3.1 Enumeration Attributes

The driver that enumerates GIO clients must create the following enumeration attributes and pass them to the Management Agent in the ***attr_list*** parameter of the udi_enumerate_ack operation (see "Device Management Operations" on page 25-26).

Table 26-1 GIO Enumeration Attributes

ATTRIBUTE NAME	TYPE	SIZE	DESCRIPTION
gio_type	UDI_ATTR_STRING	1..32	Type of GIO client driver
gio_instance	UDI_ATTR_UBIT32	4	Instance number of GIO client relative to the parent GIO provider

The "gio_type" attribute specifies the type of GIO client driver being enumerated. The size of the "gio_type" attribute must be in the range 1..32, including the null-terminator character. If enumerating a GIO client to be a Diagnostics child, the value for this attribute is defined in Chapter 27, "*Diagnostics Support*". Otherwise, the contents of "gio_type" strings are driver-defined.

26.2.3.2 Filter Attributes

There are no filter attributes defined for the GIO Metalanguage.

26.2.3.3 Generic Enumeration Attributes

As defined in "Enumeration Attributes" on page 16-2, there are four generically-accessible enumeration attributes: "identifier", "address_locator", physical_locator", and "physical_label". These attributes, of type UDI_ATTR_STRING, are defined so as to allow environments to use these attributes in generic algorithms to identify and compare information about the devices in the system. This is useful in keeping the UDI environment isolated from the specifics of metalanguages and bus bindings.

In GIO, the "identifier" attribute must have the same value as the "gio_type" attribute. The "address_locator" attribute must have the same value as the "gio_instance" attribute. The "physical_locator" and "physical_label" attributes are not defined for the GIO metalanguage and must not be set by the enumerating driver.

26.2.4 Bindings for Trace Events

The following defines the rules and conventions in the GIO Metalanguage for the use of the metalanguage-selectable trace events (see the "Metalanguage-Selectable Trace Events" #defines in udi_trevent_t on page 18-3).

- UDI_TREVENT_IO_SCHEDULED
  - The provider should trace at least the corresponding GIO transfer control block pointer and GIO opcode.
- UDI_TREVENT_IO_COMPLETED
  - The provider should trace at least the corresponding GIO transfer control block pointer, GIO opcode, and *status*.

- `UDI_TREVENT_META_SPECIFIC_1`,
  `UDI_TREVENT_META_SPECIFIC_2`,
  `UDI_TREVENT_META_SPECIFIC_3`,
  `UDI_TREVENT_META_SPECIFIC_4`,
  `UDI_TREVENT_META_SPECIFIC_5`
    - Reserved for future use.

Note – All returned status values other than `UDI_OK` that indicate exceptional conditions must be logged and, when enabled, may also be traced, even if such events are expected.

26.3 Metalanguage State Diagram

See “Driver Instantiation” on page 25-2 for the general configuration sequence of UDI drivers. The following state diagram shows the GIO metalanguage state diagram, which illustrates the set of states specific to use of the GIO metalanguage. This same state diagram applies to both GIO clients and GIO providers.

Figure 26-1 GIO Metalanguage State Diagram

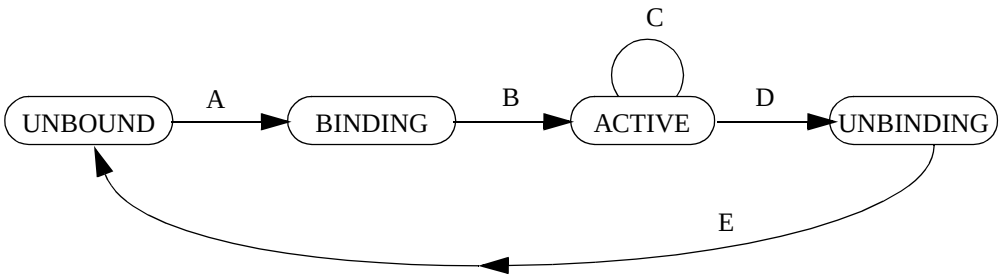


Table 26-2 GIO Metalanguage Events

Event	Operation
A	`udi_gio_bind_req`
B	`udi_gio_bind_ack`
C	`udi_gio_xfer_req`, `udi_gio_xfer_ack`, `udi_gio_xfer_nak`, `udi_gio_event_ind`, `udi_gio_event_res`
D	`udi_gio_unbind_req`
E	`udi_gio_unbind_ack`

26.3.1 GIO Metalanguage States

UNBOUND — A GIO channel in the unbound state has been established between the two regions but has not yet been initialized in those regions for general use. The client side of the GIO channel should initiate the GIO bind operation when in this state.

BINDING — This indicates that the client side of the GIO channel has initiated a bind operation and is waiting for the provider side of the GIO channel to complete its initialization and acknowledge that bind request.

ACTIVE — This indicates that the GIO channel is fully bound between the two regions and that it may be used for GIO transfer operations or event indications.

UNBINDING — This indicates that the GIO channel is being shut down. The client driver can cause this state to be entered by issuing a `udi_gio_unbind_req`. When the unbind operation is acknowledged, both the client and the provider return to the UNBOUND state.

26.4 Channel Ops Vectors

This section defines the channel ops vector types for use with the Generic I/O Metalanguage. There are two ops vector types in the Generic I/O Metalanguage: one that a GIO provider uses on its end of a GIO channel (`udi_gio_provider_ops_t`) and one that a GIO client uses on its end of a GIO channel (`udi_gio_client_ops_t`).

NAME

udi_gio_provider_ops_t *Provider entry point ops vector*

SYNOPSIS

```
#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_event_res_op_t *gio_event_res_op;
} udi_gio_provider_ops_t;

/* Ops Vector Number */
#define UDI_GIO_PROVIDER_OPS_NUM          1
```

DESCRIPTION

A Generic I/O provider uses the `udi_gio_provider_ops_t` structure in a `udi_ops_init_t` as part of its `udi_init_info` in order to register its entry points for receiving generic I/O bind and transfer requests, and event responses.

EXAMPLE

The driver's `udi_init_info` might include the following:

```
#define MY_GIO_OPS  1 /* Ops for my child GIO client */

#define MY_GIO_META 1 /* Meta index for GIO meta */

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_event_res
};

...

static const udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_GIO_OPS,
          MY_GIO_META,
          UDI_GIO_PROVIDER_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_gio_provider_ops },
     { 0 }
};
```

NAME

udi_gio_client_ops_t *Client entry point ops vector*

SYNOPSIS

```
#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_event_ind_op_t *gio_event_ind_op;
} udi_gio_client_ops_t;

/* Ops Vector Number */
#define UDI_GIO_CLIENT_OPS_NUM            2
```

DESCRIPTION

A Generic I/O client uses the `udi_gio_provider_ops_t` structure in a `udi_ops_init_t` as part of its `udi_init_info` in order to register its entry points for receiving generic I/O bind and transfer acknowledgements, and event indications.

EXAMPLE

The driver's `udi_init_info` might include the following:

```
#define MY_GIO_OPS  1 /* Ops for my parent GIO provider */

#define MY_GIO_META 1 /* Meta index for GIO meta */

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_event_ind
};

...

static const udi_ops_init_t ddd_ops_init_list[] = {
     {    MY_GIO_OPS,
          MY_GIO_META,
          UDI_GIO_CLIENT_OPS_NUM,
          0, /* chan_context_size */
          (udi_ops_vector_t *)&ddd_gio_client_ops },
     { 0 }
};
```

26.5 Binding and Unbinding Operations

NAME

udi_gio_bind_cb_t *Control block for GIO binding operations*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
} udi_gio_bind_cb_t;

/* Control Block Group Number */
#define UDI_GIO_BIND_CB_NUM              1
```

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 GIO client and the GIO provider to complete initial binding over the bind channel.

In order to use this type of control block it must be associated with a control block index by including `UDI_GIO_BIND_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

REFERENCES

`udi_init_info, udi_cb_init_t, udi_cb_alloc`

NAME **udi_gio_bind_req** *Request a binding to a GIO provider*

SYNOPSIS

```
#include <udi.h>

void udi_gio_bind_req (
     udi_gio_bind_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a GIO bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a GIO client to a GIO provider.

DESCRIPTION

A Generic I/O client uses this operation to bind to a Generic I/O provider.

The client must prepare for the `udi_gio_bind_req` operation by allocating a GIO bind control block (calling `udi_cb_alloc` with a ***`cb_idx`*** that was previously associated with `UDI_GIO_BIND_CB_NUM`).

Next, the client sends the GIO bind control block to the provider with a `udi_gio_bind_req` operation.

The `udi_gio_bind_req` operation must be the first channel operation sent on the bind channel. The GIO client must not send any further operations on the bind channel until it receives the corresponding `udi_gio_bind_ack` from the GIO provider.

REFERENCES

`udi_gio_bind_cb_t, udi_gio_bind_ack`

NAME **udi_gio_bind_ack** *Acknowledge a GIO binding*

SYNOPSIS

```
#include <udi.h>

void udi_gio_bind_ack (
     udi_gio_bind_cb_t *cb,
     udi_ubit32_t device_size_lo,
     udi_ubit32_t device_size_hi,
     udi_status_t status );
```

ARGUMENTS

cb is a pointer to a GIO bind control block.

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 GIO provider.

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 GIO provider.

status indicates whether or not the binding was successful.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a GIO provider to a GIO client.

DESCRIPTION

The `udi_gio_bind_ack` operation is used by a Generic I/O provider to acknowledge binding with a Generic I/O client (or failure to do so, as indicated by ***status***), as requested by a `udi_gio_bind_req` operation.

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 `udi_gio_rw_params_t` indicate the starting device offset for each transfer and transfers may be sent to the provider in any order. The client 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***.

If ***device_size_lo*** and ***device_size_hi*** are both zero, and the client uses standard read/write operations, then it must send them to the provider in device order, and ***offset_lo*** and ***offset_hi*** must be ignored by the provider.

STATUS VALUES

UDI_OK

UDI_STAT_CANNOT_BIND

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_gio_bind_req` operation.

