Micrel KSZ88xx Programmer's Guide
For Generic Bus /PCI Bus Interface

Rev 1.2

06/30/2006

Table of Contents

Revision History

Revision	Date	Summary of Changes
0.3	6/28/05	First released, preliminary information only for generic bus version of driver description.
0.4	9/16/05	Added support platform drivers block diagram.
1.0	10/28/05	Added PCI bus version of driver description.
1.1	03/21/06	Update Table 2-4 for more OS the driver support.
1.2	06/30/06	Added EEPROM access support. Include KSZ886x family support.

1 Overview

The KS88xx driver works for KSZ8841M, KSZ8842M, KSZ8861M, KSZ8862M, KSZ8841P, or KSZ8842P device, and is independent from hardware platforms and operating systems. The KS88xx driver also works for Generic bus or PCI bus interface.

The KS88xx driver provides a set of API (Application Programming Interface) for the user to:

- Initialize KSZ88xx device.
- Set PHY link speed and duplex.
- Transmit the packets to KSZ88xx device.
- Receive the packets from KSZ88xx device.
- Diagnose cable status with LinkMD cable diagnostics function.

It provides the following API for KSZ8842/8862-only features:

- Read VLAN table, static MAC table, dynamic MAC table, and MIB counters.
- Create VLAN table and static MAC table.
- Configure VLAN functions.
- Configure IEEE 802.1d Spanning Tree Protocol.
- Configure per-port broadcast storm protection.
- Configure per-port rate limiting at the ingress and egress.
- Configure QoS (port-base, 802.1p and DiffServ) function.
- Configure Mirroring function.

It provides the following API for KSZ8841-only features:

- Configure Wake-on-LAN function.

It also provides a set of CLI functions for user to diagnose the KSZ88xx device.

When reading this document, please have the relative device datasheet as cross reference.

2 KS88xx Driver Source and Include Files

This section describes the KS88xx driver associated source and header files.

2.1 Driver Source and Header File Descriptions

The KS88xx hardware driver is platform and OS independent. It can be ported to any platform and OS. The KS88xx driver source and header files are:

File Name	Description
hardware.c	- Initialize KSZ88xx device. - Transmit the packets to KSZ88xx device. - Receive the packets from KSZ88xx device. - Configure Early transmit/receive function.
ks_config.c	- Set PHY link speed and duplex on a per port basis.
ks_stp.c	- Set port base STP states (disabled, blocking, listening, learning, and forwarding) from Spanning Tree Protocol.
ks_table.c	- Read VLAN table. - Read static MAC, dynamic MAC table. - Read SNMP MIB counters. - Create VLAN table. - Create static MAC table.
ks_vlan.c	- Configure port base VLAN on a per port basis. - Configure VLAN ingress, and egress function on a per port basis.
ks_rate.c	Configure broadcast storm protection on per port basis - Configure rate limiting at ingress and egress ports on a per port basis.
ks_qos.c	- Set DiffServ priority values. - Configure DiffServ, and 802.1p function on a per port basis. - Configure re-mapping 802.1p priority field on a per port basis.
ks_mirror.c	- Configure port mirroring/sniffing on a per port basis.
ks_Diag.c cliTable.h	- Provides all CLI functions to diagnosis the device.
hardware.h	Device driver header file. - Driver structure. - Device registers definitions.
ks_def.h	- KSZ8842/8862 device switch register definitions header file
ks_config.h	- Device Driver configuration functions header file.
target.c	- Platform or OS dependence functions.
target.h	- Platform or OS dependence read/write device registers functions.

Table 2-1. Driver Source and Header Files Descriptions

2.2 Driver Data Structure

The data structure used by KS88xx hardware driver is HARDWARE. The structure is defined in hardware.h.

```
typedef struct
{
    struct hw_fn*           m_hwfn;

    UCHAR                   m_bPermanentAddress[ MAC_ADDRESS_LENGTH ];
    UCHAR                   m_bOverrideAddress[ MAC_ADDRESS_LENGTH ];

    /* PHY status info. */
    ULONG                   m_ulHardwareState;
    ULONG                   m_ulTransmitRate;
    ULONG                   m_ulDuplex;

    /* hardware resources */
    PUCHAR                  m_pVirtualMemory;
    ULONG                   m_ulVIoAddr;    /* device's base address */
    ULONG                   m_boardBusEndianMode; /* board bus endian
     .                                        mode board specific */
     .
} HARDWARE, *PHARDWARE;
```

The first thing in the driver initialization function is to allocate a system memory for the HARDWARE structure.

Secondly, allocate the KS884x internal set APIs structure **struct hw_fn,** and initialize the API routines that are for Generic bus or PCI bus.

Then assign the KSZ88xx device base address to **m_ulVIoAddr** before driver accessing the device. For the generic bus interface, the lower 16-bit value of **m_ulVIoAddr** must be the same as the device register **BAR** (Base Address Register).

2.2.1 Generic Bus Data Structure Initialization

For example, to initialize **struct hw_fn** of KSZ88xxM generic bus interface:

```
void ksSetApiFunctions (struct hw_fn *pks_fn)
{
    pks_fn->m_fPCI = FALSE;

    pks_fn->fnSwitchDisableMirrorSniffer = SwitchDisableMirrorSniffer_ISA;
    pks_fn->fnSwitchEnableMirrorSniffer = SwitchEnableMirrorSniffer_ISA;
    pks_fn->fnSwitchDisableMirrorReceive = SwitchDisableMirrorReceive_ISA;
    pks_fn->fnSwitchEnableMirrorReceive = SwitchEnableMirrorReceive_ISA;
    pks_fn->fnSwitchDisableMirrorTransmit = SwitchDisableMirrorTransmit_ISA;
    pks_fn->fnSwitchEnableMirrorTransmit = SwitchEnableMirrorTransmit_ISA;
    pks_fn->fnSwitchDisableMirrorRxAndTx = SwitchDisableMirrorRxAndTx_ISA;
```

```
    pks_fn->fnSwitchEnableMirrorRxAndTx = SwitchEnableMirrorRxAndTx_ISA;

    pks_fn->fnHardwareConfig_TOS_Priority = HardwareConfig_TOS_Priority_ISA;
    pks_fn->fnSwitchDisableDiffServ = SwitchDisableDiffServ_ISA;
    pks_fn->fnSwitchEnableDiffServ = SwitchEnableDiffServ_ISA;

    pks_fn->fnHardwareConfig802_1P_Priority = HardwareConfig802_1P_Priorit_ISA;
    pks_fn->fnSwitchDisable802_1P = SwitchDisable802_1P_ISA;
    pks_fn->fnSwitchEnable802_1P = SwitchEnable802_1P_ISA;
    pks_fn->fnSwitchDisableDot1pRemapping = SwitchDisableDot1pRemapping_ISA;
    pks_fn->fnSwitchEnableDot1pRemapping  = SwitchEnableDot1pRemapping_ISA;

    pks_fn->fnSwitchConfigPortBased = SwitchConfigPortBased_ISA;

    pks_fn->fnSwitchDisableMultiQueue = SwitchDisableMultiQueue_ISA;
    pks_fn->fnSwitchEnableMultiQueue = SwitchEnableMultiQueue_ISA;

    pks_fn->fnSwitchDisableBroadcastStorm = SwitchDisableBroadcastStorm_ISA;
    pks_fn->fnSwitchEnableBroadcastStorm = SwitchEnableBroadcastStorm_ISA;
    pks_fn->fnHardwareConfigBroadcastStorm = HardwareConfigBroadcastStorm_ISA;

    pks_fn->fnSwitchDisablePriorityRate = SwitchDisablePriorityRate_ISA;
    pks_fn->fnSwitchEnablePriorityRate = SwitchEnablePriorityRate_ISA;

    pks_fn->fnHardwareConfigRxPriorityRate = HardwareConfigRxPriorityRate_ISA;
    pks_fn->fnHardwareConfigTxPriorityRate = HardwareConfigTxPriorityRate_ISA;

    pks_fn->fnPortSet_STP_State = PortSet_STP_State_ISA;

    pks_fn->fnPortReadMIBCounter = PortReadMIBCounter_ISA;
    pks_fn->fnPortReadMIBPacket = PortReadMIBPacket_ISA;

    pks_fn->fnSwitchEnableVlan = SwitchEnableVlan_ISA;
}
```

2.2.2 PCI Bus Data Structure Initialization

For example, to initialize **struct hw_fn** of KSZ88xxP PCI bus interface:

```
void ksSetApiFunctions (struct hw_fn *pks_fn)
{
    pks_fn->m_fPCI = TURE;

    pks_fn->fnSwitchDisableMirrorSniffer = SwitchDisableMirrorSniffer_PCI;
    pks_fn->fnSwitchEnableMirrorSniffer = SwitchEnableMirrorSniffer_PCI;
    pks_fn->fnSwitchDisableMirrorReceive = SwitchDisableMirrorReceive_PCI;
    pks_fn->fnSwitchEnableMirrorReceive = SwitchEnableMirrorReceive_PCI;
    pks_fn->fnSwitchDisableMirrorTransmit = SwitchDisableMirrorTransmit_PCI;
    pks_fn->fnSwitchEnableMirrorTransmit = SwitchEnableMirrorTransmit_PCI;
    pks_fn->fnSwitchDisableMirrorRxAndTx = SwitchDisableMirrorRxAndTx_PCI;
    pks_fn->fnSwitchEnableMirrorRxAndTx = SwitchEnableMirrorRxAndTx_PCI;

    pks_fn->fnHardwareConfig_TOS_Priority = HardwareConfig_TOS_Priority_PCI;
    pks_fn->fnSwitchDisableDiffServ = SwitchDisableDiffServ_PCI;
    pks_fn->fnSwitchEnableDiffServ = SwitchEnableDiffServ_PCI;

    pks_fn->fnHardwareConfig802_1P_Priority = HardwareConfig802_1P_Priorit_PCI;
```

```
    pks_fn->fnSwitchDisable802_1P = SwitchDisable802_1P_PCI;
    pks_fn->fnSwitchEnable802_1P = SwitchEnable802_1P_PCI;
    pks_fn->fnSwitchDisableDot1pRemapping = SwitchDisableDot1pRemapping_PCI;
    pks_fn->fnSwitchEnableDot1pRemapping  = SwitchEnableDot1pRemapping_PCI;

    pks_fn->fnSwitchConfigPortBased = SwitchConfigPortBased_PCI;

    pks_fn->fnSwitchDisableMultiQueue = SwitchDisableMultiQueue_PCI;
    pks_fn->fnSwitchEnableMultiQueue = SwitchEnableMultiQueue_PCI;

    pks_fn->fnSwitchDisableBroadcastStorm = SwitchDisableBroadcastStorm_PCI;
    pks_fn->fnSwitchEnableBroadcastStorm = SwitchEnableBroadcastStorm_PCI;
    pks_fn->fnHardwareConfigBroadcastStorm = HardwareConfigBroadcastStorm_PCI;

    pks_fn->fnSwitchDisablePriorityRate = SwitchDisablePriorityRate_PCI;
    pks_fn->fnSwitchEnablePriorityRate = SwitchEnablePriorityRate_PCI;

    pks_fn->fnHardwareConfigRxPriorityRate = HardwareConfigRxPriorityRate_PCI;
    pks_fn->fnHardwareConfigTxPriorityRate = HardwareConfigTxPriorityRate_PCI;

    pks_fn->fnPortSet_STP_State = PortSet_STP_State_PCI;

    pks_fn->fnPortReadMIBCounter = PortReadMIBCounter_PCI;
    pks_fn->fnPortReadMIBPacket = PortReadMIBPacket_PCI;

    pks_fn->fnSwitchEnableVlan = SwitchEnableVlan_PCI;
}
```

2.3 KS88xx Driver Platform Support

Other than the platform/OS independent KS88xx hardware driver, we also provide following platform/OS dependent sample drivers listed in the Table 2-4.

Buses	Microprocessor	OS	Protocol Stack	Compiler Option Definition	Software Driver[1]
8-bit generic bus	ZiLog eZ80L92 [2]	ZTP 1.3.2 [3]	ZTP 1.3.2	_EZ80L92	KS88xx
	Renesas M16C/62P[4]	None	OpenTCP 1.0.4 [5]	M16C_62P	KS88xx
16-bit generic bus	Renesas M16C/62P	None	OpenTCP 1.0.4	M16C_62P	KS88xx
32-bit generic bus	Renesas SH7751R[6]	vxWorks 5.5.1 Tornado 2.2.1[7]	vxWorks 5.5.1 Tornado 2.2.1	DEF_VXWORKS	KS88xx
	Renesas SH7760	WinCE 5.0	WinCE 5.0	_WIN32	KS88xx
		Linux 2.4 / 2.6	Linux 2.4 / 2.6	DEF_LINUX	KS88xx
		Windows 2000/XP	Windows 2000/XP	_WIN32	KS88xx
PCI bus	Renesas SH7751R	vxWorks 5.5.1 Tornado 2.2.1	vxWorks 5.5.1 Tornado 2.2.1	DEF_VXWORKS	KS88xx
		Linux 2.4 / 2.6	Linux 2.4 / 2.6	DEF_LINUX	KS88xx
		Windows 2000/XP	Windows 2000/XP	_WIN32	KS88xx

[1] All the KS88xx drivers are available upon request.

