Uniform Driver Interface

UDI Network Driver Specification Version 0.90

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

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

Table Of Contents

List of Reference Pages by Chapter

Chapter 1 Network Interface Metalanguage

Alphabetical List of Reference Pages

UDI Network Driver Specification

Abstract

The UDI Network Driver Specification defines the required interfaces and semantics for UDI environments that support Network Interface Card (NIC) drivers. This is an optional extension to the UDI Core Specification, which is defined in a separate book. See the Document Organization chapter in the UDI Core Specification for a description of the other books in the UDI Specification, as well as references to additional tutorial materials. The intended audience for this book includes driver writers, environment implementors, and metalanguage implementors.

Network Interface Metalanguage *1*

This specification details the channel operations, parameters and sequences for the UDI Network Interface Metalanguage which is used to support Network Interface (*NIC*) Adapters.

Each subsection defines the channel operation calls, control structure type declarations, the rationale for the operation's existence, constraints and guidelines for the use of each operation, and error conditions that can occur.

1.1 Overview

The networking device driver framework in UDI defines a set of channel operations and rules which allow for writing a NIC driver that works with existing and future networking protocol stacks regardless of the OS and protocol stack characteristics. This specification describes both the UDI *NIC Driver* (ND) and the UDI *Network Service Requester* (NSR) which communicate via the Network Interface Metalanguage. The ND is completely portable within the UDI environment and is intended to accompany the distribution of the NIC hardware. The NSR is typically provided by the UDI environment implementation or via additional UDI modules. The Network Interface Metalanguage completely defines the interface between the two components.

The UDI Network Interface Metalanguage was developed to represent a universal set of connectionless network-related functions that provide all of the needed functionality in an OS, protocol, and transport independent manner.

1.1.1 General Requirements

Before including any UDI header files, the driver must define the preprocessor symbol, UDI_NET_VERSION, to indicate the version of the UDI Network Interface Metalanguage to which it conforms. For this version of the specification, UDI_NET_VERSION must be set to 0x090:

```
#define UDI_NET_VERSION 0x090
```

The channel ops, types, and definitions defined in this specification may be found in the `udi_net.h` header file which should be included (along with `udi.h`) in all ND or NSR modules The `udi_net.h` header file must be included after the version definition above and after the inclusion of `udi.h`, although it may follow or preceed other UDI header files:

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

1.1.2 Documentation Conventions

Throughout this document, the following conventions will be used:

- Operations of type “_req” are used to request an activity.
- Operations of type “_ack” are used to acknowledge the request (but not necessarily the success or failure in implementing that request).
- Operations of type “_ind” are used for indications of an event, typically asynchronous in nature.
- The term *ND* is used to refer to the driver supplied by the adapter manufacturer which is used to manage and operate that adapter.
- The term *NSR* is used to refer to the agent which handles network operations on behalf of the OS and interfaces with the ND. The NSR may be a mapper or another UDI module.

1.1.3 Normative References

The UDI Network Interface Metalanguage is designed to support a wide variety of network topologies. Although there is no single standard or reference for this implementation, the designer is expected to be familiar with the appropriate network protocol standards and technology specifications that apply to the implementation being developed. This includes International Standards Organization (ISO), American National Standards Inc. (ANSI), and Internet Engineering Task Force (IETF) publications for general networking standards as well as technology-specific standards from specific groups such as the ATM Forum, the ANSI X3T11 Fibre Channel committee, the various 802.x standards committees for Ethernet and FDDI, and other organizations and specifications as appropriate.

1.2 Network Interface Metalanguage Environment

The UDI Network Interface Metalanguage environment may be implemented in one of two possible configurations as shown in Figure 1-1: either UDI protocol support or a mapper to native OS protocol stacks. The implementation chosen and the associated details are determined by the provider of the UDI environment and do not affect the UDI NIC Driver implementation.

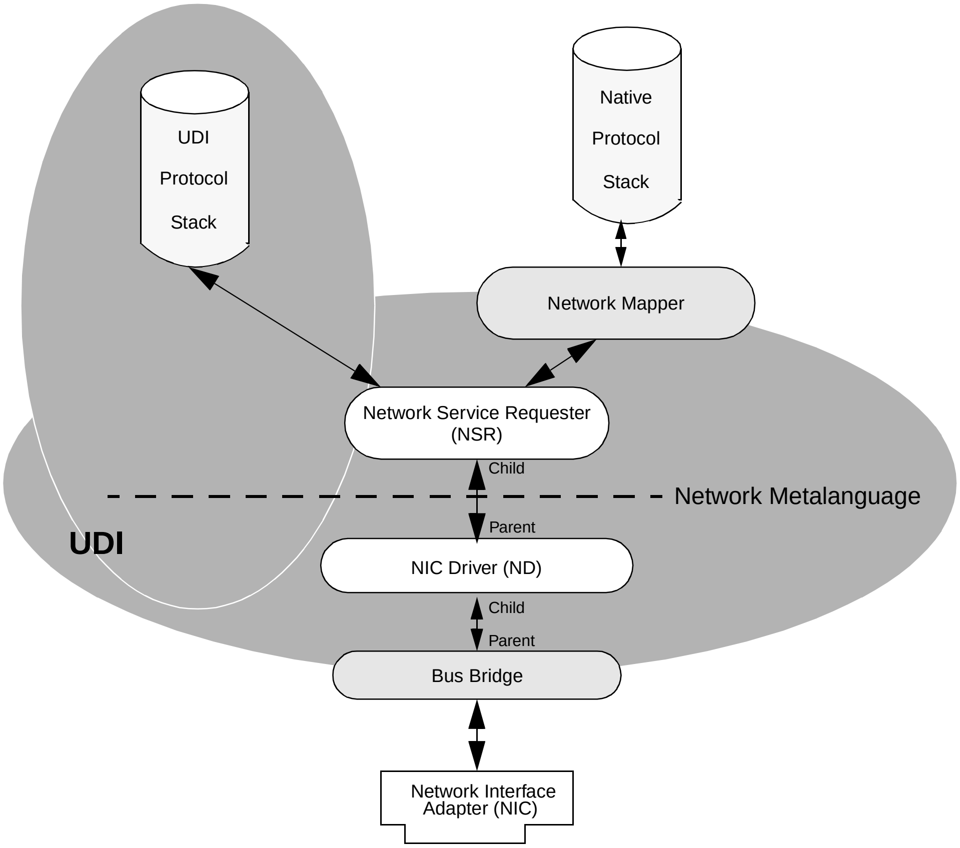

Figure 1-1 UDI Networking Environment

In the first configuration scenario, the protocol stack is implemented directly within the UDI environment. It is attached directly to the UDI ND in the customary UDI parent-child relationship. It implements the transport protocol mechanism using native UDI services (e.g. buffer services, timer services, etc.) and scheduling. The Protocol Stack operates as the Network Service requester (NSR) in this configuration, communicating with the ND via the Network Interface Metalanguage.

The second configuration scenario is used when the transport protocol mechanism is implemented in a manner native to that particular OS (e.g. Streams/DLPI, BSD ifnet/2, NDIS, ODI, CDLI, NDI, etc.). In this scenario the protocol stack is independent of UDI and does not run under the UDI paradigm and an additional component, the UDI Network Mapper is provided to act as the NSR agent on behalf of the native protocol stack. The UDI environment implementation must support the Network Interface Metalanguage by implementing one or the other of these support scenarios.

Please note that there are a number of “flow” diagrams presented throughout this chapter. They are intended to provide examples of how an existing Streams/DLPI protocol stack interface would be mapped to the corresponding Network Interface Metalanguage operations. The UDI Network Interface

Metalanguage can be implemented as an addition to or as a replacement for any existing network adapter interface, so the examples shown here should not be interpreted as the only possible implementation of this metalanguage.

1.2.1 Network Service Access

The UDI Network Adapter Metalanguage, as with many network driver architectures, defines that architecture in terms of a client-server arrangement. In terms of this arrangement, the ND provides various “services” to the protocol stack.

There may be multiple protocol stacks or other modules utilizing an ND driver at any single time, depending on the system configuration; these are all channeled through the NSR to provide a single access point between the ND and the NSR for all network traffic. Correspondingly, there may be multiple ND’s providing network access services to the protocol stack or stacks within the environment.

The services provided by a ND and their parameters are described by the UDI Network Interface Metalanguage. The way in which these services are used by the protocol stack is not described by the Metalanguage and is a prerogative of that protocol stack and accompanying environment.

One example of network access is an Ethernet datalink path between an Ethernet Adapter’s ND and the TCP/IP protocol stack via the NSR. TCP/IP packets are presented to the ND through Network Metalanguage channel; the NSR will prepare the packet format for transmission and build the packet headers and the ND will perform the appropriate actions to post the buffer to the Adapter’s MAC chip to transmit it on the network. Incoming packets are verified, the Ethernet header is removed, and the packet is passed to the NSR over the Network Metalanguage channel. The NSR then interprets the frame header and removes it and the subsequent data buffer is passed to the TCP/IP protocol stack agent corresponding to the Ethertype in the Ethernet header.

1.2.2 Network Interface Multiplexing

As defined, the Network Interface Metalanguage provides an interface to the NIC driver that is nearly independent of the network protocol type. The NIC Driver’s primary responsibility is sending and receiving network packets via the associated hardware.

The NSR provides the network header creation and manipulation operations as well as multiplexing between the NIC and the various protocol stacks. The NIC therefore can be implemented very simply since it only manages one hardware device and passes packets to one NSR.

In this paradigm, it is expected that an OS implementing multiple upper-layer protocols (e.g. IP, IPX, ARP, Appletalk, etc.) over a single network interface will attach these protocols to the NSR for header manipulation and incoming packet filtering and distribution. The NIC Driver will have only the single connection to the NSR for all of these simultaneous protocols. Accordingly, the only incoming packet filtering performed by the NIC Driver is to only pass up packets intended for the local host (unless promiscuous mode is set, in which case ALL packets should be passed to the NSR).

Only one NSR may connect to a NIC Driver at any time. If a NIC Driver instance is bound via the Network Metalanguage to a specific NSR child, no other Network Metalanguage bind operations will be issued to that NIC Driver instance until the existing NSR child is unbound.

In addition, any Generic Metalanguage channels bound to a NIC Driver will be used in a manner defined by that NIC Driver and the associated client. However, it is customary for the NIC to refrain from passing any received packets to the Generic Metalanguage unless specifically requested via a Generic Metalanguage control operation.

1.2.3 Network Addressing

The Network Service requester (NSR) is responsible for determining the network (MAC) destination address for transmit packets and for building the datalink header (*e.g*. Ethernet V2, 802.2, SNAP) in the packet before delivering the packet to the ND for transmission. On receives, the ND will validate any addressing information (if necessary) and then pass the entire packet to the NSR. The NSR will parse and remove the datalink header(s) from the packet and then pass it to the appropriate protocol stack entity, thereby providing the receive demultiplexing functionality.

When the Network Interface Metalanguage binding between the NSR and the ND is performed, the ND will indicate the media type to provide basic framing information to the NSR. The NSR will use this information to generate the proper device or technology headers for transmit operations and to correspondingly interpret the receive headers.

1.2.3.1 Address Specifications

During normal operations, it isn’t necessary for the NSR and ND to exchange address information; the initial indication of the media type allows both the NSR and ND to parse the packet buffers to obtain or set MAC addresses as needed. In the event that it is necessary for the NSR and ND to exchange MAC addresses directly, they will use explicit fields in the associated operation control blocks to do so. Some examples of situations requiring the exchange of addresses are: ND reporting of current/factory MAC address to the stack, and NSR specification of multicast addresses to be accepted.

