Texas Instruments

USB 2.0 RNDIS Driver Design

Version 1.0

June 29, 2005

Confidential

Important Notice

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TIs terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address:
Texas Instruments
Post Office Box 655303, Dallas, Texas 75265

About This Guide

This design guide describes the USB 2.0 RNDIS driver, TI's new initiative on USB front. The USB 2.0 device controller empowers communication processors with the distinct advantage of a widely popular interface and a much higher data rate than the earlier generation USB 1.1 device controllers used in current communication processors. In addition to driver details, this document describes various user interfaces and management mechanisms.

Note:

This document does not discuss either USB 2.0 Protocol or RNDIS, and does not describe how each specific routine should be implemented.

Deliverables

The deliverable of the project is the USB 2.0 RNDIS device driver.

Platform	Deliverables
VxWorks	USB 2.0 RNDIS END Driver
Linux	USB 2.0 RNDIS Net Driver

Terms and Abbreviations

Term/Abbreviation	Description
CLI	Command Line Interface
END	Enhanced Network Driver (VxWorks)
HAL	Hardware Abstraction Layer
NDIS	Network Driver Interface Specification (Microsoft)
RNDIS	Remote NDIS
USB	Universal Serial Bus

Related Documentation

- USB 2.0 Specification
  (http://www.usb.org/developers/)
- RNDIS Specification
  (http://www.microsoft.com/hwdev/resources/hwservices/rndis.asp)
- USB 2.0 Silicon Design
  (http://www.dal.asp.ti.com/dsl/projects/ip-mod/usb20-device/module-usb20.htm)
- CPPI USB 2.0 HAL API Document
  (http://www.india.ti.com/~sabya/project/USB/DOCS/usb_2.0_hal.pdf)

Revision History

Revision	Author	Date	Comments	Version
0.1	Sabyasachi Dey	Jul 31, 2003	Initial draft created.	0.1
0.2	Sabyasachi Dey	Sep 10, 2003	Modified Protocol Driver section for APIs, buffer management, task description. Completed IOCTL commands.	0.2
1.0	Eyal Reizer	June 29, 2005	Updated	1.0

Trademarks

Table of Contents

List of Figures

Chapter 1

System Overview

USB mandates a host-driven communication flow and allows only pull-mode data transfer. That means the data is pulled from a device by the host. The host initiates any communication to and from any device. The USB elements hierarchy is shown in Figure 1-1. There can be only one host in any USB system.

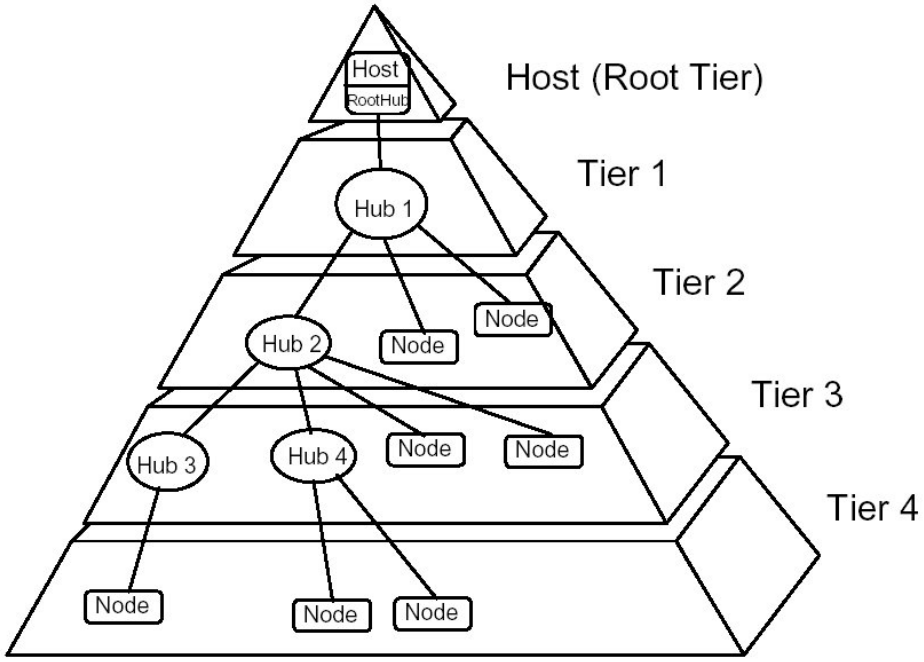

Figure 1-1. USB Bus Topology

The USB host is the root-most point in the USB connectivity hierarchy. The USB devices are connected in a tiered star topology. The root hub (integrated in the host) is the root of this tree. Data communication always takes place between a device and USB host.

The USB 2.0 device can only communicate with a 2.0 host controller.