REFERENCES

`udi_gio_bind_cb_t, udi_gio_bind_req`

NAME

udi_gio_unbind_req *Request to unbind from a GIO provider*

SYNOPSIS

```
#include <udi.h>

void udi_gio_unbind_req (
     udi_gio_bind_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a GIO bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a GIO client to a GIO provider.

DESCRIPTION

A Generic I/O client uses this operation to unbind from a Generic I/O provider.

The GIO client must prepare for the `udi_gio_unbind_req` operation by allocating a GIO bind control block (calling `udi_cb_alloc` with a ***`cb_idx`*** that was previously associated with `UDI_GIO_BIND_CB_NUM`).

Next, the GIO client sends the GIO bind control block to the GIO provider with a `udi_gio_unbind_req` operation.

REFERENCES

`udi_gio_bind_cb_t, udi_gio_unbind_ack`

NAME **udi_gio_unbind_ack** *Acknowledge a GIO unbind request*

SYNOPSIS

```
#include <udi.h>

void udi_gio_unbind_ack (
     udi_gio_bind_cb_t *cb );
```

ARGUMENTS

cb is a pointer to a GIO bind control block.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a GIO provider to a GIO client.

DESCRIPTION

The `udi_gio_unbind_ack` operation is used by a Generic I/O provider to acknowledge unbinding from a Generic I/O client as requested by a `udi_gio_unbind_req` operation.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_gio_unbind_req` operation.

REFERENCES

`udi_gio_bind_cb_t, udi_gio_unbind_req`

26.6 Data Transfer and Control Operations

NAME

udi_gio_xfer_cb_t *Control block for GIO transfer operations*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
     udi_gio_op_t op;
     void *tr_params;
     udi_buf_t *data_buf;
} udi_gio_xfer_cb_t;

/* Control Block Group Number */
#define UDI_GIO_XFER_CB_NUM              2
```

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.

op is a code designating the specific operation to be performed. Operation codes may be used to indicate different operation semantics. Custom operations are supported, as well as standard operations. See `udi_gio_xfer_req` for a description of these operations.

tr_params is a pointer to an inline memory structure that is used to hold operation-specific parameters. The pointer itself is set by the environment when the control block is allocated, and must not be modified by the driver.

data_buf is a pointer to a buffer used to carry the data portion of a transfer. See `udi_gio_xfer_req` and `udi_gio_xfer_ack` for details on buffer usage.

DESCRIPTION

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

In order to use this type of control block it must be associated with a control block index by including `UDI_GIO_XFER_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

The size and layout of the ***tr_params*** structure must be specified using the ***inline_size*** and ***inline_layout*** members of that `udi_cb_init_t` structure (i.e. ***tr_params*** is a `UDI_DL_INLINE_DRIVER_TYPED` field).

REFERENCES

`udi_init_info, udi_cb_init_t, udi_cb_alloc, udi_gio_op_t`

NAME

udi_gio_op_t *GIO operation type*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit8_t udi_gio_op_t;

/* Limit values for udi_gio_op_t */
#define UDI_GIO_OP_CUSTOM                16
#define UDI_GIO_OP_MAX                   64

/* Direction flag values for op */
#define UDI_GIO_DIR_READ                 (1U<<6)
#define UDI_GIO_DIR_WRITE                (1U<<7)

/* Standard Operation Codes */
#define UDI_GIO_OP_READ       UDI_GIO_DIR_READ
#define UDI_GIO_OP_WRITE      UDI_GIO_DIR_WRITE
```

DESCRIPTION

This type is used to hold an operation code, including direction flags, for a Generic I/O transfer control block.

The ***data_buf*** parameter is used to specify the data buffer to be read or written in this GIO transfer operation. The ***data_buf->buf_size*** field indicates the number of bytes that are to be transferred; if no actual data is to be transferred then ***data_buf*** may be NULL.

The operation code includes a bitmask of zero, one, or both direction flags from the following list:

UDI_GIO_DIR_READ - from provider to client

UDI_GIO_DIR_WRITE - from client to provider

These indicate the direction of data flow for the ***data_buf*** buffer. They do not imply any particular operation semantics.

The standard operation codes listed above are defined below with specific semantics. Additional, optional, standard operation codes are defined for device diagnostics in Chapter 27, *"Diagnostics Support"*. The GIO provider may define additional custom operations, whose semantics and parameters are completely defined by the GIO provider. However, the basic rules for use of ***data_buf***, ***data_buf->buf_size***, and the direction flags must be followed in all cases. Driver-defined custom operations must use ***op*** values of UDI_GIO_OP_CUSTOM or greater.

The UDI_GIO_OP_READ and UDI_GIO_OP_WRITE operations use a udi_gio_rw_params_t structure for ***tr_params*** in the transfer control block.

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_buf->buf_size*** bytes remaining on the device at the time the request is processed, then those bytes that are present must be returned and ***data_buf->buf_size*** adjusted accordingly. If there are no data bytes

available, and the possibility exists of more data arriving eventually, the provider must wait until at least one byte becomes available before responding.

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_buf->buf_size`*** additional bytes at the time the request is processed, then those bytes that fit must be sent to the device and ***`data_buf->buf_size must`*** be set to that value. 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 must continue to process the write operation once the device is no longer busy, and must not respond early with a short count.

Transfer constraints (see Chapter 13, *"Constraints Management"*) apply to the standard read/write operations, but not to any other standard or custom operations.

REFERENCES `udi_gio_xfer_cb_t, udi_gio_rw_params_t`

NAME

udi_gio_rw_params_t *Parameters for standard GIO read/write ops*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_ubit32_t offset_lo;
     udi_ubit32_t offset_hi;
     udi_ubit16_t timeout;
} udi_gio_rw_params_t;
```

MEMBERS

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`.

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`.

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`.

DESCRIPTION

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 memory structure of the `udi_gio_xfer_cb_t`, which must have been initialized with an ***inline_size*** of `sizeof(udi_gio_rw_params_t)`.

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.

REFERENCES

`udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_ack`

NAME **udi_gio_xfer_req** *Request a Generic I/O transfer*

SYNOPSIS

```
#include <udi.h>

void udi_gio_xfer_req (
     udi_gio_xfer_cb_t *cb );
```

ARGUMENTS

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 to a GIO provider.

DESCRIPTION

A Generic I/O client uses this operation to send a transfer request to a Generic I/O provider.

The GIO client 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 associated with `UDI_GIO_XFER_CB_NUM`) and filling in all of its members.

The client driver must then set ***data_buf->buf_size*** to the amount of data to be transferred via ***data_buf*** in the direction(s) indicated by the setting of the direction flags in ***op***: `UDI_GIO_DIR_READ` and/or `UDI_GIO_DIR_WRITE`. If no data is to be transferred in either direction, the client may set ***data_buf*** to `NULL`.

Finally, the client sends the GIO transfer control block to the provider with a `udi_gio_xfer_req` operation.

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 26-17 for descriptions of valid operation codes.

This operation is abortable with `udi_channel_op_abort` if ***op*** is `UDI_GIO_OP_READ` or `UDI_GIO_OP_WRITE`.

If ***op*** does not include `UDI_GIO_DIR_WRITE`, any data in ***data_buf*** is not guaranteed to be preserved by this channel operation. That is, when the provider driver receives this operation, the contents (but not the size) of the buffer are unspecified unless `UDI_GIO_DIR_WRITE` is set.

REFERENCES

`udi_gio_xfer_cb_t, udi_gio_op_t, udi_gio_xfer_ack`

NAME

udi_gio_xfer_ack *Acknowledge a GIO transfer request*

SYNOPSIS

```
#include <udi.h>

void udi_gio_xfer_ack (
     udi_gio_xfer_cb_t *cb );
```

ARGUMENTS

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 provider to a GIO client.

DESCRIPTION

The `udi_gio_xfer_ack` operation is used by a Generic I/O provider to acknowledge a transfer request back to a Generic I/O client (indicating success), as requested by a `udi_gio_xfer_req` operation. The `udi_gio_xfer_nak` operation is used to indicate failure or other exceptional conditions.

The ***op*** member of the control block must have the same value as at the time of the `udi_gio_xfer_req` operation. The contents of the ***tr_params*** inline memory are ignored for `udi_gio_xfer_ack`.

If ***data_buf*** is not `NULL`, ***data_buf->buf_size*** must be the same as in the original request and must equal the number of bytes actually transferred (overruns and underruns are handled with `udi_gio_xfer_nak`). The ***data_buf*** pointer must either be the same as in the original request, or a direct "descendant" of the original buffer (i.e. results from a chain of one or more service calls such as `udi_buf_write` that replace the original buffer with a modified version).

If ***op*** does not include `UDI_GIO_DIR_READ`, any data in ***data_buf*** is not guaranteed to be preserved by this channel operation. That is, when the provider driver receives this operation, the contents (but not the size) of the buffer are unspecified unless `UDI_GIO_DIR_READ` is set.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_gio_xfer_req` operation.

REFERENCES

`udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_nak, udi_buf_copy`

NAME

udi_gio_xfer_nak *Abnormal completion of a GIO transfer request*

SYNOPSIS

```
#include <udi.h>

void udi_gio_xfer_nak (
     udi_gio_xfer_cb_t *cb,
     udi_status_t status );
```

ARGUMENTS

cb is a pointer to a GIO transfer control block.

status indicates why the transfer was unsuccessful.

TARGET CHANNEL

The target channel for this operation is the bind channel connecting a GIO provider to a GIO client.

DESCRIPTION

The `udi_gio_xfer_nak` operation is used by a Generic I/O provider to send a negative acknowledgement of a transfer request (indicating failure, overruns, and underruns) back to the Generic I/O client that requested the transfer using a `udi_gio_xfer_req` operation. Whether or not overruns and underruns are considered errors is defined by the semantics of the particular ***op*** used and the needs of the client.

The ***op*** member of the control block must have the same value as at the time of the `udi_gio_xfer_req` operation. The contents of the ***tr_params*** inline memory are ignored for `udi_gio_xfer_nak`.

If ***data_buf*** is not `NULL`, the provider driver must set ***data_buf->buf_size*** to the number of bytes actually transferred, which must be less than or equal to the requested size. The ***data_buf*** pointer must either be the same as in the original request, or a direct "descendant" of the original buffer (i.e. results from a chain of one or more service calls such as `udi_buf_write` that replace the original buffer with a modified version).

If ***flags*** in the control block include `UDI_GIO_DIR_WRITE`, the contents of the data buffer must be the same as in the original request. This allows the client driver to retry failed operations if it so chooses.

Data in ***data_buf*** is always preserved by this channel operation.

STATUS VALUES

many

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_gio_xfer_req` operation.