In order to allow the Network Interface Metalanguage to support a wide variety of media types, the size of the MAC addresses is not restricted to the traditional 6 bytes. Instead, the size of the MAC address will be explicitly stated when exchanging these addresses, subject to the maximum size of `UDI_NET_MAC_ADDRESS_SIZE`[1]. Within the context of the ND, it may be reasonable to assume a MAC address length relating to the hardware in question and the ND may be written using this assumption, but in exchanges with the NSR, the size of the MAC address must be explicitly specified.

The current value of UDI_NET_MAC_ADDRESS_SIZE is 20 bytes, which allows complete specification of E.164 addresses.

1.2.3.2 Address Implementations

This metalanguage specification uses the term “MAC” (Media Access Control) address to refer to the addresses which the ND and NSR will both evaluate. While this term is used extensively in conventional LAN implementations (e.g. Ethernet or FDDI), it usage is somewhat more vague in non-LAN or extended functionality protocols such as ATM or Fibre Channel. The contents of Table 1-1 on page 6 identifies media types and the recommended MAC address for those media. Media types not specifically identified in the table should interpret the MAC address to correspond to the static node identifying

[1] The UDI_NET_MAC_ADDRESS_SIZE specification is defined as part of the UDI specification and may not be tailored by the individual UDI environment implementations. The value of this definition may only change as part of the evolution of this specification.

address that ARP would typically use to map an IP address. A NIC driver is not required to use the address values in the table below but it should be prepared to detect and process the contents of any address resolution protocol packets (*e.g.* ARP) if it does not use the conventional addressing format.

Table 1-1 Media Types and MAC Address Lengths

Media Type	**Description**	**MAC Address**
UDI_NET_ETHER	10 Mbit Ethernet	6-byte MAC
UDI_NET_TOKEN	Token-ring Media	6-byte MAC address
UDI_NET_FASTETHER	100 Mbit Ethernet	6-byte MAC address
UDI_NET_GIGETHER	1 Gbit Ethernet	6-byte MAC address
UDI_NET_VGANYLAN	100 MBit 100VG-AnyLAN	6-byte MAC address
UDI_NET_FDDI	100 Mbit FDDI	6-byte canonical MAC address
UDI_NET_ATM	25 Mbit, 155 Mbit, or 622 Mbit ATM	Up-to 20-byte E.164 node address
UDI_NET_FC	1 Gbit (or greater) Fibre Channel	8-byte WWN
UDI_NET_MISCMEDIA	Unknown or unspecified media type	6-byte MAC

1.2.4 Network Services

The network services defined by the UDI Network Interface Metalanguage specification which are provided by the ND may be divided into two areas: control operations and transfer operations. Each type of operation is handled by one or more UDI channels between the ND and the NSR. The UDI channels are specific to the type of operation associated with that channel (e.g. it is an error to send control operations over a transfer channel).

1.2.4.1 Control Operations

The control operations specified by the UDI Network Interface Metalanguage are used to specify the binding and to handle protocol-level control operations such as hardware MAC address registration, statistics, multicast addressing, etc.

1.2.4.2 Transfer Operations

The transfer operations handle simple datagram packet transmission and reception operations.

When the ND is provided with a packet buffer to transmit, it will also be given a buffer containing the data to transmit including any protocol and frame level headers. If the device does not require any special handling of the header information, the packet is simply queued to the device. If header translation or manipulation is required, the ND is free to perform these operations before queueing the packet for transmission. When the packet transmission has completed, the ND will indicate this completion to the NSR.

When the ND receives a packet from the line, it will examine the link-level headers and trailers to determine both the source and destination addresses. The destination address will be verified as the link-level MAC address of the ND's interface or as an appropriate multicast or broadcast address (either perfect or hashed). Packets failing this verification will be discarded. The ND will then pass the packet to the NSR via a Network Interface Metalanguage receive channel. The ND may chain multiple packets together for receive indications in a similar manner to the transmit packet chaining described above; the NSR may divide up the chain however it desires to acknowledge these receives in any sequence or sub-chain it desires.

The NSR is responsible for evaluating the link level header information to determine which local protocol stack agent the message should be directed to; if no local stack agent matches the link level addressing information, the NSR will discard the packet.

1.2.4.2.1 Transmit Flow Control

Transmit flow control is implemented by selective supply of transmit control blocks to the NSR by the ND. When the network connection is enabled, the ND will allocate a number of transmit control blocks and then provide these to the NSR. The NSR can then use these for transmit requests. The NSR may not internally allocate any additional transmit control blocks; it must wait for the ND to return transmit control blocks after transmission completes if it exhausts its supplied pool of control blocks.

The transmit control block also implements a completion urgency hint which is used to indicate to the ND the relative urgency of completing the transmit quickly. If not marked to indicate an urgency desire, the ND is free to complete the transmit at any point in the future that it desires. If urgency is indicated, it typically means that the packet buffer has been loaned to the networking subsystem by another OS module (e.g. NFS) and that the buffer should be returned as soon as possible. This indicator allows the ND to optimize transmit completion interrupts as desired. Note that even if no urgency is indicated, the ND should insure that it quickly indicates a transmit completion if the NSR has passed all transmit control blocks to the ND, otherwise the flow control mechanism will prevent any further transmit requests.

To reduce overhead and allow batch optimizations in the ND, the NSR may pass a chain of multiple packets as part of a single Network Interface Metalanguage transmit request. Each packet should be transmitted in turn and when the transmission completes, the transmit buffer is deallocated and the transmission is acknowledged to the NSR. The ND may acknowledge on a packet-by-packet basis or on the basis of all (or part) of the original chain of packets. The driver is free to divide the chain into as many pieces as desired, as is the NSR or the UDI environment.

1.2.4.2.2 Receive Flow Control

Receive flow control is implemented in essentially the same manner as transmit flow control but in the reverse direction. The NSR will provide the ND with a supply of receive control blocks and associated buffers into which the received data is to be placed. Once a packet has been received into the buffer, the ND will pass the receive control block back to the NSR. The ND will never internally allocate receive control blocks and will instead wait for the NSR to supply them. As a result, the ND is not expected to retain any packets received while there are no available receive control blocks (although it may optionally queue them internally at its prerogative).

1.2.4.3 Transfer Channels

In order to allow separate handling of receive and transmit operations, the UDI Network Interface Metalanguage defines two separate transfer channels for the connection between the NSR and the ND: the transmit channel and the receive channel. Both channels may be handled by a single driver region or if desired they may each be handled by a separate region within the driver instance.

In addition, each channel provides channel operations for both normal and expedited data. This allows the NSR and ND to differentiate expedited, out-of-band, or other non-standard QOS data from the normal in-band data being transferred. Protocols which support this model of data transfer (e.g. 100VG-AnyLAN) can use different channel operation routines to handle the different traffic types separately. For protocols which do not support this differentiation, the normal and expedited channel operations may point to the same routine for either the ND or the NSR.

1.2.5 Configuration Parameters

Different network technologies have associated parameters and modes of operation which are unique to that network technology (e.g. auto-negotiation for speed). Additionally, many network adapters have varying degrees of support for network-specific functionality in the hardware itself. To attempt to capture all of these variations in this metalanguage as part of the description of the communication between the NSR and the ND would be impossible and arbitrarily restrict current and future innovation. Additionally, this level of functionality is really only used for adapter hardware configuration and the results of this configuration are not interesting to nor do they affect the NSR and the general communications between the NSR and the ND.

Therefore, these types of adapter-specific configurations are not defined as part of this metalanguage but instead should be implemented as custom attribute declarations as part of the driver's static configuration properties (see the "Custom Declaration" section of *"Static Driver Properties"* in the *UDI Core Specification Version 0.90*). In this way, the configuration parameters can be defined on a per-driver basis with defaults and values provided by the manufacturer of the adapter as appropriate to that adapter. Example configuration parameters of this type are:

- Full or Half Duplex Configuration
- Link Speed setting (including auto-negotiation)
- Port Type selection (e.g. BNC, AUI, TP)

To change the any of these configuration parameters, the system administrator should update the values for these parameters in the system's persistent storage database to the newly desired values. The ND should re-scan the persistent storage database for any configuration parameter changes each time a link enable request is received, thereby allowing the system administrator to verify that the parameters have taken effect by (re)enabling the interface.

1.2.6 Hardware Checksum Offloads

Many hardware adapters provide the capability of calculating and/or verifying network packet checksums via offload capabilities of the hardware. This can significantly improve performance for networking operations and unlike the configuration parameters discussed above this capability does affect the operation of the NSR and its children. However, hardware checksum offloads are implemented in different manners and some adapters have different offloading capabilities than others.

To support hardware checksum offloading, the ND should utilize buffer tags[2] and the buffer tag assist functions: `udi_buf_tag_compute` and `udi_buf_tag_apply`.

[2] See the the "Buffer Tags" section of *"Buffer Management"* of the *UDI Core Specification Version 0.90* for more information on buffer tags.

For a transmit operation, the NSR (or its children) will set various checksum set-request tags (e.g. UDI_BUFTAG_SET_iBE16_CHECKSUM, UDI_BUFTAG_SET_TCP_CHECKSUM, and UDI_BUFTAG_SET_UDP_CHECKSUM) for the data buffer to indicate what region of the buffer requires a checksum. If the hardware supports checksum offloading, it should use these tags to program the hardware accordingly. If the hardware does not support checksum offloading, the ND should use the udi_buf_tag_apply utility function to manually calculate and set the checksum(s) in the outgoing packet.

For received packets, if the hardware has performed checksum calculation on the incoming packet, that checksum value should be set via one or more UDI_BUFTAG_BE16_CHECKSUM tags on the received buffer to allow the NSR (or its children) to easily obtain the calculated buffer checksum. The NSR will use the udi_buf_tag_compute utility function to determine the checksum for received packets; the udi_buf_tag_compute utility will make use of any existing UDI_BUFTAG_BE16_CHECKSUM tags to avoid performing manual calculations, however, those tags are not required which effectively supports the case where the adapter hardware does not provide incoming packet checksum calculation.

If the hardware also (or instead) performs received packet checksum validation (i.e. it calculates the incoming packet's checksum as described above and verifies it against the checksum value(s) written by the transmitting host) it should indicate the success or failure of the validation via one or more of the following buffer status tags attached to the corresponding portion of the buffer data:

- UDI_BUFTAG_TCP_CKSUM_GOOD
- UDI_BUFTAG_UDP_CKSUM_GOOD
- UDI_BUFTAG_IP_CKSUM_GOOD
- UDI_BUFTAG_TCP_CKSUM_BAD
- UDI_BUFTAG_UDP_CKSUM_BAD
- UDI_BUFTAG_IP_CKSUM_BAD

Packets which have had this checksum validation performed may or may not additionally have the UDI_BUFTAG_BE16_CHECKSUM tag set but the ND should set this tag whenever the corresponding value is known.

1.3 Network Management Operations

These operations are intended to configure the NIC Driver (ND) and the Network Service requester (NSR) and to establish a binding between these two entities. The following Network Management Operations have been defined:

- UDI Initialization and Registration
- Network Control Block Management
- Network Metalanguage Binding
- Network Adapter Control

This section assumes that the reader is familiar with the UDI Management Metalanguage and the UDI configuration process.

1.3.1 Initialization and Registration Operations

The initialization and registration operations are used to prepare the driver and environment for support of the Network metalanguage and the corresponding Network Adapter Driver (ND). These operations are performed in the `init_module` routine of the ND or NSR.