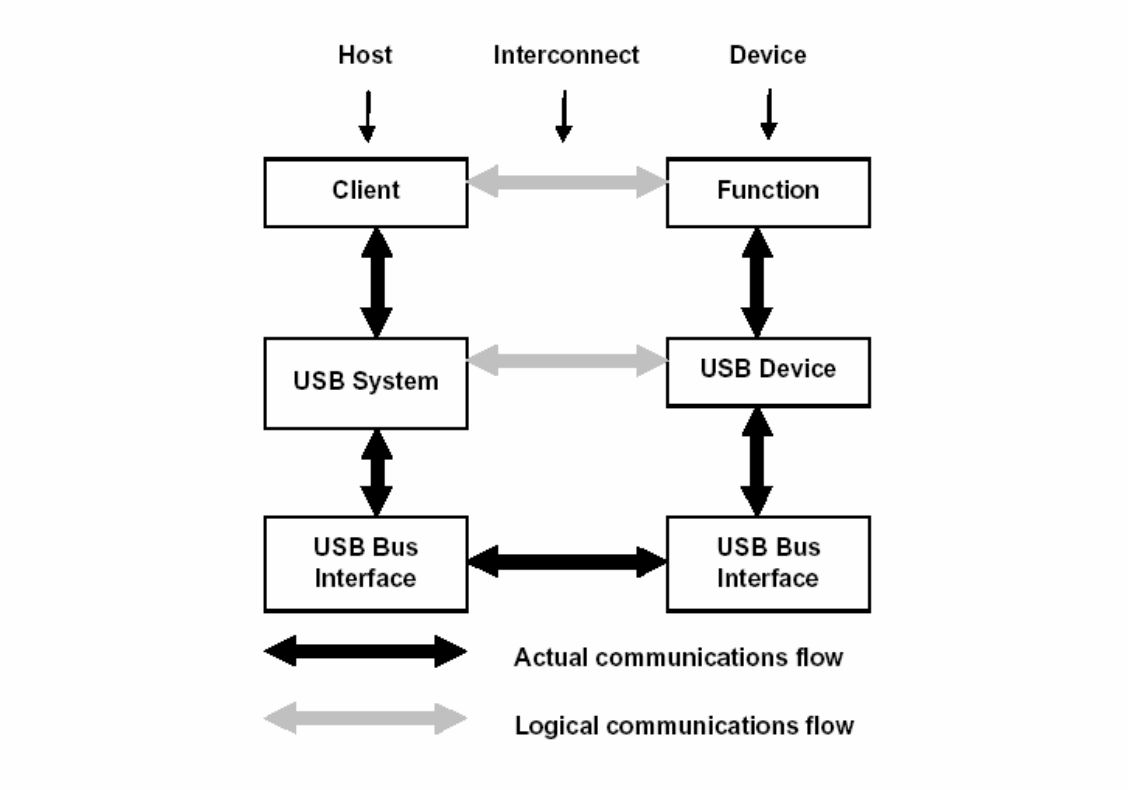

Figure 1-2. Data Flow in USB

The software developed is a driver for a USB 2.0 device function. In addition, the driver contains an RNDIS driver to communicate with the RNDIS protocol stack on the host side. The software also comprises a network interface to bind the software to the device operating system (VxWorks, Linux, WinCE) IP Protocol Stack.

Chapter 2

Design Considerations

2.1 Assumptions and Dependencies

The following assumptions are made for USB 2.0 software design:

- A well-documented HAL (Hardware Abstraction Layer) is available.
- Implementation is performed only in ANSI C. However, no assumptions should be made regarding the compiler.
- Only the management and configuration interface is exposed to the end user. Proper documentation should accompany the software for the users reference.

2.2 Hardware Limitations

- The hardware does not support an MIB-II counter.
- All USB 2.0 CPPI channels must be torn down together. Individual channel tear-down is not possible.

2.3 Design Goals and Guidelines

The following design goals and guidelines were considered when designing USB 2.0 RNDIS driver software. These design goals and guidelines are applicable at a macro level and deviation from them may be required at the micro level in the submodules in the USB software. Some of the guidelines are also applicable for coding.

- Portability across VxWorks, Windows CE, Linux
- Standard compliance (USB 2.0, RNDIS)
- Quality strategies conforming to TII processes
- Hooks for easy debugging and testing
- Performance and scalability

- Focus on space optimization, since memory becomes expensive for SOCs
- Reusable and modular design of software
- Localize C variables, to enable the compiler to optimize register and stack usage
- Try not to use more than four formal parameters for C functions
- Try to align buffers and commonly used data structures on a 16-byte boundary rather than the commonly used 32-bit boundary. This uses MIPS cache lines more efficiently. (Although the driver does not make any assumption about the processor, this is useful as all TI communication processors are based on MIPS. Similar processors tend to offer 16-byte cache lines, making this proposition more valuable). All routines should return a status and should not return void.
- The driver should not try to use direct C data types. Instead it should use either OSAL- or PSP-defined data types.

Chapter 3

System Architecture

The USB 2.0 driver software enables application software to communicate with the host using the USB device function as an Ethernet-type communication interface. The high-level architecture of the driver software is shown in the following diagram.

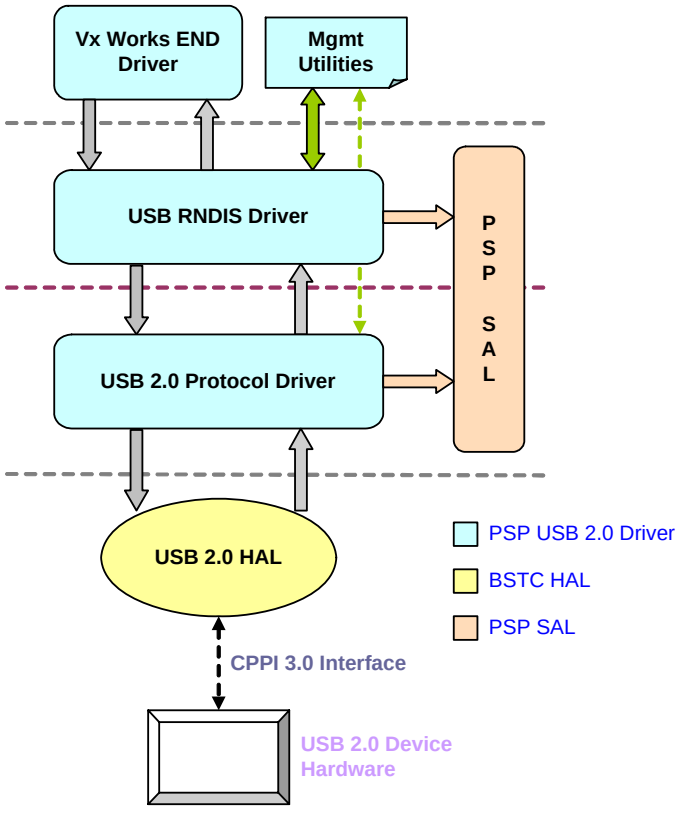

Figure 3-1. Architecture Overview

As shown in Figure 3-1, the USB system software consists of the following components:

- **USB 2.0 HAL:** CPPI 3.0-based USB 2.0 CP HAL. This provides support for accessing 2.0 Silicon in an abstract hardware-neutral way. Data communication is governed by TI proprietary CPPI-based technology. This is same as other CP HALs, except regarding endpoint 0. This HAL defines a set of routines to control, receive and transmit data over endpoint 0.

  Note:

  USB HAL is not discussed further in this document.

  Details of HAL APIs and data structures can be found in the HAL API document (refer to the *Related Documents* section). This module is operating-system-neutral, but hardware-dependent.

- **USB Protocol Driver:** This module implements the USB 2.0 protocol. It takes care of endpoint initialization, enumeration and control of data transfer. The upper-layer RNDIS stack depends on this layer for communicating with the other side of the RNDIS pipe. This module is hardware-independent and OS-abstracted.

- **USB RNDIS Driver:** The RNDIS driver enables the USB 2.0 device function to implement an Ethernet-type interface, which can provide TCP/IP connectivity. The driver software in discussion implements the RNDIS stack on top of the USB 2.0 protocol. This module is hardware-independent and OS-abstracted.

- **VxWorks END Driver:** The USB RNDIS driver implements the END interface to allow VxWorks applications to transfer IP packets over this communication interface. This is the only OS-dependent part of the driver software.

- **Management and Configuration Utility:** This is a simple command line interface (CLI). This utility is helpful in status monitoring, event logging, statistics, device testing, and so on. Some data structures in the USB driver software are accessible through this interface. The management utilities may vary from release to release. In the future, a web-based interface may be exposed.

A well-defined and clean interface is maintained between all layers in the driver stack.

Throughout the document the word “task” and “thread” are used interchangeably, and always mean exactly the same entities. An independent control of action. And of course each “thread” or “task” can see each other’s memory.