[2] Please reference the **ps0130.pdf *"eZ80Acclaim!*** ™ flash Microcontrollers eZ80L92 MCU Product Specification**"** for detailed information about the eZ80L92 MCU, via the website www.zilog.com.

[3] Please reference "ZiLog TCP/IP Software Suite Programmer's Guide Reference Manual" (RM000806) for detailed information about the ZTP, via the website www.zilog.com.

[4] Please reference the **M16C62_Hardware_Manual** for detail information about M16C/62P microprocessor, via the website www.renesas.com.

[5] OpenTCP® is an Open Source project that brings a TCP/IP stack to embedded systems, and available under Open Source license. The code is supported and distributed via the website www.opentcp.org. Information about OpenTCP license may also be obtained by visiting http://www.opentcp.org/license.txt. Please reference the **OpenTCP_App_Note** manual for detail information about OpenTCP protocol stack microprocessor via the Micrel KS88xxM Eval Kit CD-ROM.

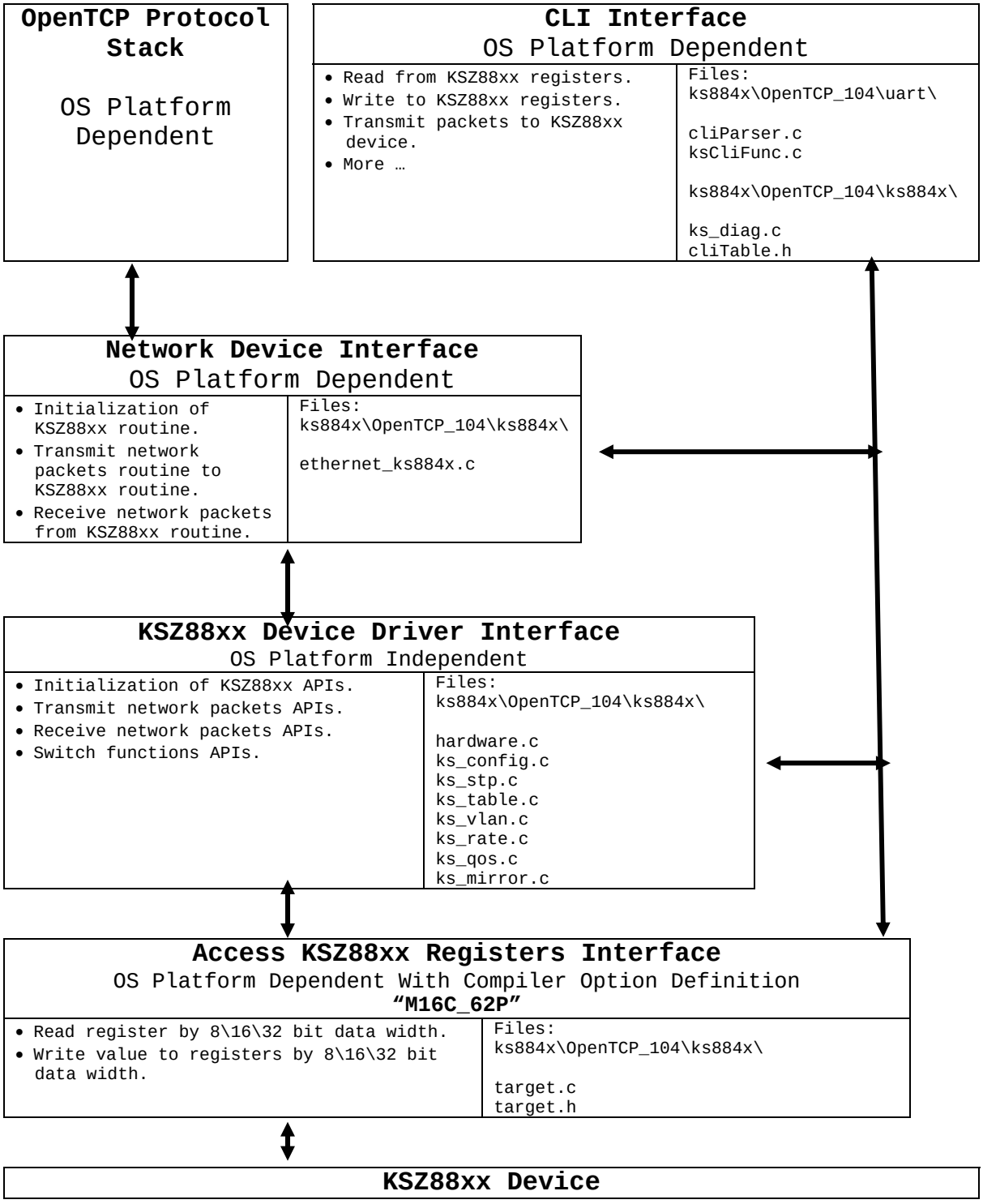
[6] Please reference the **e602201_sh7751 "**Hitachi SuperHÔ RISC engineSH7751 Series SH7751,SH7751R Hardware Manual "for detail information about SH7751R microprocessor via the website www.renesas.com.

[7] VxWorks ® and Tornado ® are license from Wind River System, Inc. For contact information, please visit the website www.windriver.com.

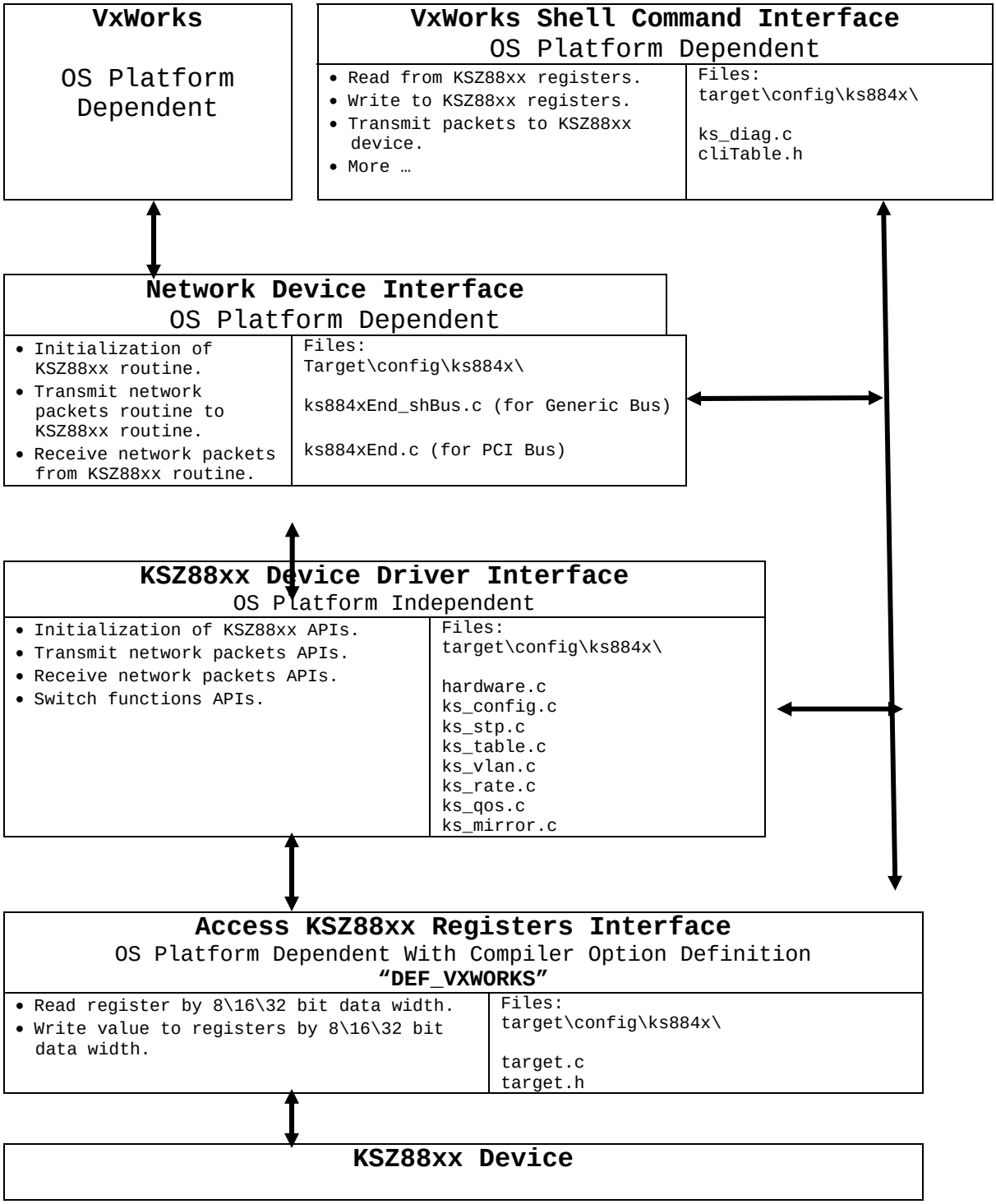
Table 2-3. Driver Platform Support

The following sections are KS88xx drivers block diagram of the some of platforms we provided.

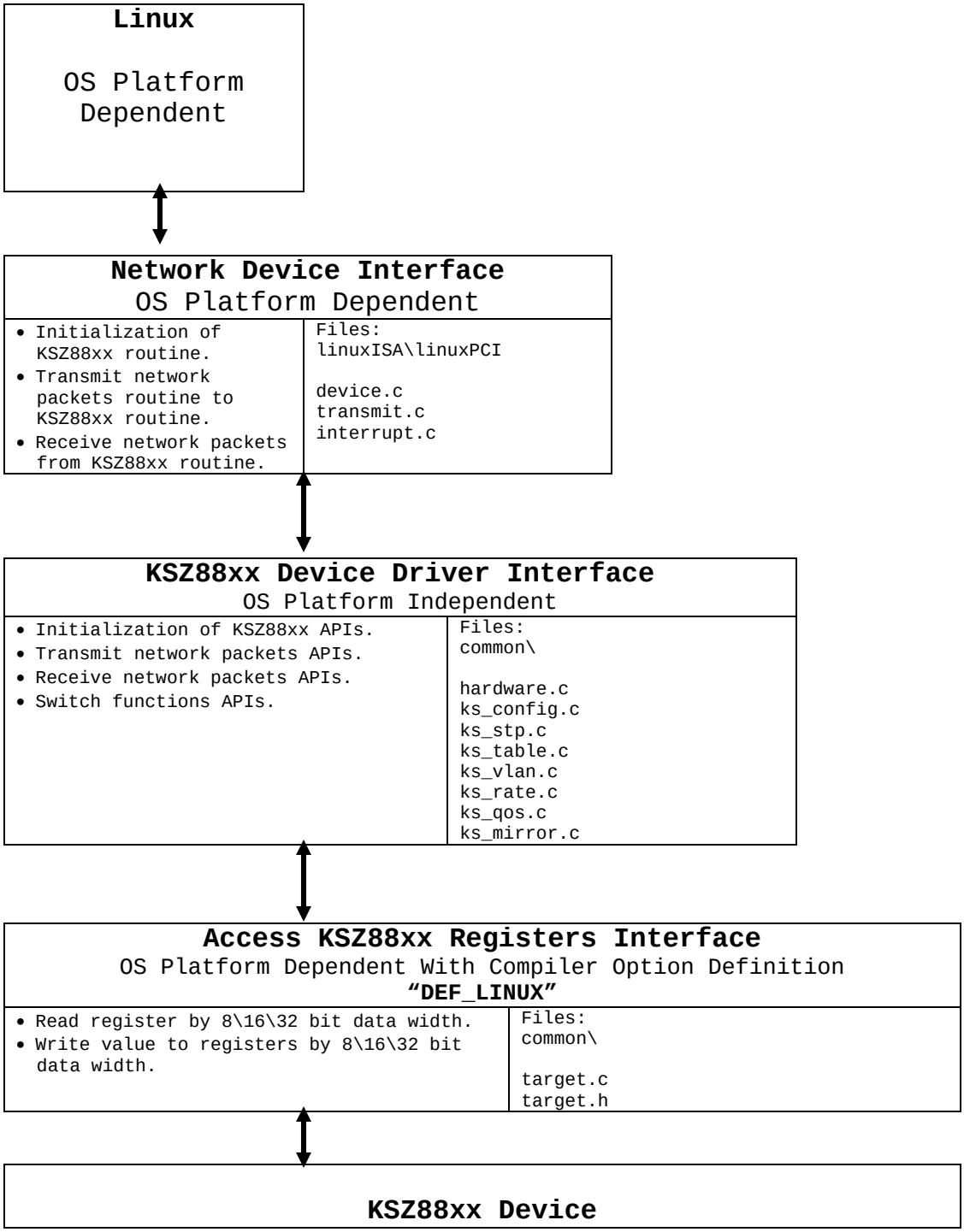
2.3.1 M16C With OpenTCP Driver Block Diagram



2.3.2 SH7751R With VxWorks OS Driver Block Diagram



2.3.3 Linux OS Driver Block Diagram



2.3.4 Definitions to Generate a Particular KS88xx Driver

The KS88xx driver can be built for different sets of drivers:

- KS8841 driver for KSZ8841M/KSZ8861M generic bus device
- KS8841 driver for KSZ8841P PCI bus device
- KS8842 driver for KSZ8842M/KSZ8862M generic bus device
- KS8842 driver for KSZ8842P PCI bus device

The drivers are designed to operate the KSZ88xx devices and demonstrate their hardware features. They share some common code that is called the KS884X library. This library provides an application programming interface to program the KSZ88xx hardware. Because the library code is shared in several platforms, the drivers may not run efficiently. To increase performance, the conditional INLINE can be defined to put some functions inline.