REFERENCES

`udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_ack, udi_buf_copy`

26.7 Event Handling Operations

NAME

udi_gio_event_cb_t *Control block for GIO event operations*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_cb_t gcb;
     udi_ubit8_t event_code;
     void *event_params;
} udi_gio_event_cb_t;

/* Control Block Group Number */
#define UDI_GIO_EVENT_CB_NUM              3
```

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.

event_code is a driver-specific code that indicates the type of event which occured.

event_params is a pointer to additional parameters for this type of event. The structure and size of these parameters are defined by the GIO provider. The pointer itself is set by the environment when the control block is allocated, and must not be modified by the driver.

DESCRIPTION

The Generic I/O event control block is used between a GIO client and a GIO provider to notify the client of an asynchronous event.

In order to use this type of control block it must be associated with a control block index by including `UDI_GIO_EVENT_CB_NUM` in a `udi_cb_init_t` in the driver's `udi_init_info`.

The size and layout of the ***event_params*** structure must be specified using the ***inline_size*** and ***inline_layout*** members of that `udi_cb_init_t` structure (i.e. ***event_params*** is a `UDI_DL_INLINE_DRIVER_TYPED` field).

If there are no parameters for this type of event, ***event_params*** must be NULL and ***inline_size*** must be zero.

REFERENCES

`udi_init_info, udi_cb_init_t, udi_cb_alloc`

NAME

udi_gio_event_ind *GIO event indication*

SYNOPSIS

```
#include <udi.h>

void udi_gio_event_ind (
     udi_gio_event_cb_t *cb );
```

ARGUMENTS

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 provider to GIO client.

PROXIES

udi_gio_event_ind_unused *Proxy for udi_gio_event_ind*

```
udi_gio_event_ind_op_t udi_gio_event_ind_unused;
```

`udi_gio_event_ind_unused` may be used as a GIO client's `udi_gio_event_ind` entry point if the client expects that the GIO provider will never send it any event indications.

DESCRIPTION

A Generic I/O provider uses this operation to send an event notification to a Generic I/O client.

The GIO provider must prepare for the `udi_gio_event_ind` operation by allocating a GIO event control block (calling `udi_cb_alloc` with a ***`cb_idx`*** that was previously associated with `UDI_GIO_EVENT_CB_NUM`).

Next, the provider sends the GIO event control block to the GIO client with a `udi_gio_event_ind` operation. The provider does not need to wait to receive a response before sending another `udi_gio_event_ind`; multiple indications may be pending at once.

Whether or not a provider supports event notification, and whether or not the client must enable events explicitly (via custom operations), is defined by the GIO provider. There are no standard events.

REFERENCES

`udi_gio_event_cb_t, udi_gio_event_res`

NAME **udi_gio_event_res** *GIO event response*

SYNOPSIS

```
#include <udi.h>

void udi_gio_event_res (
     udi_gio_event_cb_t *cb );
```

ARGUMENTS

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 to a GIO provider.

PROXIES **udi_gio_event_res_unused** *Proxy for udi_gio_event_res*

```
udi_gio_event_res_op_t udi_gio_event_res_unused;
```

`udi_gio_event_res_unused` may be used as a GIO provider's `udi_gio_event_res` entry point if the provider never sends any event indications (and therefore expects no responses).

DESCRIPTION

The `udi_gio_event_res` operation is used by a Generic I/O client to acknowledge an event indication from a Generic I/O provider, as delivered by a `udi_gio_event_ind` operation.

WARNINGS

The control block must be the same control block as passed to the driver in the corresponding `udi_gio_event_ind` operation.

REFERENCES

`udi_gio_event_cb_t, udi_gio_event_ind`

Diagnostics Support *27*

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.

27.1 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 must 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 might refuse new child bindings. Once the driver has agreed to allow the diagnostics session to begin, it must not allow normal activities that would interfere with diagnostics (or vice versa) 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`.

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 (Diagnostics) *Diagnostics control operations*

SYNOPSIS

```
#include <udi.h>

typedef udi_ubit8_t udi_gio_op_t;

/* Diagnostics 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)
```

DESCRIPTION

The following optional Generic I/O standard operations are defined to support diagnostics operations on drivers. These supplement the operations defined for udi_gio_op_t on page 26-17.

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.

No data payload is used with this operation, so ***data_buf*** must be NULL.

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.

No data payload is used with this operation, so ***data_buf*** must be NULL.

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.

In the udi_gio_xfer_ack returned in response to the UDI_GIO_OP_DIAG_RUN_TEST operation, the status may be UDI_OK if the test passed, UDI_STAT_HW_PROBLEM if the test failed due to a hardware problem, UDI_STAT_NOT_SUPPORTED if the specified test is not supported on this device, UDI_STAT_ABORTED if the test was aborted, or UDI_STAT_INVALID_STATE if the driver is not in the proper state for running diagnostics.

In the case of UDI_STAT_HW_PROBLEM, the buffer pointed to by ***data_buf*** is filled in with a message string containing additional information to help isolate the failure to a specific field replaceable unit. The ***data_buf->buf_size*** field must be set to the length of that string (without any null terminator). The buffer returned in the ***data_buf*** parameter of udi_gio_xfer_ack must be the same buffer as received in the udi_gio_xfer_req or a direct decendent of that buffer (i.e. a ***dst_buf*** of the original buffer as processed by udi_buf_copy or udi_buf_write).

STATUS VALUES

UDI_OK – The operation completed successfully.

UDI_STAT_BUSY – The device or driver is not in a safe state for diagnostics.

UDI_STAT_NOT_SUPPORTED – The specified test number, or diagnostics in general, are not supported by this driver.

UDI_STAT_INVALID_STATE – The driver is not in diagnostics mode when requestes to run tests or disable diagnostics, or is already in diagnostics mode when requested to enable diagnostics.

UDI_STAT_HW_PROBLEM – The requested diagnostics test failed.

UDI_STAT_ABORTED – A test in progress was aborted.

NAME

udi_gio_diag_params_t *Parameters for standard GIO diagnostic ops*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_ubit8_t test_num;
     udi_ubit8_t test_params_size;
} udi_gio_diag_params_t;
```

MEMBERS

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.

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.

DESCRIPTION

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.

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.

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).

REFERENCES

udi_gio_xfer_cb_t, udi_gio_xfer_req, udi_gio_xfer_ack, udi_gio_xfer_cb_init

UDI Core Specification

Section 6: MEI Services

Introduction to MEI *28*

28.1 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.

MEI Services must not be used directly by driver modules.

This chapter defines requirements for the design of new metalanguages.

28.2 Requirements on Metalanguage Specifications

28.2.1 General Requirements & Conventions

A UDI metalanguage specification must define a version number for all its functions and structures. A driver that confoms to and uses that metalanguage must include the appropriate "requires" versioning declaration in its `udiprops.txt` file (see Chapter 31, *"Static Driver Properties"*).

In each UDI driver source file, before including any metalanguage-specific header files, the driver must define a preprocessor symbol to indicate the version of each metalanguage to which it conforms. This version number must be the same as the "requires" version number defined above. Metalanguage-specific header files must be included after "`udi.h`".

A portable implementation of any Metalanguage Library must include a corresponding "provides" declaration in its `udiprops.txt` file and must also define the preprocessor symbol.

As described in Section 31.4.6, "Requires Declaration," on page 31-6, the two least-significant hexadecimal digits of the interface version represent the minor number; the rest of the hex digits represent the major number. Versions that have the same "major version number" as an earlier version shall be backward compatible with that earlier version (i.e. a strict superset).

28.2.2 Bindings to the Core Specification

Each metalanguage definition must specify how each of the following generic concepts apply specifically to that metalanguage.

28.2.2.1 Bindings for Static Driver Properties

Each metalanguage definition must specify the relevant interface name(s) (i.e., the <interface_name> parameter on the "requires" and "provides" and "meta" property declarations), and the definition of the interface version number for this version of this metalanguage.

Each metalanguage definition must also specify the string to use for the "category" declaration.

28.2.2.2 Bindings for Transfer Constraints

The applicability and semantics of the UDI transfer constraints, defined in udi_constraints_attr_t on page 13-4, are metalanguage-specific and must be specified as part of each metalanguage definition.

28.2.2.3 Bindings for Instance Attributes

Each metalanguage can specify a list of instance attributes appropriate to that metalanguage. There are four principle classes of driver instance attributes: instance-private attributes, enumeration attributes; sibling-group attributes, and parent-visible attributes. Of these, metalanguages typically specify enumeration and, in some cases, parent-visible attributes.

There are four generic enumeration attributes whose specific content must be defined for each metalanguage that can be used between drivers: "identifier", "address_locator", "physical_locator" and "physical_label".

See Section 16.2, “Instance Attribute Names,” on page 16-1 for more details on Instance Attributes.

28.2.2.4 Bindings for Custom Parameters

Many metalanguages also specify minimum requirements for driver’s use of “custom” parameters, by defining a set of private instance attributes each driver must support.

28.2.2.5 Bindings for Trace Events

Each metalanguage must specify how the metalanguage-selectable trace events apply to that metalanguage. This must include a definition of the metalanguage-specific semantics for `UDI_TREVENT_IO_SCHEDULED` and `UDI_TREVENT_IO_COMPLETED`, as well as any metalanguage-specific events. See Section 18.2.3, “Trace Event Types,” on page 18-2.

28.2.2.6 Abortable Ops

Each metalanguage must specify which (if any) of its metalanguage operations are abortable (see udi_channel_op_abort on page 17-7). By default, any operation that is not explicitly identified as abortable may be assumed to not be abortable.

28.2.3 State Diagram

Each metalanguage should include a state diagram which indicates the valid set of state transitions for a driver implementing that metalanguage, as well as the valid set of operations for each state.

Metalanguage-to-Environment Interface 29

29.1 Overview

The Metalanguage-to-Environment Interface (MEI) is a set of interfaces designed to allow for the creation of portable metalanguage libraries. This chapter defines the data structures, macros, and service calls that make up the UDI MEI services.

Metalanguage stubs are the pieces of code that implement metalanguage channel operations. In the UDI execution model each channel operation requires a front-end stub, a back-end stub, and a direct-call 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; 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.