Note:

The detailed startup sequence is platform-specific and will be added later. However, there will be an option to start the USB driver during system boot time, or later as a standalone driver. This can be a boot parameter like bootline in VxWorks.

Chapter 4

Detailed Design

4.1 USB Protocol Driver

4.1.1 Component Description

The USB protocol driver is a USB 2.0 protocol-specific implementation of the driver. This module is responsible for silicon initialization, device function enumeration, control and bulk data transfer with the USB host.

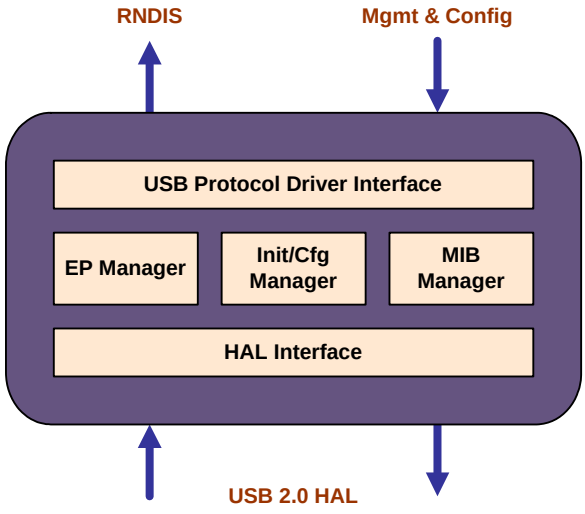

Figure 4-1. USB 2.0 Protocol Driver

The USB protocol driver has the following high-level functional components, as shown in Figure 4-1:

- **USB Protocol Driver Interface (UPDI):** This interface enables any upper-layer component to interact with the host-side driver using the USB protocol driver. This interface also enables multiple higher-layer components to connect to the same USB 2.0 protocol driver layer. As a result, the USB protocol driver can be reused for various device functionalities in addition to RNDIS. The interface has routines to initialize the protocol driver, as well as all necessary data transfer and control routines. Upper-layer drivers are normally USB class drivers.
- **Endpoint (EP) Manager (EPM):** The endpoint manager performs all control and bulk data transfer with the HAL. All data packets are delivered to upper-layer class drivers using multiplexing pipes. Control packets targeted to the protocol driver are either processed by it or passed over to the upper-layer class driver.
- **Initialization and Configuration Manager (ICM):** This module initializes the USB 2.0 hardware with the help of HAL. USB enumeration, device class information and configuration information is kept in a configuration repository. This repository is modifiable dynamically through the management interface.
- **MIB Manager:** Implement MIB-II RFCs. MIB statistics are available through OS-specific driver interfaces (END driver IOCTL, and so on), or through USB management APIs.
- **HAL Interface:** This module interfaces with the USB 2.0 CPPI HAL.

The protocol driver actually implements the USB device function. It initializes all endpoints, responds to all host driver queries, and enables class drivers to establish communication.

4.1.2 Interface Description

All USB protocol interface routines are defined below.

NAME	**`usb_drv_init`**	Initializes the USB protocol driver.

SYNOPSIS	**`USB_STATUS`**	`usb_drv_init`
		`(`
		`  [in]  CLASS_DEV  *class_dev,`
		`  [out] USB_DEV  **dev`
		`);`

	class_dev	This structure contains all class driver-related information (such as, RNDIS communication class driver, and so on).
	dev	USB device structure. This structure contains all device-specific information and is passed during initialization.
DESCRIPTION	USB protocol driver initialization happens when this routine is called. Before calling this routine, nothing can be assumed about the device state. The class driver calls this function to initialize the USB protocol stack and to populate all class driver related information.	
RETURN VALUES	USB_STATUS_OK: Initialization successful.	
	Negative value: Error. See error list.	

NAME	**usb_drv_start**	Class drivers register themselves by calling this protocol driver API.
SYNOPSIS	**USB_STATUS**	usb_drv_start ([in] USB_DEV *p_dev, [in] USB_CONFIG *usbcfg);
	p_dev	Pointer to the USB device previously obtained by calling usb_drv_init.
	usbcfg	Pointer to USB protocol configuration information (descriptors).
DESCRIPTION	Starts the protocol driver. Brings it to a state for data transfer. Interrupt service routines are connected to the OS. Timers (if they exist) are started.	
RETURN VALUES	USB_STATUS_OK: Start succeeded.	
	Negative value: Error. See error list.	

NAME	**usb_drv_stop**	Class driver (RNDIS) stops the protocol driver.
SYNOPSIS	**USB_STATUS** usb_drv_stop ([in] USB_DEV *p_dev);	
	p_dev	Pointer to the USB device obtained through the previous call to usb_drv_init().
DESCRIPTION	Class drivers stop the protocol driver. It is then in a state where no data transfer is possible. All pending transfers are aborted. Interrupt service routines are deregistered with the OS. Any running timers are canceled.	
RETURN VALUES	USB_STATUS_OK: Stop operation successful.	
	Negative value: Error. See error list.	

NAME	**usb_drv_shutdown**	Shuts down the protocol driver.
SYNOPSIS	**USB_STATUS**	usb_drv_shutdown ([in] USB_DEV *p_dev);
	p_dev	Pointer to the USB device obtained through the previous call to *usb_drv_init()*.
DESCRIPTION	Shuts down the driver. All memory and buffers are released. Any other OS resources (semaphore, mutex, timer, and so on) are released. HAL is lost.	
RETURN VALUES	USB_STATUS_OK: Clean shutdown.	
	Negative value: Error. See error list.	

NAME **usb_drv_send** Class drivers send data over USB using this routine.

SYNOPSIS

```
USB_STATUS
usb_drv_send
(
   [in] USB_DEV      *p_dev,
   [in] USB_EP  *ep,
   [in] USB_PKT      *pkt,
   [in] VOID              *sendInfo
);
```

p_dev — The descriptor obtained after calling usb_drv_init().

ep — Endpoint (opened before) over which communication takes place.

pkt — USB data packet. Class drivers (such as RNDIS) create this packet, which has a packet ID and a data buffer.

sendInfo — Any private information. This is returned to the caller with the send_complete() routine.

DESCRIPTION Send is a non-blocking call. This means, that send returns immediately. However, data transmission over the USB bus may not be complete. When data transfer is completed, the USB protocol driver calls the send_complete() routine corresponding to the class driver.

RETURN VALUES USB_STATUS_OK: Send accepted by the protocol driver.

Negative value: Error. See error list.

NAME **usb_drv_ioctl** This API provides a strong control/configuration interface for the protocol driver.

SYNOPSIS

```
USB_STATUS
usb_drv_ioctl
(
   [in] USB_DEV     *p_dev,
   [in] UINT32          command,
   [inout] VOID         *data
);
```

p_dev	Pointer to the USB device obtained through the previous call to *usb_drv_init()*.
command	USB device structure. This structure contains all device-specific information and is passed during initialization.
data	Data buffer related to the command.