The generic version of the KS884X driver is used for generic bus. It uses a list of banks to access registers.

The PCI version of the KS884X driver is used for PCI bus. It uses a flat address space to access registers. It uses lists of descriptors to send and receives packets.

There are some define identifiers that must be defined from the compiler options to generate a particular driver for a particular device.

DEF_KS8841	DEF_KS8842	KS_ISA_BUS	KS_ISA	KS_PCI_BUS	KS_PCI	Driver
Yes	No	Yes	Yes	No	No	KSZ8841/8861 Generic Bus Driver
Yes	No	No	No	Yes	Yes	KSZ8841 PCI Bus Driver
No	Yes	Yes	Yes	No	No	KSZ8842/8862 Generic Bus Driver
No	Yes	No	No	Yes	Yes	KSZ8842 PCI Bus Driver

Table 2-3-4. Definitions to Generate a Particular KSZ88xx Driver

3 Mapping KSZ88xx Device to Host CPU Memory Area

There are different ways for host CPU system to map KSZ88xx device base memory address to system memory address when it is generic bus interface or PCI bus interface.

Note: Mapping KSZ88xx device to the host CPU memory address is a major important job and more difficult than port KS88xx driver. User need to configure host CPU bus control to meet all the KSZ88xx timer requirements in order to read\write to the device.

3.1 Mapping KSZ88xxM Generic Bus Device to Host CPU Memory Area

3.1.1 Assign Base Memory Address to the Device

The KSZ88xx with generic bus interface is an external device. It can be mapped to either memory space or I/O space on the host CPU memory.

However, the I/O space is accessed using special I/O instructions on some host CPU platforms.

The following processes are mapping KSZ88xxM to a host CPU memory space:

- Configures one of available CS (Chip Select) on your host CPU processor to access a external device.
- Configures the CS data width according to your KSZ88xxM device generic data bus, 8, 16, or 32 bits.
- Configures the CS wait state according to KSZ88xxM hardware timing specification.
- Configures the CS memory base address according to KSZ88xxM Base Address Register (BAR). The BAR holds the base address for decoding a device access, and must be the same as the CS memory base address.
- Configures the CS memory size large enough for KSZ88xxM device (16 bytes).
- Sets the CS memory base address to KSZ88xxM hardware structure phw->m_ulVIoAddr.

For example,

- Mapping KSZ88xxM device to Renesas M16C/62P platform (16bit generic bus).

- Selects CS2 to map KSZ88xxM memory space.

- Sets M16C/62P "byte" pin pulled low and this will set the external bus as a 16-bit bus.

- Configures CS2 to 3 wait states.

- Configures CS2 memory base address to 0x10000, and KSZ88xxM is mapped to address 0x10300. KSZ88xxM Base Address Register (BAR) is 0x0300.

- Sets phw->m_ulVIoAddr = 0x10300.

• Mapping KSZ88xxM device to Renesas SH7751R platform (32bit generic bus).

- Selects CS4 (Expansion Area4) to map KSZ88xxM memory space.

- Sets CS4 to 32-bit bus size. And also configure CS4 to “byte control mode” to allow accessing KSZ88xxM by 8-bit, 16-bit, or 32-bit.

- Configures CS4 to 3 wait states.

- Configures CS4 memory base address to logical address 0xB0000000 and KSZ88xxM is mapped to logical address 0xB0000000. KSZ88xxM Base Address Register (BAR) is 0x0000.

- Sets phw->m_ulVIoAddr = 0xB0000000.

• Mapping KSZ88xxM device to ZiLog eZ80L92 platform (8bit generic bus).

- Selects CS3 to map KSZ88xxM memory space.

- Configures CS3 to 0 wait states.

- Configures CS3 memory base address to 0x100000 and KSZ88xxM is mapped to address 0x100000. KSZ88xxM Base Address Register (BAR) is 0x0000.

- Sets phw->m_ulVIoAddr = 0x100000.

3.1.2 Connecting KSZ88xxM Device Interrupt to Host CPU Interrupt Source

KSZ88xxM device interrupt source are level trigger--active low. Configure KSZ88xxM interrupt pin to one of your host CPU interrupt source.

The following processes are connecting KSZ88xxM interrupt source to a host CPU interrupt source:

- Selects a specified interrupt source from your host CPU according to KSZ88xx BSP board design.

- Configures the interrupt source to level trigger active low.

- Connects the KSZ88xxM ISR routine to a specified interrupt vector.

For example,

• Connecting KSZ88xxM interrupt to Renesas M16C/62P platform.

- Selects INT2 as host CPU interrupt source.

- Configures INT2 as priority level 4, falling edge triggered.

- Connects KSZ88xxM ISR routine ks884xIntr() to vector table interrupt number 31.

• Connecting KSZ88xxM interrupt to Renesas SH7751R platform.

- Selects SLOT_IRQ1 as host CPU interrupt source.

- Configures SLOT_IRQ1 as priority level 2, active low edge triggered.

- Connects KSZ88xxM ISR routine ks8842xEndSHInt() to vector table interrupt number 29.

• Connecting KSZ88xxM interrupt to ZiLog eZ80L92 platform.

- Selects GPIO PD5 as host CPU interrupt source.

- Configures GPIO PD5 as active low triggered.

- Connects KSZ88xxM ISR routine emacisr() to vector table interrupt number 0x50.

3.2 Assign PCI Memory Space for KSZ88xxP PCI Bus Device

The KSZ88xxP with PCI bus interface implement PCI v2.2 bus protocols and configuration space. It supports bus master reads and writes to CPU memory, and CPU access to on-chip register space. When the CPU reads And writes the configuration registers of the KSZ88xxP, it is as a slave. So the KSZ88xxP can be either a PCI bus master or slave.
The PCI bus interface also manages interrupt generation for a host processor.

When the KSZ88xxP target is first powered on, the configuration software, which is frequently referred to as the **PCI bus enumerator** that provided by OS, must scan the PCI bus to determine what PCI devices exit and what configuration requirements they have.

The **PCI bus enumerator** reads a subset of a device's configuration registers in order to determine the presence of the device and its type. Having determined the presence of the device, the software then accesses the device's other configuration registers to determine how many blocks of memory and/or IO space the device requires. Since KSZ88xxP PCI device is operation on memory base only, it then programs the device's memory address to its configuration registers **CBMA**.

Since the KSZ88xxP device indicates usage of a PCI interrupt request line (via its configuration registers **CFIT**), The **PCI bus enumerator** also programs it with routing information indicating what system interrupt request (IRQ) line the device's PCI interrupt request line is routed to by the system.

The following processes are mapping KSZ88xxP to a host CPU PCI memory space:

- The **PCI bus enumerator** scans all the PCI devices on the system PCI bus.
- The **PCI bus enumerator** assigns the KSZ88xxP PCI device memory base address (via its configuration registers **CBMA**).
- The **PCI bus enumerator** routes the KSZ88xxP PCI interrupt request line to system interrupt request (IRQ).
- The driver configures the rest of KSZ88xxP PCI configuration register.
- Sets PCI device memory base address (**CBMA**) to KSZ88xxP hardware structure phw->m_ulVIoAddr.

- The **PCI bus enumerator** in vxWorks OS platform is pciAutoConfig()from VxWorks PCI Library.
- The **PCI bus enumerator** is included in the kernel of the Linux OS platform.

4 KSZ88xx Device Initialization

The initialization routine is the first routine in the hardware-specific sublayer that the system calls during initialization. The purpose of this routine is to prepare the KSZ88xx device and the software driver for immediate use by the system.

The most of initialization functions are same for both generic bus interface and PCI bus interface of the KSZ88xx. But, the device for PCI bus interface, it needs additional initializations, such as for PCI Configuration Space registers, and device's DMA transmit\receive descriptors list.

4.1 KSZ88xxM Generic Bus Interface Device Initialization

4.1.1 Register Setting for Switch

This section describes the typical register settings for the switch function with generic bus interface. For most of the switch registers, just use the default value. The following table only describes the bit setting other than the default value.

Register Name [bit] [8]	Description
SGCR3 [5]	Enable Switch MII flow control. Flow control between QMU and the host port of the switch.
SIDER [0]	Set 1 to start the switch function[9].

Table 4-1-1. Typical Switch Register Settings for Generic Bus

4.1.2 Register Setting for Transmit

This section describes the typical register settings for transmitting packets from CPU processor to KSZ88xx[10] with generic bus interface.

Register Name[bit]	Description
TXCR [3]	Enable transmit flow control. The KSZ88xx will transmit a PAUSE frame when the Receive Buffer capacity has reached a threshold level that may cause the buffer to overflow.
TXCR [2]	Enable transmit padding. The KSZ88xx will automatically add a padding field

[8] Using "Register Name[n]" to indicate bit number in the Register. E.g. "TXCR [3] "is bit 3 of TXCR Transmit Control register.
[9] Start switch function is for KS8842/KSZ8862 only.
[10] KZS88xx means KSZ8841M. KSZ8842M, KSZ8861M, or KSZ8862M device.

	to a packet before transmits to network if packet from CPU port is shorter than 64 bytes.
TXCR [1]	Enable transmit CRC. The KSZ88xx will automatically add CRC checksum field to the end of a transmit packet from CPU port.
TXCR [0]	Enable transmit. Place KSZ884x transmits processor in running state. Software driver can start sending packet to the KSZ88xx.
ETXR [0]	Write value 1 to set early transmit threshold for 64-byte.
ETXR [7]	Enable early transmit function if you want to perform the early transmit[11].
IER [14]	Enable transmit interrupt.
IER [12]	Enable transmit underrun interrupt[12].

Table 4-1-2. Typical Transmit Register Settings for Generic Bus

4.1.3 Register Setting for Receive

This section describes the typical register settings for receiving packets from KSZ88xx to CPU processor with generic bus interface.

Register Name [bit]	Description
RXCR [10]	Enable receive flow control. The KSZ88xx will acknowledge a PAUSE frame from the receive interface; i.e., the outgoing packets will be pending in the TXQ until the PAUSE frame control timer expires.
RXCR [7]	Enable KSZ88xx receive all broadcast frames.
RXCR [6]	Enable KSZ88xx receive all multicast frames.
RXCR [5]	Enable KSZ88xx receive unicast frames that match the 48-bit Station MAC address.
RXCR [3]	Enable KSZ88xx strip the CRC on received frames before sending to CPU processor.
RXCR [0]	Enable Receive. Place KSZ88xx receives processor in running state.
ERXR [4~0]	Write value 1 to set early receive threshold for 64-byte.
ERXR [7]	Enable early receive function if you want to perform the early receive[11].
IER [13]	Enable receive interrupt[13].
IER [10]	Enable receive early interrupt[14].

Table 4-1-3. Typical Receive Register Settings for Generic Bus

[11] The early transmit or early receive function are valid only on KS8841/KSZ8861.
[12] Enable transmit underrun interrupt only when activated early transmit function.
[13] Enable receive interrupt if the software driver wants to receive a packet from the interrupt.
[14] Enable early receive interrupt only when under early receive function.