29.1.1 Versioning

All functions and structures defined in the MEI Services section of the UDI Core Specification are part of the "`udi_mei`" interface, currently at version "`0x100`". A library module that conforms to and uses the MEI Services of the UDI Core Specification, Version 1.0, must include the following declaration in its `udiprops.txt` file (see Chapter 31, *"Static Driver Properties"*):

```
requires udi_mei 0x100
```

29.2 Initialization Structures

Every metalanguage library must contain a global variable named `udi_meta_info`, of type `udi_mei_init_t`, declared as follows:

```
const udi_mei_init_t udi_meta_info;
```

This structure contains information describing the metalanguage-specific properties of control blocks and ops vectors used with the particular metalanguage. The environment uses this information to initialize drivers that use each metalanguage, before executing any code in either driver or metalanguage library.

This section contains descriptions of the various components of the `udi_meta_info` structure.

NAME

udi_meta_info *Metalanguage initialization structure*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     const udi_mei_ops_vec_template_t
               *ops_vec_template_list;
} udi_mei_init_t;

const udi_mei_init_t udi_meta_info;
```

MEMBERS

ops_vec_template_list is a pointer to a list of structures containing information about each type of ops vector supported by this metalanguage.

DESCRIPTION

This structure contains information describing the metalanguage-specific properties of control blocks and ops vectors used with the particular metalanguage. The environment uses this information to initialize drivers that use each metalanguage, before executing any code in either driver or metalanguage library.

REFERENCES

`udi_init_info, udi_mei_ops_vec_template_t`

NAME **udi_mei_ops_vec_template_t** *Metalanguage ops vector template*

SYNOPSIS

```
#include <udi.h>

typedef struct {
     udi_index_t meta_ops_num;
     udi_ubit8_t relationship;
     const udi_mei_op_template_t *op_template_list;
} udi_mei_ops_vec_template_t;

/* Flag values for relationship */
#define UDI_MEI_REL_INITIATOR            (1U<<0)
#define UDI_MEI_REL_BIND                 (1U<<1)
#define UDI_MEI_REL_EXTERNAL             (1U<<2)
#define UDI_MEI_REL_INTERNAL             (1U<<3)
#define UDI_MEI_REL_SINGLE               (1U<<4)
```

MEMBERS

meta_ops_num is a number that identifies this ops vector type with respect to others in this metalanguage, or zero to terminate the ***ops_vec_template_list*** array to which this structure belongs (see udi_mei_init_t). If ***meta_ops_num*** is zero, all other members of this structure are ignored.

relationship defines the valid relationships between the regions on opposite ends of a channel when using an ops vector of this type.

Relationship must include at least one of UDI_MEI_REL_EXTERNAL or UDI_MEI_REL_INTERNAL. If and only if ***relationship*** includes UDI_MEI_REL_EXTERNAL, then this ops type can be used for an external (driver-to-driver) channel. If and only if ***relationship*** includes UDI_MEI_REL_INTERNAL, then this ops type can be used for an internal (within one driver instance) channel.

If and only if ***relationship*** includes UDI_MEI_REL_INITIATOR, then this ops type can be used for the initiator side of a channel (the side that sends the first operation). Otherwise, this ops type can only be used for the non-initiator (responder) side of a channel.

If and only if ***relationship*** includes UDI_MEI_REL_BIND, then this ops type can be used for a bind channel. Otherwise, this ops type can only be used for an auxiliary, non-bind, channel.

Both ends of a channel must be paired appropriately: both must have the same combination of UDI_MEI_REL_EXTERNAL, UDI_MEI_REL_INTERNAL, and UDI_MEI_REL_BIND; exactly one must have UDI_MEI_REL_INITIATOR set.

`UDI_MEI_REL_SINGLE` is legal only if both `UDI_MEI_REL_EXTERNAL` and `UDI_MEI_REL_BIND` are also set and `UDI_MEI_REL_INITIATOR` is not. If and only if ***`relationship`*** does not include `UDI_MEI_REL_SINGLE`, then a driver using this ops type for a child bind channel must be prepared to have multiple child instances bound to it for each enumerated child context.

`op_template_list` is a pointer to a list of structures containing information about each type of ops vector supported by this metalanguage. The `udi_channel_event_ind_op_t` at the beginning of every ops vector type is not included in this list; the first entry in the list corresponds to index *one*, rather than zero, in the ops vector.

DESCRIPTION

This structure is used to describe the set of metalanguage operations that correspond to the specified ***`meta_ops_num`*** for the role indicated by the relationship parameter. The ops are described in terms of their parameters and the associated ownership and data transfer of those parameters as a result of performing the specified operation.

One structure of this type must be defined for each ops vector types. These will generally come in pairs: one for the ops at each end of the channel.

REFERENCES

`udi_channel_event_ind, udi_mei_init_t, udi_mei_op_template_t`

NAME **udi_mei_op_template_t** *Metalanguage channel op template*

SYNOPSIS

```
#define <udi.h>

typedef struct {
     const char *op_name;
     udi_ubit8_t op_category;
     udi_ubit8_t op_flags;
     udi_index_t meta_cb_num;
     udi_index_t completion_ops_num;
     udi_index_t completion_vec_idx;
     udi_index_t exception_ops_num;
     udi_index_t exception_vec_idx;
     udi_mei_direct_stub_t *direct_stub;
     udi_mei_backend_stub_t *backend_stub;
     const udi_layout_t *visible_layout;
     const udi_layout_t *marshal_layout;
} udi_mei_op_template_t;

/* Values for op_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
#define UDI_MEI_OPCAT_RDY                6

/* Values for op_flags */
#define UDI_MEI_OP_ABORTABLE             (1U<<0)
#define UDI_MEI_OP_RECOVERABLE           (1U<<1)
#define UDI_MEI_OP_STATE_CHANGE          (1U<<2)

/* Maximum Sizes For Control Block Layouts */
#define UDI_MEI_MAX_VISIBLE_SIZE         2000
#define UDI_MEI_MAX_MARSHAL_SIZE         4000
```

MEMBERS

op_name is the name of the entry point for the channel operation (exactly as documented for that operation; e.g. “udi_gio_xfer_req” for `udi_gio_xfer_req`), or NULL to terminate the ***op_template_list*** list to which this structure belongs (see `udi_mei_ops_vec_template_t`). Some environments may use this information to selectively trace channel operations. If ***op_name*** is NULL, all other members of this structure are ignored.

op_category is a number that identifies the category of the channel op described by this template, as indicated by its suffix. Channel op suffixes are described in Section 24.3, “Channel Operation Suffixes,” on page 24-2. Some environments may use this information to selectively trace channel operations.

op_flags is a bitmask of optional flags for this template, described below.

meta_cb_num is a number that identifies the control block group used with this operation, with respect to others in this metalanguage. It must be greater than zero.

completion_ops_num is a number that identifies the ops vector type that contains the completion operation, if any, that is the normal response to this operation. If zero, then there is no such response operation; otherwise, ***completion_ops_num*** must match a ***meta_ops_num*** in a `udi_mei_ops_vec_template_t` for this metalanguage.

completion_vec_idx is a number that identifies the index within the above ops vector that contains the function pointer for the completion operation, if any, that is the normal response to this operation, starting from zero. This is used if and only if ***completion_ops_num*** is non-zero.

exception_ops_num is a number that identifies the ops vector type that contains the exception operation, if any, that is the error response to this operation. If zero, then there is no such response operation; otherwise, ***completion_ops_num*** must match a ***meta_ops_num*** in a `udi_mei_ops_vec_template_t` for this metalanguage.

exception_vec_idx is a number that identifies the index within the above ops vector that contains the function pointer for the exception operation, if any, that is the error response to this operation, starting from zero. This is used if and only if ***exception_ops_num*** is non-zero.

direct_stub is a pointer to the function that implements the direct-call stub for this operation.

backend_stub is a pointer to the function that implements the back-end stub for this operation.

visible_layout is a pointer to the layout specifier for the visible part of the control block type used with this operation, excluding the generic `udi_cb_t` header.

marshal_layout is a pointer to the layout specifier for any marshalling space used to marshal extra parameters for this operation.

DESCRIPTION

The `udi_mei_ops_template_t` structure contains information describing the metalanguage-specific properties of a channel operation and its associated control block type.

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).

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

If and only if ***op_flags*** includes UDI_MEI_OP_ABORTABLE, the channel operation described by this structure is abortable, and drivers may use udi_channel_op_abort to abort control blocks previously passed to this operation. The udi_channel_op_abort service call will deliver a udi_channel_event_ind operation of type UDI_CHANNEL_OP_ABORTED to the target region if the corresponding completion operation (as indicated by ***completion_ops_num*** and ***completion_vec_idx***) or exception operation (as indicated by ***exception_ops_num*** and ***exception_vec_idx***) has not yet been invoked.

If and only if ***op_flags*** includes UDI_MEI_OP_RECOVERABLE, the channel operation described by this structure is recoverable; if an operation of this type has been sent to a region that is abruptly terminated ("region-killed"), and the target region has not yet responded with the corresponding completion or exception operation, then the environment will automatically construct an exception operation to inform the initiating region of the failure, passing it the special status code, UDI_STAT_TERMINATED. If this flag is set, ***exception_ops_num*** must be non-zero. and the exception operation must contain exactly one UDI_DL_STATUS_T in either its ***visible_layout*** or its ***marshal_layout***.

If and only if ***op_flags*** includes UDI_MEI_OP_STATE_CHANGE, the channel operation is considered to cause a change in the metalanguage-related state of the driver. Environments can use this to trace state changes externally to the driver.

WARNINGS

If ***visible_layout*** includes an inline pointer element (UDI_DL_INLINE_UNTYPED, UDI_DL_INLINE_TYPED, or UDI_DL_INLINE_DRIVER_TYPED), there must be exactly one op_template for this ***meta_cb_num*** of this metalanguage.

The ***marshal_layout*** specifier must include no inline pointers.

REFERENCES

udi_mei_ops_vec_template_t, udi_cb_t, udi_layout_t, udi_mei_direct_stub_t, udi_mei_backend_stub_t

NAME

udi_mei_direct_stub_t *Metalanguage direct-call stub type*

SYNOPSIS

```
#include <udi.h>

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

ARGUMENTS

op is a pointer to the driver entry point function in the target region that will be called to handle this operation. This function was declared in a `udi_ops_init_t` structure by the driver for the corresponding ops vector.

gcb is the pointer to the control block that is to be used for this operation (as passed by the driver requesting the operation).

arglist is the list of arguments that are to be passed to the ***op*** function.