DESCRIPTION USB protocol driver initialization is performed when this routine is called. Before calling this routine, nothing can be assumed about the device state. A detailed list of IOCTL commands can be found in Chapter 6.

RETURN VALUES USB_STATUS_OK: Success.

Negative value : Error. See error list.

NAME **usb_drv_ep_open** Callback routine to be called by HAL.

SYNOPSIS

```
USB_STATUS
usb_drv_ep_open
(
   [in] USB_DEV  *p_dev,
   [in] USB_PIPE  *pipe,
   [out] USB_EP    **p_ep
);
```

p_dev	Pointer to the USB device obtained through the previous call to usb_drv_init().
pipe	Pointer to the PIPE structure. A pipe is defined by the class driver and sent to the USB protocol driver. The pipe is a bi-directional communication channel between the protocol driver and the class driver.
p_ep	Pointer to the initialized endpoint.

DESCRIPTION USB protocol driver initialization happens when this routine is called. Before calling this routine, nothing can be assumed about the device state.

RETURN VALUES USB_STATUS_OK: Success.

Negative value: Error. See error list.

NAME **usb_drv_ep_close** Callback routine called by HAL.

SYNOPSIS

```
USB_STATUS      usb_drv_ep_close
                (
                    [in] USB_EP  *p_ep

                );
```

dev — USB device structure. This structure contains all device-specific information and is passed during initialization.

DESCRIPTION USB protocol driver initialization is performed when this routine is called. Before calling this routine, nothing can be assumed about the device state.

RETURN VALUES USB_STATUS_OK: Success.

Negative value: Error. See error list.

NAME **usb_drv_send_complete** Routine to be called by HAL

SYNOPSIS

```
INT32           usb_drv_send_complete_cb
                (
                    [in] USB_PIPE  *pipe,
                    [in] VOID   *sendInfo

        );
```

sendInfo Pointer to OS-specific information. This structure was passed to the CPHAL via Send(), and returned to the OS here.

DESCRIPTION Indicates to the class driver that the transmission is complete. The USB driver may not have checked the return value. This routine is implemented by the class driver for each communication pipe (**endpoint**) that the class driver intends to send data through.

RETURN VALUES 0: Initialization successful.

Non-zero: Error. (Currently, not checked by the USB protocol driver).

NAME **usb_drv_os_receive** OS receive function called by HAL.

SYNOPSIS

```
INT32           usb_drv_os_receive
                (
                    [in] OS DEVICE  *OsDev,
                    [in] FRAGLIST  *FragList,
                    [in] bit32u             FragCount,
                    [in] bit32u            PacketSize,
                    [in] HAL_RECEIVEINFO HalReceiveInfo,
                    [in] bit32u             mode

        );
```

OsDev	OS device structure (refer HAL document).
FragList	Array of structure consisting of three 32-bit words (containing length, data address, and OsInfo).
FragCount	Number of {length,data} pairs in FragList.
PacketSize	Number of bytes received.
HalReceiveInfo	Pointer to HAL information, returned to the HAL via RxReturn().
Mode	The lower eight bits of this value is the channel number.

Note: Mode and 0xff = channel number undocumented.

DESCRIPTION USB protocol driver initialization happens when this routine is called. Before calling this routine nothing can be assumed about the device state.

RETURN VALUES 0: Success.

Non-zero: Error. See error list.

Note:

For HAL OS service routines, refer to HAL documentation.

4.1.3 Design Description

This section defines key data structures used by USB Protocol Driver and important data/control flow diagrams.

Key Data Structures:

- USB_DEV
- USB_EP
- USB_PKT

USB_DEV: Defines the USB protocol driver master control block.

```
struct USB_DEV_T
{
    OSAL_MUTEX              mutex;
    HAL_DEVICE              hal_dev;
    HAL_FUNCTIONS           *hal_funcs;
    OS_FUNCTIONS            *os_funcs;
    USB_CONFIG              *usb_cfg;
    USB_CD_CONFIG           *cd_cfg;      /* Class Driver Configuration */
    USB_EP                  p_control;   /* Control End Point */
    USB_EP                  ep_table[USB20_NUM_EP];
    USB_CCPU                *ccpu;        /* Control Command Processor Unit */
    USB_CTL_TX_QUEUE_NODE   *ctl_tx_queue;
    USB_CTL_TX_QUEUE_NODE   *ctl_tx_free_list;
    USB_PARAMS              params;
    USB_STATS               stats;
    USB_CHANNEL             channel[USB20_NUM_CHANNEL]; /*Status per channel*/
```

```
    UINT32                status;       /* Status of driver */
    UINT32                link_status;  /* Link is up or down */
    VOID                  *priv;        /* Private Data for USB */
} ;
```

USB_EP: Defines the USB endpoint structure.

```
struct USB_EP_T
{
    UINT8       id;        /* End Point Number */
    UINT8       type;      /* BULK | CONTROL | INT */
    UINT8       dir;       /* End Point Direction */
    UINT8       max_size;  /* Maximum Transfer Size on this End Point */
    USB_PIPE   *pipe;      /* Pipe configured by Class Driver */
    USB_DEV    *usb_dev;   /* Pointer to USB Device structure, driver MCB */
    CHANNEL_INFO *channel; /* HAL Specific channel information, type is opaque
                        to upper layer */
    UINT32      state: 8;
    UINT32      status: 24;
} ;
```

USB_PKT:

```
struct USB_BUF_T {
    UINT8   *data;
    UINT32  len;
    void    *info;
} ;

struct USB_PKT_T
{
    USB_BUF  *buf_list;
    UINT32   num_buf;
    UINT32   pkt_size;
} ;
```

Bipartite Buffer Management

Buffer management responsibilities are split between the protocol driver and the class driver. For bulk and isochronous data transfer, buffer management is performed by the class driver, whereas for control and interrupt data transfer, buffer management is performed by the protocol driver. This eliminates the need for the protocol driver to understand the details of bulk/isochronous data transfer policies and mechanisms. However, the protocol driver facilitates a generic buffer management policy for the class driver.

Driver Tasks

The driver has two tasks, as follows:

- **Control Task:** All USB interrupts (basically, control endpoint-related interrupts) are serviced in this task. Task Priority for this should be high, as all such interrupts and associated responses must meet strict timing requirements.
- **Data Task:** Reception of bulk/isochronous packets and dispatching of them to upper layers is performed in this task. This does not require high priority.

Both the above tasks are configurable and can be excluded/included in the driver configuration (optional).

4.1.3.1 Enumeration Sequence

The following diagram describes USB device enumeration sequence.