1.3.2 Channel Ops Vector Registration

The channel operations vector registration functions are used to register the driver entry points for the parent driver (ND) and/or the child driver (NSR network interface mapper or protocol module). The ND performs registration in its *`init_module`* routines by calling one or more of:

- `udi_nd_ctrl_ops_init`
- `udi_nd_tx_ops_init`
- `udi_nd_rx_ops_init`

The NSR performs registration in its *`init_module`* routines by calling:

- `udi_nsr_ctrl_ops_init`
- `udi_nsr_tx_ops_init`
- `udi_nsr_rx_ops_init`

The primary function used for network metalanguage channel creation and manipulation is specified by the `udi_nxx_ctrl_ops_init`. The primary channel created between the ND and the NSR will use these binding operations to establish the network metalanguage data transfer channels between the ND and the NSR.

NAME

udi_nd_ctrl_ops_init *Register ND control ops entry points*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nd_bind_req_op_t *nd_bind_req_op;
     udi_nd_unbind_req_op_t *nd_unbind_req_op;
     udi_nd_enable_req_op_t *nd_enable_req_op;
     udi_nd_disable_req_op_t *nd_disable_req_op;
     udi_nd_ctrl_req_op_t *nd_ctrl_req_op;
     udi_nd_info_req_op_t *nd_info_req_op;
} udi_nd_ctrl_ops_t;

void udi_nd_control_ops_init (
     udi_index_t ops_idx,
     udi_nd_ctrl_ops_t *ops );
```

ARGUMENTS

ops_idx, ops are standard arguments described in the “Channel Ops Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

`udi_nd_ctrl_ops_init` is called by the driver’s *`init_module`* routine to register the ND’s entry points for its Networking Metalanguage control channel.

When a driver is loaded into a given domain, UDI will call the driver’s `init_module` entry point. The `init_module` entry point is expected to make calls to register the entry points for each type of channel needed for each metalanguage supported by the driver and associate those entry points with the ops index supplied by the driver. This function performs the registration for the Network Driver’s control operations.

WARNINGS

This call is only made by the `init_module` routine of the driver at load time.

NAME

udi_nd_tx_ops_init *Register ND transmit ops entry points*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nd_tx_req_op_t *nd_tx_req_op;
     udi_nd_exp_tx_req_op_t *nd_exp_tx_req_op;
} udi_nd_tx_ops_t;

void udi_nd_tx_ops_init (
     udi_index_t ops_idx,
     udi_nd_tx_ops_t *ops );
```

ARGUMENTS

ops_idx, ops are standard arguments described the “Channel Ops Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

The `udi_nd_tx_ops_init` function is called only by the driver’s `init_module` routine to set up the vector of entry points for the data transmit handling channels of all instances of the driver in a domain.

WARNINGS

This call is only made by the `init_module` routine of the driver at load time.

NAME

udi_nd_rx_ops_init *Register ND receive ops entry points*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nd_rx_rdy_op_t *nd_rx_rdy_op;
} udi_nd_rx_ops_t;

void udi_nd_rx_ops_init (
     udi_index_t ops_idx,
     udi_nd_rx_ops_t *ops );
```

ARGUMENTS

ops_idx, ops are standard arguments described the “Channel Ops Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

The `udi_nd_rx_ops_init` function is called only by the driver’s `init_module` routine to setup the vector of entry points for the data receive handling channels of all instances of the driver in a domain.

WARNINGS

This call is only made by the `init_module` routine of the driver at load time.

NAME

udi_nsr_ctrl_ops_init *Register NSR control ops entry points*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nsr_bind_ack_op_t *nsr_bind_ack_op;
     udi_nsr_unbind_ack_op_t *nsr_unbind_ack_op;
     udi_nsr_enable_ack_op_t *nsr_enable_ack_op;
     udi_nsr_ctrl_ack_op_t *nsr_ctrl_ack_op;
     udi_nsr_info_ack_op_t *nsr_info_ack_op;
     udi_nsr_status_ind_op_t *nsr_status_ind_op;
} udi_nsr_ctrl_ops_t;

void udi_nsr_ctrl_ops_init (
     udi_index_t ops_idx,
     udi_nsr_ctrl_ops_t *ops );
```

ARGUMENTS

ops_idx*, *ops are standard arguments described in the “Channel Ops Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

`udi_nsr_ctrl_ops_init` is called only by the NSR’s `init_module` routine to set up the vector of entry points for the control channel of all instances of the NSR in a domain.

UDI will call the NSR’s `init_module` entry point when the NSR is loaded; the `udi_nsr_ctrl_ops_init` routine will register the UDI Network Metalanguage NSR control channel operations for the indicated ops index.

WARNINGS

This call is only made by the `init_module` routine of the driver at load time.

NAME

udi_nsr_tx_ops_init *Register NSR transmit ops entry points.*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nsr_tx_rdy_op_t *nsr_tx_rdy_op;
} udi_nsr_tx_ops_t;

void udi_nsr_tx_ops_init (
     udi_index_t ops_idx,
     udi_nsr_tx_ops_t *ops );
```

ARGUMENTS

ops_idx, ops are standard arguments described in the “Channel Ops Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

`udi_nsr_tx_ops_init` is called only by the NSR’s `init_module` routine to set up the vector of entry points for the data transmit channels of all instances of the driver in a domain.

UDI will call the NSR’s `init_module` entry point when the NSR is loaded; the `udi_nsr_tx_ops_init` routine should be called at that time to register the UDI Network Adapter Metalanguage NSR transmit channel operations.

WARNINGS

This call is only made by the `init_module` routine of the NSR at load time.

NAME

udi_nsr_rx_ops_init *Register NSR receive ops entry points*

SYNOPSIS

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

typedef struct {
     udi_channel_event_ind_op_t *channel_event_ind_op;
     udi_nsr_rx_ind_op_t *nsr_rx_ind_op;
     udi_nsr_exp_rx_ind_op_t *nsr_exp_rx_ind_op;
} udi_nsr_rx_ops_t;

void udi_nsr_rx_ops_init (
     udi_index_t ops_idx,
     udi_nsr_rx_ops_t *ops );
```

ARGUMENTS

ops_idx, ops are standard arguments described in the "Channel Ops Properties Initialization" section of *"Standard Calling Sequences"* in the UDI Core Specification.

DESCRIPTION

`udi_nsr_rx_ops_init` is called only by the NSR's `init_module` routine to set up the vector of entry points for the data receive channels of all instances of the driver in a domain.

UDI will call the NSR's `init_module` entry point when the NSR is loaded; the `udi_nsr_rx_ops_init` routine should be called at that time to register the UDI Network Adapter Metalanguage NSR receive channel operations.

WARNINGS

This call is only made by the `init_module` routine of the NSR at load time.

1.3.3 Network Control Block Registration

The Network Interface Metalanguage uses a number of different types of control blocks for it's various metalanguage operations. These control blocks contain supplementary information for the operation being performed, scratch space for utilization by the owner of the control block, and environment management information for controlling each operation and marshalling arguments when necessary (the environment management portion of the control block is not visible to the ND).

For ease of use, these control blocks have been divided into three types: control blocks for network control operations, control blocks for network data transmit operations, and control blocks for network data receive operations.

Each type is declared with a separate control block initialization routine and they are managed via separate control block index values (see the "Control Block Properties Initialization" section of *"Standard Calling Sequences"* in the UDI Core specification). It is important to note that all control blocks of a given type (control, transmit, or receive) are declared using a single control block index for that type so the scratch space requirements for that control block type should represent the maximum of the various operational requirements.

The typical usage of these control blocks by an ND driver or NSR is as follows (from the ND perspective):

1. Register the control block index for each type of control block in the `init_module` routine by calling `udi_net_ctrl_cb_init`, `udi_net_tx_cb_init`, and `udi_net_rx_cb_init`.
2. Register the ops index for each type of channel in the `init_module` routine by calling `udi_nd_ctrl_ops_init`, `udi_nd_tx_ops_init`, and `udi_nd_rx_ops_init`.
3. All subsequent channel operations will result in the environment passing a ctrl or transfer control block on the corresponding channel which is big enough for any control or transfer usage, respectively.
4. When allocating a control block internally for use, the control block index argument passed to `udi_cb_init` specifies either a ctrl or xfer type of control block; the resulting allocated control block will be large enough to be used for any of the control blocks within that index group.

NAME

udi_net_ctrl_cb_init *Network control channel control block definition*

SYNOPSIS

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

void udi_net_ctrl_cb_init (
     udi_index_t cb_idx,
     udi_size_t scratch_requirement );
```

ARGUMENTS

cb_idx, scratch_requirement are standard arguments described in the "Control Block Properties Initialization" section of *"Standard Calling Sequences"* in the UDI Core Specification.

DESCRIPTION

The `udi_net_ctrl_cb_init` function is called to specify the scratch space requirements and establish the control block index for a network control channel control block.

The following control blocks can be represented by this ***cb_idx***:

```
udi_net_bind_req_cb_t
udi_net_bind_ack_cb_t
udi_net_unbind_cb_t
udi_net_enable_cb_t
udi_net_disable_cb_t
udi_net_ctrl_cb_t
udi_net_status_cb_t
udi_net_info_cb_t
```

The scratch space requirement specified for this ***cb_idx*** should be large enough to provide enough scratch space for all control blocks allocated with this index or received on any channel associated with this ***cb_idx*** for inbound control blocks via `udi_cb_select`.

REFERENCES

`udi_net_ctrl_cb_t`

NAME

udi_net_tx_cb_init *Network transmit control block definition*

SYNOPSIS

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

void udi_net_tx_cb_init (
     udi_index_t cb_idx,
     udi_size_t scratch_requirement );
```

ARGUMENTS

cb_idx, scratch_requirement are standard arguments described in the "Control Block Properties Initialization" section of "*Standard Calling Sequences*" in the UDI Core Specification.

DESCRIPTION

The `udi_net_tx_cb_init` function is called to specify the scratch space requirements and establish the control block index for a network transmit channel control block.

The following control blocks can be represented by this ***cb_idx***:

`udi_net_tx_cb_t`

REFERENCES

`udi_net_tx_cb_t`

NAME

udi_net_rx_cb_init *Network receive control block definition*

SYNOPSIS

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

void udi_net_rx_cb_init (
     udi_index_t cb_idx,
     udi_size_t scratch_requirement );
```

ARGUMENTS

cb_idx, scratch_requirement are standard arguments described in the “Control Block Properties Initialization” section of *“Standard Calling Sequences”* in the UDI Core Specification.

DESCRIPTION

The `udi_net_rx_cb_init` function is called to specify the scratch space requirements and establish the control block index for a network receive channel control block.

The following control blocks can be represented by this ***cb_idx***:

```
udi_net_rx_cb_t
```

REFERENCES

`udi_net_rx_cb_t`

1.3.4 Network Interface Metalanguage Binding

1.3.4.1 Network Bind Operation

Once the ND has been initialized and it has registered the UDI Network Interface Metalanguage channel operations, it may receive bind requests from the NSR. As the first stage of this binding, the UDI Management Agent (MA) will bind the child (NSR) to the parent (ND) using the techniques described in the UDI Management Metalanguage chapter. A typical binding can be summarized by the following procedure:

1) The MA creates the initial bind channel between the ND and the NSR. This channel is the network control channel and uses the UDI Network Interface Metalanguage `ctrl_ops` role. The ND or NSR can use "loose end" roles if desired for further region/channel control as specified in the UDI Management Metalanguage chapter.
2) The MA sends a `udi_usage_ind` and (possibly) `udi_region_attach_ind` operations to the new region to initialize it.
3) The UDI MA sends a `udi_prep_for_child_req` to the ND via the ND's management channel.
4) The ND responds to the UDI MA via the `udi_prep_for_child_ack` indicating that it has anchored its end of the channel and is ready for the next stage of the binding process.
5) The UDI MA then sends a `udi_bind_to_parent_req` to the NSR, specifying the other end of the channel created in Step 1.
6) The NSR prepares and anchors its end of the channel. The UDI Network Interface Metalanguage control channel is now established between the ND and the NSR.

At this point, the UDI Network Interface Metalanguage control channel is established between the ND and the NSR; the NSR may now initiate Network Bind operations to the ND.

The Network Bind operation (`udi_nd_bind_req)` exchanges basic information about the media type and supported packet information between the NSR and the ND, as well as identifying the transfer channel indices to be used for communication.

7) The NSR spawns the transmit channel, specifying that the spawned channel should use the transmit channel ops vector. This may be done in the primary region or in another region created specifically to handle transmit operations.
8) The NSR spawns the receive channel in a manner similar to the transmit channel.
9) The NSR issues a `udi_nd_bind_req` operation over the newly established bind channel to the ND. Part of the `udi_nd_bind_req` operation information specifies the indices of the spawned transfer channels.

The NSR will typically use well-known values for the transmit and receive channel indices, but may choose any values it desires to accommodate multiple simultaneous bindings to the same region.

The ND and the NSR have now agreed to communicate and have a common control channel which allows them to communicate directly without requiring assistance from the UDI MA. However, as noted in Section 1.2.4.3, "Transfer Channels", the ND and NSR cannot yet perform data transfer operations; the data handling channels have not been established and will be created by subsequent operations.

10)The ND verifies the request, prepares itself for handling that NSR binding, and then responds to the NSR with a `udi_nsr_bind_ack` operation over the network control channel.

11)The NSR then completes its internal preparations and then responds to the UDI MA over the management channel with a `udi_bind_to_parent_ack` to indicate completion of the bind operation.

The expedited data ops vectors allow expedited data to be handled separately from normal data and thereby allows a higher priority to be assigned to managing expedited data. If no expedited data is expected, or if the network type or protocol stack cannot support expedited data, the expedited data ops vectors should point to the normal data handling routines.

The data channels are established by spawning the network control channel as indicated by the contents of the network bind control block exchanged in Step 9 above.

When the NSR initiates the `udi_nd_bind_req`, the associated control block contains a channel spawn index for the network transmit and network receive channels to be established.

12)Once the ND completes Step 10 above it should then call `udi_channel_spawn` with the spawn index indicated in the Network Bind control block to establish the transmit data channel to the NSR. Simultaneously or on completion of that spawn it should also call `udi_channel_spawn` to establish the receive data channel to the NSR.

13)Once the NSR completes Step 11 above, it should await the completion of the transmit and receive channel spawn operations that were initiated in Step 7.

14)When both the ND and the NSR rendezvous in the corresponding channel spawn operations, the UDI environment spawns the corresponding transmit or receive data channel, assigns the proper channel operations to the newly created channel(s), and returns a handle to the channel(s) to both the ND and the NSR via `udi_channel_spawn` callback operations.

At this point, the communications pathways between the ND and the NSR have been fully established. The NSR may now enable the interface, which causes the ND to enable the hardware and to create a pool of transmit control blocks to post to the NSR for sending or receiving network packets. The normal flow of calls associated with the Network Bind operation which supports expedited data operations is illustrated in Figure 1-2.

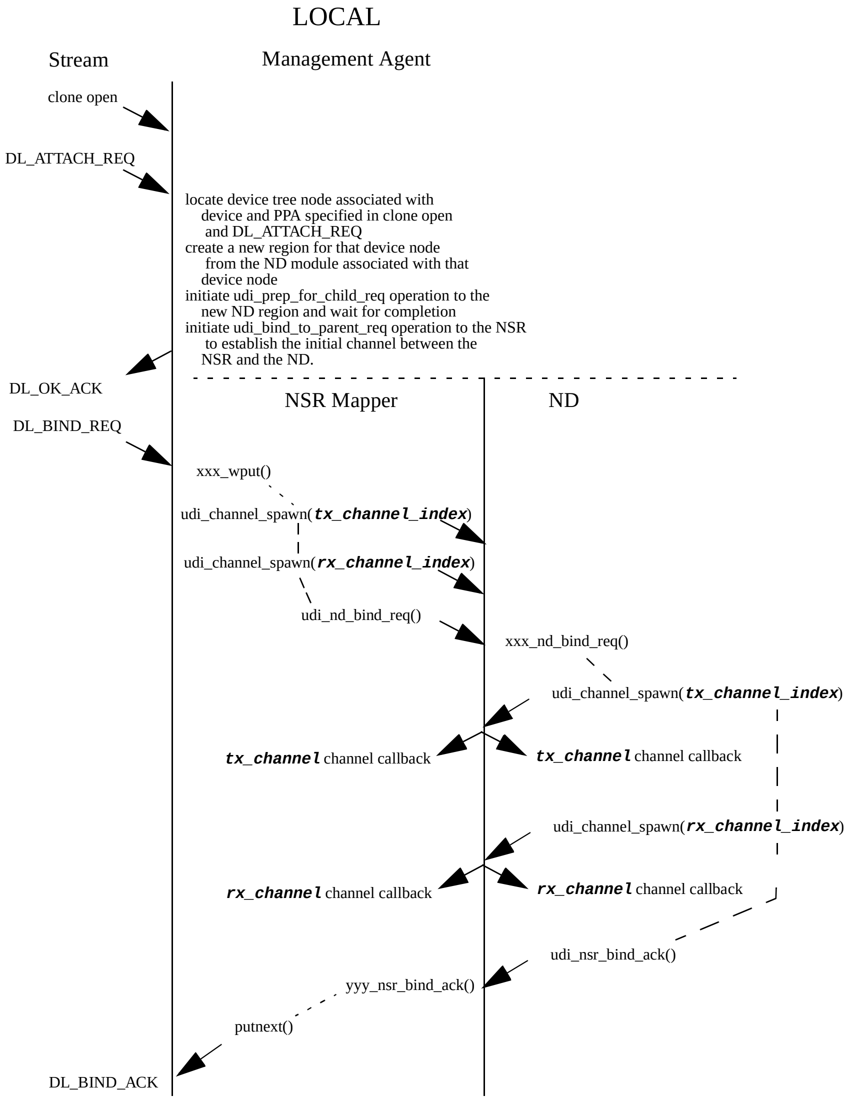
LOCAL
Stream
Management Agent
clone open
DL_ATTACH_REQ
locate device tree node associated with
device and PPA specified in clone open
and DL_ATTACH_REQ
create a new region for that device node
from the ND module associated with that
device node
initiate udi_prep_for_child_req operation to the
new ND region and wait for completion
initiate udi_bind_to_parent_req operation to the NSR
to establish the initial channel between the
NSR and the ND.
DL_OK_ACK
NSR Mapper
ND
DL_BIND_REQ
xxx_wput()
udi_channel_spawn(tx_channel_index)
udi_channel_spawn(rx_channel_index)
udi_nd_bind_req()
xxx_nd_bind_req()
udi_channel_spawn(tx_channel_index)
tx_channel channel callback
tx_channel channel callback
udi_channel_spawn(rx_channel_index)
rx_channel channel callback
rx_channel channel callback
udi_nsr_bind_ack()
yyy_nsr_bind_ack()
putnext()
DL_BIND_ACK

Figure 1-2 Connectionless Network Bind Operation

1.3.4.2 Network Unbind Operation

The Network Unbind process is the converse of the Network Bind operation. During the Network Unbind operation the driver is instructed to stop forwarding any frames and shutdown the channels. The driver acknowledges the Network Unbind operation through a `udi_nsr_unbind_ack` operation. Once the network unbind has been performed, the control channel and both data channels should be released by calling `udi_channel_close.` Either end may initiate the closure, but both sides must release the data channels after the Network Unbind acknowledgement (one or both of the ND and NSR may receive `udi_channel_closed` indications depending on the sequence of events; these are assumed to be indications of the proper unbinding operation and are typically ignored).

The Network Unbind process is used to gracefully terminate network connectivity. If the normal data channels or the control channel is closed by either end, a Network Unbind condition is assumed and the channels are shutdown by both the NSR and the ND.

The normal flow of this operation is illustrated in Figure 1-3.

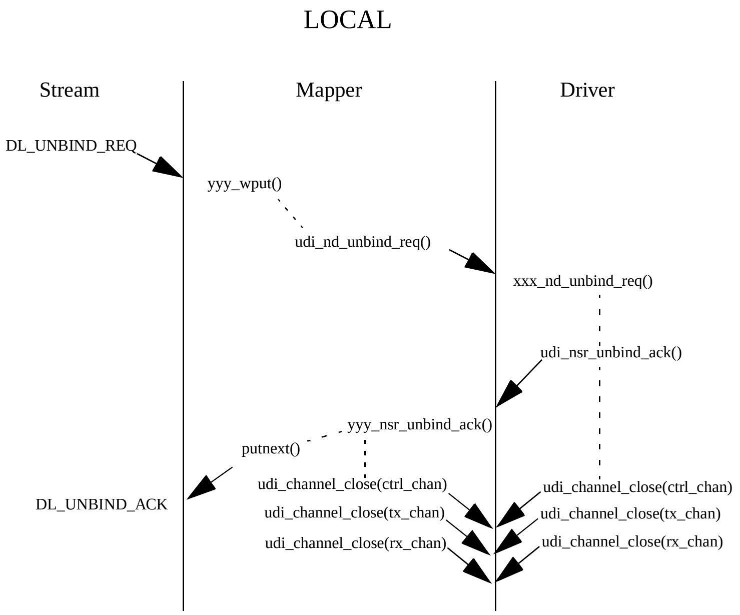
LOCAL
Stream
Mapper
Driver
DL_UNBIND_REQ
yyy_wput()
udi_nd_unbind_req()
xxx_nd_unbind_req()
udi_nsr_unbind_ack()
yyy_nsr_unbind_ack()
putnext()
DL_UNBIND_ACK
udi_channel_close(ctrl_chan)
udi_channel_close(tx_chan)
udi_channel_close(rx_chan)
udi_channel_close(ctrl_chan)
udi_channel_close(tx_chan)
udi_channel_close(rx_chan)

Figure 1-3 Unbind Operation

NAME **udi_net_bind_req_cb_t** *Network bind control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_index_t tx_chan_index;
     udi_index_t rx_chan_index;
} udi_net_bind_req_cb_t;
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

tx_chan_index is the index of the data transmit channel which is to be created between the ND and the NSR by synchronizing via `udi_channel_spawn` operations.

rx_chan_index is the index of the data receive channel which is to be created between the ND and the NSR by synchronizing via `udi_channel_spawn` operations.

DESCRIPTION

The `udi_net_bind_req_cb_t` structure is used between a child (NSR) and its parent (ND) at network specific bind time. This structure is used to supply Network Metalanguage specific information at bind time.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES `udi_nd_bind_req`

NAME

udi_nd_bind_req *Network driver bind request operation*

SYNOPSIS

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

void  udi_nd_bind_req (
     udi_channel_t target_channel,
     udi_net_bind_req_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

The `udi_nd_bind_req` uses the Network Interface Metalanguage control channel which is the same as the primary channel created by the UDI Management Agent to initially bind the NSR and ND regions.

DESCRIPTION

`udi_nd_bind_req` is used to associate, or “bind” a channel between the NSR and the ND. The `net_bind_cb` provides information for creating the data transfer channel(s) and any other configuration information for supporting this network adapter.

REFERENCES

`udi_nsr_bind_ack, udi_net_bind_t`

NAME **udi_net_bind_ack_cb_t** *Network bind ack control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_ubit8_t media_type;
     udi_ubit32_t min_pdu_size;
     udi_ubit32_t max_pdu_size;
     udi_ubit32_t rx_hw_threshold;
     udi_ubit8_t mac_addr_len;
     udi_ubit8_t mac_addr[UDI_NET_MAC_ADDRESS_SIZE];
} udi_net_bind_ack_cb_t;

#define UDI_NET_MAC_ADDRESS_SIZE     20

/* media_type values */
#define   UDI_NET_ETHER              0
#define   UDI_NET_TOKEN              1
#define   UDI_NET_FASTETHER          2
#define   UDI_NET_GIGETHER           3
#define   UDI_NET_VGANYLAN           4
#define   UDI_NET_FDDI               5
#define   UDI_NET_ATM                6
#define   UDI_NET_FC                 7
#define   UDI_NET_MISCMEDIA          0xff
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

media_type is the network media type supported by the ND and associated Adapter hardware. This information is used by the NSR to parse and create frame headers for packets. Valid values for ***media_type*** are specified in Table 1-1 on page 6.

If there is not an exact match for your media type appearing in the table, select the media type with the framing style closest to the device's network type. The ***media_type*** value suggests default values for the remaining `net_bind_ack_cb_t` configuration values and also hints to the NSR about handling packets for this network interface.

min_pdu_size is the minimum size of a data packet including the headers that the NSR or other modules will supply. The ND must return this information in the ack operation. If specified as 0, the default is used based on the specified ***media_type***.

max_pdu_size is the maximum size of a data packet including the headers that the NSR or other modules will supply. The ND must return this information in the ack operation. If specified as 0, the default is used based on the specified ***media_type***.

rx_hw_threshold is the number of receive requests that may be posted to the hardware adapter at any one time. This value is used as a hint to the NSR to indicate the number of receive control blocks and buffers that should be provided to the ND for receive operations. The ND should be prepared to operate with more or less than this value but this should represent the optimal requirements of the ND and its hardware adapter.

mac_addr_len is the number of bytes that should be used by the NSR for MAC addresses for this ND and overrides any default size for this media type. If specified as 0, the default MAC Address size is used based on the specified ***media_type***.

mac_addr is the current (factory) MAC address of the adapter being managed by the ND. The current MAC address of the adapter will be written to the first ***mac_addr_len*** bytes (or the number of bytes appropriate to that ***media_type*** if ***mac_addr_len*** is zero).

DESCRIPTION

The udi_net_bind_ack_cb_t structure is used for the udi_nsr_bind_ack response operation between the parent (ND) and the child (NSR) which issued the udi_nd_bind_req operation. This structure establishes the parameters for the associated NIC Adapter for subsequent network data handling.

This control block is declared with the udi_net_ctrl_cb_init routine.

REFERENCES

udi_nsr_bind_ack

NAME

udi_nsr_bind_ack *Network bind acknowledgment operation*

SYNOPSIS

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

void  udi_nsr_bind_ack (
