Network Developer's Kit (NDK) Support Package Ethernet Driver

Design Guide

Literature Number: SPRUFP2B
December 2015

Preface

About This Guide

This document describes the design of the Ethernet and Serial driver architecture in the NDK. Drivers packaged as a part of the Network Developer's Kit (NDK) Support Package in NDK 2.x follow the generic architecture described in this document.

Important Note:

- The setup and installation steps for each NDK Support Package (NSP) are provided in the Release Notes provided with that NSP.

Intended Audience

This document is intended for writers of Ethernet and serial mini-drivers. This document assumes you have knowledge of Ethernet and serial concepts.

Related Documents

The following books describe the Network Developer's Kit (NDK)"

- SPRU523 - *Network Developer's Kit (NDK) Software User's Guide*. Describes how to use the NDK libraries, how to develop networking applications, and ways to tune the NDK to fit a particular software environment.
- SPRU524 - *Network Developer's Kit (NDK) Software Programmer's Reference Guide*. Describes the various API functions provided by the stack libraries, including the low level hardware APIs.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `mono-spaced font`. Examples use **`bold`** for emphasis, and interactive displays use **`bold`** to distinguish commands you enter from items that the system displays (such as prompts, command output, error messages, etc.).
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Trademarks

December 2015

Contents

Chapter 1

Architecture Overview

This chapter provides an overview of the terminology and components involved in the Network Developer's Kit Support Package (NSP) Ethernet driver. It also describes the architecture of such drivers.

Topic **Page**

1.1 Acronyms

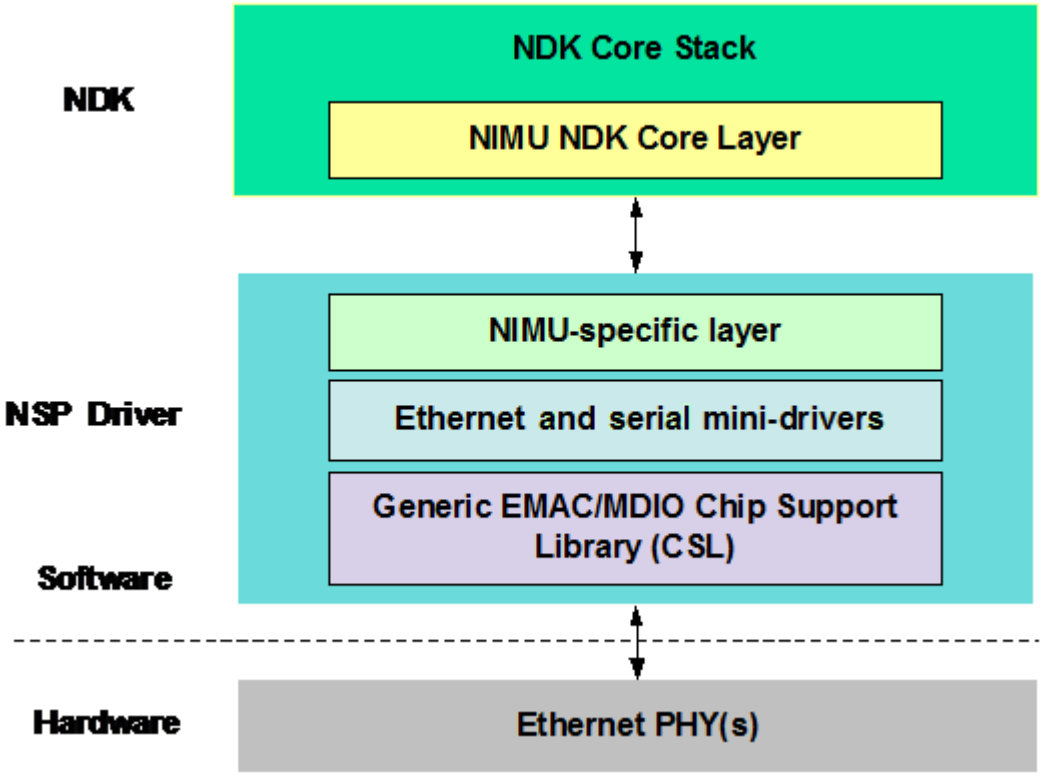
The following acronyms are used in this document:

Table 1–1. Acronyms

Acronym	Description
API	Application Programming Interface
BD	Buffer Descriptor
CSL	Chip Support Library
DSP	Digital Signal Processor
EMAC	Ethernet Medium Access Protocol
LL	Low Level Packet Driver
MDIO	Management Data Input/Output Interface
NDK	Network Developer's Kit
NIMU	Network Interface Management Unit
NSP	NDK Support Package
OSAL	Operating Systems Abstraction Layer
Rx	Receive Operation
SGMII	Serial Gigabit Media Independent Interface
Tx	Transmit Operation

1.2 Ethernet Driver Architecture

The following diagram shows the architecture of the Ethernet driver design in the NDK 2.0 Support Package (NSP).



This new NSP Ethernet driver architecture consists of the following components:

- **NIMU-specific layer**, which acts as the interface between the NDK stack and the Ethernet driver. See Section 1.2.1.
- **Ethernet mini-driver**, which manages the EMAC configuration using the CSL. Also manages DSP interrupts and memory allocation for packet buffers in buffer descriptors using the NDK Operating Systems Abstraction Layer (OSAL). See Section 1.2.2.
- **Serial mini-driver**, which manages all device configuration, communication, memory and packet buffer allocations, and hardware interrupts as required by the serial device.
- **Generic EMAC/MDIO Chip Support Library (CSL)**, which contains the generic APIs and data structures needed to control and configure EMAC/MDIO peripherals. Also manages buffer descriptors and interrupt service routines. The CSL layer is optional—the TI-RTOS Ethernet and serial drivers do not use the CSL layer. See Section 1.2.4.

The NIMU-specific layer in previous versions of the NDK was generic enough to be ported to different platforms with ease. However, the mini-driver was not easily portable and had to be rewritten from scratch every time it had to be ported to a new platform. This led to different flavors of the Ethernet device drivers for different platforms—thus increasing the development, maintenance, and debugging effort.

To overcome the limitations of this architecture, the architecture of Ethernet drivers in the NSPs have been reorganized to optimize for a better development and debugging experience. The Generic EMAC/MDIO Chip Support Library (CSL) component is new; it has been split apart from the Ethernet mini-driver component to better isolate portions that commonly require changes when porting.

1.2.1 NIMU-Specific Layer

The Network Interface Management Unit (NIMU) specific layer acts as the interface between the Ethernet, serial, or other physical device and the NDK core stack. It provides an implementation for the APIs defined by the NIMU specification for this EMAC device. These APIs let the NDK core stack control and configure the physical device at runtime and transmit packets. They also enable the driver to hand any received packets back to the stack.

This layer is fairly generic and doesn't change between different platforms.

This layer's functionality and role are the same as in versions of NDK prior to v2.0.

1.2.2 Ethernet Mini-Driver

The Ethernet mini-driver layer is responsible for setting up parameters for EMAC and MDIO configuration according to system needs. If available, it uses APIs and data structures exported by the underlying Chip Support Library (CSL) layer. It is also responsible for setting up EMAC interrupts into the DSP using data structures and APIs exposed by the NDK OSAL.