DESCRIPTION

The `udi_mei_direct_stub_t` type is used for metalanguage “direct” stub functions. These are used by `udi_mei_call` when it makes a direct call to the target region, without marshalling parameters.

Direct-call stubs are automatically generated by the UDI_MEI_STUBS macro.

REFERENCES

`udi_ops_init_t, udi_mei_call, UDI_MEI_STUBS`

NAME **udi_mei_backend_stub_t** *Metalanguage back-end stub type*

SYNOPSIS

```
#include <udi.h>

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

ARGUMENTS

op is a pointer to the driver entry point function in the target region that will be called to handle this operation. This function was declared in a `udi_ops_init_t` structure by the driver for the corresponding ops vector.

gcb is the pointer to the control block that is to be used for this operation (as passed by the driver requesting the operation).

marshal_space is a pointer to the marshalling space containing the marshalled parameters to pass as arguments to the ***op*** function.

DESCRIPTION

The `udi_mei_backend_stub_t` type is used for metalanguage "back-end" stub functions. These are used by `udi_mei_call` when it makes a call to the target region for an operation that has previously been marshalled.

Back-end stubs are automatically generated by the UDI_MEI_STUBS macro.

REFERENCES `udi_ops_init_t, udi_mei_call, UDI_MEI_STUBS`

29.3 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.

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.

29.4 MEI Stubs

The following section describes the stubs used by metalanguage libraries and the implementation of their functions. Each invocation of `UDI_MEI_STUBS` generates 3 stubs functions for a channel operation: front-end, direct-call, and back-end, with the following pseudo-code.

The front-end stub implements the exported interface for the caller side of a channel operation:

```
void
<<meta>>_<<op>> (
   <<meta>>_<cbtype>>_cb_t *cb,
   ...<<call-dependent parms>>... )
{
   udi_mei_call(UDI_GCB(cb), &udi_meta_info, \
         ZZZ_OPS_NUM, ZZZ_VEC_IDX, \
         ...<<call-dependent parms>>...);
}
```

The direct-call stub implements the call into the target driver when the environment wishes to invoke it directly from the original calling context:

```
static void
<<meta>>_<<op>>_direct (
   udi_op_t *op,
   udi_cb_t *gcb,
   va_list arglist )
{
   arg1_type arg1 = va_arg(arglist, arg1_type);
   ...

   (*(<<meta>>_<<op>>_op_t)op)
         (UDI_MCB(gcb, <<meta>>_<cbtype>>_cb_t),
               arg1... );
}
```

The back-end stub implements the call into the target driver when the environment wishes to invoke it after having queued the channel operation:

```
static void
<<meta>>_<<op>>_backend (
   udi_op_t *op,
   udi_cb_t *gcb,
   void *marshal_space )
{
   struct <<meta>>_<<op>>_marshal {
         arg1_type arg1;
         ...
   } *mp = marshal_space;

   (*(<<meta>>_<<op>>_op_t)op)
         (UDI_MCB(gcb, <<meta>>_<cbtype>>_cb_t),
               mp->arg1... );
}
```

NAME **UDI_MEI_STUBS** *Metalanguage stub generator macro*

SYNOPSIS

```
#include <udi.h>

#define UDI_MEI_STUBS ( op_name, cb_type,
          argc, args, arg_types,
          meta_ops_num, vec_idx )
```

ARGUMENTS

op_name is a token specifying the name of the channel operation for which to create stub functions.

cb_type is the data type of control blocks used with this operation.

argc is the number of additional arguments to the operation.

args is a comma-separated list, enclosed in parentheses, of the names of the additional arguments.

arg_types is a comma-separated list, enclosed in parentheses, of the data types of the additional arguments.

meta_ops_num is the metalanguage-defined identifier for the ops vector type to which this operation belongs. (See udi_mei_ops_vec_template_t on page 29-4.)

vec_idx is the index into the ops vector identified by ***meta_ops_num*** that corresponds to this operation, starting from zero. (A vec_idx of zero corresponds to the `udi_channel_event_ind_op_t` at the beginning of every ops vector type, and is not actually used in metalanguage libraries.)

DESCRIPTION

Each invocation of **UDI_MEI_STUBS** creates the definition of the following three functions needed to support a metalanguage-specific channel operation:

Front-end stub:

```
void op_name ( cb_type *cb
          _UDI_ARG_LIST_##argc args );
```

Direct-call stub:

```
static udi_mei_direct_stub_t op_name##_direct;
```

Back-end stub:

```
static udi_mei_backend_stub_t op_name##_backend;
```

REFERENCES

`udi_mei_ops_vec_template_t, udi_channel_event_ind, udi_mei_call, udi_mei_direct_stub_t, udi_mei_backend_stub_t`

EXAMPLES

The following examples illustrate the use of UDI_MEI_STUBS.

The `udi_gio_bind_ack` channel operation has three extra parameters and could be implemented using the stubs macro as follows:

```
UDI_MEI_STUBS(udi_gio_bind_ack, udi_gio_bind_cb_t,
          3, (device_size_lo, device_size_hi, status),
           (udi_ubit32_t, udi_ubit32_t, udi_status_t),
          UDI_GIO_CLIENT_OPS_NUM,
          UDI_GIO_BIND_ACK)
```

The `udi_scsi_io_req` channel operation has no extra parameters and could be implemented using the stubs 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)
```

NAME

udi_mei_call *Channel operation invocation*

SYNOPSIS

```
#include <udi.h>

void udi_mei_call (
     udi_cb_t *gcb,
     const udi_mei_init_t *meta_info,
     udi_index_t meta_ops_num,
     udi_index_t vec_idx,
     ... );
```

ARGUMENTS

gcb is a pointer to the control block passed to the actual channel operation.

meta_info is a pointer to this metalanguage's `udi_meta_info` structure, which can be used to uniquely identify this metalanguage.

meta_ops_num is the metalanguage-defined identifier for the ops vector type to which this operation belongs. (See udi_mei_ops_vec_template_t on page 29-4.)

vec_idx is the index into the ops vector identified by ***meta_ops_num*** that corresponds to this operation, starting from zero. (A vec_idx of zero corresponds to the `udi_channel_event_ind_op_t` at the beginning of every ops vector type, and is not actually used in metalanguage libraries.)

... are zero or more additional metalanguage-specific parameters for this channel operation. This will be passed to the callee-side entry point in the target driver, immediately following the control block argument.

DESCRIPTION

This 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.

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 ***`gcb->channel`*** and ***`vec_idx`*** or ***`meta_init`*** and ***`meta_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 unmarshals the parameters from the marshalling space (pointed to by ***`marshal_space`***) and calls the driver entry point with these parameters

REFERENCES

`udi_mei_ops_vec_template_t, udi_channel_event_ind, udi_mei_direct_stub_t, udi_mei_backend_stub_t, UDI_MEI_STUBS`

29.5 MEI Stub Implementation

This section presents a typical implementation of the UDI_MEI_STUBS macro. Actual implementations of `UDI_MEI_STUBS` may vary, but must generate equivalent code.

```
#define UDI_MEI_STUBS(op_name, cb_type, \
                      argc, args, arg_types, \
                      meta_ID, ops_num, vec_idx) \
   void op_name ( cb_type *cb \
                 _UDI_ARG_LIST_##argc args ) { \
         udi_mei_call ( UDI_GCB(cb), &udi_meta_info, \
                        ops_num, vec_idx \
                        _UDI_ARG_VARS_##argc ); \
   } \
   static void op_name##_direct ( \
                     udi_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_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 ); \
   }

#define _UDI_ARG_LIST_0()
#define _UDI_ARG_LIST_1(a)             ,a arg1
#define _UDI_ARG_LIST_2(a,b)           ,a arg1,b arg2
#define _UDI_ARG_LIST_3(a,b,c)         ,a arg1,b arg2,c arg3
#define _UDI_ARG_LIST_4(a,b,c,d)       \
               ,a arg1,b arg2,c arg3,d arg4
#define _UDI_ARG_LIST_5(a,b,c,d,e)     \
               ,a arg1,b arg2,c arg3,d arg4,e arg5
#define _UDI_ARG_LIST_6(a,b,c,d,e,f)   \
               ,a arg1,b arg2,c arg3,d arg4,e arg5,f arg6
#define _UDI_ARG_LIST_7(a,b,c,d,e,f,g) \
               ,a arg1,b arg2,c arg3,d arg4,e arg5,f arg6,g arg7

#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
```

UDI Core Specification

Section 7: Packaging and Distribution

Introduction to Packaging and Distribution *30*

30.1 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 31 defines the Static Driver Properties file that is used to provide global driver attributes.

Chapter 32 defines the packaging and distribution formats for UDI drivers.

Chapter 33 describes build and packaging utilities provided by UDI build environments.

Static Driver Properties 31

31.1 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.

31.1.1 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 31-6 and the “Provides Declaration” on page 31-8).

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 31.6.8, “Region Declaration,” on page 31-15 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 `udi_init_info` structure (see udi_init_info on page 10-3).

UDI libraries each consist of at most one library module and zero or more exported header files. UDI libraries provide functions that can be called by UDI drivers, but maintain no state of their own. Library modules do not have `udi_init_info` structures and will not have any region data associated with

them. However, if a library is a *metalanguage library* (i.e. implements a metalanguage API), then it will have a `udi_meta_init` structure, which serves a similar purpose as a driver's `udi_init_info` structure.

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 utility program called "`udimkpkg`" (see Section 33.3 on page 33-1) attaches the property values from `udiprops.txt` to the binary object file for the driver's primary module or the library's sole module, in a fashion appropriate to the particular binary object file format used. Header-only libraries have no modules to compile, so their static properties remain as a text file.

31.2 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 ('#') preceeds 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.

31.3 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 "`0x100`":

```
properties_version 0x100
```

Properties version `0x100` 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. The value of <msgnum> must be $0..2^{16}$-1 (i.e. a 16-bit value with 0 reserved as illegal).

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.

31.4 Common Property Declarations

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

31.4.1 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.

31.4.2 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.

31.4.3 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.

31.4.4 Shortname Declaration

Exactly one “shortname” 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.

31.4.5 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 31-1, “Enumeration Attribute Value Encoding,” on page 31-13) 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.