      udi_channel_t target_channel,
      udi_net_bind_ack_cb_t *cb,
      udi_status_t status );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

status indicates the success or failure of the network bind request

TARGET CHANNEL

The `udi_nsr_bind_ack` uses the Network Interface Metalanguage bind channel over which the `udi_nd_bind_req` operation was received.

DESCRIPTION

Handshakes `udi_nd_bind_req` operation to the NSR. The ND uses it to inform the NSR about the driver operational parameters and signify its readiness for network data transfer operations.

It is expected that the data transfer channels will have been fully spawned and configured before completion of the `udi_net_bind_ack` operation. The NSR will always issue the `udi_channel_spawn` for its end of the data transfer channels before issuing the `udi_nd_bind_req` operation.

STATUS VALUES

`UDI_OK` indicates that the network bind operation succeeded.

`UDI_STAT_INVALID_STATE` indicates that the ND is already bound to another NSR and cannot satisfy this bind request.

REFERENCES

`udi_nd_bind_req`

NAME

udi_net_unbind_cb_t *Network interface unbind control block*

SYNOPSIS

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

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

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. 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_net_unbind_cb_t` is the control block used for processing network metalanguage unbind requests and acknowledgements.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES

`udi_nd_unbind_req`, `udi_nsr_unbind_ack`

NAME

udi_nd_unbind_req *Network unbind request operation*

SYNOPSIS

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

void  udi_nd_unbind_req (
     udi_channel_t target_channel,
     udi_net_unbind_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

The `udi_nd_unbind_req` uses the Network Interface Metalanguage control channel.

DESCRIPTION

`udi_nd_unbind_req` is used when the NSR wishes to close down an active binding between itself and the ND. All data transfer ceases, any queued data is discarded, and the data channel(s) are closed. The control channel may be used to issue a subsequent `udi_net_bind_req` to the ND if desired or it may also be closed.

REFERENCES

`udi_nsr_unbind_ack`

NAME

udi_nsr_unbind_ack *Network unbind acknowledgment operation*

SYNOPSIS

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

void  udi_nsr_unbind_ack (
     udi_channel_t target_channel,
     udi_net_unbind_cb_t *cb,
     udi_status_t status );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

The `udi_nsr_unbind_ack` uses the Network Interface Metalanguage control channel.

DESCRIPTION

Handshakes network unbind request operation to the NSR. As a result of this operation the routing information will be removed from the NSR and the control and data handling channels will be closed by both the ND and the NSR. No more data transfer will occur for this adapter instance unless another `udi_nd_bind_req` is issued by the NSR to create and bind a new set of data channels.

STATUS VALUES

`UDI_OK` indicates that the network unbind request operation succeeded.

`UDI_STAT_INVALID_STATE` indicates that the ND was not currently bound.

REFERENCES

`udi_nsr_unbind_req`

1.3.5 Interface Control

The interface control operations are used by the NSR to instruct the ND to enable or disable the network interface for link activity. Once the NSR and the ND have successfully been bound together, the `udi_nd_enable_req` operation is performed to activate the interface, usually as the result of a control operation by the system or user (e.g. ifconfig ... up).

The `udi_nd_enable_req` is successfully acknowledged by the ND using the `udi_nsr_enable_ack` operation as soon as the link activation operation is initiated. The acknowledgement is not expected to wait for the link to actually become enabled and the returned status in the `udi_net_enable_cb_t` is only used to indicated the success or failure of the ND in initiating the enable operation.

When the interface actually reports active, it will use the `udi_nsr_status_ind` operation to indicate that the link state has changed to the active state. Note that the interface may be enabled, but not necessarily have an active link status. Whenever the interface is in the enabled state, `udi_nsr_status_ind` operations should be used to indicate the link status if the link status changes.

The `udi_nd_disable_req` is used to disable an interface. The link status change may or may not be indicated by the `udi_nsr_status_ind` indication, but this request must perform the appropriate activity to force the interface into the disabled state. There is no acknowledgement to the NSR for the disable operation; the link must be disabled if currently active and there are no reasonable failure conditions.

When an interface is in the disabled state, link status changes should not be indicated by the `udi_nsr_status_ind` operation; if generated in this state, these indications will be ignored.

NAME

udi_net_enable_cb_t *Network interface enable control block*

SYNOPSIS

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

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

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. 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_net_enable_cb_t` is the control block used for processing network interface enable requests and acknowledgements.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES

`udi_nd_enable_req`, `udi_nsr_enable_ack`

NAME

udi_nd_enable_req *Network link enable request operation*

SYNOPSIS

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

void  udi_nd_enable_req (
     udi_channel_t target_channel,
     udi_net_enable_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Network Interface Metalanguage control channel.

DESCRIPTION

`udi_nd_enable_req` is used when the NSR wishes to begin data transfer over the network connection provided by the ND and it's associated adapter. The NSR uses this operation to enable the link. The link will not necessarily become immediately available, although the ND will acknowledge this request as soon as it has successfully initiated the link enable operation. The actual link up state will be indicated by the ND via the `udi_nsr_status_ind` operation.

The ND is expected to update the configuration of the driver and adapter hardware from its device instance attributes (if any) each time the `udi_nd_enable_req` is issued. This insures that any device-specific configuration changes made by the MA or system administrator will take effect at that time.

REFERENCES

`udi_nsr_enable_ack, udi_nd_disable_req`

NAME

udi_nsr_enable_ack *Network link enable acknowledgment operation*

SYNOPSIS

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

void  udi_nsr_enable_ack (
     udi_channel_t target_channel,
     udi_net_enable_cb_t *cb,
     udi_status_t status );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

status indicates the success or failure of the enable operation

TARGET CHANNEL

Network Interface Metalanguage control channel.

DESCRIPTION

Handshakes network link enable request operation to the NSR. If successful, the NSR will subsequently expect to receive a `udi_nsr_status_ind` which indicates the actual link-up state for this network adapter connection.

Once the link-up state has been reached, the ND is expected to “post” transmit control blocks to the NSR for use in transmit requests by pre-allocating one or more of these control blocks and passing them to the NSR over the transmit channel using the `udi_nsr_tx_rdy` operation. The number of these transmit control blocks that are provided to the NSR represents the flow control level of the connection between the NSR and the ND; the NSR will never internally allocate transmit control blocks to be passed to the ND and will instead wait for the ND to supply these control blocks with the `udi_nsr_tx_rdy` operation.

STATUS VALUES

`UDI_OK` indicates that the network enable request operation succeeded.

`UDI_STAT_HW_PROBLEM` indicates that the ND will not be able to enable the link due to a hardware problem.

REFERENCES

`udi_nsr_unbind_req`

NAME

udi_net_disable_cb_t *Network interface disable control block*

SYNOPSIS

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

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

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. 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_net_disable_cb_t` is the control block used for processing network interface disable requests.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES

`udi_nd_disable_req`

NAME

udi_nd_disable_req *Network link disable request operation*

SYNOPSIS

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

void  udi_nd_disable_req (
     udi_channel_t target_channel,
     udi_net_disable_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Network Interface Metalanguage control channel.

DESCRIPTION

udi_nd_disable_req is used when the NSR wishes to shutdown an active link interface and terminate any data transfer activity for that link. This does not cause an unbind operation, and all control and data transfer channels between the NSR and the ND are preserved, but the hardware link interface state is set to remove the adapter from the network. This request must be honored and implemented by the ND; there is no failure case and correspondingly there is no acknowledgement to the NSR of this operation.

The NSR may be notified of the link down state as a result of the ND issuing a udi_nsr_status_ind, but this is not required and the ND is not expected to communicate any link state changes to the NSR when the interface state is down.

REFERENCES

udi_nd_enable_req

1.3.6 Control and Status Operations

The purpose of the control operation is to perform a specific network-related function on the driver (ND) or the controlled hardware. The most common functions that have been identified are:

- setting a multicast address
- setting adapter physical address
- setting a filtering mode (i.e. promiscuous mode)
- resetting the driver/card

All of the above functions have been implemented using a single pair of channel operations: `udi_nd_ctrl_req` and `udi_nsr_ctrl_ack`. A command field in the *`udi_net_ctrl_cb_t`* control block informs the ND of the type of control function the caller wishes to execute. The normal flow for the control operation is illustrated in Figure 1-4.

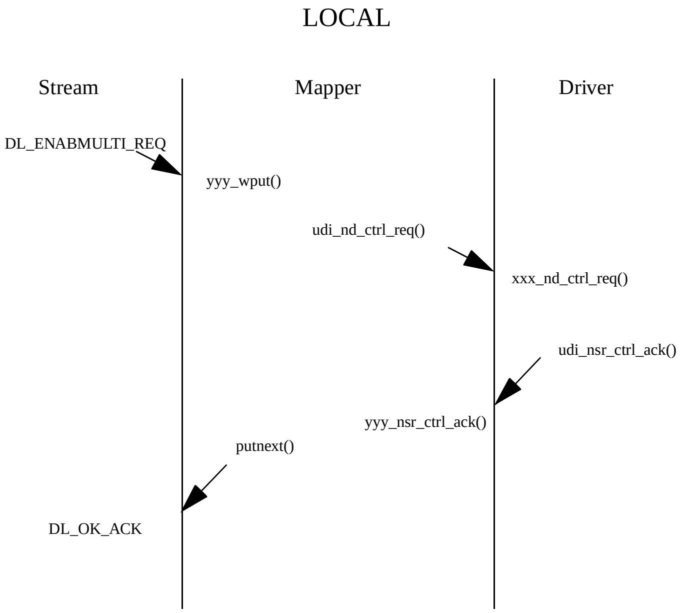

Figure 1-4 Control Operation (configure multicast address)

The information request/response operations are used by the NSR to obtain configuration and statistics information.

The status operation is an unsolicited event indication issued by the ND to the NSR to inform the NSR of a change in status. Some of the status events indicated by the `udi_nsr_status_ind` indications are similar to those that can be queried via the `udi_nd_ctrl_req` operation.

NAME

udi_net_ctrl_cb_t *Network control operation control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     void *tr_context;
     udi_ubit8_t command;
     udi_ubit32_t indicator;
     udi_buf_t data_buf;
} udi_net_ctrl_cb_t;

#define   UDI_NET_ADD_MULTI     0x1
#define   UDI_NET_DEL_MULTI     0x2
#define   UDI_NET_ALLMULTI_ON   0x3
#define   UDI_NET_ALLMULTI_OFF  0x4
#define   UDI_NET_GET_CURR_MAC  0x5
#define   UDI_NET_SET_CURR_MAC  0x6
#define   UDI_NET_GET_FACT_MAC  0x7
#define   UDI_NET_PROMISC_ON    0x8
#define   UDI_NET_PROMISC_OFF   0x9
#define   UDI_NET_HW_RESET      0xA
#define   UDI_NET_BAD_RXPKT     0xB
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

tr_context is an arbitrary transaction context which uniquely identifies the control request contained in the structure. This is set by the NSR when issuing a control request to the ND and is used to uniquely identify the acknowledgement of that control request, therefore the ND should not modify this field and should supply the same value in the control response.

command is the control command the driver must execute. The following control commands have been identified:

UDI_NET_ADD_MULTI -- Configures the multicast addresses specified in the supplied buffer. The ***data_buf*** contains the array of MAC addresses to be added to the existing filter (the number of addresses being specified is indicated by the ***indicator*** field). The remaining portion of the ***data_buf*** contains the new filter table including the newly added addresses.
Devices which can modify their address filtering simply by knowing the addresses which are to be added or removed can use the initial ***indicator*** number of

addresses to modify their filter. Other devices require access to the full table of addresses to be filtered to recompute the filter and may use the latter portion of the ***data_buf***.