4.1.4 Initialization Routine

Here is an example of routine NE2000Init() to initialize KSZ88xxM-16 device with API that the driver provides on Renesas M16C/62P platform (16bit generic bus).

	Descriptions	Driver API
1	Mapping KSZ88xx to the host CPU memory, and configure Chip Select.	M16c_ports.c / sys_init()
2	Connecting KSZ88xx device interrupt to the host CPU interrupt source INT2, and configure INT2 as priority level 4, falling edge triggered.	

3	Allocate the KS884x hardware structure.
	
	PHARDWARE phw;
	phw = (PHARDWARE)calloc (sizeof (HARDWARE), 1);
	
	Set KS884x base memory map address
	
	phw->m_ulVIoAddr = 0x10300;
		ethernet_ks884x.c / NE2000Init()

4	Allocate the KS884x internal set API structure (for Generic bus).
	
	struct hw_fn *pks_fn=NULL;
	pks_fn = (struct hw_fn *)calloc (sizeof (struct hw_fn), 1);
		ethernet_ks884x.c / NE2000Init()

5	Set KS884x driver internal API for Generic bus by call
	
	ksSetApiFunctions(pks_fn);
	phw->m_hwfn = pks_fn;
		ethernet_ks884x.c / ksSetApiFunctions()
		
		/* Sample code to generic bus API. */

6	Configure user parameters for device receive controller according to the system OS requested.
	
	phw->m_bPromiscuous=FALSE; //host not receives all the incoming packets
	phw->m_bAllMulticast=FALSE; // host not receives multicast packets
		ethernet_ks884x.c / NE2000Init()

7	Reset KSZ88xx device. HardwareReset(phw);	ethernet_ks884x.c / NE2000Init() hardware.c / HardwareReset ()
8	Check KSZ88xx device ID. HardwareInitialize (phw);	ethernet_ks884x.c / NE2000Init() hardware.c / HardwareInitialize ()
9	Read device station MAC address. HardwareReadAddress(phw); Note: If device station MAC address is not assigned by EEPROM, Set driver's MAC address to device station MAC address.	hardware.c / HardwareReadAddress ()
10	For KS8842 driver, set device switch MAC address.	ks_config.c / SwitchSetAddress ()

	SwitchSetAddress (phw, phw->m_bOverrideAddress);	
11	Clear device MIB counters. phw->m_bPortSelect = 0; /* clear Port 1 */ HardwareClearCounters(phw); #ifdef DEF_KS8842 phw->m_bPortSelect = 1; /* clear Port 2 */ HardwareClearCounters(phw); #endif phw->m_bPortSelect = 2; /* clear Port 3 */ HardwareClearCounters(phw);	hardware.c / HardwareClearCounters ()
12	- Initialization KS884x hardware structure default setting for device transmit control. ° Enable padding (device automatically adds a padding field to packet shorter than 64 bytes). ° Enable CRC (device automatically adds a CRC checksum field to the end of a transmit frame). ° Enable QMU transmit flow control. ° Setup Transmit Frame Data Pointer Auto-Increment. ° Enable transmit. - Initialization KS884x hardware structure default setting for device receive control. ° Enable receive broadcast frames. ° Enable receive unicast frames. ° Enable receive strip CRC (device strips the CRC on the received frames). ° Enable QMU receive flow control. ° Setups Receive Data Pointer Auto-Increment. ° Enable receive. - Enable Switch MII flow control. - Enable WOL by detection of magic packet. - Initialization default setting for device port control. - Initialization default setting for device port PHY control. ° Sets port(s) auto-negotiation. HardwareSetup();	hardware.c / HardwareSetup ()
13	Initialization device transmit/receive control according to KS884x hardware structure value. HardwareEnable(phw);	hardware.c / HardwareEnable ()
14	Enable device switch engine. SwitchEnable (phw, TRUE);	ks_config.c / SwitchEnable ()
15	Connects KS88xx ISR routine ks884xIntr() to vector table interrupt number 31. .glb _ks884xIntr .lword _ks884xIntr ; INT2 (vector 31)	sect30_62pskp.inc
16	Initialization KS884x hardware structure default setting for device interrupt mask. HardwareSetupInterrupt (phw);	hardware.c / HardwareSetupInterrupt ()

17	Clear the device interrupts status. HardwareAcknowledgeInterrupt(phw, 0xFFFF);	hardware.c / HardwareAcknowledgeInterrupt ()
18	Enable the device interrupt sources HardwareEnableInterrupt(phw);	hardware.c / HardwareEnableInterrupt ()
19	Enable the host CPU interrupt source for the KSZ88xx device. SYS_INT_ENABLE;	ethernet_ks884x.c / SYS_INT_ENABLE
20	Get the device port(s) link status. SwitchGetLinkStatus(phw);	ks_config.c / SwitchGetLinkStatus()
21	Show KS88xx BSP information and port(s) link status. - Bus interface of KSZ8841 or KSZ8842 interface. - OpenTCP Ethernet driver version. - KS8842 or KS8841device driver version. - KSZ8842 or KSZ8841chip ID. - KS8842 or KS8841 driver reversion. - KSZ8842 or KSZ8841 base address that is mapped to the CPU memory space. - MAC address of KS8841 or KS8842 station. - IP address of KS8841 or KS8842 target. - Port Link Status. showKsBspInfo (); showKsLinkStatus ();	ethernet_ks884x.c / showKsBspInfo () showKsLinkStatus()

Table 4-1-4. NE2000Init Initialize Routine.

4.2 KSZ88xx PCI Bus Interface Device Initialization

4.2.1 Register Setting for PCI Configuration Space

This section describes the typical settings for the PCI Configuration Registers after the **PCI bus enumerator** assigns the device memory base address to **CBMA** register.

Register Name [bit]	Description
CFCS [8]	System Error Enable. The device asserts SERR_N when it detects a parity error on the address phase.
CFCS [6]	Enable Parity Error Response. The device asserts fatal bus error after it detects a parity error.
CFCS [2]	Master Operation. The device is capable of action as a bus master.
CFCS [1]	Memory Space Access. The device responses to memory space accesses.
CFLT [15-8]	Set Configuration Latency Timer to 0x80 pc PCI bus clocks.
CFLT [7-0]	Set Cache Line Size to 8 32-bit words of the system cache line size.

Table 4-2-1. Typical PCI Configuration Register Settings

4.2.2 Register Setting for Switch

This section describes the typical register settings for the switch function with PCI bus interface. For most of the switch registers, just use the default value. The following table only describes the bit setting other than the default value.

Register Name [bit]	Description
SGCR3 [5]	Enable Switch MII flow control. Flow control between QMU and the host port of the switch.
SIDER [0]	Set 1 to start the switch function.

Table 4-2-2. Typical Switch Register Settings for PCI Bus

4.2.3 Register Setting for Transmit

This section describes the typical DMA Transmit Control register settings for transmitting packets from CPU processor to KSZ8841\2P with PCI bus interface.

Register Name[bit]	Description
MDTXC [29-24]	Set DMA Transmit Burst Size to 8 of words to be transferred in one DMA transaction.
MDTXC [18]	Enable the KSZ884x generates correct UDP checksum for outgoing UDP/IP frames.
MDTXC [17]	Enable the KSZ884x generates correct TCP checksum for outgoing TCP/IP

	frames.
MDTXC [16]	Enable the KSZ884x generates correct IP checksum for outgoing IP frames.
MDTXC [9]	Enable Transmit Flow Control. The KSZ884x will transmit a PAUSE frame when the Receive Buffer capacity has reached a threshold level that may cause the buffer to overflow.
MDTXC [2]	Enable Transmit Padding. The KSZ884x will automatically add a padding field to a packet before transmits to network if packet from CPU port is shorter than 64 bytes.
MDTXC [1]	Enable Transmit Add CRC. The KSZ884x will automatically add CRC checksum field to the end of a transmit packet from CPU port.
MDTXC [0]	Enable Transmit. Place KSZ884x transmits processor in running state. Software driver can start sending packet to the KSZ884x.
INTEN [30]	Enable Transmit Interrupt. The device issues an interrupt when it completely transmitted a packet.
INTEN [26]	Enable Transmit Stop Interrupt. The device issues an interrupt when its transmit process stop from running state[15].

Table 4-2-3. Typical Transmit Register Settings for the PCI Bus

4.2.4 Register Setting for Receive

This section describes the typical register settings for receiving packets from KSZ8841\2P to CPU processor with PCI bus interface.

Register Name [bit]	Description
MDRXC [29-24]	Set DMA Receive Burst Size to 8 of words to be transferred in one DMA transaction.
MDRXC [18]	Enable the KSZ884x checks for correct UDP checksum for incoming UDP/IP frames.
MDRXC [17]	Enable the KSZ884x checks for correct TCP checksum for incoming TCP/IP frames.
MDRXC [16]	Enable the KSZ884x checks for correct IP checksum for incoming IP frames.
MDRXC[9]	Enable receive flow control. The KSZ884x will acknowledge a PAUSE frame from the receive interface; i.e., the outgoing packets will be pending in the TXQ until the PAUSE frame control timer expires.
MDRXC [6]	Enable KSZ884x receive all broadcast frames.
MDRXC [5]	Enable KSZ884x receive all multicast frames.
MDRXC [4]	Enable KSZ884x receive unicast frames that match the 48-bit Station MAC address.
MDRXC [0]	Enable Receive. Place KSZ884x receives processor in running state.
INTEN [29]	Enable Receive Interrupt. The device issues an interrupt when it completely placed a packet in the one of Receive Descriptor's buffer.
INTEN [27]	Enable Receive Buffer Unavailable Interrupt. The device issues an interrupt when it indicates that all the Receive Descriptors are owned by Host (out of received buffers), and can't acquired by the device.
INTEN [25]	Enable Receive Stop Interrupt. The device issues an interrupt when its receivet process stop from running state[15].

Table 4-2-4. Typical Receive Register Settings for PCI Bus

[15] If software driver wants to do some debug?

4.2.5 Transmit and Receive Descriptor Lists and Data Buffers

The KSZ88xxP transfers received data frames to the receive buffer in host memory and transmits data from transmit buffers in the host memory. Descriptors that reside in the host memory act as pointers to these buffers.

There are two descriptor lists, one for receive, called receive descriptor, and one for transmit, call transmit descriptor, for the device Rx\Tx DMA. And the descriptors lists reside in the host physical memory address space.

After the system allocate the memory for the transmit and receive descriptors list, it must write the transmit descriptors list base address with LONG WORD (32bit) alignment to device register **TDLB**[31-2], and the receive descriptors list base address with LONG WORD (32bit) alignment to device register **RDLB**[31-2]

A descriptor list is forward linked. The last descriptor may point back to the first entry to create a ring link list structure. For the deep of ring link list (number of descriptors), it is all dependents on your host system memory size and overall throughput performance. But it must at least more than two descriptors for each descriptor list.

There are four control\status registers in each descriptor structures. The following section describes the default setting for transmit and receive descriptor list structures to create a ring link list.

- **Transmit Descriptors Registers Initial Setting**

Register Name [bit]		Description
Bit	Value	
TDES0		
[31]	0	Set this descriptor is owned by the host.
TDES1		
TDES2		
[31-0]	0	Set transmit buffer address point to NULL.
TDES3		
[31:0]	address	Point to next descriptor address.

Table 4-2-5-1. Typical Transmit Descriptors Register Settings for PCI Bus

Here is example of display of transmit descriptors list structure:

```
Tx Desc 00 addr:0xAC747FE0: TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC747FF0
Tx Desc 01 addr:0xAC747FF0: TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC748000
Tx Desc 02 addr:0xAC748000: TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC748010
Tx Desc 03 addr:0xAC748010: TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC748020
```

```
Tx Desc 04 addr:0xAC748020: TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC748030
•••

Tx Desc 31 addr:0xAC7481D0, TDES0:0x00000000, TDES1:0x00000000, TDES2:0x00000000,
                            TDES3:0xAC747FE0
```

- **Receive Descriptors Registers Initial Setting**

Register Name [bit]		Description
Bit	Value	
RDES0		
[31]	1	Set this descriptor is owned by the device.
RDES1		
[10-0]	0x000007FC	Set to maximum receive buffer size (2044 bytes). User can adjust to their application need.
RDES2		
[31-0]	address	Set receive buffer address which reside in the system memory.
RDES3		
[31:0]	address	Point to next descriptor address.