31.4.6 Requires Declaration

One or more “requires” declarations must be included:

```
requires <interface_name> <version_number>
```

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 UDI drivers and libraries must include the following “requires” declaration:

```
requires udi 0x100
```

Additional “requires” statements for each of the other UDI interfaces used by the driver must be included; 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 31.5.1, “Provides Declaration,” on page 31-8.

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).

If the interface name begins with a percent-sign (‘%’), the required interface must match a “provides” declaration in the same package collection.

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.

The “requires” declaration indicates both an external symbol dependency for linking/loading, and a compile-time dependency on any header files exported by the providing library. To express a dependency only on header files, use “source_requires”.

31.4.7 Module Declaration

One or more “module” declarations must be included, except in a library that only exports header files:

```
module <filename>
```

Each “module” declaration denotes a module that is part of this driver or library. Drivers must have at least one module declaration. Libraries must have at most one module declaration.

The `<filename>` string provides the name of a binary module file; it must be a local name, without any path separators. 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 on the target system.

The following declaration types are sensitive to ordering relative to “module” declarations: “region”, “requires”, “source_files”, and “source_requires”. See each of these declaration sections for more details.

31.5 Property Declarations for Libraries

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

31.5.1 Provides Declaration

One or more “provides” declarations must be included:

```
provides <interface_name> <version_number>
```

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 (defined in each specification as described in the “requires” declaration), 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).

By default, if no “symbols” declarations are associated with this “provides” declaration, 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 each of the “provides” declarations. For libraries with a single “provides” declaration, “symbols” is optional.

If the interface name begins with a percent-sign (‘%’), this interface is visible only to modules in the same package collection. Otherwise the library is available for use by any UDI package installed into the system and will be referenced by a “requires” declaration in that driver’s installation. To avoid conflicts with this global namespace, the following naming convention is recommended for the `<interface_name>` parameter when it does not begin with ‘%’:

1. It should be a trademarked name owned by the supplying company, or
2. It should begin with the supplying company’s stock symbol followed by an underscore if that company is publicly traded, or
3. It should start with an underscore followed by the company or organization’s name or commonly used acronym, or
4. It should start with two underscores followed by the developer’s name or similar identification if not affiliated with any company or organization.

The “symbols” declaration type is sensitive to ordering relative to “provides” declarations. See the “Symbols Declaration” section for more details.

31.5.2 Symbols Declaration

Zero or more “symbols” declarations may be included:

```
symbols { [ <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. “Symbols” declarations apply to the most recently declared “provides” declaration preceding the “symbols” declaration. “Symbols” declarations must not precede the first “provides” declaration. Multiple “symbols” declarations may be provided for the same “provides” declaration.

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.

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).

31.5.3 Category Declaration

One optional “category” declaration may be included in a library that is used as a metalanguage library:

```
category <msgnum>
```

The category message string is a human-readable brief (two or three word) description of the category of device supported by drivers that use this metalanguage as a child metalanguage. While the overall type of device can be inferred from the driver’s “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 category name (in English) to be used for its “category” declaration. The message text for the POSIX (“C”) locale for this “category” declaration must exactly match the specified category name, since driver documentation may refer to these strings.

Environments may choose to group drivers by category for purposes of presenting lists of drivers to administrators, and to use the category message strings from the associated libraries to present a heading for each group. If a driver falls into multiple categories (because it has multiple child metalanguages), it is recommended but not required that it be listed in all categories to which it belongs.

The category name must be phrased as a appropriate for a table heading, and thus must be a plural (or collective) noun phrase. Examples of possible category names are listed below. Refer to metalanguage specifications for the official names.

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

31.6 Property Declarations for Drivers

The property declarations in this section apply only to drivers.

31.6.1 Meta Declaration

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

```
meta <meta_idx> <interface_name>
```

A “meta” declaration indicates a metalanguage that may be used by this driver. The `<interface_name>` string must be the same as the `<interface_name>` in a “requires” declaration for this driver.

The `<meta_idx>` specified in the “meta” declaration 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) and is used to refer to this metalanguage in other declarations and in the driver’s `udi_init_info` structure.

The `<meta_idx>` number must be unique with respect to all “meta” declarations for this driver.

31.6.2 Child_bind_ops Declaration

Exactly one “child_bind_ops” declaration must be included for each type of child binding ops index supported by the driver:

```
child_bind_ops <meta_idx> <region_idx> <ops_idx>
```

A “child_bind_ops” declaration indicates a metalanguage that may be used to bind children to this driver. Some drivers may support multiple child metalanguages.

The `<meta_idx>` token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another and must match a corresponding “meta” declaration. It must also match the ***`meta_idx`*** value specified in the driver’s `udi_ops_init_t` structures corresponding to `<ops_idx>`. The ***`meta_ops_num`*** for this `udi_ops_init_t` structure must refer to an ops vector type that is suitable for use with child bind channels (as indicated by the relationship value in the metalanguage library’s `udi_ops_vec_template_t`). For more information, see udi_ops_init_t on page 10-9 and udi_mei_ops_vec_template_t on page 29-4.

When the driver is being bound to a child using the specified ops index, its end of the bind channel will be anchored using `<ops_idx>` in a region of type `<region_idx>`.

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

31.6.3 Parent_bind_ops Declaration

Exactly one “parent_bind_ops” declaration must be included for each type of parent metalanguage supported:

```
parent_bind_ops <meta_idx> <region_idx> <ops_idx>
```

A “parent_bind_ops” declaration indicates a metalanguage that may be used to bind parents to this driver. Some drivers may support multiple parent metalanguages.

The <meta_idx> token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another and must match a corresponding “meta” declaration. It must also match the ***meta_idx*** value specified in the driver’s udi_ops_init_t structures corresponding to <ops_idx>. The ***meta_ops_num*** for this udi_ops_init_t structure must refer to an ops vector type that is suitable for use with parent bind channels (as indicated by the relationship value in the metalanguage library’s udi_ops_vec_template_t). For more information, see udi_ops_init_t on page 10-9 and udi_mei_ops_vec_template_t on page 29-4.

When the driver is being bound to a parent using the specified metalanguage, its end of the bind channel will be anchored using <ops_idx>.

Drivers with no “parent_bind_ops” 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:

- 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 16.4.3, “Sibling Group Attributes,” on page 16-4.)
- Orphan driver instances are never bound to parents, so udi_bind_to_parent_req is never called in an orphan driver.

31.6.4 Internal_bind_ops Declaration

Exactly one “internal_bind_ops” declaration must be included for each type of secondary region:

```
internal_bind_ops <meta_idx> <region_idx> \
                      <primary_ops_idx> <secondary_ops_idx>
```

A “internal_bind_ops” declaration indicates a metalanguage that may be used between regions internal to this driver. There must be a one-to-one correspondence between “internal_bind_ops” declarations and “region” declarations, based on matching <region_idx> values, except for region index zero (the primary region).

The <meta_idx> token is an ASCII-encoded decimal number from 1 to 255 that is used to distinguish one metalanguage declaration from another and must match a corresponding “meta” declaration. It must also match the ***meta_idx*** value specified in the driver’s udi_ops_init_t structures corresponding to both <primary_ops_idx> and <secondary_ops_idx>. The ***meta_ops_num*** for these udi_ops_init_t structures must refer to ops vector types that are suitable for use with internal bind channels (as indicated by the relationship value in the metalanguage library’s udi_ops_vec_template_t). For more information, see udi_ops_init_t on page 10-9 and udi_mei_ops_vec_template_t on page 29-4.

When a secondary region of this type is created, an internal bind channel will be created by the environment, with the primary region’s end of the channel anchored using <primary_ops_idx> and with the secondary region’s end anchored using <secondary_ops_idx>.

31.6.5 Device Declaration

One or more “device” declarations must be included:

```
device <msgnum> <meta_idx> { <attr_name> <attr_type> <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; the corresponding message referenced is intended to be a verbose human-readable device name (see Section 31.6.10, “Message Declaration”).

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 selected parent metalanguage, as indicated by a “parent_bind_ops” declaration with matching `<meta_idx>`. 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.

If two “device” declarations for the same driver use the same `<msgnum>`, the environment may bind multiple parents to this driver of different types (as indicated by the `<meta_idx>` values, which must be different for each of these “device” declarations).

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 16-7), as indicated by `<attr_type>`, according to the following table. `<attr_type>` must be one of the tokens in the Type Name column of this table.

Table 31-1 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 Table 31-2 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.
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 31-2 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.
all others	All other escape sequences (as identified by the initial backslash character) are illegal. The result of using an illegal escape sequence is indeterminate and implementation-specific.

Values for enumeration attributes of other types not listed in Table 31-1 cannot be used in property declarations.

31.6.6 Enumerates Declaration

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

```
enumerates <msgnum> <min_num> <max_num> <meta_idx> \
                        { <attr_name> <attr_type> <attr_value> }
```

Each “enumerates” declaration describes a type of (actual or pseudo) child device that this driver is likely to enumerate when used with a device who’s “device” declaration has a matching <msgnum>. This can be used as a hint to the environment, for example to help choose drivers to pre-load in a static environment, or, on the opposite end of the spectrum to allow drivers to be automatically loaded as they are accessed by applications.

As with “device” declarations, “enumerates” specifies a metalanguage and a set of enumeration attributes. The driver is not required to guarantee that it will enumerate the devices for which it includes “enumerates” declarations, but it should only list devices that are highly likely, to avoid incurring excessive performance penalties.

The <min_num> and <max_num> values, represented as ASCII-encoded decimal numbers from zero to 2^{32}-1, indicate the expected range for the number of device instances of this type that will be enumerated per parent instance. <max_num> must be greater than or equal to <min_num>.

31.6.7 Multi_parent Declaration

One optional “multi_parent” declaration may be included:

```
multi_parent
```

The “multi_parent” declaration indicates that each instance of the driver may be bound to multiple parent instances (using either the same or different metalanguages); this is typically used for multiplexers. If a driver does not include “multi_parent” in its static properties, it is guaranteed to be bound to at most one parent per instance at any time.

31.6.8 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 31-3 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.
binding	static	Exactly one region of this type will be created by the environment for each driver instance, when that instance is created. The primary region must have this attribute value. This id the default value for this attribute.
binding	dynamic	Regions of this type are to be created only when parent or child bindings for this region index are performed. One region is created for each such binding. See the “parent_bind_ops” and “child_bind_ops” declarations.
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.

Table 31-3 Region Attributes

<region_attribute>	<value>	Meaning
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.
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.