This layer acts as the sole memory manager in the Ethernet driver. That is, it handles all memory allocations, initializations, and frees of packet buffers for use in the buffer descriptors (BDs) in the Transmit (Tx) and Receive (Tx) paths. For memory management, it again uses the data structures and APIs defined by the NDK OSAL.

For the most part, the mini-driver invokes CSL APIs for setup, Tx, and interrupt service operations. The CSL layer, however, can also invoke the mini-driver layer. The CSL layer can invoke the mini-driver registered callback functions (set up during EMAC_open) for updating statistics and reporting errors. On receiving a packet, it can hand over the packet to be passed up the stack or for memory allocation/free of buffers in BDs.

This layer is OS agnostic, since it uses the NDK OSAL for all memory and interrupt management operations. However, this layer is device-dependent since the EMAC peripheral setup requires knowledge of the capabilities of EMAC on this platform/device and will have to be customized for each platform and for application needs. So, this layer needs to be ported and customized from one platform to another.

1.2.3 Serial Mini-Driver

The serial mini-driver layer is responsible for setting up parameters for device configuration and communication, the transmission and reception of data, as well as computing and validating any required checksums. It should make use of any existing serial hardware APIs or define its own code for communicating directly with the serial device, if no API is available. Additionally, any hardware interrupt configuration should be done in this layer (if necessary).

This layer also acts as the sole memory manager in the serial driver. That is, it handles all memory allocations, initializations, and frees of packet buffers for use in the buffer descriptors (BDs) in the Transmit (Tx) and Receive (Tx) paths. For memory management, it uses the data structures and APIs defined by the NDK Packet Buffer Manager (PBM) and Memory Allocation modules.

The mini-driver code can be defined to be OS agnostic by making use of the NDK OSAL, however it is up to the author to decide whether this is appropriate or not.

Similarly, the device dependency of this layer is determined by the implementation used to communicate with the serial hardware. If device-specific code is used to define this layer, then this layer would, of course, need to be customized for compatibility between platforms. However, by using device independent serial APIs (if available), this layer could be made to be device independent.

1.2.4 Generic EMAC/MDIO Chip Support Library

This layer enables the generic driver architecture by doing the following:

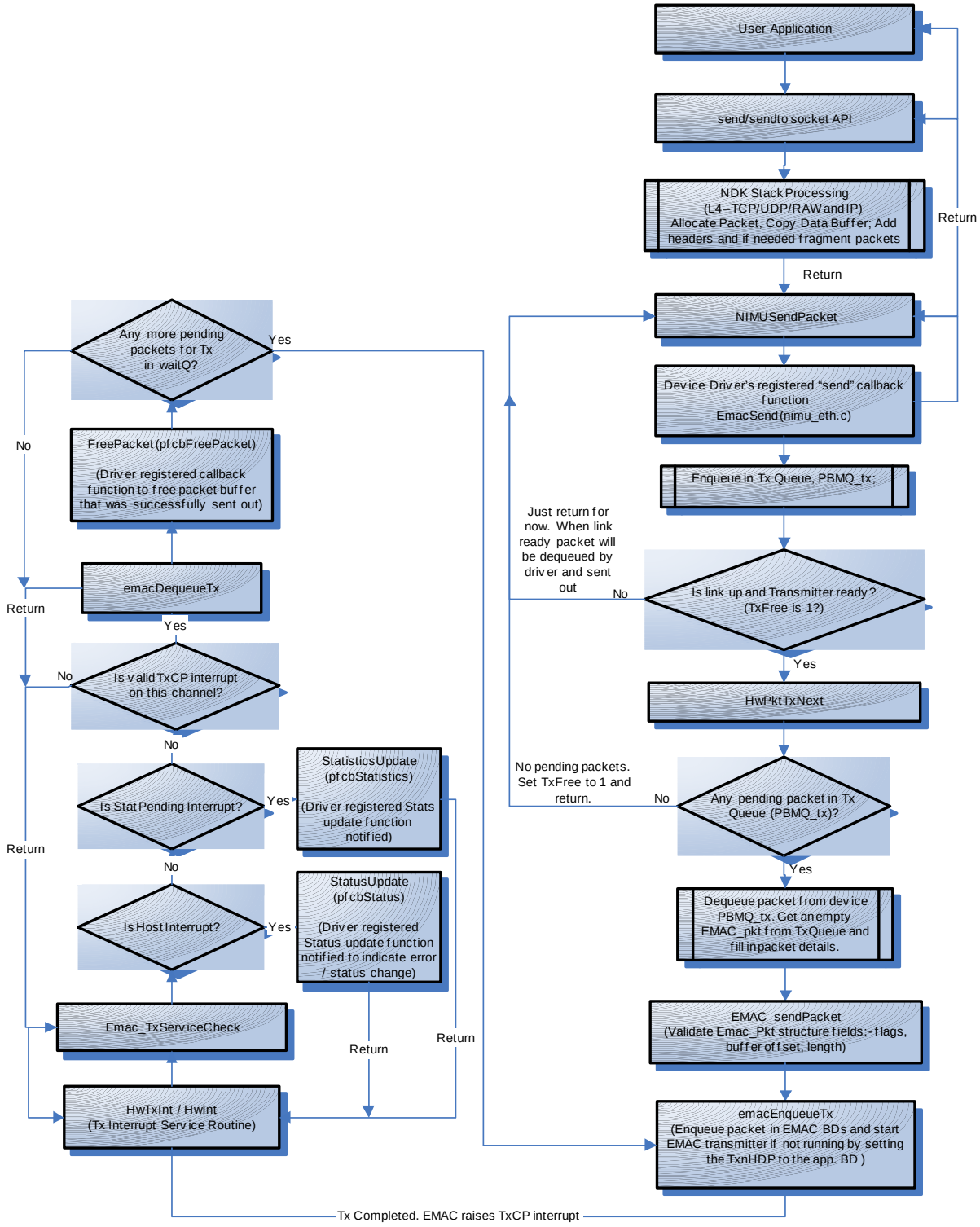
- **EMAC APIs.** It defines the data structures and interfaces (APIs) required to configure and use EMAC for transmit and receive operations.
- **MDIO and SGMII APIs.** It exposes APIs for managing the PHY-related (physical layer) configuration through the MDIO and SGMII (if the PHY is capable of gigabit speed) modules.
- **BD logic.** It implements the basic logic for CPPI Buffer Descriptor management (setup, enqueuing, and dequeuing operations).
- **ISR logic.** It contains the central logic for interrupt service routines. However, it uses the mini-driver's registered callback functions to report packet reception, statistics, errors, and obtaining or freeing a buffer for filling up a BD.

This layer is largely generic and doesn't vary much from platform to platform unless the EMAC capabilities change a whole lot. For example, the CSL for an EMAC peripheral connecting to a PHY switch would be very different from an EMAC that connects to a single PHY port. This layer is easily portable to different devices with similar capabilities.