Table 4-2-5-2. Typical Receive Descriptors Register Settings for PCI Bus

Here is example of display of receive descriptors list structure:

```
Rx Desc 00 addr:0xAC748200, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC724084,
                            RDES3:0xAC748210
Rx Desc 01 addr:0xAC748210, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC724888,
                            RDES3:0xAC748220
Rx Desc 02 addr:0xAC748220, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC72508C,
                            RDES3:0xAC748230
Rx Desc 03 addr:0xAC748230, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC725890,
                            RDES3:0xAC748240
Rx Desc 04 addr:0xAC748240, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC726094,
                            RDES3:0xAC748250
•••

Rx Desc  n addr:0xAC7483F0, RDES0:0x80000000, RDES1:0x000007FC, RDES2:0xAC733900,
                            RDES3:0xAC748200
```

4.2.6 Initialization Routine

Here is an example of routine pcidev_init() to initialize KSZ88xxP device with API that the driver provides on Linux platform (PCI bus).

	Descriptions	Driver API
1	Static allocate the KS884x hardware structure "**phw**" in Linux device structure "struct dev_info".	device.h
2	The **PCI bus enumerator** scans all the PCI devices on the system PCI bus.	Linux Kernel
3	The **PCI bus enumerator** assigns the KSZ88xx PCI device memory base address.	Linux Kernel
4	The **PCI bus enumerator** routes the KSZ88xx PCI interrupt request line to system interrupt request (IRQ).	Linux Kernel
5	The KSZ88xx driver request the device base memory address from Kernel. And Set KS884x base memory address. pHardware->m_pVirtualMemory = ioremap(reg_base, reg_len);	device.c / pcidev_init ()
6	Probe the KSZ88xx PCI device with the **PCI bus enumerator** given memory base address pHardware->**m_pVirtualMemory** by checking device ID. (1). Reset KSZ88xx device. HardwareReset(&hw); (2). Check KSZ88xx device ID. HardwareInitialize(&hw);	device.c / dev_probe ()
7	Initializes the transmit, receive descriptors list, including allocate receive data buffer for the receive descriptors list to store incoming packet data. AllocateMemory(hw_priv);	device.c / pcidev_init ()

8	Allocate the KS884x internal set API structure (for PCI bus). And Set KS884x driver internal API for PCI bus.
	
	static struct hw_fn* ks8842_fn = NULL;
	ks8842_fn = kmalloc(sizeof(struct hw_fn), GFP_KERNEL);
	
	ks8842_fn->m_fPCI = TRUE;
	ks8842_fn->fnSwitchDisableMirrorSniffer =
	SwitchDisableMirrorSniffer_PCI;
	…
		device.c /
		netdev_init()

9	Clear device MIB counters. phw->m_bPortSelect = 0; /* clear Port 1 */ HardwareClearCounters(phw); #ifdef DEF_KS8842 phw->m_bPortSelect = 1; /* clear Port 2 */ HardwareClearCounters(phw); #endif	device.c / netdev_open ()

	phw->m_bPortSelect = 2; /* clear Port 3 */ HardwareClearCounters(phw);	
10	Configure user parameters for device receive controller according to the system OS requested. phw->m_bPromiscuous=FALSE;/*host not receives all the incoming packets */ phw->m_bAllMulticast=FALSE; /* host not receives multicast packets */	device.c / netdev_open ()
11	- Initialization KS884x hardware structure default setting for device transmit control. ° Enable padding (device automatically adds a padding field to packet shorter than 64 bytes). ° Enable CRC (device automatically adds a CRC checksum field to the end of a transmit frame). ° Set DMA Transmit Burst Size to 8 of words to be transferred in one DMA transaction. ° Enable MAC DMA transmit flow control. ° Enable the device IP checksum generate. ° Enable the device TCP checksum generate. ° Enable the device UDP checksum generate ° Enable transmit. - Initialization KS884x hardware structure default setting for device receive control. ° Enable receive broadcast frames. ° Enable receive unicast frames. ° Set DMA Receive Burst Size to 8 of words to be transferred in one DMA transaction. ° Enable QMU receive flow control. ° Enable the device checks for correct IP checksum. ° Enable the device checks for correct TCP checksum. ° Enable the device checks for correct UDP checksum. ° Enable receive. - Enable Switch MII flow control. - Enable WOL by detection of magic packet. - Initialization default setting for device port control. - Initialization default setting for device port PHY control. ° Sets port(s) auto-negotiation. HardwareSetup();	device.c / netdev_open () hardware.c / HardwareSetup ()
12	Set base address of Transmit \Receive descriptor to the device register **TDLB,** and **RDLB**. HardwareSetDescriptorBase(pHardware, phw ->m_TxDescInfo.ulRing, phw ->m_RxDescInfo.ulRing);	device.c / netdev_open ()
13	If device station MAC address is not assigned by EEPROM, Set driver's MAC address to device station MAC address. And, for KS8842 driver, set device switch MAC address. HardwareSetAddress(phw);	device.c / netdev_open ()
14	Initialization KS884x hardware structure default setting for device interrupt	device.c /

	mask. HardwareSetupInterrupt (phw);	netdev_open ()
15	Initialization device transmit/receive control according to KS884x hardware structure value. HardwareEnable(phw);	device.c / netdev_open ()
16	Enable device switch engine. SwitchEnable (phw, TRUE);	ks_config.c / SwitchEnable ()
17	Clear the device interrupts status. HardwareAcknowledgeInterrupt(phw, 0xFFFFFFFF);	hardware.c / HardwareAcknowledgeInterrupt ()
19	Enable the device interrupt sources HardwareEnableInterrupt(phw);	device.c / netdev_open ()

Table 4-2-6. pcidev_init() Initialize Routine.

5 KSZ88xx Driver Transmit Packets to Device – Flowchart

The transmit routine is called by the upper layer to transmit a contiguous block of data through the Ethernet controller.

It is your choice as to how the transmit routine is implemented. The software may elect to wait until the Ethernet controller is ready to transmit new data and then synchronously send the new frame, or it may append the new frame to a queue in software and transmit the data asynchronously when a transmit completed interrupt signals that the controller is able to accept a new transmit request. The sample driver included in this BSP returns FALSE when the KSZ88xx TXQ is not ready to transmit new frame.

If the Ethernet controller encounters an error while transmitting the frame, it is your choice as to whether the driver should attempt to retransmit the same frame or discard the data. The sample driver included in this BSP does not attempt to retransmit a frame.

5.1 KSZ88xxM Generic Bus Interface Transmit Routine

There are only a few steps to transmit an Ethernet packet from upper layer to KSZ88xxM device for the generic bus interface.

1) Check the device QMU TXQ has enough amount of memory for the Ethernet packet data by read the device register TXMIR (bank 16, offset 0x08).

2) Sets “control word”, and “byte count” to the frame header through a pair of the device registers QDRL (bank 17, offset 0x08).

3) Write (copy) the Ethernet packet data to the device QMU TXQ through a pair of the device registers QDRL (bank 17, offset 0x08), and QDRH (bank 17, offset 0x0A).

3.1) the pseduo code to transmit packet to the KSZ88xxM-8 (8bit generic bus)

```
UINT8 *pTxData;
UINT32 addr;
Select bank 17;
while (txPacketLength > 0)
{
   addr = QDRL;
   *(UINT8 *) addr = *pTxData++;
   *(UINT8 *) (addr+1) = *pTxData++;
   *(UINT8 *) (addr+2) = *pTxData++;
   *(UINT8 *) (addr+3) = *pTxData++;
   txPacketLength -=4;
}
```

3.2) the pseduo code to transmit packet to the KSZ88xxM-16(16bit generic bus)

```
UINT16 *pTxData;
UINT32 addr;
Select bank 17;
while (txPacketLength > 0)
{
   addr = QDRL;
   *(UINT16 *) addr = *pTxData++;
   *(UINT16 *) (addr+2) = *pTxData++;
   txPacketLength -=4;
}
```

3.3) the pseduo code to transmit packet to the KSZ88xxM-32 (32bit generic bus)

```
UINT32 *pTxData;
UINT32 addr;
Select bank 17;
while (txPacketLength > 0)
{
   addr = QDRL;
```

```
            *(UINT32 *) addr = *pTxData++;
            txPacketLength -=4;
         }
```

4) Issue the ENQUEUE transmits command for the device transmits the Ethernet packet to the Network by write 1 to the device register TXQCR (bank 17, offset 0x0).

The sample code of transmit routine for M16C microprocessor with KSZ88xxM-16 under OpenTCP is

```
UINT8 ksSendFrame (UINT8* pData, UINT16 dataLength, ULONG port);
```

The first parameter is a pointer to a system data buffer that contains the data frame to be transmitted. The second parameter is data length. The third parameter is device destination port that this data frame transmits to.

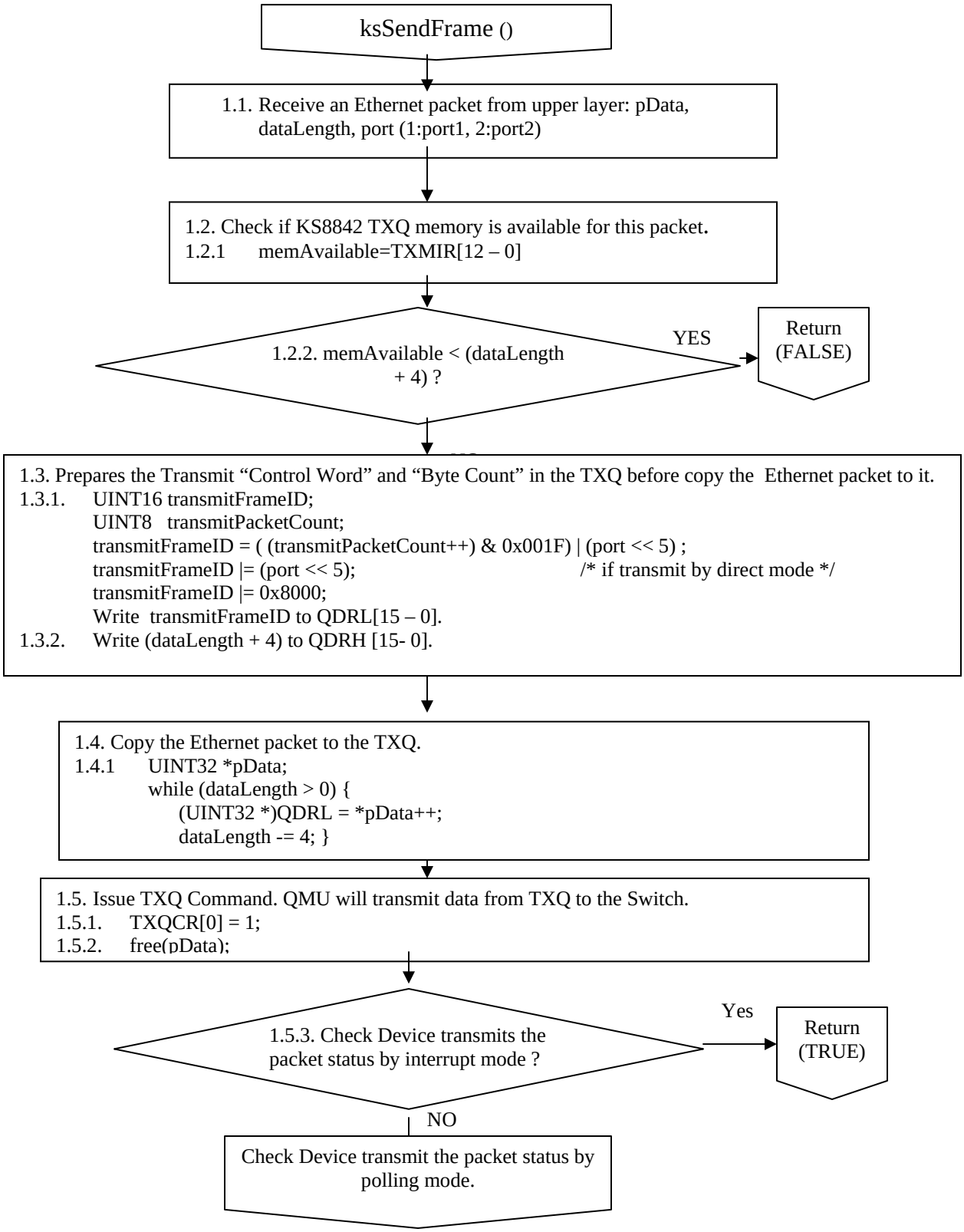
The following table describes the detail steps to transmit a data frame to KSZ88xxM-16 device:

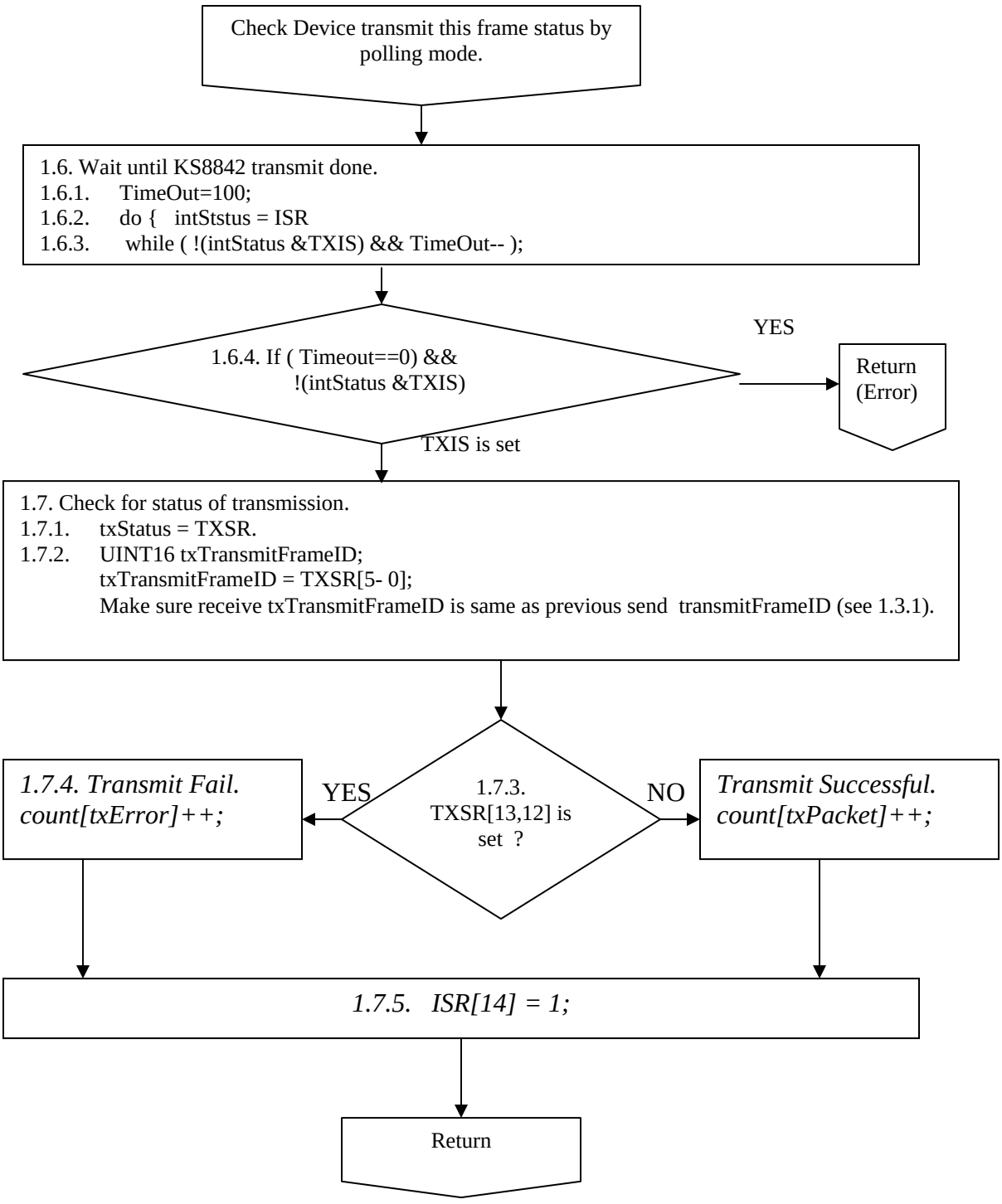
	Descriptions	Driver API
1.1	Receive an Ethernet packet from upper layer. There are three variables are needed from upper layer protocol stack. ksSendFrame (pData, dataLength, port); (1). Packet data pointer (pData). Point to the host CPU system memory space contains the completed Ethernet packet. (2). Packet length (dataLength). The Ethernet packet length not includes CRC. (3). Destination port number[16] (port).	Ethernet_ks884x.c / ksSendFrame()
1.2	Checking if KSZ8842 TXQ memory is available for this packet. HardwareAllocPacket(phw, dataLength); 1.2.1 Read memAvailable = Transmit Memory Available status from TXMIR[12 - 0]. 1.2.2 If memAvailable < packetDataLength + 4 (2bytes of "Control Word", 2bytes of "Byte Count"), return FALSE.	Ethernet_ks884x.c / ksSendFrame() hardware.c / HardwareAllocPacket ( )
1.3	Prepares the Transmit "Control Word" and "Byte Count" in the TXQ before copy the Ethernet packet to it. txCntlAndLength = HardwareSetTransmitLength(phw, dataLength); 1.3.1. Set TXIC (transmit interrupt after the present frame has been transmitted), portNumber to TXDPN [17](Transmit Destination Port Number), and Transmit Frame ID (can using transmit packet number as unique Transmit Frame ID) to form the Transmit "Control Word". Write the Transmit "Control Word" to QDRL QMU Data Register Low. This will set the "Control Word" of TXQ Frame Format. 1.3.2 Write packetDataLength + 4 (2bytes of "Control Word", 2bytes of	Ethernet_ks884x.c / ksSendFrame() hardware.c / HardwareSetTransmitLength ( )

[16] Optional, the value is needed when only transmit packet by direct mode.
[17] Set portNumber to TXDPN if transmit packet by direct mode without go through the look up engine. Otherwise, set zero to TXDPN if transmit packet by lookup mode.

	"Byte Count") to QDRH QMU Data Register High. This will set the "Byte Count" of TXQ Frame Format.	
1.4	Then, copy the Ethernet packet to the TXQ. 1.4.1. Copy Ethernet packet to TXQ by write packet data in DWORD (32-bit) to the QDRL QMU Data Register Low once a time until finished entire packet. HW_WRITE_BUFFER(phw, REG_DATA_OFFSET, pData, dataLength);	Ethernet_ks884x.c / ksSendFrame() hardware.c / HW_WRITE_BUFFER ( )
1.5	Issue TXQ Command. The QMU will transmit data from TXQ to the Switch. HardwareSendPacket(phw); 1.5.1 Set TXQCR[0] to 1. 1.5.2 Free the system memory by pPacketData. 1.5.3. Return from routine as OK if checking Device transmits the packet status by interrupt, otherwise continue.	Ethernet_ks884x.c / ksSendFrame() hardware.c / HardwareSendPacket ( )
1.6	Wait until KSZ8842 transmit done. 1.6.1. Read Interrupt Status Register by maximum 100 times if ISR[14] is not set. 1.6.2. do {Read ISR Interrupt Status Register.} 1.6.3. while ((ISR[14] is not set - transmit is not done) AND (TimoutCount != 0)) 1.6.4. If TimeCount=0 and ISR[14] is not set, something is wrong, return FALSE from routine.	
1.7	Check for status of transmission. 1.7.1. txStatus = Read TXSR Transmit Status Register. 1.7.2. Makes sure the TXFID Transmit Frame ID in the txStatus is same as you previous send packet's Transmit Frame ID (see 1.3.1). 1.7.3. If any of TXSR[13,12] is set (transmit fail), update software driver's transmit error statistics counter, otherwise 1.7.4. Transmit successful, update software driver's transmit packets statistics counter. 1.7.5. Clear Interrupt Status Register by setting ISR[14].	

Table 5-1. ksSendFrame Transmit Routine


ksSendFrame ()
1.1. Receive an Ethernet packet from upper layer: pData, dataLength, port (1:port1, 2:port2)
1.2. Check if KS8842 TXQ memory is available for this packet.
1.2.1 memAvailable=TXMIR[12 – 0]
1.2.2. memAvailable < (dataLength + 4) ?
YES
Return (FALSE)
1.3. Prepares the Transmit "Control Word" and "Byte Count" in the TXQ before copy the Ethernet packet to it.
1.3.1. UINT16 transmitFrameID;
UINT8 transmitPacketCount;
transmitFrameID = ((transmitPacketCount++) & 0x001F) | (port << 5) ;
transmitFrameID |= (port << 5); /* if transmit by direct mode */
transmitFrameID |= 0x8000;
Write transmitFrameID to QDRL[15 – 0].
1.3.2. Write (dataLength + 4) to QDRH [15- 0].
1.4. Copy the Ethernet packet to the TXQ.
1.4.1 UINT32 *pData;
while (dataLength > 0) {
(UINT32 *)QDRL = *pData++;
dataLength -= 4; }
1.5. Issue TXQ Command. QMU will transmit data from TXQ to the Switch.
1.5.1. TXQCR[0] = 1;
1.5.2. free(pData);
1.5.3. Check Device transmits the packet status by interrupt mode ?
Yes
Return (TRUE)
NO
Check Device transmit the packet status by polling mode.


Check Device transmit this frame status by polling mode.
1.6. Wait until KS8842 transmit done.
1.6.1. TimeOut=100;
1.6.2. do { intStstus = ISR
1.6.3. while (!(intStatus &TXIS) && TimeOut--);
1.6.4. If (Timeout==0) && !(intStatus &TXIS)
YES
Return (Error)
TXIS is set
1.7. Check for status of transmission.
1.7.1. txStatus = TXSR.
1.7.2. UINT16 txTransmitFrameID;
txTransmitFrameID = TXSR[5- 0];
Make sure receive txTransmitFrameID is same as previous send transmitFrameID (see 1.3.1).
1.7.3. TXSR[13,12] is set ?
YES
1.7.4. Transmit Fail.
count[txError]++;
NO
Transmit Successful.
count[txPacket]++;
1.7.5. ISR[14] = 1;
Return

5.2 KSZ88xxP PCI Bus Interface Transmit Routine

The PCI version of the transmit routine uses lists of transmit descriptors to send packets to the device transmit MAC DMA.

There are only a few steps to transmit an Ethernet packet from up layer to KSZ88xxP device for the PCI bus interface.

1) Search an available transmit descriptor, its OWN bit is owned by the host (TDES0[31]). Start from current transmit descriptor pointer, move to next descriptor if current descriptor is not available.

2) Sets this available transmit descriptor's transmit buffer address TDES2[31-0] to the output packet buffer address.

3) Sets all the control bits in the TDES1.

3.1) Write packet length to TDES1[10-0],
3.2) Write 1 to TDES1[31] to enable device generated a interrupt when this frame has been transmitted.
3.3) Write transmit destination port number to TDES1[23-20] if it available, otherwise, write 0 to TDES1[23-20].
3.4) Write 1 to TDES1[30], and TDES1[29] if this frame contains one full Ethernet packet.

4) Issue the ENQUEUE transmits command for the device transmits the Ethernet packet to the Network by write 1 to the device register MDTSC (offset 0x0).

5) Free the output packet buffer pointer by this transmit descriptor's transmit buffer address TDES2[31-0] when the driver received a transmit done interrupt from the device.

The sample code of transmit routine for the KSZ88xxP under Linux driver is

```
ks8842p_dev_transmit (struct sk_buff* skb, struct net_device* dev );
```

The first parameter is a pointer to a system data buffer sk_buff structure that contains the data frame to be transmitted. The second parameter is Linux network device structure.