31.6.9 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>.]
```

31.6.10 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 valid range for <msgnum> is $1..2^{16}-1$ (i.e. a 16-bit value with 0 being reserved as an illegal message number).

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 31-1. 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 modules themselves, for the purpose of tracing and logging, by specifying the desired `<msgnum>` as an argument to `udi_trace_write` or `udi_log_write`. These message strings may contain format codes as for `udi_snprintf`.

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.

31.6.11 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.

31.6.12 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.

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.

31.6.13 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 16.2, “Instance Attribute Names,” on page 16-1for 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`, any file used as a message file (see the “message_file” declaration), or any of the driver’s source files (see the “source_files” declaration) or module files (see the “module” declaration). Readable files must not be larger than 16 MB.

31.6.14 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 31-4 Custom Parameter Scope

Value of <scope>	Meaning
device	The parameter applies to each device instance independently, and is required for all device instances.
device_optional	The parameter applies to each device instance independently, and is optional.
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 16, *“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?”. The specification of <msgnum1> must be a single token and may using the encodings specified in Table 31-2.

<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 consists 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>. The specification of <msgnum2> must be a single token and may use the encodings specified in Table 31-2.

<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”. The specification of <msgnum3> must be a single token and may use the encodings specified in Table 31-2.

<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 | \
     only )  [ when <attr_value2> is <attr_name2> ]
```

<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 31-1, “Enumeration Attribute Value Encoding,” on page 31-13.

<default_value> provides the default value for a parameter, and is encoded as for <attr_value> in “device” declarations (see Table 31-1) except when <attr_type> is “string”. If the <attr_type> is “string” then the <default_value> and all other values specified for this parameter are ASCII encoded numeric values representing message numbers; if such attribute values are presented to a user, they shall be presented with the text of these message strings in the user’s current locale, but the C locale string shall be used when setting the actual attribute value (which will be of type UDI_ATTR_STRING).

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, which is only valid when <attr_type> is “ubit32”, indicates a range of choices beginning with <min_value>, incrementing by <stride>, until the value exceeds <max_value>. The any keyword indicates that any value appropriate for the attribute type may be used. The only keyword indicates that the only valid value is <default_value>; this is equivalent to “mutex <default_value> end”.

If the optional when clause is provided, then this “custom” declaration is only used when another custom parameter or enumeration attribute, named <attr_name2>, has the value indicated by <attr_value2>. The <attr_value2> value is encoded as appropriate for the type of <attr_name2>.

<value>, <min_value>, <max_value>, <attr_value>, <attr_value2> and <stride>, if provided, are encoded as shown in Table 31-1.

31.6.15 Config_choices Declaration

One or more “config_choices” declarations must be included for each <msgnum> used in a “device” declaration for a non-self-identifying bus (e.g. legacy ISA). The “config_choices” declaration is only supported for such devices.

```
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.

31.7 Build-Only Properties

The property declarations in this section apply only when building drivers or libraries from source. These declarations are stripped out by the udimkpkg utility program (see Section 33.3 on page 33-1) when it attaches static driver properties to binary object files for binary distributions.

31.7.1 Source_files Declaration

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

```
source_files { <c_source_file> }
```

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

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” or “.h”.

The “source_files” declaration is sensitive to ordering relative to “compile_options” declarations. See Section 31.7.2, “Compile_options Declaration”, for more details.

31.7.2 Compile_options Declaration

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

```
compile_options { <option> }
```

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

“Compile_options” declarations apply to any “source_files” declarations following the “compile_options” declaration, until the next “compile_options” declaration. Each “compile_options” decalaration overrides the preceding one.

If no “compile_options” declarations precede a particular “source_files” declaration, the corresponding source files will be compiled without any special compile options.

Valid compile options are listed in the following table.

Table 31-5 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 program (see Section 33.2 on page 33-1).

31.7.3 Source_requires Declaration

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

```
source_requires <interface_name> <version_number>
```

Each “source_requires” declaration is treated exactly like a “requires” declaration, except that it is used only when the driver or library is compiled/built from source.

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

31.8 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 0x100
supplier 1
contact 2
name 3
category 4
shortname xyznic
release 5 1.0b5
requires udi 0x100
requires udi_physio 0x100
requires udi_bridge 0x100
meta 1 udi_bridge
requires udi_network 0x100
meta 2 udi_network
parent_bind_ops 1 0 1
child_bind_ops 2 0 2
device 5 1 bus_type pci pci_vendor_id ubit32 1234 \
                    pci_device_id ubit32 19
custom driver media_type 10 11 0 string 12 \
                    mutex 12 13 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 "\m12" or
   "\m13".
message 12 Ethernet
message 12 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 "\m12" oryay
   "\m13".
message 12 Ethernetyay
message 12 Okentay Ingray
```

Packaging & Distribution Format 32

32.1 Overview

This chapter defines the UDI packaging and distribution format for both source and binary distributions of UDI drivers.

32.2 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 be provided with a driver to make it usable. This directory structure is then encapsulated in a "pax" archive to create a distributable UDI package. A single package can contain multiple drivers and/or libraries.

Each environment implementation shall provide a utility program called `udisetup`, that is used to install a driver once the UDI package has been somehow transported to the target system and possibly extracted from distribution media. This utility 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.

When presented with a distribution containing multiple independent drivers, it is implementation-dependent whether `udisetup` installs all of the drivers or prompts the user for the name of a driver package to install.

32.2.1 Directory Structure

In the directory structure shown below, all UDI files are contained under a standard top-level directory ("`udi-pkg.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 31, *"Static Driver Properties"*. If this component is being distributed in source form, the driver source and `udiprops.txt` are contained in the `src` subdirectory. If this component is being distributed in binary form, the driver binaries are contained in the `bin` directory, with the content of `udiprops.txt` embedded in the primary module's object file in an ABI-specific fasion.

Beneath the `bin` directory, the <`abi`> subdirectories contain the driver binaries built to conform to a particular Architected Binary Interface. The <`abi`> subdirectory names are defined in each ABI, with typically one subdirectory defined for binaries which can be used generically within the ABI, and an additional subdirectory defined for each supported processor subclass. For example, take a fictitious CPU architecture called, "3CPU", with specific processor models "3CPU1", "3CPU2" and so on. `bin`/`3CPU` would contain generic 3CPU binaries that would run on any processor in the family; `bin`/`3CPU1` would contain binaries specifically optimized for the 3CPU1 processors and might not even run on other processors in the family.

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.