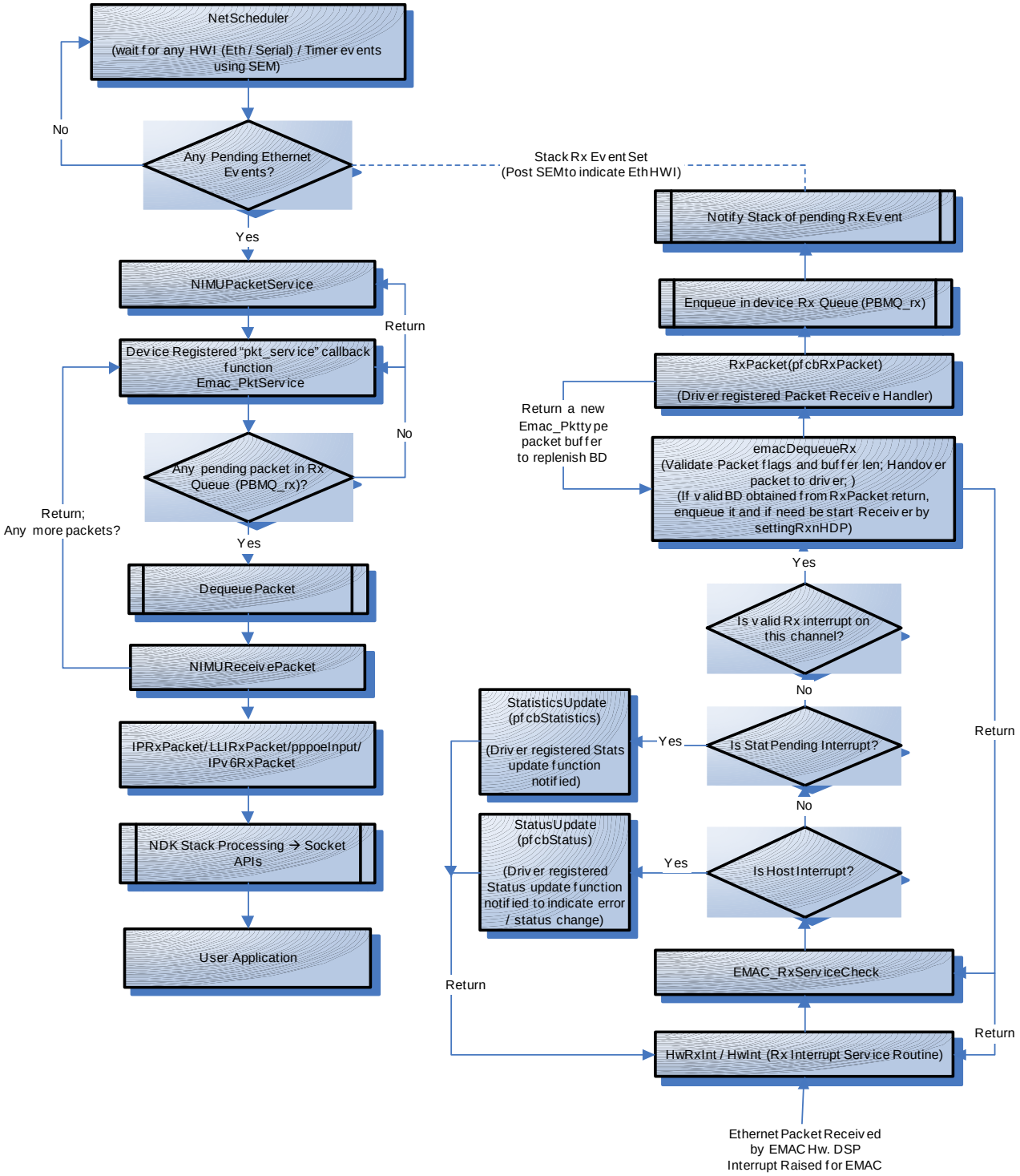
1.3 Flow Charts

The flow charts in this section do not apply to the serial mini-driver.

The transmission path for Ethernet packets is shown in the following flow chart.



The receive path for Ethernet packets is as follows:



1.4 Background

To port NDK Support Package device drivers, you should be familiar with the following constructs and concepts.

1.4.1 Network Control (NETCTRL) Module

The Network Control Module (NETCTRL) is at the center of the NDK and controls the interface of the HAL device drivers to the internal stack functions.

The NETCTRL module and its related APIs are described in both the *Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524) and the *Network Developer's Kit (NDK) Software User's Guide* (SPRU523). To write device drivers, you must be familiar with NETCTRL. The description given in the *Network Developer's Kit (NDK) Software User's Guide* (SPRU523) is more appropriate for device driver work.

1.4.2 Stack Event (STKEVENT) Object

The STKEVENT object is a central component in the low-level architecture. It ties the HAL layer to the scheduler thread in the network control module (NETCTRL). The network scheduler thread waits on events from various device drivers in the system, including the Ethernet, serial, and timer drivers.

Device drivers use the STKEVENT object to inform the scheduler that an event has occurred. The STKEVENT object and its related API are described in the *Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). Device driver writers need to be familiar with STKEVENT.

1.4.3 Packet Buffer (PBM) Object

The PBM object is a packet buffer that is sourced and managed by the Packet Buffer Manager (PBM). It provides packet buffers for all packet-based devices in the system. Therefore, the serial port and Ethernet drivers both make use of this module.

The PBM module manages packet buffers up to 3 KB in size. Any packet buffer allocation larger than 3 KB is managed by the Jumbo Packet Buffer Manager (Jumbo PBM). A default Jumbo PBM implementation is provided in the NDK; this implementation might need customization according to the application needs and system's memory constraints.

The PBM object, its related API, and the Jumbo PBM API are described in the *Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). The *Network Developer's Kit (NDK) Software User's Guide* (SPRU523) also includes a section on adapting the PBM to a particular included software.

1.4.4 NDK Interrupt Manager

The NDK Interrupt Manager is a module in the NDK OSAL that abstracts out the OS (SYS/BIOS) specific APIs and data structures required for interrupt configuration and management. It exposes a simple interface to the driver writer to configure EMAC interrupts into the DSP core. Interrupt Setup (IntSetup) Object is a data structure defined by this module.

Depending on the system specification, there can be a single or multiple system event/interrupt numbers defined for the EMAC module's Transmit (Tx) and Receive (Rx) events. Also based on the system specification, one could register a single Interrupt Service Routine (ISR) for both Tx and Rx events or register separate ISRs for each event.

The following NDK Interrupt Manager APIs can be used by the Ethernet driver in setting up the interrupts:

- Interrupt_add
- Interrupt_delete
- Interrupt_enable
- Interrupt_disable

Please see the sample Ethernet driver code packaged as the NDK Support Package (NSP) for any C64x+ device for an illustration of interrupt configuration using NDK Interrupt Manager APIs. The NDK Interrupt Manager, along with its related API and data structures, are described in the *Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524).

1.4.5 Data Alignment

The NDK libraries have been built with the assumption that the IP header in a data packet is 16-bit aligned. In other words, the first byte of the IP packet (the version/length field) must start on an even 16-bit boundary. In any fixed-length header protocol, this requirement can be met by backing off any odd byte header size, and adding it to the header padding specified to the stack. For Ethernet and peer-to-peer protocol (PPP), the only requirement is that the Ethernet or PPP packet not start on an odd byte boundary.

In addition, some drivers in the NDK are set up to have a 22-byte header. This is the header size of a PPPoE packet when sent using a 14-byte Ethernet header. When all arriving packets use the 22-byte header, it guarantees that they can be routed to any egress device with a header requirement up to that size. For Ethernet operation, this requires that a packet has 8 bytes of pre-pad to make its total header size 22 bytes.