Although it is possible for the network stack or user-level applications to register multiple times for a specific multicast address, the NSR is responsible for handling this redundancy; the ND is only notified of multicast address modifications when the actual received packet filter must be modified.

UDI_NET_DEL_MULTI -- Removes the multicast addresses specified in the supplied buffer. In a manner similar to the UDI_NETADD_MULTI operation, the initial ***indicator*** number of addresses in the ***data_buf*** lists the addresses to be removed, and the latter portion of the data_buf lists the new filter set after removing the specified addresses.

UDI_NET_ALLMULTI_ON -- Enables the network adapter to receive all multicast addressed packets. The ***indicator*** and ***data_buf*** fields are unused and ignored. This operation additionally indicates that any specific multicast addresses registered (via UDI_NET_ADD_MULTI should be removed).

UDI_NET_ALLMULTI_OFF -- Disables the reception of all multicast addresses. The ***indicator*** and ***data_buf*** arguments are used in the same manner as in the UDI_NET_ADD_MULTI operation to specify the new list of specific multicast addresses (if any) that are to be passed to the NSR.

The adapter hardware is to be returned to the standard reception mode with no multicast addresses being received unless UDI_PROMISC_ON is in effect or unless specifically registered via this operation.

UDI_NET_GET_CURR_MAC -- Reads the network adapter card's current physical MAC address. The MAC address is placed into the ***data_buf*** and the size of the MAC address is placed into the ***indicator*** field.

UDI_NET_SET_CURR_MAC -- Configures the network adapter card's current physical MAC address. The MAC address to be set is contained in the ***data_buf*** and its size is in the ***indicator*** field.

UDI_NET_GET_FACT_MAC -- Reads the network adapter card's factory installed physical MAC address. The address is specified in the same manner as for the GET_CURR_MAC command.

`UDI_NET_PROMISC_ON` -- Enables the promiscuous mode on the network adapter card. The ***`indicator`*** and ***`data_buf`*** fields are unused and ignored.

`UDI_NET_PROMISC_OFF` -- Disables the promiscuous mode on the network adapter card. The ***`indicator`*** and ***`data_buf`*** fields are unused and ignored. The NSR should now be passed only unicast addresses specific to this adapter or any multicast addresses enabled by any previous `UDI_NET_ADD_MULTI` or `UDI_NET_ALLMULTI_ON` operations.

`UDI_NET_HW_RESET` -- Resets the network adapter card.

`UDI_NET_BAD_RXPKT` -- Specifies ND handling of received packets which have an error indication. If the ***`indicator`*** is zero, the ND should simply discard the bad packet and await the next received packet.

If non-zero, the ***`indicator`*** specifies how many bytes of the bad packet should be passed to the NSR (with the appropriate rx_status indication). This is intended as a hint to allow the ND to terminate or discard reception of bad packets if needed; the bad packet is not required or guaranteed by the ND to match the length field in the ***`indicator`***. A receive indication with a bad packet status should be passed to the NSR even if no bytes from the received packet can be passed.

`indicator` is a field which supplies additional information for the control operation as specified on a per-command basis.

`data_buf` is the handle to the data buffer associated with the control request, e.g., the buffer containing the multicast list addresses.

DESCRIPTION

The network control structure is used to configure, retrieve information or request a specific configuration action from the driver and/or the network adapter card. This structure is used with network control operations issued to the control channel.

Table 1-2 udi_net_ctrl_cb_t Argument usage

Command (UDI_NET_xxx)	**Indicator**	**Data_buf contents**
ADD_MULTI	number of new MAC addresses	MAC addresses (new/all)
DEL_MULTI	number of MAC addresses being removed	MAC addresses (removed/remaining)
ALLMULTI_ON	-	-
ALLMULTI_OFF	number of new MAC addresses	MAC addresses (new/all)

Table 1-2 udi_net_ctrl_cb_t Argument usage

Command (UDI_NET_xxx)	**Indicator**	**Data_buf contents**
GET_CURR_MAC	returns MAC address size	returns MAC address
SET_CURR_MAC	MAC address size	MAC address
GET_FACT_MAC	returns MAC address size	returns MAC address
PROMISC_ON	-	-
PROMISC_OFF	-	-
HW_RESET	-	-
BAD_RXPKT	0 = ND discards bad RX packets non-zero = number of bytes of bad RX packets that the ND passes the NSR	-

The `UDI_NET_PROMISC_ON` and `UDI_NET_ALLMULTI_ON` shall be individually applied and the enabling or disabling of one shall not affect the setting of the other. The `UDI_NET_PROMISC` settings do not affect the current multicast address table but the `UDI_NET_ALLMULTI_ON` will delete the current list of explicit multicast addresses (which may be re-established in whole or in part by the `UDI_NET_ALLMULTI_OFF`).

This control block is declared with the `udi_net_ctrl_cb_init` routine.

NAME

udi_nd_ctrl_req *Network control operation request*

SYNOPSIS

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

void  udi_nd_ctrl_req (
     udi_channel_t target_channel,
     udi_net_ctrl_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Network Interface Metalanguage control channel.

DESCRIPTION

The `udi_nd_ctrl_req` operation is used to perform various network control functions on the ND and/or controlled adapter card.

The network control operations are described in more detail in the description of the `udi_net_ctrl_cb_t` structure.

REFERENCES

`udi_net_ctrl_ack, udi_net_ctrl_cb_t`

NAME

udi_nsr_ctrl_ack *Network control acknowledgment operation*

SYNOPSIS

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

void  udi_nsr_ctrl_ack (
     udi_channel_t target_channel,
     udi_net_ctrl_cb_t *cb,
     udi_status_t status );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

status indicates success or failure of the control request

TARGET CHANNEL

Network Interface Metalanguage control channel

DESCRIPTION

`udi_nsr_ctrl_ack` confirms the `udi_nd_ctrl_req` operation that has been issued to the driver and passes back any requested information in the control block or the associated buffer structure.

STATUS VALUES

`UDI_OK` indicates that the network control request operation succeeded.

`UDI_STAT_NOT_UNDERSTOOD` indicates that the control operation parameters were invalid.

REFERENCES

`udi_nd_ctrl_req, udi_net_ctrl_cb_t`

NAME

udi_net_status_cb_t *Network status indication control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_ubit8_t event;
} udi_net_status_cb_t;

#define  UDI_NET_LINK_DOWN     0x0
#define  UDI_NET_LINK_UP       0x1
#define  UDI_NET_LINK_RESET    0x2
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may used the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

event is the status event code being indicated by the ND driver to the NSR. The following status events have been defined:

UDI_NET_LINK_UP -- Link active transition

UDI_NET_LINK_DOWN -- Link inactive transition

UDI_NET_LINK_RESET -- Link reset occurred. A link reset has the same effect as a UDI_NET_LINK_DOWN event with the additional indication that any link state was lost and that a link-level error recovery was initiated. The NSR should be prepared to re-establish the link and remote node configuration information when a link reset occurs.

DESCRIPTION

The network status indication control block structure is used to notify the NSR of asynchronous events. This structure is used with network status indications issued to the control channel.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES

`udi_nsr_status_ind`

NAME

udi_nsr_status_ind *Network status indication*

SYNOPSIS

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

void  udi_nsr_status_ind (
     udi_channel_t target_channel,
     udi_net_status_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

Network Metalanguage control channel.

DESCRIPTION

The `udi_nsr_status_ind` indication is used by the ND to asynchronously notify the NSR of various status information such as any change in link status. The NSR should evaluate the status indication, perform the appropriate action, and then deallocate the status indication control block.

REFERENCES

`udi_net_status_cb_t`

NAME **udi_net_info_cb_t** *Network information control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_boolean_t interface_is_active;
     udi_boolean_t link_is_active;
     udi_boolean_t is_full_duplex;
     udi_ubit32_t link_mbps;
     udi_ubit32_t link_bps;
     udi_ubit32_t tx_packets;
     udi_ubit32_t rx_packets;
     udi_ubit32_t tx_errors;
     udi_ubit32_t rx_errors;
     udi_ubit32_t tx_discards;
     udi_ubit32_t rx_discards;
     udi_ubit32_t tx_underrun;
     udi_ubit32_t rx_overrun;
     udi_ubit32_t collisions;
} udi_net_info_cb_t;
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may used the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

interface_is_active is a boolean indication of whether the interface has been activated or not (via `udi_nd_enable_req`).

link_is_active is a boolean indication of whether the link is active or not (line status).

is_full_duplex is the boolean indication of whether the link is operating in full-duplex or half-duplex mode.

link_mbps is the current link data rate in megabits-per-second.

link_bps is the current link data rate in bits-per-second.

tx_packets is the number of packets transmitted by this interface (inclusive of discarded packets and those in which the transmit request completed with errors).

rx_packets is the number of packets received by this interface (inclusive of discarded packets and those indicating receive errors).

tx_errors is the number of packet transmissions that encountered an error of some form.

rx_errors is the number of receive packet operations that encountered an error of some form.

tx_discards is the number of packets which were discarded before transmission for internal reasons (e.g. timeouts or unavailable resources).

rx_discards is the number of packets which were discarded after being received but before being passed to the NSR for internal reasons (e.g. unavailable resources).

tx_underrun is the number of transmit underrun errors (which is also included in ***tx_errors***).

rx_overrun is the number of receive overrun errors (which is also included in ***rx_errors***).

collisions is the number of transmit packet collisions (if applicable).

DESCRIPTION

The network information control block structure is used to provide configuration and statistics information to the NSR.

Fields which represent counters are subject to overflow and will silently wraparound to continue counting from zero. The statistics values should be read frequently enough to detect this wraparound condition if it is significant.

The ***link_bps*** field is customarily used for slower WAN protocols where the link data rate may range from 300 bps to 128 Kbps. The ***link_mbps*** field is used for higher-speed LAN protocols. Note that there is an overlap between these two fields; it is expected that if the ***link_mpbs*** field is non-zero that the ***link_bps*** field may be ignored. These values are supplied to provide diagnostic information and allow predictive scheduling of data exchange and therefore are not required to be exact nor do they control the associated hardware settings.

This control block is declared with the `udi_net_ctrl_cb_init` routine.

REFERENCES

`udi_nd_info_req, udi_nsr_info_ack`

NAME

udi_nd_info_req *Network information request*

SYNOPSIS

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

void  udi_nd_info_req (
      udi_channel_t target_channel,
      udi_net_info_cb_t *cb,
      udi_boolean_t reset_statistics );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

reset_statistics is a boolean value which indicates whether the ND is to reset the statistics counters or not.

TARGET CHANNEL

Network Metalanguage control channel.

DESCRIPTION

The udi_nd_info_req is used by the NSR to request information from the NIC Driver. The NSR supplies a udi_net_info_cb_t control block which is filled in with the appropriate information by the ND and then passed back to the NSR.

The fields in the control block which is passed in are uninitialized and should be overwritten by the ND.

If the ***reset_statistics*** argument is true, the ND should reset the statistics after providing their current values in the ***cb*** control block being returned.

REFERENCES

udi_net_info_cb_t, udi_nsr_info_ack

NAME **udi_nsr_info_ack** *Network information response*

SYNOPSIS

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

void  udi_nsr_info_ack (
     udi_channel_t target_channel,
     udi_net_info_cb_t *cb );
```