The following table describes the detail steps to transmit a data frame to KSZ88xxP device:

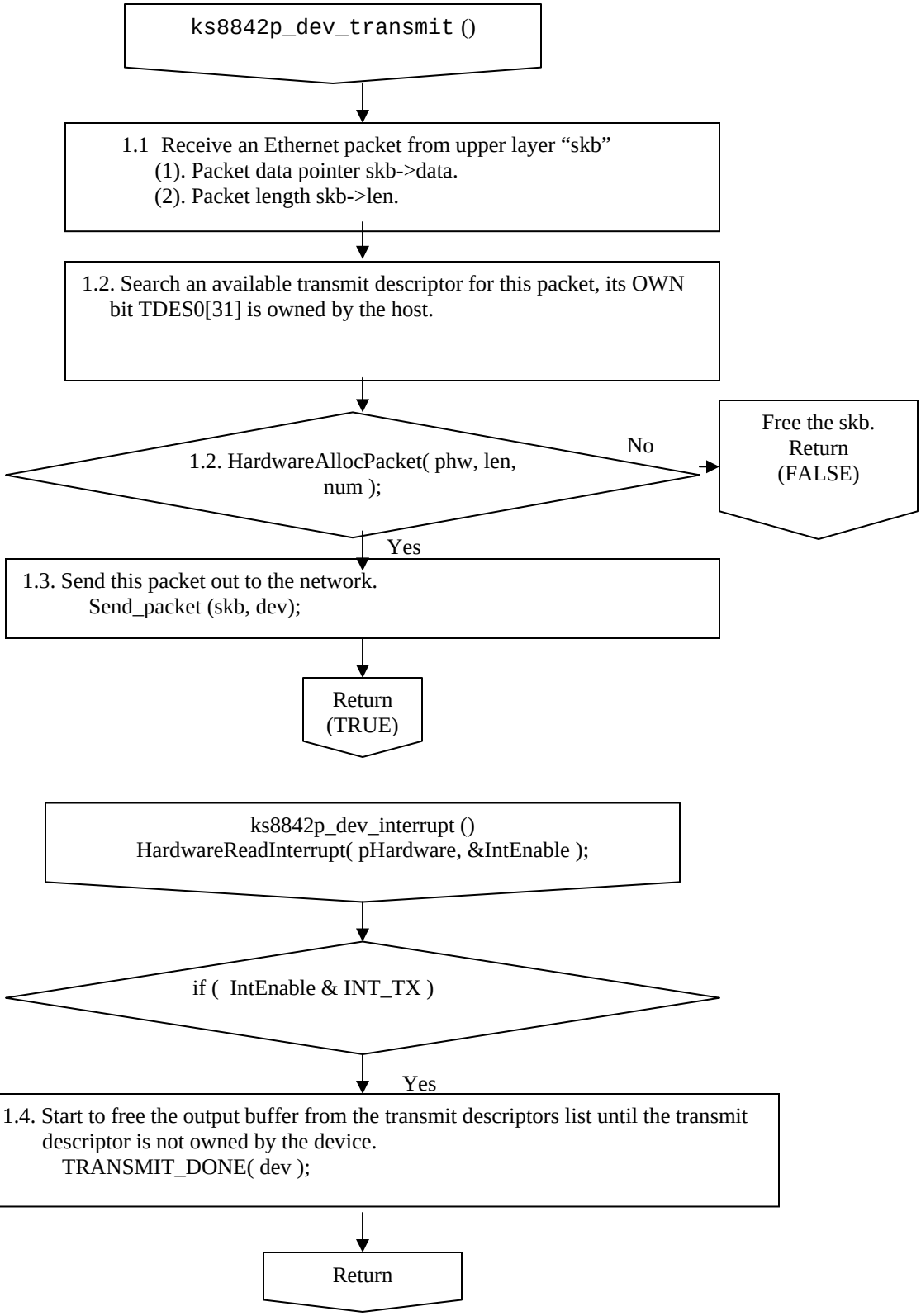
	Descriptions	Driver API
1.1	Receive an Ethernet packet from upper layer. There are two variables are needed from upper layer protocol stack. ks8842p_dev_transmit (struct sk_buff* skb, struct net_device* dev); (1). Packet data pointer (skb->data). Point to the host CPU system memory space contains the completed Ethernet packet. (2). Packet length (skb->len). The Ethernet packet length not includes CRC.	transmit.c / ks8842p_dev_transmit ()
1.2	Search an available transmit descriptor for this packet, its OWN bit TDES0[31] is owned by the host. HardwareAllocPacket(phw, len, num); 1.2.1 Call GetTxPacket() to get next available transmit descriptor for this packet, and pointer by phw ->m_TxDescInfo.pCurrent 1.2.2 Write 1 to the transmit descriptor's First Segment bit TDES1[30].	transmit.c / ks8842p_dev_transmit () hardware.c / HardwareAllocPacket ( )

1.3	Send this packet out to the network.
	
	send_packet(skb, dev);
	
	1.3.1. Record this available transmit descriptor by "pDma", the PDMA_BUFFER structure for free the output buffer later.
	
	pDma = alloc_tx_buf(&hw_priv->m_TxBufInfo, pDesc);
	
	1.3.2. Sets this available transmit descriptor's transmit buffer address TDES2[31-0] to the output packet buffer address "skb->data" by call
	
	SetTransmitBuffer(pDesc, pDma->dma);
	
	1.3.3. Sets the output packet length "skb->len"to this available transmit descriptor's transmit buffer size TDES1[10-0] by call
	
	SetTransmitLength(pDesc, pDma->len);
	
	1.3.4. (1) Write 1 to the transmit descriptor's Last Segment bit TDES1[29] to indicate this frame contains one full Ethernet packet.
	(2) Write 1 to the transmit descriptor's TDES1[31] to enable device generated a interrupt when this frame has been transmitted.
	(3) Set transmit destination port number "phw ->m_bPortTX" to the transmit descriptor's TDES1[23-20].
	(4). Issue the ENQUEUE transmits command for the device transmits the Ethernet packet to the Network by call
	
	HardwareSendPacket(phw);
		transmit.c / ks8842p_dev_transmit ()
		
		transmit.c / send_packet ()
		
		hardware.c / HardwareSetTransmitBuffer ( )
		
		HardwareSetTransmitLength()
		
		HardwareSendPacket ()

1.4	Free the output packet buffer "pDma", when the driver ISR received transmit done INT_TX (INTST[30] interrupt from the device.	interrupt.c / ks8842p_dev_interrupt ()

	TRANSMIT_DONE(dev); 1.4.1. Get the previous last release buffer “pDma” from pInfo->iLast. 1.4.2. while not finished one full transmit descriptor ring 1.4.3. Start free output packet buffer from “pDma” from pInfo->iLast . 1.4.4. Break the while loop when the transmit descriptor is not owned by device. 1.4.5. Get next transmit descriptor to free.	transmit.c / ks8842p_transmit_done ( )

Table 5-2. ks8842p_dev_transmit Transmit Routine


ks8842p_dev_transmit ()
1.1 Receive an Ethernet packet from upper layer "skb"
(1). Packet data pointer skb->data.
(2). Packet length skb->len.
1.2. Search an available transmit descriptor for this packet, its OWN bit TDES0[31] is owned by the host.
1.2. HardwareAllocPacket(phw, len, num);
No
Free the skb.
Return
(FALSE)
Yes
1.3. Send this packet out to the network.
Send_packet (skb, dev);
Return
(TRUE)
ks8842p_dev_interrupt ()
HardwareReadInterrupt(pHardware, &IntEnable);
if (IntEnable & INT_TX)
Yes
1.4. Start to free the output buffer from the transmit descriptors list until the transmit descriptor is not owned by the device.
TRANSMIT_DONE(dev);
Return

6 KSZ88xx Driver Receive Packets from Device – Flowchart

It is your choice as to how the driver receives data frames from the KSZ88xx device either as a result of polling or servicing an interrupt. When an interrupt is received, the OS invokes the interrupt service routine that is in the interrupt vector table.

If your system has OS support, to minimize interrupt lockout time, the interrupt service routine should handle at interrupt level only those tasks that require minimum execution time, such as error checking or device status change. The routine should queue all the time-consuming work to transfer the frame from the KSZ88xx RXQ into system memory at task level.

6.1 KSZ88xxM Generic Bus Interface Receive Routine

It just takes a few steps to receive an Ethernet packet from KSZ88xxM device to upper layer when the interrupt serve routine detect a received frame interrupt on the generic bus interface.

1) While loop to do following steps until there is no receive packet data in the device QMU RXQ by read the device register RXMIR (bank 16, offset 0x0A).

2) Checks the received frame status by read the device register RXSR (bank 18, offset 0x04) or read “status word” from the received frame header through a pair of the device registers QDRL (bank 17, offset 0x08).

3) If it is a valid and good frame, get the received frame length by read the device register RXBC (bank 18, offset 0x06) or read “byte count” from the received frame header through a pair of the device registers QDRL (bank 17, offset 0x08).

4) Read (copy) the Ethernet packet data from the device QMU RXQ through a pair of the device registers QDRL (bank 17, offset 0x08), and QDRH (bank 17, offset 0x0A) to a already allocated system memory buffer.

4.1) the pseduo code to receive a packet from the KSZ88xxM-8 (8bit generic bus)

```
UINT8 *pRxData;
UINT32 addr;
Select bank 17;
while (rxPacketLength > 0)
{
   addr = QDRL;
   *pRxData++ = *(UINT8 *) addr;
   *pRxData++ = *(UINT8 *) (addr+1);
   *pRxData++ = *(UINT8 *) (addr+2);
   *pRxData++ = *(UINT8 *) (addr+3);
   rxPacketLength -=4;
}
```

4.2) the pseduo code to receive a packet from the KSZ88xxM-16(16bit generic bus)

```
UINT16 *pRxData;
UINT32 addr;
Select bank 17;
while (rxPacketLength > 0)
{
   addr = QDRL;
   *pRxData++ = *(UINT16 *) addr;
   *pRxData++ = *(UINT16 *) (addr+2);
   rxPacketLength -=4;
}
```

4.3) the pseduo code to receive a packet from the KSZ88xxM-32 (32bit generic bus)

```
UINT32 *pRxData;
UINT32 addr;
Select bank 17;
while (rxPacketLength > 0)
{
   addr = QDRL;
   *pRxData++ = *(UINT32 *) addr;
   rxPacketLength -=4;
}
```

5) Issue the RELEASE frame command for the device to release this frame buffer memory space from QMU RXQ by write 1 to the device register RXQCR (bank 17, offset 0x02).

The sample driver provides in the BSP is based on Renesas M16C/62P OpenTCP platform. There is no OS involved; the receive routine is called at interrupt level[18].

The following table describes the detail steps to receive a data frame from the KSZ88xxM-16 device:

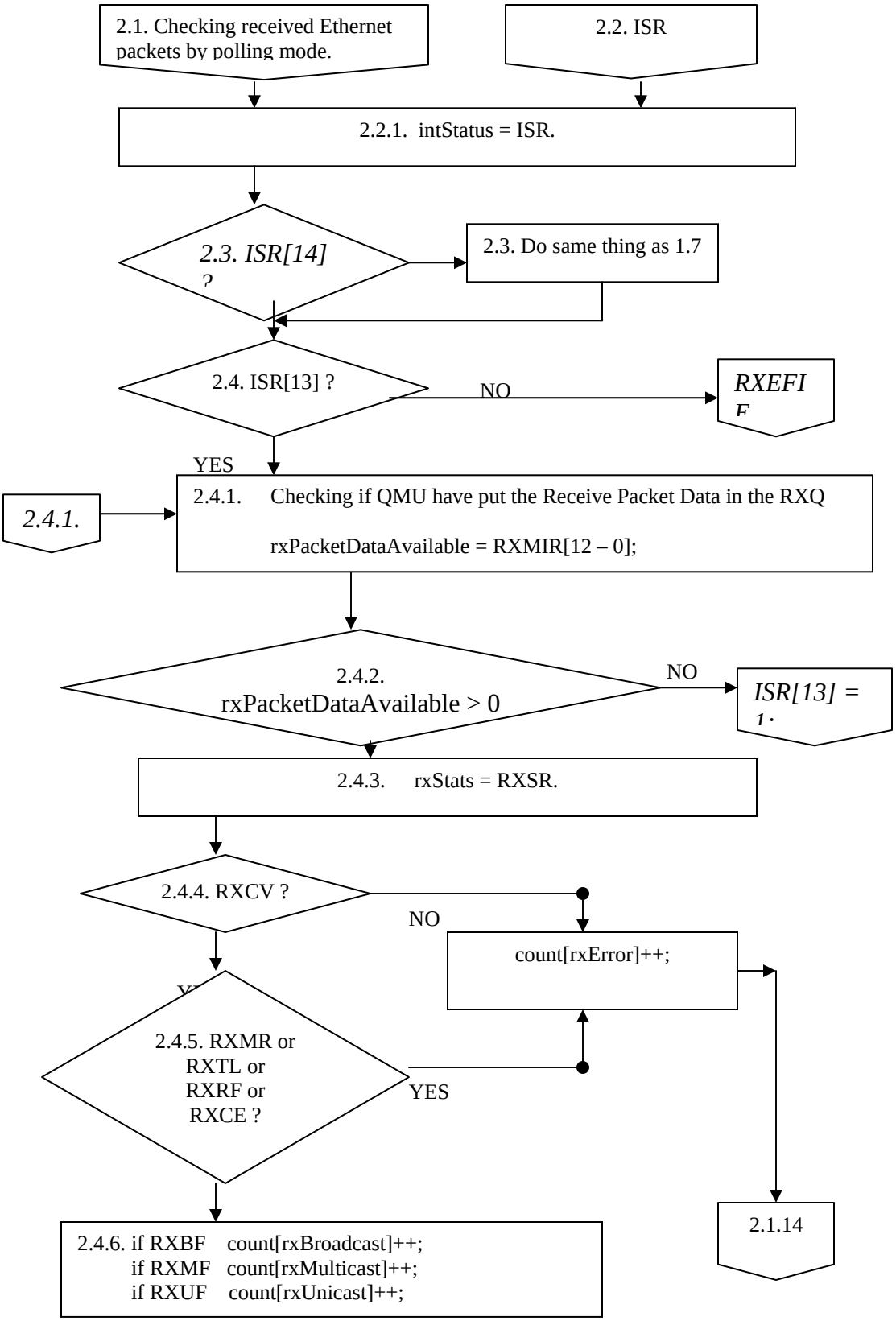
	Descriptions	Driver API
2.1	Checking if received Ethernet packets by polling mode. 2.1.1. While loop reading ISR Interrupt Status Register - ISR. NE2000ReceiveFrame();	gui_demo.c / main() ethernet_ks884x.c / NE2000ReceiveFrame ()
2.2	Checking if received Ethernet packets by interrupt driven ks884xIntr (). 2.2.1. intStatus= read Interrupt Status Register - ISR. HardwareReadInterrupt(phw, &wIntStatus);	ethernet_ks884x.c / ks884xIntr () hardware.c / HardwareReadInterrupt ( )
2.3	If ISR [14] is set, do 1.7 processes. HardwareTransmitDone(phw); HardwareAcknowledgeTransmit(phw);	ethernet_ks884x.c / ks884xIntr () hardware.c / HardwareTransmitDone ( ); HardwareAcknowledgeTransmit();
2.4	If ISR[13] is set, call ksReceiveFrame() 2.4.1 Checking if QMU have put Receive Packet Data in the RXQ by read RXMIR[12 – 0]. rxPacketDataAvailable = RXMIR[12 – 0]. HardwareReceiveMoreDataAvailable (phw); 2.4.2 If no Receive Packet Data in the RXQ (rxPacketDataAvailable <= 0), exist while loop.	Ethernet_ks884x.c / ksReceiveFrame () hardware.c / HardwareReceiveMoreDataAvailable();