The value of this pre-pad is #defined as PKT_PREPAD in the Ethernet driver include files.

1.5 API Overview

The various APIs exposed by the three main layers—the NIMU-specific layer, mini-driver, and generic EMAC/MDIO CSL layer—can be classified based on their functionality into the following categories:

- **Initialization and Shutdown APIs.** These APIs are called during Ethernet device start up to initialize the EMAC environment or during shutdown to bring down the Ethernet controller and its subsystems.
- **Configuration APIs.** These APIs are called to get/set the EMAC configuration. The configuration APIs are generally useful in setting the following parameters:
  - multicast configuration
  - receive filters on the Ethernet device
- **Transmit APIs.** These APIs provide a well-defined interface for the NDK stack to pass down any available Ethernet packets onto the wire using the Ethernet driver.
- **Receive APIs.** These APIs provide a well-defined interface for the driver to pass up an Ethernet packet to the NDK stack and into an application. Frames are enqueued in an RX ISR, then dequeued in EmacPktService, and passed up the stack via the NIMUReceivePacket function.
- **Polling APIs.** These APIs provide an interface for the NDK core stack to monitor the status of the Ethernet link on a periodic basis and to perform any necessary configuration of the EMAC depending on a change of state, if any.

The following table groups the APIs defined by each of the Ethernet driver layers under one of these five categories.

Table 1-2. API Mapping between the Ethernet driver layers

API Category	NIMU Layer	Mini-Driver Layer	CSL Layer
Initialization	EmacInit	HwPktInit	EMAC_initialize (optional)
	EmacStart	HwPktOpen	EMAC_open / MDIO_open
Shutdown	EmacStop	HwPktClose, HwPktShutdown	EMAC_close / MDIO_close
Configuration	Emacioctl	HwPktSetRx	EMAC_setReceiveFilter, EMAC_getReceiveFilter, EMAC_setMulticast, EMAC_getConfig (optional), EMAC_getStatus, EMAC_getStatistics, EMAC_enumerate
Transmit	EmacSend, NIMUSendPacket	HwPktTxNext	EMAC_sendPacket, EMAC_TxServiceCheck (Tx ISR)
Receive	EmacPktService, NIMUReceivePacket	HwInt / HwRxInt (depends on whether the EMAC has separate system events mapped into DSP for Rx/Tx or just one for both)	EMAC_RxServiceCheck (Rx ISR)
Polling	EmacPoll	_HwPktPoll	EMAC_TimerTick (optional), MDIO_timerTick, MDIO_getStatus

The following chapters discuss each of the layers and APIs in detail.

Chapter 2

NIMU Layer

This chapter describes Network Interface Management Unit (NIMU) layer API.

Topic **Page**

2.1 Overview of the NIMU Layer

The Network Interface Management Unit (NIMU) layer interfaces with the NDK core stack. It enables the stack to control the device at runtime. This layer is platform-independent and is easily portable across various platforms.

2.2 NIMU APIs

Driver writers need to implement APIs as follows to make their driver NIMU-compliant:

1. Register a driver Init callback function with the core NDK NIMU layer by populating the function in the NIMUDeviceTable.
2. Allocate and initialize the NETIF_DEVICE structure for this device with the appropriate parameters and callback functions defined for the following NIMU-defined APIs:
   - start
   - stop
   - poll
   - send
   - pkt_service
   - ioctl
   - add_header
3. Invoke the NIMURegister API to register this device with the NDK core's NIMU layer for further management.

4. Finally, implement all the callback functions as per the NIMU architecture guidelines and the API descriptions described in "Network Interface Management Unit" section of the *Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524).

Please see the nimu_eth.c file in the sample Ethernet driver code packaged as NDK Support Package (NSP) for any C64x+ device for an example NIMU API implementation.

Chapter 3

Ethernet Mini-Driver Layer

This chapter describes Ethernet mini-driver layer interface.

Topic **Page**

3.1 Overview

The Ethernet mini-driver layer in the new driver architecture is responsible for setting up the EMAC subsystem configuration. It exposes various APIs to the NIMU layer through which the NDK stack can configure, control, transmit, and receive packets using the Ethernet controller. It sets up configuration for the EMAC, MDIO, and SGMII (if the physical layer is capable of gigabit speed) modules and acts like glue between the NIMU-specific layer and the low level EMAC configuration layer—that is, the Chip Support Library (CSL) layer—for those modules.

This layer is platform-dependent. Driver writers will need to know the PHY and EMAC capabilities and interrupt definitions for the specific system and will need to configure the Ethernet module accordingly.

3.2 Data Structures

Device configuration information is stored in a private device instance structure called "PDINFO" that is used to communicate the device configuration between the NIMU and the mini-driver layers.

```
typedef struct _pdinfo
{
    uint       PhysIdx;      /* physical index of device */
    HANDLE     hEther;       /* handle to logical driver */
    STKEVENT_Handle  hEvent; /* semaphore handle */
    UINT8      bMacAddr[6];  /* MAC address */
    uint       Filter;       /* current RX filter */
    uint       MCastCnt;     /* current MCast addr count */
    UINT8      bMCast[6*PKT_MAX_MCAST]; /* multicast list */
    uint       TxFree;       /* transmitter "free" flag */
    PBMQ       PBMQ_tx;      /* Tx queue */

#ifdef _INCLUDE_NIMU_CODE
    PBMQ       PBMQ_rx;      /* Rx queue */
#endif

} PDINFO;
```

The following list describes the structure items in more detail:

- **PhysIdx.** Physical Index of this device (>=0). The PhysIdx may range from 0 to n-1. Care should be to taken to ensure that the physical index of a device is unique if multiple instances of devices exist in the system. This attribute is an auxiliary field that can be used by the NIMU and mini-driver to communicate any data at run-time. For example, the physical index can be used to hold the EMAC channel number on which packets using this device should be transmitted, and the mini-driver can be changed to use this info when transmitting the packet.
- **hEther.** This field is no longer being used after the switch to NIMU style drivers in NDK 2.0.
- **hEvent.** The handle to the semaphore object shared by the NDK stack and the driver to communicate pending network Rx events. This handle is used with the STKEVENT_signal() function to signal the NDK stack that a packet has been received and enqueued by the driver for hand off to the NDK Ethernet stack.
- **bMacAddr.** The Mac (Hardware) address of this interface. This is set to a default value by the NIMU layer. The default value can be overridden by the mini-driver with a value received from the EEPROM during device open.
- **Filter.** The current receive filter setting, which indicates which types of packets are accepted. The receive filter determines how the packet device should filter incoming packets. This field is set by the NIMU layer/stack and used by the mini-driver to program the EMAC. Legal values include:
  - ETH_PKTFLT_NOTHING. no packets
  - ETH_PKTFLT_DIRECT. only directed Ethernet
  - ETH_PKTFLT_BROADCAST. directed plus Ethernet broadcast
  - ETH_PKTFLT_MULTICAST. directed, broadcast, and selected Ethernet multicast
  - ETH_PKTFLT_ALLMULTICAST. directed, broadcast, and all multicast
  - ETH_PKTFLT_ALL. All packets