ARGUMENTS ***target_channel, cb*** are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL Network Metalanguage control channel.

DESCRIPTION The `udi_nsr_info_ack` is the ND response to the `udi_nd_info_req` issued by the NSR. The ND passes back the information in the fields of the associated control block.

REFERENCES `udi_net_info_cb_t, udi_nd_info_req`

1.4 Data Transfer Operations

These operations are used to allow the local protocol stack to communicate with other hosts on the network by passing requests to or from the ND via the data transfer channels connected to the NSR. The associated data buffers for these requests are the data “packets” to be sent over the network and are assumed (by the ND) to have the proper network headers already constructed and provided, although the ND is free to add device or network specific headers as needed.

If the semantics of the underlying network technology are based on unacknowledged broadcast methodologies (e.g. Ethernet) then the ND will acknowledge transmit requests back to the NSR as soon as the ND has completed the transmit operation itself. If the network technology incorporates packet or frame level acknowledgement[3], this should be managed by the ND and the transmit operations should not be acknowledged to the NSR until the remote host has acknowledged them back to the ND. The ND is not expected to implement retransmissions in either case unless desirable by the ND implementation or the network technology.

The ND is also expected to generate packet or frame level acknowledgements if so required by the network technology.

Any technology-specific operations must be implemented by the ND internally (e.g. ATM protocol segmentation/reassembly of packets into ATM cells).

The normal sequence of events for data transfer is illustrated in Figure 1-5.

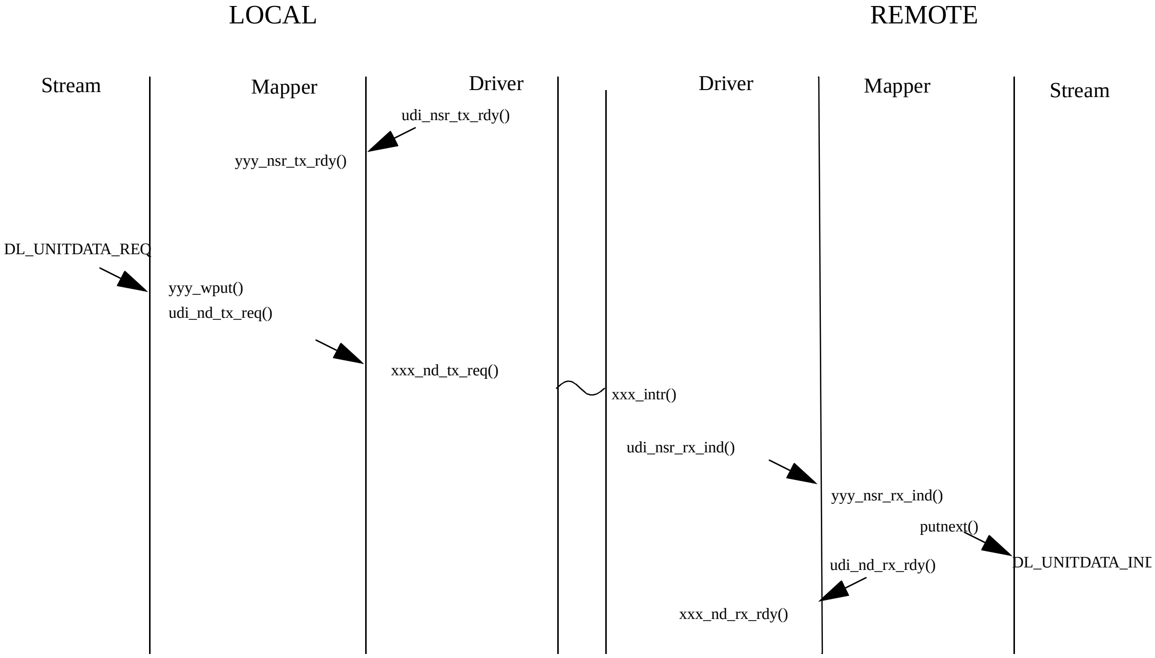

Figure 1-5 Connectionless data transfer operation

[3] Packet or frame level acknowledgement is differentiated from protocol-level acknowledgement; the latter is implemented by higher layers of the protocol stack than that represented by the ND. An example of protocol-level acknowledgement is the TCP ACK packet.

NAME

udi_net_tx_cb_t *Network transmit control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_net_tx_cb_t *chain;
     udi_buf_t tx_buf;
     udi_boolean_t completion_urgent;
} udi_net_tx_cb_t;
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may used the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

chain is a pointer to the next `udi_net_tx_cb_t` structure (and associated packet buffer) for this operation. The ND and NSR will use this field to "batch" a number of identical transmit requests or ready's into a single metalanguage operation. The ND, NSR, or environment are free to divide the chain at any point and implement explicit operations for each resulting portion of the chain, but performance concerns would indicate that processing the entire chain as a batch is highly desirable. The end of the chain is indicated by a NULL pointer.

tx_buf is the buffer describing the packet to be transmitted This field is ignored for the `udi_nsr_tx_rdy` operation, only for the `udi_nd_tx_req` operation.

completion_urgent is a hint to the ND that the NSR or UDI environment considers the associated packet buffer to be a critical resource and that it should be returned (via `udi_buf_free`) as quickly as possible after the transmit completes. This field is ignored for the `udi_nsr_tx_rdy` operation.

DESCRIPTION

The `udi_net_tx_cb_t` structure is used to pass packets from the NSR to the ND for transmission, and to handle flow control between the ND and the NSR by requiring the ND to supply the NSR with all usable `udi_net_tx_cb_t` structures.

This control block is declared with the `udi_net_tx_cb_init` routine.

NAME

udi_nsr_tx_rdy *Network driver ready to transmit packet*

SYNOPSIS

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

void  udi_nsr_tx_rdy (
     udi_channel_t target_channel,
     udi_net_tx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

Transmit data transfer channel

DESCRIPTION

The `udi_nsr_tx_rdy` is used to indicate to the NSR that the ND can transmit a packet when the NSR has a packet available. When the NSR receives a packet from the protocol stack that should be transmitted, it looks for a `udi_net_tx_cb_t` to associate with that packet. If no `udi_net_tx_cb_t` is available, it must queue the packet and exert flow control “back pressure” into the protocol stack. When a `udi_net_tx_cb_t` is available (either immediately or as a result of this command), the NSR may use that structure to pass the packet buffer to the ND for transmission.

The ND may chain multiple `udi_net_tx_cb_t` structures together as a linked list via the ***chain*** field in the structure and pass the entire chain to the NSR in a single `udi_nsr_tx_rdy` operation.

Following a `udi_nd_disable_req`, the NSR will return all transmit control blocks to the ND driver. This is done via the `udi_nd_tx_req` transmit path, but there will be no associated buffer for these operations.

Following a `udi_nd_unbind_req` or a `udi_channel_closed` operation, the ND and NSR are each responsible for deallocating the transmit control blocks held by each; no transmit control blocks are passed between the ND and NSR following these operations.

The ND may increase or decrease the number of transmit opportunities available to the NSR at any point in time by making correspondingly more or less numbers of transmit control blocks available to the NSR. Neither the ND or the NSR should expect the number of transmit control blocks to stay constant over the lifetime of the ND instance.

REFERENCES

`udi_nd_tx_req, udi_net_tx_cb_t`

NAME

udi_nd_tx_req *Network send packet*

SYNOPSIS

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

void  udi_nd_tx_req (
      udi_channel_t target_channel,
      udi_net_tx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Transmit data transfer channel

DESCRIPTION

The `udi_nd_tx_req` is called by the NSR to pass one or more packet buffers to the ND for transmission on the network. The NSR has already build the datalink frame header for the packet and obtained a `udi_net_tx_cb_t` from the ND to associate with the packet. The NSR issues the `udi_nd_tx_req` to the ND; once the packet has been successfully transmitted the ND will deallocate the packet buffer and pass the `udi_nd_tx_cb_t` back to the NSR in the `udi_nsr_tx_rdy` operation.

The ND may chain multiple `udi_net_tx_cb_t` structures together as a linked list via the *chain* field in the structure and pass the entire chain to the ND in a single `udi_nd_tx_req` operation; each `udi_net_tx_cb_t` has an associated buffer which represents a separate packet. When passing the `udi_net_tx_cb_t` structures back to the NSR via the `udi_nsr_tx_rdy` operation the ND is free to subdivide the chain and pass it back in pieces, or it may return the entire chain at once.

Each `udi_nd_tx_cb_t` contains a completion urgency hint field which is used to indicate to the ND that the corresponding packet transmission should be completed as quickly as possible to return the packet buffer. If this hint does not indicate any packet transmission urgency, the ND is not required to detect transmission completion in any set time period.

The ND is responsible for removing the buffer from the `net_tx_cb` structure and deallocating the buffer (`udi_buf_free`) before posting the structure back to the NSR via the `udi_nsr_tx_rdy` operation.

Note that in the special case where the NSR has initiated a `udi_nsr_disable_req` operation to this ND, it will return any transmit control blocks it owns via the `udi_nd_tx_req` operations; these control blocks have no attached buffers and are simply being recycled to the ND device.