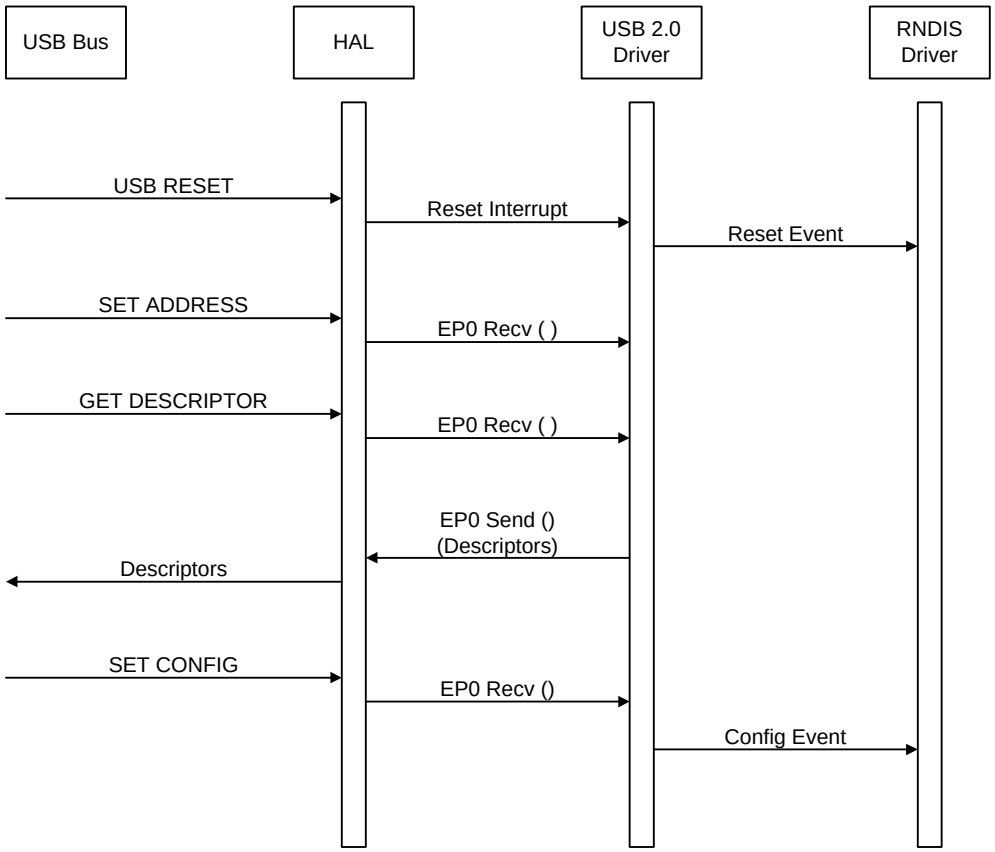

Figure 4-2. USB Device Enumeration

4.1.3.2 Control Data Transfer Sequence

The following diagram describes the control data transfer sequence

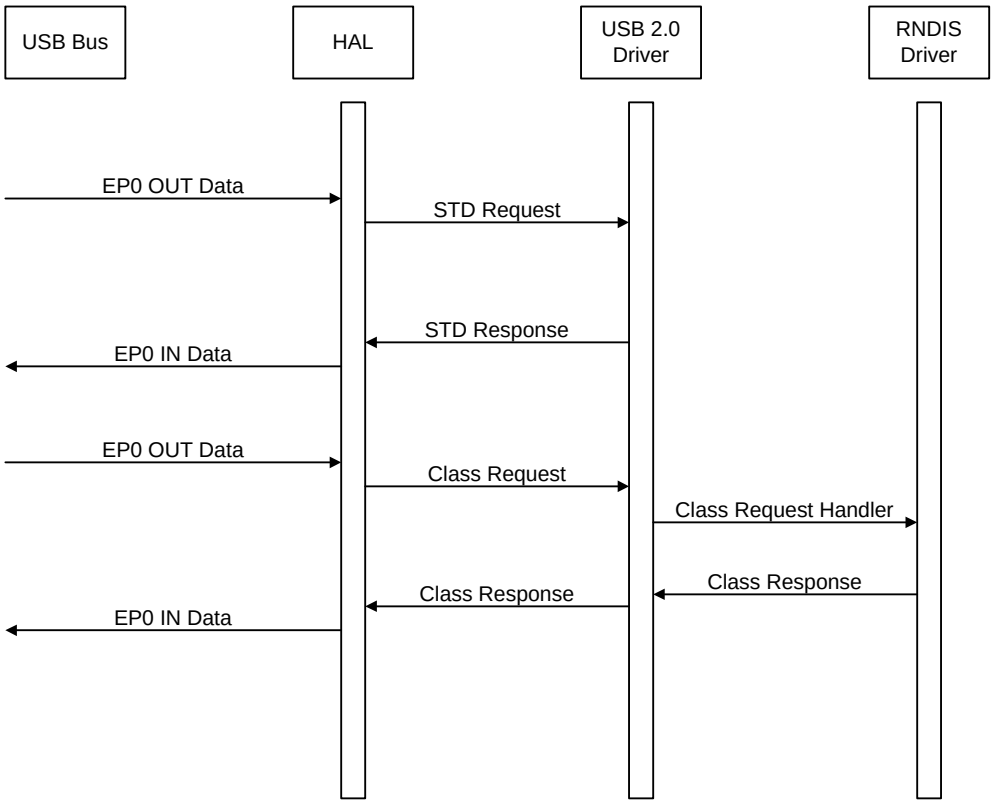

Figure 4-3. USB Control Data Flow

4.1.3.3 Bulk Data Transfer Sequence

The following diagram describes the bulk data transfer sequence

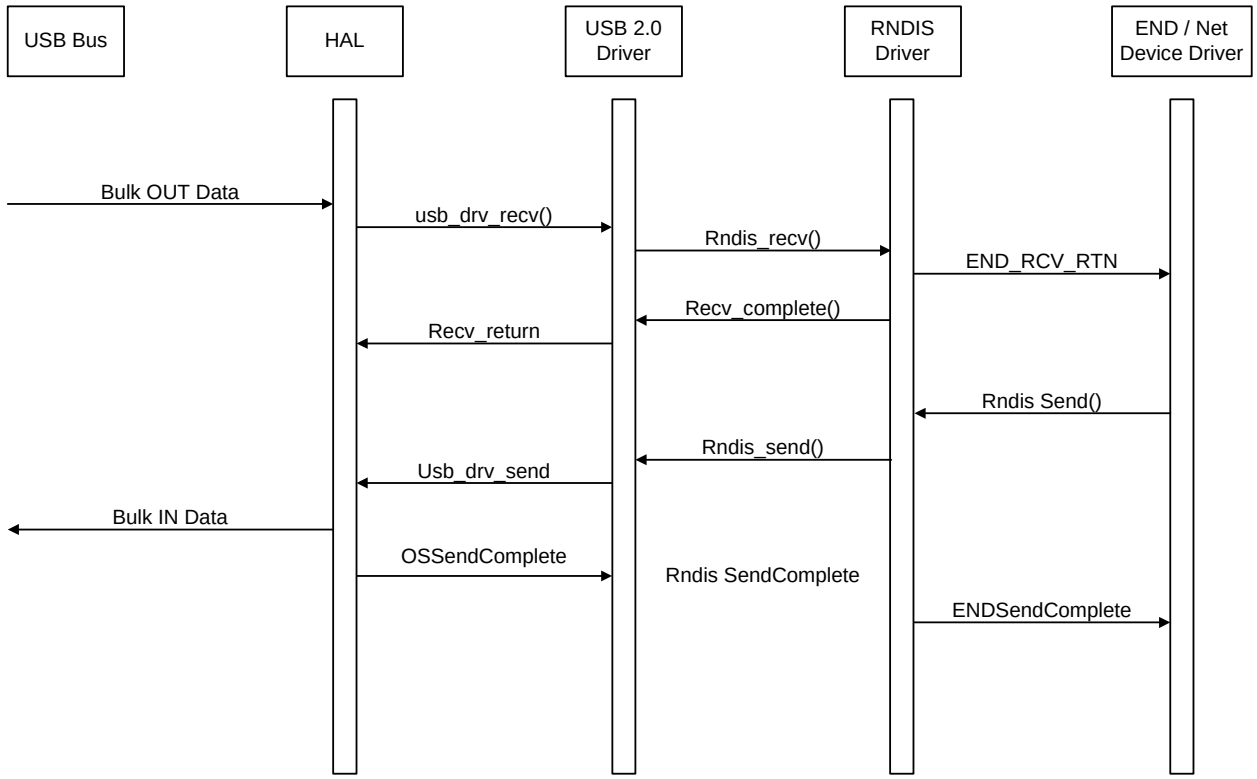

Figure 4-4. Bulk Data Transfer Sequence Diagram

4.2 USB RNDIS Driver

4.2.1 Component Description

The PSP USB RNDIS protocol stack implements the RNDIS 1.1 protocol specification from Microsoft. All RNDIS components are shown in Figure 4-5.

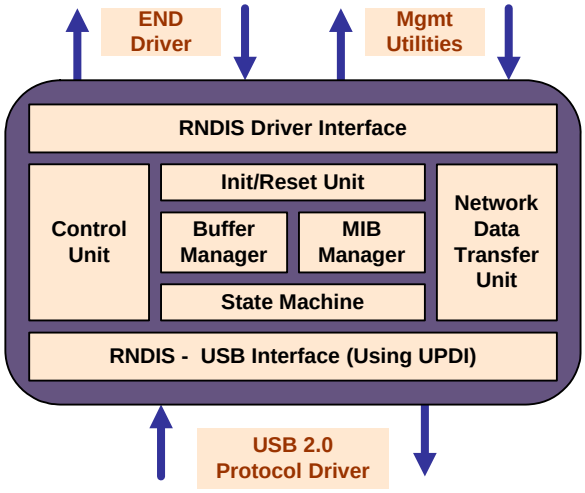

Figure 4-5. USB RNDIS Driver

USBD has the following functional components:

- **Network Data Transfer Unit:** This unit provides a communication channel through logical pipes and IRPs. Client software sends and receives data to/from devices through this module.
- **Control Unit:** This unit provides a set of APIs for pipe management, data transfer, and so on.
- **Buffer Manager:** Controls the bus with the help of HCD. This module manages bus state. In addition, if there is an error on the bus, this module notifies all clients about the bus-related events, such as suspension, resumption and reset.
- **MIB Manager:** This unit maintains the active topology. It is responsible for configuring newly attached devices and adding them to the active topology, and for removing detached devices from the active topology.
- **Init/Reset Unit:** Manages all hubs attached to the bus. The hub driver runs as an independent separate task. If a new device is attached, the hub is first notified through the control pipe of the hub. When a device is detached, the hub driver is notified. If the detached device is a hub, this unit ensures that all children of the hub are also disconnected and removed from the active topology.
- **State Machine:** This consists of a set of routines that calculate bandwidth requirements and availability for a particular endpoint. Depending on this, a particular bus transaction may be accepted or refused.

- **RNDIS Driver Interface:** These APIs form a library that is used by both the HCD as well as the root hub driver. The API set includes bandwidth management routines, HCD registration routines, bus and device structures management routines, device configuration routines, and so on.