- **MCastCnt.** Number of multicast addresses installed.
- **bMCast.** Multicast address list. This field is a byte array of consecutive 6-byte multicast MAC addresses. The number of valid addresses is stored in the MCastCnt field. The multicast address list determines what multicast addresses (if any) the MAC is allowed to receive. The multicast list is configured by the application.
- **TxFree.** Transmitter free flag. The TxFree flag is used by NIMU layer to determine if a new packet can be sent immediately by the mini-driver, or if it should be placed on the transmit pending queue for later. If the flag is not zero, the mini-driver function HwPktTxNext() is called when a new packet is queued for transmission. This flag is maintained by the mini-driver.
- **PBMQ_tx.** Transmit pending queue. The transmit pending queue holds all the packets waiting to be sent on the Ethernet device. The mini-driver pulls PBM packet buffers off this queue in its HwPktTxNext() function and posts them to the Ethernet MAC for transmit. Once the packet has been transmitted, the packet buffer is freed by the mini-driver calling PBM_free(). There is one Tx queue for each PKT device.
- **PBMQ_rx.** Receive queue. All packets received by the EMAC and handed over to the mini-driver are enqueued to the Rx queue. The mini-driver also signals the NDK stack of the pending receive packet that needs to be serviced in this queue. When the NDK scheduler thread next runs and finds this pending event to service, it invokes the NIMU layer EmacPktService function, which dequeues any pending packets on this queue and hands it over to the stack for further processing. There is one Rx queue for each PKT device.

3.3 Ethernet Mini-Driver APIs

The following APIs are exported by the Ethernet mini-driver layer:

- HwPktInit
- HwPktOpen
- HwPktClose
- HwPktShutdown
- HwPktSetRx
- HwPktTxNext
- _HwPktPoll

As described in Section 1.5, the APIs exposed by this layer can be conveniently grouped according to their functionality into the following categories:

1. **Initialization.** HwPktInit, HwPktOpen
2. **Shutdown.** HwPktClose, HwPktShutdown
3. **Configuration.** HwPktSetRx
4. **Transmit.** HwPktTxNext
5. **Receive.** HwInt, HwRxInt
6. **Polling.** _HwPktPoll

3.3.1 HwPktInit — Initialize Packet Driver Environment

Syntax

```
uint HwPktInit();
```

Parameters

None

Return Value

The number of Ethernet devices initialized. 0 indicates an error. All other positive values are considered success.

Description

This function is called to initialize the mini-driver environment and enumerate the number of devices in the system. A device instance may be opened for each device represented in the return count. If the function returns zero, no devices are supported.

3.3.2 HwPktOpen — Open Ethernet Device Instance

Syntax

```
uint HwPktOpen (PDINFO *pi);
```

Parameters

pi - Pointer to Ethernet device instance structure.

Return Value

Returns 0 on success and a positive value to indicate an error.

Description

This function is called to open a packet device instance. When HwPktOpen is called, the PDINFO structure is assumed to be valid. This function sets up the EMAC configuration and invokes the CSL layer's EMAC_open function to configure the EMAC peripheral. As part of the configuration passed to EMAC_open, the driver sets up the required callback functions that the CSL layer in turn invokes to allocate/free packet buffers, update statistics or status, and to hand over received packets.

This function is also responsible for setting up the interrupts and any other PHY related configuration to ready it for Tx/Rx operations.

3.3.3 HwPktClose — Close Ethernet Device and Disable Interrupts

Syntax

```
void HwPktClose (PDINFO *pi);
```

Parameters

pi - Pointer to Ethernet device instance structure.

Return Value

None.

Description

This function is called to close a packet device instance. When called, this function invokes the CSL layer EMAC_close function to disable EMAC Tx/Rx operations and free up any enqueued packets. This function also disables the EMAC interrupts.

3.3.4 HwPktSetRx — Configure the Ethernet Receive Filter Settings

Syntax

```
void HwPktSetRx( PDINFO *pi );
```

Parameters

pi - Pointer to Ethernet device instance structure.

Return Value

None

Description

This function is called when the values contained in the PDINFO instance structure for the Rx filter or multicast list are altered. The mini-driver calculates hash values based on the new settings if multicast lists are maintained through hash tables on this platform, and updates the EMAC settings by calling the CSL layer's EMAC_setReceiveFilter API.

3.3.5 HwPktIoctl — Execute Driver-Specific IOCTL Commands

Syntax

```
uint HwPktIoctl(PDINFO *pi, uint cmd, void *arg);
```

Parameters

pi - Pointer to Ethernet packet device instance structure.

cmd - Device-specific command.

arg - Pointer to command specific argument.

Return Value

Returns 1 on success and 0 on error.

Description

This function is called to execute a driver-specific IOCTL command. Not all Ethernet drivers support this API.

3.3.6 HwPktTxNext — Transmit Next Buffer in the Transmit Queue

Syntax

```
void HwPktTxNext( PDINFO *pi );
```

Parameters

pi - Pointer to Ethernet packet device instance structure.

Return Value

None

Description

This function is called to indicate that a packet buffer has been queued in the transmit pending queue contained in the device instance structure and the NIMU layer believes the transmitter to be free. This function dequeues any pending packets in the transmit queue of this device, allocates a EMAC_Pkt structure (a data structure understood by the CSL layer) and fills in the packet details and invokes the CSL layer function EMAC_sendPacket to finally transmit the packet.

3.3.7 _HwPktPoll — Mini-Driver Polling Function

Syntax

```
void _HwPktPoll( PDINFO *pi, uint fTimerTick );
```

Parameters

pi - Pointer to Ethernet packet device instance structure.

fTimerTick - Flag indicating whether this function has been called because the 100 ms timer expired or if it was called by some other function randomly.

Return Value

None

Description

This function is called by the NIMU layer at least every 100 ms, but calls can come faster when there is network activity. The mini-driver is not required to perform any operation in this function, but it can be used to check for device lockup conditions. When the call is made due to the 100 ms time tick, the fTimerTick calling parameter is set.

Note that this function is not called in kernel mode (hence, the underscore in the name). This is the only mini-driver function called from outside kernel mode (to support polling drivers).

3.4 Configuration Variables

The following configuration variables are defined by the Ethernet mini-driver layer to control various features:

- **EXTERNAL_MEMORY.** Enable this flag to compile the code to support the cache cleaning and synchronization required when the packet buffer memory is allocated from external memory.

- **EXTMEM.** Define this bit mask to indicate the external memory address location for this platform.
- **PKT_MAX.** Use this constant to control the number of "EMAC_Pkt" type packet buffers that are allocated and initialized on Receive and Transmit paths respectively at this layer to optimize the data paths. During EMAC start up, in the HwPktOpen() function, buffers of type "EMAC_Pkt" structure are allocated and enqueued to a free queue/receive queue. Packets from this "RxQueue" are used to replenish the CSL layer with buffers for its BDs. Similarly, for the Tx path a queue of such EMAC_Pkt initialized structures are held. A packet buffer from the "TxQueue" is dequeued and used in filling up the NDK packet buffer details before being handed over to the CSL layer. This constant controls the number of such replenishing buffers at this layer.