REFERENCES

`udi_nsr_tx_rdy, udi_nd_exp_tx_req`

NAME **udi_nd_exp_tx_req** *Expedited data transmit request*

SYNOPSIS

```
#include <udi.h>

void udi_nd_exp_tx_req (
     udi_channel_t target_channel,
     udi_net_tx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of “*Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

Transmit data transfer channel

DESCRIPTION

The `udi_nd_exp_tx_req` is called by the NSR to pass one or more packet buffers to the ND for expedited transmission on the network. This operation is functionally equivalent to the `udi_nd_tx_req` except that the packet(s) associated with this request are “high-priority” and should be handled immediately by the ND, before any current or future low-priority traffic is handled.

REFERENCES

`udi_nd_tx_req`, `udi_nsr_tx_rdy`

NAME

udi_net_rx_cb_t *Network receive control block*

SYNOPSIS

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

typedef struct {
     udi_cb_t gcb;
     udi_net_rx_cb_t *chain;
     udi_buf_t rx_buf;
     udi_ubit8_t rx_status;
     udi_ubit8_t addr_match;
} udi_net_rx_cb_t;

/* values for rx_status */
#define   UDI_NET_RX_BADCKSUM        (1<<0)
#define   UDI_NET_RX_UNDERRUN        (1<<1)
#define   UDI_NET_RX_OVERRUN         (1<<2)
#define   UDI_NET_RX_DRIBBLE         (1<<3)
#define   UDI_NET_RX_FRAME_ERR       (1<<4)
#define   UDI_NET_RX_MAC_ERR         (1<<5)
#define   UDI_NET_RX_OTHER_ERR       (1<<7)

/* values for addr_match */
#define   UDI_NET_RX_UNKNOWN         0x0
#define   UDI_NET_RX_EXACT           0x1
#define   UDI_NET_RX_HASH            0x2
#define   UDI_NET_RX_BROADCAST       0x3
```

MEMBERS

gcb is the generic control block header which includes a pointer to the scratch space associated with this block and the channel context for the associated channel. The driver may use the scratch space while it owns the control block, but the values are not guaranteed to persist across channel operations.

chain is a pointer to the next `udi_net_rx_cb_t` structure (and associated packet buffer) for this operation. The ND and NSR will use this field to “batch” a number of identical receive indications or responses into a single metalanguage operation. The ND, NSR, or environment are free to divide the chain at any point and implement explicit operations for each resulting portion of the chain, but performance concerns would indicate that processing the entire chain as a batch is highly desirable. The end of the chain is indicated by a NULL pointer.

rx_buf is the buffer containing the packet that has been received. This field is used to pass an empty buffer to the ND in the `udi_nd_rx_rdy` operation which will be filled in with an incoming packet and returned with the `udi_nsr_rx_ind` indication.

rx_status is a bitmask indicating if there were any problems with the received packet. This field supports the following bit values:

UDI_NET_RX_BADCKSUM - Indicates that the adapter hardware or associated driver performed checksum calculation and validation and determined that the calculated checksum value for the packet did not match one or more of the corresponding checksum values indicated internally in the packet. If this status is not indicated for the packet it only indicates that there was no specific checksum failure detected by the ND or the adapter but not that the checksums have been validated to be correct. More explicit checksum information is usually specified via buffer status tags if available (see Section 1.2.6, “Hardware Checksum Offloads”).

UDI_NET_RX_UNDERRUN - Indicates that the packet is too short (did not meet minimum PDU size requirements) or that the reception terminated before the entire packet was received. Also known as a runt packet.

UDI_NET_RX_OVERRUN - Indicates that the packet is too large (exceeds maximum PDU size requirements).

UDI_NET_RX_DRIBBLE - Indicates that the frame contained a non-integer multiple of 8 bits (“dribbling” bits).

UDI_NET_RX_FRAME_ERR - Indicates that the packet had a framing error. These are technology-specific but include such causes as: incorrect frame header characters, incorrect frame trailer characters, invalid characters in the frame header or trailer, or an invalid CRC value for the frame.

UDI_NET_RX_MAC_ERR - Indicates that the packet was damaged by a media access error. These types of errors are network specific. A typical use of this bit is for a late collision that has occured after the valid collision period for a broadcast media technology (e.g. Ethernet).

UDI_NET_RX_OTHER_ERR - Indicates that some type of error was detected by the adapter or driver that does not fit into one of the other error classes. The ND is not expected to signal errors related to the operation of the local hardware or the driver; the ***rx_status*** field should be used only for packet or network related errors encountered during receives.

When the ND indicates a receive packet error via a non-zero ***rx_status*** value, no guarantees are made regarding whether data will be returned in the associated buffer and any data present should only be used for diagnostic purposes; under no circumstances should the this data be used as part of the normal transfer operations.

addr_match is an indicator of the ND’s knowledge of the MAC address match for the current packet. The ND is not required to know how the receive packet matched the set of acceptable addresses, but if a determination can easily be made from hardware registers or additional packet information, the ND can supply this information to the NSR in this field to optimize processing of the packet.

Valid values for this field are:

UDI_NET_RX_UNKNOWN - The ND does not know how the received packet matched

UDI_NET_RX_EXACT - The received packet exactly matched a valid unicast or multicast address for which the ND and associated adapter were registered.

UDI_NET_RX_HASH - The received packet matched based on a hashing algorithm and the address needs to be further verified by the NSR at some point.

UDI_NET_RX_BROADCAST - The received packet used the media broadcast address.

The addr_match field is a hint to the NSR to optimize processing of the packet, therefore the ND should never indicate any type of match that is not known to be valid (e.g. indicate an EXACT match when it was a BROADCAST). When in doubt, the ND should always use the UNKNOWN indicator.

DESCRIPTION

The udi_net_rx_cb_t structure is used to pass packets from the ND to the NSR after they have been received. They are returned to the ND via the udi_nd_rx_rdy response operation.

This control block is declared with the udi_net_rx_cb_init routine.

NAME

udi_nsr_rx_ind *Network receive packet indication*

SYNOPSIS

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

void  udi_nsr_rx_ind (
     udi_channel_t target_channel,
     udi_net_rx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Receive data transfer channel

DESCRIPTION

The `udi_nsr_rx_ind` is used when the ND has received a packet and is passing it to the NSR for handling by the protocol stack. The ***cb*** can be an individual receive control block or a chain of receive control blocks; each control block references a separate received packet. The packet buffer will be parsed and then demultiplexed to the interested protocol stack entities by the NSR, and the ***cb*** will subsequently be returned via the `udi_nd_rx_rdy` operation.

REFERENCES

`udi_nd_rx_rdy, udi_nsr_exp_rx_ind`

NAME

udi_nsr_exp_rx_ind *Network receive packet indication*

SYNOPSIS

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

void  udi_nsr_exp_rx_ind (
      udi_channel_t target_channel,
      udi_net_rx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the “Channel Operations” section of *“Standard Calling Sequences”* in the UDI Core Specification.

TARGET CHANNEL

Receive data transfer channel

DESCRIPTION

The `udi_nsr_exp_rx_ind` is used when the ND has received an expedited (high-priority) packet and is passing it to the NSR for handling by the protocol stack. The functionality of this operation is identical to the `udi_nsr_rx_ind` operation except that this interface should be used if the packet was determined to be urgent or high-priority.

REFERENCES

`udi_nd_rx_rdy, udi_nsr_rx_ind`

NAME

udi_nd_rx_rdy *Network receive packet response*

SYNOPSIS

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

void  udi_nd_rx_rdy (
     udi_channel_t target_channel,
     udi_net_rx_cb_t *cb );
```

ARGUMENTS

target_channel, cb are standard arguments described in the "Channel Operations" section of *"Standard Calling Sequences"* in the UDI Core Specification.

TARGET CHANNEL

Receive data transfer channel

DESCRIPTION

The udi_nd_rx_rdy is used by the NSR to provide one or more udi_net_rx_cb_t structures and associated receive buffers to the ND. The ND may then use these buffers to receive packets from the network and pass those packets to the NSR in udi_nsr_rx_ind or udi_nsr_exp_rx_ind indications. The ND should not allocate udi_net_rx_cb_t control blocks internally because these are regulated by the NSR to implement flow control.

The ND should also refrain from allocating receive buffers internally if possible. In any event, the incoming data should always be passed to the NSR in the same buffer (or a direct udi_buf_write/udi_buf_copy version thereof) that was originally passed to the ND by the NSR. If the ND has initially received the packet into a separate buffer it should issue a udi_buf_copy operation to copy that packet into the buffer provided by the NSR to send that packet to the NSR.

The NSR will not expect the ND to preserve or receive packets during any period of time wherein the NSR has not provided the ND with enough receive control blocks and buffers to accomodate the incoming data. It is recommended but not required that the NSR provide the ND with buffers that are at least ***max_pdu_size*** in length (as specified in the udi_net_bind_ack_cb_t) and that it provide at least ***rx_hw_threshold*** number of receive control blocks at any point in time.

Note that if a udi_nd_unbind_req or udi_channel_closed operation occurs, the NSR will not return the receive control blocks or buffers to the ND device and will deallocate them internally.

REFERENCES

udi_nsr_rx_ind, udi_nsr_exp_rx_ind

1.5 Network Trace Events

The following definitions describe usage of the trace events for the Network Interface Metalanguage and NIC Drivers. These trace events are defined in the "Trace Event Types" section of *"Tracing and Logging"* of the UDI Core Specification.

- UDI_TREVENT_IO_SCHEDULED
    - Trace indicating a new network transmit request is being handled.
- UDI_TREVENT_META_SPECIFIC_1
    - Trace indicating a network transmit request encountered an error.
- UDI_TREVENT_IO_COMPLETED
    - Trace indicating a network packet was received and is being handled.
- UDI_TREVENT_META_SPECIFIC_2
    - Trace indicating an incoming network packet has an error.
- UDI_TREVENT_STATE_CHANGE
    - Trace indicating the network link has changed state. If the first byte of the trace data is non-zero it indicates that the link has transitioned to an active state. If the first byte of the trace data is zero it indicates that the link has transitioned to a link inactive state.

1.6 Bindings for Instance Attributes

In each fo the attribute tables below, the **ATTRIBUTE NAME** is a null-terminated string (see Section 15.2, "Instance Attribute Names," on page 15-1 of the *UDI Core Specification Version 0.90*); the TYPE column specifies an attribute data type as defined in udi_instance_attr_type_t on page 15-6 of the *UDI Core Specification Version 0.90*; and the SIZE column specifies the valid size range, in bytes, for each attribute.

1.6.1 Enumeration Attributes

The UDI Network Adapter Driver enumerates network access points provided by that adapter. The following table specifies the enumeration attributes that can be specified for each network access point enumerated:

Table 1-3 Network Enumeration Attributes

ATTRIBUTE NAME	**TYPE**	**SIZE**	**Description**
!if_num	UDI_ATTR_UBIT32	4	Relative instance number for the adapter media interface.
!if_media	UDI_ATTR_UBIT32	4	Media type as defined in Table 1-1 on page 1-6.
!locator	UDI_STRING		Media string concatenated with !if_num where the media string is one of the entries in Table 1-4 on page 1-64.

The following table defines the media string prefixes for the different media types to be specified in the locator attribute:

Table 1-4 Media Strings

Media Type	**Locator Media String**
UDI_NET_ETHER	“eth”
UDI_NET_TOKEN	“tr”
UDI_NET_FASTETHER	“fe”
UDI_NET_GIGETHER	“ge”
UDI_NET_VGANYLAN	“vg”
UDI_NET_FDDI	“fddi”
UDI_NET_ATM	“atm”
UDI_NET_FC	“fc”
UDI_NET_MISCMEDIA	“net”

1.6.2 Child Attributes

There are no defined child instance attributes for the Network Adapter Metalanguage.

1.6.3 Filter Attributes

There are no defined filter attributes for the Network Adapter Metalanguage.

1.7 Instance Category

The recommended driver category to be specified in the configuration parameters for drivers implementing the Network Adapter Metalanguage is “Network”.