- **USB Protocol Driver Interface:** Exposes a set of routines that enable an application to list all devices attached, bus status, modify device configuration, and to see the USBD log.

4.2.2 Interface Description

USBD exposes two independent sets of interfaces. The first is called the USBD Interface (USBDI). This interface is documented and published. The second is a collection of management and configuration routines. This interface is neither documented nor published. This second interface is intended to enable TI internal teams to provide utilities and a framework for USB management and configuration.

RNDIS Driver Interface

NAME	**usb_rndis_init**	Initializes RNDIS driver.

SYNOPSIS	**RNDIS_HANDLE**	`usb_rndis_init`
		`(`
		`  [in] rndis_net_t  *rndis_net`
		`);`

	rndis_net	This structure contains all information required for initializing the driver and is passed during initialization. This includes the callback routines and configuration data.
DESCRIPTION	RNDIS driver initialization is performed when this routine is called. It is typically called as part of the END driver load function (Ref: VxWorks Network Programming). The callback routines are for the following functionality: • OS (system) sending complete notification • OS (system) receiving data • Getting buffers (for receiving) • Freeing-up of buffers • Event transfer callback	
RETURN VALUES	Non-null value: Initialization successful. NULL value: Error. See error list.	

NAME	**usb_rndis_start**	Starts the RNDIS driver.

SYNOPSIS	**RNDIS_STATUS**	`usb_rndis _start (`
		`  [in] RNDIS_HANDLE  handle`
		`);`

	handle	Handle to the RNDIS layer.

DESCRIPTION This starts the RNDIS driver and brings up the stack so that data transmission can be started.

Here, the endpoints are opened for the control and data channels. Callback routines for these endpoints are registered with the calls to **usb_drv_ep_open**.

RETURN VALUES RNDIS_STATUS_OK: Call successful.

Negative value: Error. See error list.

NAME **usb_rndis _stop** Halts the RNDIS driver.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_stop (
                   [in] RNDIS_HANDLE  handle
```

handle Handle to the RNDIS layer.

DESCRIPTION This is a complementary function to rndis_usb_start. The function halts the RNDIS stack and disables all further bulk data transactions.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_shutdown** Unloads the RNDIS driver.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_unload (
                   [in] RNDIS_HANDLE handle
```

handle Handle to the RNDIS layer.