  This constant can be fine-tuned during performance tuning to suit the application's needs. For example, increasing this constant helps in cases where the NDK stack or application is transmitting packets at a faster rate than the EMAC hardware. In this case, the packets are buffered up here at the mini-driver and get transmitted at the next suitable opportunity. But, it's important to note that this constant needs to be tuned according to the memory available in the system. The smallest number this can be set to is 8.
- **PKT_PREPAD.** The number of bytes to reserve before the Ethernet header for any additional headers like PPP. This is typically defined to be 8 to include the PPP header.
- **RAM_MCAST.** Define this configuration variable as 1 if the EMAC on this device supports RAM-based multicast lists. That is, if the EMAC is capable of storing multicast addresses in RAM and has defined appropriate registers to store them.
- **HASH_MCAST.** Enable this or define this as 1 if the EMAC on this device is capable of maintaining the multicast address list using hash tables.
- **PKT_MAX_MCAST.** This constant defines the maximum number of multicast addresses that can be configured and supported by the EMAC peripheral on this device. This is typically set to 31.

Chapter 4

Serial Mini-Driver Layer

This chapter describes serial mini-driver layer interface.

Topic **Page**

4.1 Overview

The serial port driver is divided into two distinct parts, a hardware-independent module (`llserial.c`) that implements the llSerial API (where "ll" is lowercase Ls for "low-level"), and a hardware-specific module that interfaces to the hardware independent module. The llSerial API is described in the *TCP/IP NDK Programmer's Reference Guide,* Appendix D. This section describes this small hardware-specific module, or "mini-driver".

Note that this module is purely optional. A valid serial port driver can be developed by directly implementing the llSerial API described in the Programmer's Reference Guide. Even if the mini-driver is used, the driver writer has the option of changing any of the internal data structures so long as the llSerial interface remains unchanged.

4.2 Global Instance Structure

Nearly all the functions in the mini-driver API take a pointer to a serial driver instance structure called SDINFO. This structure is defined in `llserial.h`:

```
/*
 *  Serial device information
 */
typedef struct _sdinfo {
    uint            PhysIdx;                    /* Physical index of device (0 to n-1) */
    uint            Open;                       /* Open counter used by llSerial */
    HANDLE          hHDLC;                      /* Handle to HDLC driver (NULL=closed) */
    STKEVENT_Handle hEvent;                     /* Handle to scheduler event object */
    UINT32          PeerMap;                    /* 32 bit char escape map (for HDLC) */
    uint            Ticks;                      /* Track timer ticks */
    uint            Baud;                       /* Baud rate */
    uint            Mode;                       /* Data bits, stop bits, parity */
    uint            FlowCtrl;                   /* Flow Control Mode */
    uint            TxFree;                     /* Transmitter "free" flag */
    PBMQ            PBMQ_tx;                    /* Tx queue (one for each SER device) */
    PBMQ            PBMQ_rx;                    /* Rx queue (one for each SER device) */
    PBM_Handle      hRxPend;                    /* Packet being rx'd */
    UINT8          *pRxBuf;                     /* Pointer to write next char */
    uint            RxCount;                    /* Number of bytes received */
    UINT16          RxCRC;                      /* Receive CRC */
    UINT8           RxFlag;                     /* Flag to "un-escape" character */
    PBM_Handle      hTxPend;                    /* Packet being tx'd */
    UINT8          *pTxBuf;                     /* Pointer to next char to send */
    uint            TxCount;                    /* Number of bytes left to send */
    UINT16          TxCRC;                      /* Transmit CRC */
    UINT8           TxFlag;                     /* Flag to insert character */
    UINT8           TxChar;                     /* Insert character */
    void           (*cbRx)(char);               /* Charmode callback (when open) */
    void           (*cbTimer)(HANDLE h);        /* HDLC Timer callback (when open) */
    void           (*cbInput)(PBM_Handle hPkt); /* HDLC Input callback (when open) */
    uint            CharReadIdx;                /* Charmode read index */
    uint            CharWriteIdx;               /* Charmode write index */
    uint            CharCount;                  /* Number of charmode bytes waiting */
    UINT8           CharBuf[CHAR_MAX];          /* Character mode recv data buffer */
} SDINFO;
```

4.2.1 PhysIdx: Physical index of this device (0 to n-1)

The physical index of the device is how the device instance is represented to the outside world. The mini-driver need not be concerned about the physical index.

4.2.2 Open: Open flag

This flag is used by `llserial.c` to track whether the mini-driver has been opened. It should not be modified by the mini-driver code.

4.2.3 hHDLC: Handle to HDLC driver

The handle to the HDLC device is how the system tracks where HDLC data should be sent. When this field is NULL, the driver is not open for HDLC mode, and all data should be treated as character mode. When the field is not NULL, any incoming serial data should be treated as potential HDLC data, and any output packet is treated as an egress HDCL frame. HDLC packets received in HDLC mode are tagged with this handle so that the upper layers can identify the packet's source.

4.2.4 hEvent: Handle to scheduler event object

The handle hEvent is used with the STKEVENT function STKEVENT_signal() to signal the system whenever new data is received. In character mode, this event is fired for each character. In HDLC mode, the event is fired when a good HDLC packet is received.

4.2.5 PeerMap: 32 bit char escape map (for HDLC)

The peer map is a 32 bit bitmap coded as (1<<char) where char is an ASCII character 0 through 31. When the bit is set, an outgoing HDLC frame must have the corresponding character escaped in a HDLC frame transmission.

4.2.6 Ticks: Track timer ticks

The field is used to convert 100ms timer ticks to 1 second timer ticks. It is not used by mini-drivers.

4.2.7 Baud: Serial Device Baud Rate

The field holds the current physical baud rate of the serial port in bps (e.g.: 9600, 19200, 153600, etc).

4.2.8 Mode: Device Mode

The mode field holds the mode of the serial port in terms of data bits, stop bits, and parity. These values appear in `hal.h`. Currently defined values are as follows:

```
#define HAL_SERIAL_MODE_8N1    0
#define HAL_SERIAL_MODE_7E1    1
```

4.2.9 FlowCtrl: Flow Control Mode

The FlowCtrl field determines the flow control mode. These values appear in `hal.h`. Currently defined values are as follows:

```
#define HAL_SERIAL_FLOWCTRL_NONE    0
#define HAL_SERIAL_FLOWCTRL_HARDWARE    1
```

4.2.10 TxFree: Transmitter Free Flag

The TxFree flag is used by `llserial.c` to determine if new data should be sent immediately by the mini-driver, or placed on the transmit pending queue for later. If the flag is not zero, the mini-driver function HwSerTxNext() is called when any new data is queued for transmission. This flag is maintained by the mini-driver.

4.2.11 PBMQ_tx: Tx queue

The PBMQ_tx queue is a queue of packets waiting to be transmitted. When the transmitter is free and the HwSerTxNext() function is called, the mini-driver removes the next packet off this queue and starts transmission.

The PBMQ object is a queue of PBM packet buffers and it is operated on by the PBMQ functions defined in the TCP/IP NDK Programmer's Reference Guide.

4.2.12 PBMQ_rx: Rx queue

The PBMQ_rx queue is a queue of packets that have been received on the interface. When a new packet is received, the mini-driver enqueues it onto this queue, and fires a serial event to the STKEVENT handle.

The PBMQ object is a queue of PBM packet buffers and it is operated on by the PBMQ functions defined in the TCP/IP NDK Programmer's Reference Guide.