The resulting package directory hierarchy for one component would look like the following:

```
udi-pkg.1/<drv_xxx>/<alt_yyy>/<comp_zzz>/
                                    src/udiprops.txt
                                        <source files>...
                                    bin/<abi>/<primary module file>
                                              <secondary modules>...
                                    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.

32.3 Archive Format

In order to create a UDI driver package, the entire package directory hierarchy described above is encapsulated in a "pax" archive, as defined in the IEEE Std. 1003.1-1988 "Archive/Interchange File Format", with relative pathnames. The resultant archive file, referred to as a *UDI package file*, can be given as an argument to the `udisetup` utility to install the driver on a target system.

All UDI build environments shall support a "`udimkpkg`" utility, which takes a collection of UDI driver files and creates a UDI package file.

32.4 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 one or more UDI package files (output from `udimkpkg`), stored on the media in a specified storage format at a specified location. In order to allow the target environment to locate the UDI package files, which may be interspersed with other files on the same media, UDI package files stored on physical distribution media must be placed (directly) in a top-level directory named "`/udi-dist.1`", referred to as the distribution directory.

Further, since some media formats are limited to 8.3 filenames (up to 8 characters, optionally followed by a dot ('.') and 1-3 additional characters), UDI package files stored on physical distribution media must be named "`xxxxxxxx.udi`", where "`xxxxxxxx`" is replaced by 1-8 characters chosen by the driver developer (or distributor). There must be no other files ending with the "`.udi`" extension in the distribution directory, except other UDI package files.

32.4.1 Floppy Storage Format

For floppy disks, the storage format shall 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 distribution directory, "`udi-dist.1`", must be placed in the root of the DOS filesystem.

32.4.2 CD-ROM Storage Format

For CD-ROMs, the storage format shall be an ISO-9660 filesystem. (RockRidge extensions are allowed, but not required since the UDI package file names will fit in 8.3). The UDI distribution directory, "`udi-dist.1`", must be placed in the root of the directory hierarchy identified by the Primary Volume Descriptor.

Build & Packaging Utility Programs 33

33.1 Overview

This chapter describes utilities that UDI build environments and UDI runtime environments must provide. The build environment consists of the tools and utilities used to create driver binaries (the `udibuild` utility) and to create driver source and/or binary installation packages (the `udimkpkg` utility). The runtime environment consists of the tools necessary to install driver packages (the `udisetup` utility), as well as the software modules necessary to configure and execute UDI drivers.

Many environments will be a combination of both build and runtime environment.

33.2 The udibuild Utility

All UDI build environments, and all runtime environments that support source distributions, shall include a utility called `udibuild`. This utility, when invoked with no command-line arguments, will search the current working directory for a `udiprops.txt` file and source files, then attempt to build a driver binary using build rules from `udiprops.txt`, placing the resultant binary object modules in the same directory.

The behavior of `udibuild` when given any command-line arguments is implementation-dependent. The command "`udibuild -?`" shall cause a usage message to be displayed, which will list the parameters specific to this implementation.

33.3 The udimkpkg Utility

All UDI build environments shall include a utility called `udimkpkg`. This utility uses the static driver properties file, `udiprops.txt`, to find the various pieces of a UDI driver or library component (including message files, readable files, as well as source and/or object files), and gathers them into a UDI package file, ready for distribution or installation. For binary distributions, `udimkpkg` also attaches the content of the `udiprops.txt` file to the primary binary module (typically a relocatable object file) of the driver, in an ABI-specific fashion, stripping out any build-only properties in the process.

The output of `udimkpkg` is a UDI package file, as described in Section 32.3, "Archive Format," on page 32-2. Additional environment-specific tools may be required to copy this package file 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 UDI package file in the current working directory, named "`nnnnnnnn.udi`", where "`nnnnnnnn`" is replaced by the `<name_string>` from the "shortname" declaration in `udiprops.txt`.

When stored in the UDI package file, the component must have associated *driver*, *alternate*, and *component* names. By default, `udimkpkg` takes these from the "shortname", "release", and "module" declarations, respectively, from `udiprops.txt`. (If there are multiple "module" declarations for a component, the first one is used.) All `udimkpkg` utilities must also provide a way to specify other values for each of these names, but the command syntax for doing so is implementation-defined, as is the syntax for selecting source vs binary distributions. By default, `udimkpkg` will include both source and binary files in the package, if both are present in the input directory.

For binary distributions, `udimkpkg` will use a build environment default value for the `<abi>` sub-directory name, but shall provide an implementation-defined way to override this default.

The `udimkpkg` utility may also support additional implementation-defined arguments. The command "`udimkpkg -?`" shall cause a usage message to be displayed, which will list the parameters specific to this implementation.

33.4 The udisetup Utility

All UDI build environments shall include a utility called `udisetup`. This utility extracts driver files from the UDI package file, installs them in environment-specific locations, and prepares the driver for use on the target system.

With no command-line arguments, `udisetup` shall search the current working directory for all files ending in "`.udi`" or "`.UDI`", and either install all of the drivers in all of the packages, or provide the user a way to interactively pick and choose amongst them if there are more than one. With a single command-line argument, `udisetup` shall interpret that argument as the filename of a single UDI package file to use, and either install all of the drivers in the package, or provide the user a way to interactively pick and choose amongst them if there are more than one.

`udisetup` may have additional implementation-specific parameters for items such as distribution package location, driver versions and alternates to be installed, and so forth. The command "`udisetup -?`" shall cause a usage message to be displayed, which will list the parameters specific to this implementation.

UDI Core Specification

Section 8: ABI Bindings

Introduction to ABI Bindings *34*

34.1 Introduction

Most of the UDI specification documents (see **Chapter 2, *"Document Organization"***) define programming interfaces and conventions that are applicable to any processor or platform architecture; these specifications, referred to as *source-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, that 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. The ABI Binding specifications are also refered to as *binary-level specifications*.

A UDI ABI binding consists of the following components:

1. A processor architecture, instruction set definition, and endianness. This defines the supported instruction sets, and endianness of the compiled UDI driver, as well as corresponding subdirectory names for the UDI packaging format.
2. Runtime architecture. This defines the procedure calling conventions, register usage, stack conventions, data layouts, etc.
3. Binary bindings to the source-level UDI specifications. This specifies the sizes of fundamental UDI data types, the binary-portability requirements of *implementation-dependent UDI macros* and functional interfaces, and other miscellaneous binary bindings related to the source-level specifications.
4. Building the driver object. This specifies the object file format, and the encapsulation of the static driver properties in the object files.

34.2 Processor Architecture

Each ABI binding specification must specify a processor architecture with its associated instruction set(s) 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.

Second, each ABI binding must specify the specific subclasses of processor architectures that are supported; e.g., an IA-32 binding might specify support for 386, 486, P1, P2, and P3 processors.

Third, the ABI binding must specify the <abi> subdirectory names for use in the UDI packaging format specification (see Section 32.2.1, "Directory Structure," on page 32-1); there should generally be one generic subdirectory defined for driver binaries that are useable across the ABI's processor subclasses, and an additional subdirectory for each processor subclass. For example, an IA-32 binding might specify

a subdirectory named `IA32` for generic IA-32 binaries, and might specify additional subdirectories named `IA32_386`, `IA32_486`, `IA32_P1`, `IA32_P2`, and `IA32_P3` for binaries specific to a given IA-32 processor subclass.

34.3 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.

34.4 Binary Bindings to the Source-Level Specifications

A few aspects of the *source-level specifications* are implementation-dependent and thus require additional specification for binary portability. These fall into four general categories: (1) the sizes of fundamental UDI data types, (2) the binary-portability requirements of implementation-dependent UDI macros, (3) the specification of UDI functional interfaces (other than the UDI utility functions) that are allowed to be implemented as macros, and (4) other miscellaneous binary bindings related to the source-level specifications.

Only the UDI Core Specification and the UDI Physical I/O Specification are allowed to contain fundamental UDI data types and *implementation-dependent macros*. Furthermore, only these two Specifications are allowed to have functional interfaces that can be implemented as macros, or to have any other aspects that require binary bindings to source-level specifications. Therefore, the only two source-level specifications which require bindings for binary portability are the UDI Core Specification and the UDI Physical I/O Specification. This allows the ABI bindings to be specified independently from the existence or the evolution of other source-level specifications such as metalanguages and bus bindings.

Utility functions in UDI can always be implemented as macros without breaking binary portability; see the Utility Functions Chapter for details.

Note – The values for symbolic constants are all defined in the source-level specifications, and thus require no specification in the ABI bindings.

34.4.1 Sizes of UDI Data Types

As part of its C language binding, UDI defines a set of fundamental data types (see Chapter 9, *"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, opaque types, and pointer types ("void *" as well as specific pointer types) have implementation-dependent sizes. 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 34-1 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_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 – Only the UDI Core Specification and the UDI Physical I/O Specification are allowed to define UDI fundamental data types.

34.4.2 Implementation-Dependent Macros

In UDI, an *implementation-dependent macro* is an interface that is defined to be a macro in a source-level specification (other than utility macros), but whose macro expansion definition is implementation-specific. Such macros may contain environment and platform dependencies which, to provide binary portability, must be hidden behind an external function call; any such external symbol references must be specified in the ABI.

Only the UDI Core Specification and the UDI Physical I/O Specification are allowed to specify implementation-dependent macros. In the 1.0 version of these Specifications two implementation-dependent macros are specified: the UDI_HANDLE_ID macro on page 9-25 and UDI_HANDLE_IS_NULL macro on page 9-24.

Each ABI must specify the binary-portability requirements of each macro. This means that the ABI must specify whether or not the macro produces any external symbol references and, if so, the names and semantics of those external symbols. The ABI must also specify the behavior of the inline portion of the macro, in an environment-neutral fashion.

34.4.3 UDI Functions implemented as macros

As previously noted, utility functions in UDI can always be implemented as macros without breaking binary portability. However, ABI bindings must require each other UDI functional interface to be implemented in one of two ways. One way is to implement the UDI functional interface as an external

function call. Another way is to optionally implement the interface as a macro with any environment or platform dependencies hidden behind an external function call. This external function call is specified by the ABI as part of the macro expansion.

A common example of a UDI functional interface that is implemented as a macro is the `udi_assert` interface. When `udi_assert` is implemented as a macro, any environment or platform dependencies must be hidden behind a call to an external function specified in the ABI. This allows the performance path to be done inline, only taking the overhead of a function call in the exception case.

34.4.4 Miscellaneous Binary Bindings

Each ABI must specify the following binary bindings related to the source-level specifications or the UDI architectural model:

- Maximum stack usage per call into the driver.
- Maximum instruction (code) size per driver object module, and per driver.
- Maximum static data size per driver object module, and per driver.
- Maximum binary object file size per driver object module.

34.5 Building the Driver Object

The Core Specification defines the generic aspects of building UDI driver object code. Each 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.

34.5.1 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.

34.5.2 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 31, *"Static Driver Properties"*) are attached to the driver's object files.

UDI Core Specification

Section 9: Appendices

Glossary *A*

anchored channel a channel end that 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 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 ***bind process***.

buffer an opaque object used to carry "application" or "wire" data within the UDI environment. A UDI buffer is logically contiguous but may be virtually or physically segmented.

buffer handle an opaque reference to a UDI buffer.

buffer tag a tag associating a type code and a value with a particular range of data in a logical buffer, which moves with the data in the presence of insertions and deletions prior to the tagged range. If data associated with a buffer tag is modified or deleted, the tag is removed and discarded. Buffer tags are often used for network checksums.

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 `foo` requests a service, e.g., allocation of a block of memory. This service request specifies the address of procedure `foo_callback`, 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, `foo_callback` 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 *channel operations* in another region. Example of channels include the *bind channel* between an adapter driver instance and its parent driver instance, and the *management channel* 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 is passed to the driver as the first parameter for every channel operation that 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.

channel handle 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.

channel operation 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”.

channel operations vector 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.”

child driver instance 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.

communications channel see ***channel***.

control block 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 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. See also ***generic control block*** and ***metalanguage-specific control block***.

destructive diagnostic request an operation that may have effects external to the driver to which the request was directed.

external mapper 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.

handle see ***opaque handle***.

IMC 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

implicit synchronization 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.

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.

logical buffer see ***buffer***.

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 ***Management Agent***.

Management Agent the agent or set of cooperating agents within the environment that 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 a UDI software-only driver that maps one metalanguage to another, for example a Fibre Channel to SCSI mapper. A mapper is 100% UDI code, unlike an ***external mapper***. Also known as an ***internal mapper***.

marshalling see parameter marshalling.

MEI Metalanguage-to-Environment Interface. Defines the interfaces needed to implement portable metalanguage libraries. See Chapter 28, *"Introduction to MEI"*.

metalanguage the communication protocol used by two or more cooperating modules. A metalanguage includes interface definitions for associated channel operations, control block structures, and service calls, as well as bindings to the use of UDI trace events and the definition of various types of UDI instance attributes. E.g., the SCSI Metalanguage is used for communication between SCSI peripheral drivers and SCSI HBA drivers; and the USBDI Metalanguage is used for communication between USB peripheral drivers and the USBD driver layer. When refering to a metalanguage used by a particular type of driver the adjectives "top-side" and "bottom-side" are sometimes applied: e.g., the SCSI Metalanguage is the top-side metalanguage for SCSI HBA drivers; the USBDI Metalanguage is the bottom-side metalanguage for USB peripheral drivers.

module A set of ISO C routines that 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.

opaque 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 a domain-independent, protected reference to an object. UDI handles are all region-local handles; i.e., they are only useable in the context of the caller's region and may require translation if transferred to another region (see ***transferable handle*** and ***nontransferable handle***). See also the definitions for specific types of handles: e.g., ***buffer handle***, ***channel handle***, and ***constraints handle***.

operation See *channel operation*.

parameter marshalling taking parameters of a channel operation and saving them in a storage area, such as in a *control block*. 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 *server* in this relationship.

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 directly visible to UDI drivers.

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 that is shared between drivers and the environment but may contain environment-private data that 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 *control block* 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 a channel operation. The operation is sent to the other end of the target channel. The target channel is passed to the channel operation via the ***channel*** member of the control block.

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.

visible fields those members of the C structure representing the driver-visible part of a semi-opaque object.

Index