DESCRIPTION This is a complementary function to rndis_usb_load. The function cleans up all the initializations performed in rndis_usb_load.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_send** Sends a data packet over the USB bus using the RNDIS protocol.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_send
                (
                   [in] RNDIS_PKT   *packet,
                );
```

packet Array of buffers containing the packet.

DESCRIPTION This function is used by the OS-specific driver to send a packet over the USB bus (as bulk data).

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_ioctl** Configures/controls API to the RNDIS driver.

SYNOPSIS

```
RNDIS_STATUS        usb_rndis_ioctl
                    (
                      [in] RNDIS_HANDLE handle,
                      [in] INT32                    command,
                      [inout] UINT32          *data
                    );
```

command — Identifier for the command to execute.

data — Data for the command.

DESCRIPTION This API gives an interface to the OS driver to configure and control the RNDIS driver. It also provides an interface to query for some data (such as, MIB data). It also acts as a method of reporting events to the RNDIS-USB drivers.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

USB Protocol Driver Interface

NAME **usb_rndis_notify_cb** Routine for the USB protocol driver to notify RNDIS about events.

SYNOPSIS

```
RNDIS_STATUS        usb_rndis_notify_cb
                    (
                       [in] INT32     event,
                       [in] UINT32  *data

                    );
```

command — Identifier for the event.

data — Data associated with the event.

DESCRIPTION This function is registered with the USB protocol driver during initialization. The USB driver uses this API to notify the RNDIS driver about events that occur at the USB layer, hardware layer, and so on. A typical example of an event that is notified is the USB reset event.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_query_cb** Routine for the USB protocol driver to query RNDIS for information.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_query_cb
                (
                   [in] INT32       query_id,
                   [out] UINT32  *data
                );
```

command — Identifier for the query.

data — Data associated with the query.

DESCRIPTION This function is registered with the USB protocol driver during initialization. The USB driver uses this API to query the RNDIS driver for information it requires. A typical example of a query is when the USB driver queries for descriptor data.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_control_rcv_cb** Function to receive data on the control endpoint.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_control_rcv_cb
                (
                  [in] ep_t * ep
                  [in] RNDIS_PKT  *pkt,
                );
```

ep — Handle to the endpoint.

pkt — Packet with received data.

DESCRIPTION This function is registered with the USB protocol driver during initialization as the receive callback function for EP0. The USB driver uses this API to send the RNDIS driver data from EP0 endpoint.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_control_tx_complete_cb** This function notifies completion of transmit on the control endpoint.

SYNOPSIS

```
RNDIS_STATUS    usb_rndis_control_tx_complete_cb
                (
                   [in] ep_t   *ep
                   [in] VOID  *sendInfo,

        );
```

ep — Handle to the endpoint.

sendInfo — Private information sent during send call.

DESCRIPTION During initialization, this function is registered with the USB protocol driver as the transmit-complete notification callback function for EP0. The USB driver uses this API to indicate the completion of the transmit to the RNDIS driver, and also whether any error occurred during transmit.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_int_tx_complete_cb** This function notifies of completion of transmit on the interrupt endpoint.

SYNOPSIS

```
RNDIS_STATUS        usb_rndis_int_tx_complete_cb
                    (
                      [in] ep_t * ep,
                      [in] VOID *sendInfo,
                    );
```

ep — Handle to the endpoint.

sendInfo — What was passed during the send() call.

DESCRIPTION During initialization, this function is registered with the USB protocol driver as the transmit-complete notification callback function for EP0. The USB driver uses this API to indicate the completion of the transmit to the RNDIS driver, and also whether any error occurred during transmit.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_bulk_rcv_cb** Function to receive data on the bulk endpoint.

SYNOPSIS

```
INT32               usb_rndis_bulk_rcv_cb
                    (
                      [in] ep_t * ep
                      [in] RNDIS_PKT  *pkt,

                    );
```

ep — Handle to the endpoint.

pkt — Packet with received data.

DESCRIPTION This function is registered with the USB protocol driver during initialization as the receive callback function for the bulk/data endpoint. The USB driver uses this API to send the RNDIS driver data from the bulk OUT endpoint of the RNDIS data interface.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

NAME **usb_rndis_bulk_tx_complete_cb** Function to notify of completion of transmit on the bulk endpoint.

SYNOPSIS

```
INT32                    usb_rndis_bulk_tx_complete_cb
                          (
                            [in] ep_t * ep
                            [in] int pkt_id,
                            [in] int tx_status
                         );
```

ep	Handle to the endpoint.
buf	ID of the packet that was sent.
tx_status	Error status of the transmission.

DESCRIPTION During initialization, this function is registered with the USB protocol driver as the transmit-complete notification callback function for the bulk endpoint. The USB driver uses this API to indicate the completion of the transmit of bulk data to the RNDIS driver, and also whether any error occurred during the transmission. Here, the OS END driver is notified of the transmit-complete event and the freeing-up of buffers can take place.

RETURN VALUES RNDIS_STATUS_OK: Success.

Negative value: Error.

4.2.3 Design Description

The USB driver is built around several key data structures. This section describes the key data structures maintained by the USB driver.

```
typedef struct RNDIS_MCB
{
    /********************* USB Descriptors ******************************/
    USB_DEVICE_DESCRIPTOR   *ptr_device_desc;
    RNDIS_USB_CONFIG*ptr_config_desc;
    HAL_USB_STRING_ENTRY*ptr_string_desc;

    /********************** HANDLE **************************************/
    USB_HND         h_drv;        /* USB Driver handle                  */
    HAL_HND         h_end;        /* Handle of the END driver.          */
    HAL_HND         ctrl_in;      /* Control IN endpoint handle         */
    HAL_HND         ctrl_out;     /* Control OUT endpoint handle        */
    HAL_HND         intr_in;      /* INTERRUPT endpoint handle          */
    HAL_HND         bulk_in;      /* BULK IN endpoint handle            */
    HAL_HND         bulk_out;     /* BULK OUT endpoint handle           */


    /********************* RNDIS Protocol *******************************/

    RNDIS_STATE       state;               /* RNDIS Stack Current State          */
    UINT8             host_macadd_cur[6];  /* HOST MAC Address            */
    UINT8             host_macadd_default[6];   /* Permanent HOST MAC Address   */
```

```
    UINT32            packet_filter;       /* PACKET Filter (Multicast / Promiscuous
/ Broadcast / Directed */

/* These lists handle the responses for control messages. The FREE list is a list
* of responses messages that are free and can be used by the function layer. The
* response lists contains all the responses that have been queued and will be txed
* to the HOST as and when the HOST will request for them through a GET_ENCAPSULATED
* message. */
RNDIS_RESPONSE* p_free_list;
RNDIS_RESPONSE* p_response_list;

/* Notification response to be sent to the HOST is always constant. Create it once
* and store it in the structure. */
USB_SETUP         response_available;  /* To be sent over interrupt endpoint */