4.2.13 hRxPend: PBM_Handle to packet being received

When in HDLC mode, this value holds a handle to the packet that is currently being received by the mini-driver. When the packet is complete, the mini-driver places this packet in the PBMQ_rx queue, and allocates another free packet by calling PBM_alloc().

4.2.14 pRxBuf: Pointer to next character in packet to receive

When in HDLC mode, this is a pointer where to write the next character of received data. The pointer points somewhere in the current packet buffer who's handle is stored in hRxPend.

4.2.15 RxCount: Number of bytes written to RX packet buffer so far

When in HDLC mode, this value is the number of characters that have been written to the current packet being received.

4.2.16 RxCRC: RX CRC running total

When in HDLC mode, this value is a running total of the current CRC value of the packet being received.

It is used as a temporary CRC holding value while packet data is still being received. It is then compared to the CRC contained in the packet to validate the incoming CRC.

4.2.17 RxFlag: Flag indicating that next byte is the second half of an escape sequence

When in HDLC mode, this flag is set when an escape character is seen. It prompts the RX state machine in the mini-driver to "un-escape" the next character received.

4.2.18 hTxPend: PBM_Handle to packet being transmitted

This value holds a handle to the packet that is currently being transmitted by mini-driver. When the packet is completely transmitted, the mini-driver frees this packet by calling PBM_free().

4.2.19 pTxBuf: Pointer to next character in packet to transmit

This is a pointer where to read the next character of transmit data. The pointer points somewhere in the current packet buffer who's handle is stored in hTxPend.

4.2.20 TxCount: Number of bytes yet to send from to TX packet

This value is the number of characters that have yet to be read and transmitted from the current packet being sent.

4.2.21 TxCRC: RX CRC running total

When in HDLC mode, this value is a running total of the current CRC value of the packet being transmitted. It is used as a temporary CRC holding value while packet data is still being sent. It is used to patch in the correct CRC value as the last two bytes of the packet data.

4.2.22 TxFlag: Flag indicating that next byte is the second half of an escape sequence

When in HDLC mode, this flag is set when an escape character has to be generated. It prompts the TX state machine in the mini-driver to write the second half of the escape sequence next. This value is stored in TxChar.

4.2.23 cbRx: Pointer to character mode callback function

This character mode callback function is called by `llserial.c` whenever there is character mode data queued up by the serial driver. This is not used by the mini-driver.

4.2.24 cbTimer: Pointer to HDLC timer callback function

The serial driver (`llserial.c`) calls this function once every second. The callback function is not used by the mini-driver.

4.2.25 cbInput: Pointer to HDLC input callback function

The serial driver (`llserial.c`) calls this function with new HDLC packets. The callback function is not used by the mini-driver.

4.2.26 TxChar: Second half of escape sequence

When in HDLC mode and TxFlag is set, this variable holds the next value to send out the serial port

4.2.27 CharReadIdx: Character buffer read index

This index is used by `llserial.c` to read character data out of the circular character buffer. It is not used by a mini-driver.

4.2.28 CharWriteIdx: Character buffer write index

This index is used by a mini-driver in character mode to write newly received character data to circular character buffer array contained in this structure. As data is written, this index is increased and the CharBufUsed value is increased. Once it reaches the value CHAR_MAX, it is reset to zero.

4.2.29 CharCount: Characters stored in character buffer

Data received in "character mode" are not placed in an serial frame buffer, but are stored in a circular buffer contained in this instance structure. The maximum number of characters that can be stored is determined by CHAR_MAX. The number of characters currently stored is determined by this value. The value is increased as characters are written to the buffer. The `llserial.c` module will decrement this value as characters are read out, so it should only be altered in a critical section.

4.2.30 CharBuf: Character mode input data buffer

This array acts as the input buffer for "character mode" data. Unlike HDLC data, individual characters are not built into serial packet buffers. Instead, they are queued for immediate consumption by the character mode user - most likely an AT command set modem state machine, but it could also be a serial console program. The size of this buffer is set by CHAR_MAX.

4.3 Serial Mini-Driver Operation

Only some of these fields are used in a mini-driver. The structure entries as defined as follows:

The serial mini-driver is charged with maintaining the serial device hardware, and servicing any required communications interrupts. It is built around a simple open/close concept. When open, the driver is active, and when closed is it not. In general, it must implement the mini-driver API described in the following section. Here are some additional notes on its internal operation.

4.3.1 Receive Operation

The mini-driver receives serial data and must classify it as HDLC data or character mode data. It is sufficient to use the current "mode" of the driver to determine how to classify data. For example:

```
// If HDLC handle valid, driver is open on HDLC mode
// Else use charmode
if( MyInstancePtr->hHDLC )
    Treat_Data_as_HDLC();
else
    Treat_Data_as_CharacterMode();
```

Of course, more advanced classification heuristics can be attempted (auto recognition of HDLC frames). Once the data is classified, it is placed either in a PBM packet buffer (if HDLC), or the circular character buffer (if character mode data). Empty packet buffers are acquired by calling the PBM_alloc() function.

The character mode buffer array for non-HDLC data is located in the mini-driver device instance, using the structure fields: CharBuf, CharCount, and CharWriteIdx. When CharCount equals CHAR_MAX, and no more data can be written to the buffer, any new data is discarded.

When the driver is in HDLC mode, the driver receives serial data as HDLC packets, and creates a PBM packet buffer object to hold each HDLC frame. Note that the HDLC flag character (0x7E) is always removed from the HDLC packets. The completed HDLC packet written to the PBM packet buffer has following format.

Table 4-1. HDLC Packet Format

Addr (FF)	Control (03)	Protocol	Payload	CRC
1	1	2	1500	2

When a HDLC packet is ready, the mini-driver adds it to the PQMQ_rx queue and signals an event to the STKEVENT object.

On receive, the mini-driver must remove all HDLC escape sequences, and validate the HDLC CRC. Packets with an invalid CRC are discarded. CRC calculation for both receive and transmit is done "in-line" as the packet is being received. Also, the CRC code in the example driver is based on a 4 bit algorithm. This allows for the use of a 16 entry lookup table instead of a 256 entry table.

4.3.2 Transmit Operation

Unlike receive, transmit uses PBM packet buffers to send regardless whether its in character mode or HDLC mode. The only difference in that in HDLC mode, the data must be formatted. The mini-driver gets the next packet to send off the PBMQ_tx queue when its HwSerTxNext() function is called. When all the characters from the packet have been read and transmitted, the PBM packet buffer is freed by calling PBM_free().

On transmit, the mini-driver must use escape sequences when necessary, and compute the HDLC CRC. Note on transmitted packet, the 2 byte HDCL CRC is present, just not valid. The mini-driver must validate the CRC when it sends the packet. CRC calculation for both receive and transmit is done "in-line" as the packet is being received. Also, the CRC code in the example driver is based on a 4 bit algorithm. This allows for the use of a 16 entry lookup table instead of a 256 entry table.

4.4 Serial Mini-Driver API

The following API functions must be provided by a mini-driver.

4.4.1 HwSerInit - Initialize Serial Port Environment

Syntax:

```
uint HwSerInit();
```

Parameters:

None

Description:

Called to initialize the serial port mini-driver environment, and enumerate the number of devices in the system. A device instance may be opened for each device represented in the return count. If the function returns zero, no serial devices are supported.