/* Transmit - For BULK data. */
RNDIS_DATA_HEADER     *ptr_bulk_free_list;

/* Multicast Address. The RNDIS layer needs to keep track of the Multicast
* address information that is passed from the HOST. */
UINT8         mcast_list[HAL_USB_MAX_MAC_MCAST_LIST][6]; /* Array for Multicast
address  */
int           mcast_list_size;      /* List size in BYTES.       */

/* Statistics */
RNDIS_STATS   stats;                /* Add up count */

} RNDIS_MCB;

typedef struct  rndis_ep_t
{
      UINT8 type;     /*  EP Type (CONTROL|BULK|INT),  Dir , Status */
      INT32 (*receive)(struct USB_CLASS_DRV_EP *ep, struct USB_CLASS_DRV_PKT *pkt);
      INT32 (*sendComplete)(struct USB_CLASS_DRV_EP *ep, void *priv);
      USB_ENDPOINT_DESCRIPTOR *epd;
} rndis_ep_t;

struct buf_node {
      char *buf;
      UINT32 len;
      struct buf_node *next;
} buf_node_t;

struct RNDIS_PKT
{
      UINT32 os_pkt_id;
      struct buf_node *head;
};
```

```
#define RNDIS_MAX_BUFFER_SIZE 512

struct RNDIS_RESPONSE
{
    char                    data[RNDIS_MAX_BUFFER_SIZE];
    UINT16              len;
    struct RNDIS_RESPONSE *next;
};
```

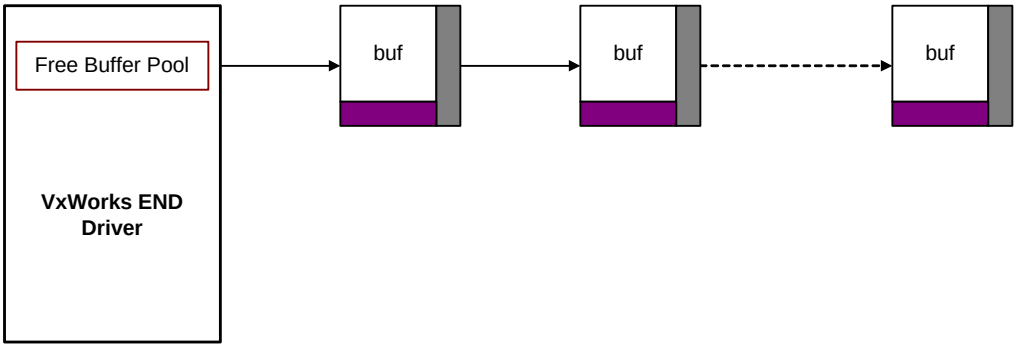

Figure 4-6. RNDIS Response Buffer Pool

```
typedef struct RNDIS_DATA_HEADER
{
    char       message[44];
    UINT32     chain_id;
    struct RNDIS_DATA_HEADER  *next;
}RNDIS_DATA_HEADER;
```

RNDIS State Transition Diagram

See the *Microsoft RNDIS 1.1 Specification*.

RNDIS Initialization Flow

See the *Microsoft RNDIS 1.1 Specification*.

4.3 VxWorks END Driver

4.3.1 Component Description

As per the USB 1.1 RNDIS driver.

4.3.2 Interface Description

As per the USB 1.1 RNDIS driver.

4.3.3 Design Description

As per the USB 1.1 RNDIS driver.

Chapter 5

USB IOCTL Operations

IOCTL_GET_USB_CONFIG - Get all descriptors

IOCTL_SET_USB_CONFIG - Set descriptors

IOCTL_GET_STATS_USB - Get USB related status / stats (MIB)

IOCTL_SET_SERIAL_NO - Set USB serial number

IOCTL_GET_SERIAL_NO - Get USB serial number

IOCTL_SET_VENDOR_ID - Set USB vendor ID

IOCTL_GET_VENDOR_ID - Get USB vendor ID

IOCTL_SET_PRODUCT_ID - Set USB product ID

IOCTL_GET_PRODUCT_ID - Get USB product ID

IOCTL_SET_MAX_POWER - Set USB max power

IOCTL_GET_MAX_POWER - Get USB max power

IOCTL_GET_BULK_EP_SIZE - Get USB endpoint size

IOCTL_SET_BULK_EP_SIZE - Set USB endpoint size

IOCTL_GET_INT_EP_SIZE - Get USB endpoint size

IOCTL_SET_INT_EP_SIZE - Set USB endpoint size

IOCTL_GET_CTRL_EP_SIZE - Get USB endpoint size

IOCTL_SET_CTRL_EP_SIZE - Set USB endpoint size

IOCTL_GET_ATTRIB - Get USB attributes
(Bus/Self-power, remote wakeup support)

Chapter 6

RNDIS IOCTL Operations

IOCTL_NOTIFY_LINK_STATE - To notify about connection to or disconnection from the host

IOCTL_GET_USB_INFO - Get USB level details, such as, vendor ID, product ID, and so on

IOCTL_GET_STATS_NET - Get network related statistics (Tx/Rx Stats)

IOCTL_GET_STATS_USB - Get USB related status/statistics (MIB)

IOCTL_SET_MIB_INTF_PARAMS - MIB interface functions

IOCTL_GET_MIB_INTF_PARAMS - MIB interface functions

Chapter 7

USB Error List

Not available.

Chapter 8

RNDIS Error List

RNDIS_MEMORY_ALLOC_ERROR

RNDIS_INIT_ERROR

RNDIS_START_ERROR

RNDIS_STOP_ERROR

RNDIS_SHUTDOWN_ERROR

RNDIS_SEND_ERROR

RNDIS_RCV_ERROR

RNDIS_USB_PROTO_INIT_ERROR

RNDIS_USB_PROTO_START_ERROR

RNDIS_USB_PROTO_STOP_ERROR

RNDIS_USB_PROTO_SHUTDOWN_ERROR

RNDIS_USB_EP_OPEN_ERROR

RNDIS_USB_EP_SEND_ERROR

RNDIS_USB_EP_RCV_ERROR

RNDIS_USB_EP_CLOSE_ERROR

RNDIS_UNSUPPORTED_IOCTL_ERROR

RNDIS_IOCTL_ERROR

RNDIS_OUT_OF_SYNC_ERROR

RNDIS_INVALID_PROTOCOL_PACKET_ERROR

RNDIS_INVALID_CONTROL_PACKET_ERROR

RNDIS_INVALID_ENCAPSULATED_COMMAND_ERROR

RNDIS_NO_RESPONSE_AVAILABLE_ERROR

RNDIS_OUT_OF_HEADER_BUFFERS_ERROR

RNDIS_INCORRECT_VERSION_ERROR

RNDIS_UNSUPPORTED_QUERY_ERROR

RNDIS_BAD_SET_REQUEST_ERROR