Returns:

The number of serial devices in the system.

4.4.2 HwSerShutdown - Shutdown Serial Port Environment

Syntax:

```
void HwSerShutdown();
```

Parameters:

None

Description:

Called to indicate that the serial port environment should be completely shutdown.

Returns:

Nothing.

4.4.3 HwSerOpen - Open Serial Port Device Instance

Syntax:

```
uint HwSerOpen( SDINFO *pi );
```

Parameters:

pi -- Pointer to serial device instance structure

Description:

Called to open a serial device instance. When called, SDINFO structure is valid.

Returns:

Returns 1 if the driver was opened, or 0 on error.

4.4.4 HwSerClose - Close Serial Port Device Instance

Syntax:

```
void HwSerClose( SDINFO *pi );
```

Parameters:

pi -- Pointer to serial device instance structure

Description:

Called to close a serial device instance. When called, any PBM packet buffers held by the driver instance including hRxPend, hTxPend, and PBMQ_tx. Packets from all three are freed by calling PBM_free(). In addition, the character mode buffer is reset (read pointer, write pointer, and character count all set to NULL). Packets that have been placed on the PBMQ_rx queue are flushed by `llserial.c`.

Returns:

Nothing.

4.4.5 HwSerTxNext - Transmit next buffer in transmit queue

Syntax:

```
void HwSerTxNext( SDINFO *pi );
```

Parameters:

pi -- Pointer to serial device instance structure

Description:

Called to indicate that a PBM packet buffer has been queue in the transmit pending queue (PBMQ_tx) contained in the device instance structure, and `llserial.c` believes the transmitter to be free (TxFree set to 1). The mini-driver uses this function to start the transmission sequence.

Returns:

Nothing.

4.4.6 HwSerSetConfig - Set Serial Port Configuration

Syntax:

```
void HwSerSetConfig( SDINFO *pi );
```

Parameters:

pi -- Pointer to serial device instance structure

Description:

Called when the values contained in the SDINFO instance structure are altered. The structure fields used for configuration are Baud, Mode, and FlowCtrl. The mini-driver should update the serial port configuration with the current SDINFO settings.

Returns:

Nothing.

4.4.7 HwSerPoll - Serial Polling Function

Syntax:

```
void _HwSerPoll( SDINFO *pi, uint fTimerTick );
```

Parameters:

pi -- Pointer to serial device instance structure fTimerTick Flag indicating the 100ms have elapsed.

Description:

Called by `llserial.c` at least every 100ms, but calls can come faster when there is serial activity. The mini-driver is not required to perform any operation in this function, but it can be used to check for device lockup conditions. When the call is made due to the 100ms time tick, the fTimerTick calling parameter is set.

Note that this function is not called in kernel mode (hence the underscore in the name). This is the only mini-driver function called from outside kernel mode (done to support polling drivers).

Returns:

Nothing.

Chapter 5

Generic EMAC/MDIO CSL Layer

This chapter describes the EMAC/MDIO CSL layer interface.

Topic **Page**

5.1 Overview

The EMAC/MDIO CSL layer defines data structures and APIs that enable the driver to configure the EMAC hardware and send and receive packets.

This CSL layer is fairly generic and can be ported easily across different platforms so long as the EMAC hardware specification don't vary a lot. For example, the CSL for an EMAC with switch capabilities would be very different from the CSL for an EMAC with support for a single PHY. This layer abstracts out all the EMAC/MDIO register layer configuration details from the higher layers and makes them easier to write and understand.

5.2 CSL Data Structures

The CSL layer exports various data structures to enable configuration of EMAC, MDIO, and other Ethernet associated modules. Discussing all the data structures is beyond the scope of this document. The definitions can be viewed from the code or by obtaining a doxygen output of the code.

5.3 EMAC APIs

The following APIs are exported by the CSL EMAC layer. Several APIs are optional, and a driver can choose not to implement these APIs.

- EMAC_enumerate
- EMAC_open
- EMAC_initialize (optional)
- EMAC_close
- EMAC_setReceiveFilter
- EMAC_getReceiveFilter
- EMAC_setMulticast
- EMAC_getStatus
- EMAC_getConfig (optional)
- EMAC_getStatistics
- EMAC_sendPacket
- EMAC_RxServiceCheck
- EMAC_TxServiceCheck
- EMAC_TimerTick (optional)

As described in Section 1.5, the APIs exposed by this layer can be conveniently grouped according to their functionality into the following categories:

1. **Initialization.** EMAC_open, EMAC_initialize
2. **Shutdown.** EMAC_close
3. **Configuration.** EMAC_setReceiveFilter, EMAC_getReceiveFilter, EMAC_setMulticast, EMAC_getStatus, EMAC_getConfig, EMAC_getStatistics, EMAC_enumerate
4. **Transmit.** EMAC_sendPacket, EMAC_TxServiceCheck (Tx ISR)
5. **Receive.** EMAC_RxServiceCheck (Rx ISR)
6. **Polling.** EMAC_TimerTick

The Ethernet mini-driver layer can invoke CSL APIs to perform any configuration or interrupt related processing only after opening and setting up the EMAC peripheral successfully using the "EMAC_open" API. All the error codes, macros, and constants used are defined in the header files included with the source code and can be found in the "inc" directory.

5.4 Callback Functions

The CSL layer doesn't perform any OS specific operations such as memory allocation, free, initialization, copy etc. Instead, this layer defines the required callback functions in the "EMAC_Config" data structure and mandates that the driver implement these functions and register them with the driver during the "EMAC_open" call. The callback functions that need to be implemented by the driver and their description are describe in the subsections that follow.

See the Ethernet driver code for sample implementations of these functions.

5.4.1 pfcbGetPacket

This function is called by the CSL layer when it needs an empty packet buffer to replenish a receive EMAC Buffer Descriptor (BD) in the EMAC RAM. This function needs to implement logic to allocate an EMAC packet (of type "EMAC_Pkt") and to initialize the buffers and offsets appropriately for use by the CSL layer. This function is typically called during EMAC initialization to initialize the Receive BDs or can be called during a receive interrupt servicing to re-fill any empty BDs.

5.4.2 pfcbFreePacket

This function is called by the CSL layer to free the memory allocated for an EMAC packet (of type "EMAC_Pkt") and any buffers held within it. This function is typically called during EMAC close, when an error occurs, or during a Transmit complete interrupt handling for cleaning up the associated buffers.

5.4.3 pfcbRxPacket

This function is the driver-registered receive handler for all Ethernet packets received and validated by the EMAC and handed over to the CSL layer when a receive interrupt occurs.

This function is required to save the packet buffer received to hand it over to the stack for further processing. At that point, it is the responsibility of the driver/stack to free the packet buffer. This function is also required to return a new EMAC packet buffer in return to replenish the BD just serviced.

5.4.4 pfcbStatus

This function is called by the CSL to notify the driver of a status change or the occurrence of an error during EMAC processing (HOSTPEND interrupt).

5.4.5 pfcbStatistics

This function is called by the CSL to update the driver with the latest snapshot of statistics (STATPEND interrupt).

Index

I

J

L

M

N

O

P

R

S

T

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions of TI 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video & Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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