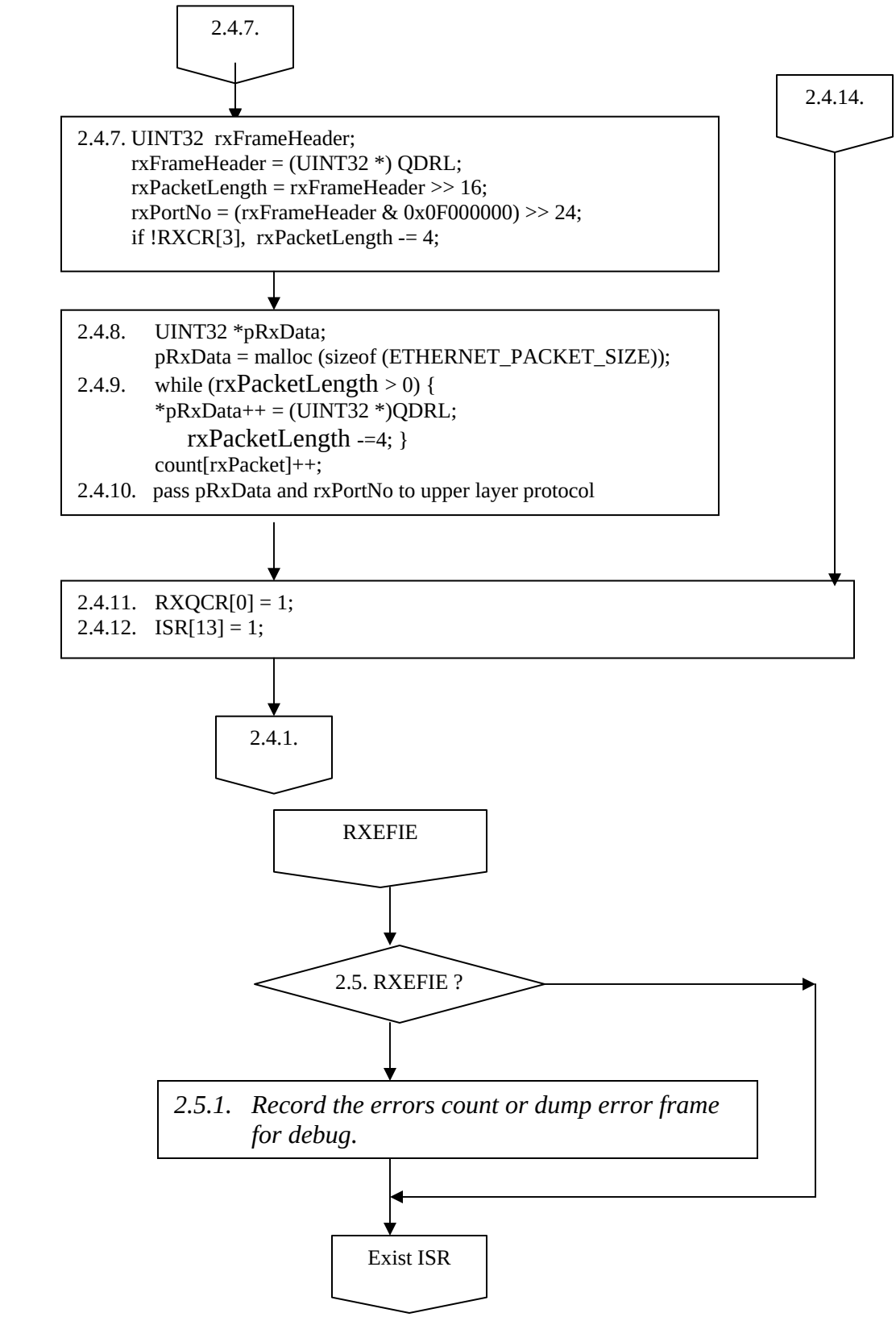
[18] We also provide others platform drivers which receive routines are done at task level.
[19] Indicated the source port where the received packet was received if upper layer protocol requested.

	2.4.3 Checking the status of the current received frame by read RXSR.
	HardwareReceiveStatus(phw, &rxCntl, &rxLength);
	
	2.4.4 If RXSR [15] (Receive Frame Valid) is not set, go to 2.4.14.
	2.4.5 Otherwise, if any of RXSR [4, 2, 1, 0] (Receive errors) is set, go to 2.4.14.
	2.4.6 Received the packet without error, update software driver's receive packet type statistics counter.
	2.4.7 Get Received Packet Length and Received Source Port Number from QDRL QMU Data Register Low (read 4-byte).
	rxPacketLength = "Byte Count", and subtract by 4-byte of CRC if RXCR[3] is not set.
	rxPortNo = RXSPN;
	
	rxPacketLength =HardwareReceiveLength(phw);
	rxPortNo = phw->m_bPortRX;
	
	2.4.8 Allocate a system memory space (address by pRxData) which big enough to hold a Ethernet packet.
	2.4.9 Copy the received Ethernet packet from RXQ to system memory space by read DWORD (32-bit) from QDRL QMU Data Register Low once a time until finished entire frame (count by rxPacketLength) .
	Update the software driver receive packet statistics counter.
	
	HardwareReceiveBuffer(phw, & pPacketData rxPacketLength);
	
	2.4.10 Pass received Ethernet packet data pointer by pPacketData to the upper layer protocol along with Receive Source Port information [19]("rxPortNo") to process.
	2.4.11 Release the RXQ memory by set RXQCR[0] to 1.
	
	HardwareReleaseReceive(phw);
	
	2.4.12 Clear Interrupt Status Register by set ISR [13] to 1.
	
	HardwareAcknowledgeReceive(phw);
	
	2.4.13 Go to 2.4.1. Return.
	2.4.14 Updates the software driver receive packet error type statistics counters.
	2.4.15 Release the RXQ memory by set RXQCR[0] to 1.
	2.4.16 Clear Interrupt Status Register by set ISR [13] to 1.
		hardware.c / HardwareReceiveStatus ();
		
		hardware.c / HardwareReceiveLength ();
		
		hardware.c / HardwareReceiveBuffer ();
		
		hardware.c / HardwareReleaseReceive ();

2.5	If ISR[7] bit is set, 2.5.1. Record the error count for debug. 2.5.2. Clear Interrupt Status Register by set ISR [7] to 1.	ethernet_ks884x.c / ks884xIntr ()

Table 6-1. ksReceiveFrame() Receive Routine


2.1. Checking received Ethernet packets by polling mode.
2.2. ISR
2.2.1. intStatus = ISR.
2.3. ISR[14] ?
2.3. Do same thing as 1.7
2.4. ISR[13] ?
NO
RXEFI F
YES
2.4.1.
2.4.1. Checking if QMU have put the Receive Packet Data in the RXQ
rxPacketDataAvailable = RXMIR[12 – 0];
2.4.2. rxPacketDataAvailable > 0
NO
ISR[13] = 1
2.4.3. rxStats = RXSR.
2.4.4. RXCV ?
NO
count[rxError]++;
2.4.5. RXMR or RXTL or RXRF or RXCE ?
YES
2.1.14
2.4.6. if RXBF count[rxBroadcast]++;
if RXMF count[rxMulticast]++;
if RXUF count[rxUnicast]++;


2.4.7.
2.4.14.
2.4.7. UINT32 rxFrameHeader;
rxFrameHeader = (UINT32 *) QDRL;
rxPacketLength = rxFrameHeader >> 16;
rxPortNo = (rxFrameHeader & 0x0F000000) >> 24;
if !RXCR[3], rxPacketLength -= 4;
2.4.8. UINT32 *pRxData;
pRxData = malloc (sizeof (ETHERNET_PACKET_SIZE));
2.4.9. while (rxPacketLength > 0) {
*pRxData++ = (UINT32 *)QDRL;
rxPacketLength -=4; }
count[rxPacket]++;
2.4.10. pass pRxData and rxPortNo to upper layer protocol
2.4.11. RXQCR[0] = 1;
2.4.12. ISR[13] = 1;
2.4.1.
RXEFIE
2.5. RXEFIE ?
2.5.1. Record the errors count or dump error frame for debug.
Exist ISR

6.2 KSZ88xxP PCI Bus Interface Receive Routine

The PCI version of receive routine uses lists of receive descriptors to receive packets from the device receive MAC DMA.

There are only a few steps to receive an Ethernet packet from KSZ88xxP device to the upper layer when the interrupt serve routine detect a received frame interrupt on the PCI bus interface.

1) While loop in the receive descriptors list ring to do following steps until the receive descriptor's OWN bit is owned by the device (RDES0[31]).

2) Checks the received frame status by read the receive descriptor's register RDES0[28,27,26,25,19,18,17,16].

3) If it is a good frame, get the received frame length by read the receive descriptor's register RDES0[10-0].

4) Get the received frame data buffer pointer (a system memory buffer that allocated from initialize receive descriptors list) from the receive descriptor's register RDSE2[31-0] that the device receive MAC DMA already put the receive data into it. And pass this buffer pointer to the upper layer protocol.

5) Allocate a new system memory data buffer, and replace this buffer pointer to the receive descriptor's register RDSE2[31-0].

6) Return this receive descriptor to the device receive MAC DMA by set RDSE0[31] OWN bit to 1.

The sample code of receive routine for the KSZ88xxP under Linux driver is dev_rcv_packets(dev);

The following table describes the detail steps to receive a data frame from the KSZ88xxP device:

	Descriptions	Driver API
2.1	Checking if received Ethernet packets by interrupt driven ks884xIntr (). 2.1.1. IntEnable = read Interrupt Status Register - ISR. HardwareReadInterrupt(phw, & IntEnable); 2.1.2. If INT_RX (INTST[29] is set, call dev_rcv_packets()	interrupt.c / ks8842p_dev_interrupt () hardware.c / HardwareReadInterrupt ( )
2.2	While loop for only the full ring of receive descriptors list int cnLeft = pInfo->cnAlloc;	interrupt.c / dev_rcv_packets()

	iNext = pInfo->iNext; while (cnLeft--)	
2.3	2.3.1. Start from the previous last receive descriptor, indicate by "pInfo->iNext", check the next receive descriptor if its OWN bit RDES0[31] is owned by the host by call GetReceivedPacket(pInfo, iNext, pDesc, status.ulData); 2.3.2. If this receive descriptor is not owned by the host, break from while loop 2.2.	interrupt.c / dev_rcv_packets() hardware.h/ GetReceivePacket()

2.4	2.4.1. If the buffer pointed by this receive descriptor is the last buffer of the frame by RDES0[29].
	
	if (status.rx.fLastDesc)
	
	2.4.2. Check the received frame, if it is a good frame,
	
	if (!status.rx.fError)
		interrupt.c / dev_rcv_packets()

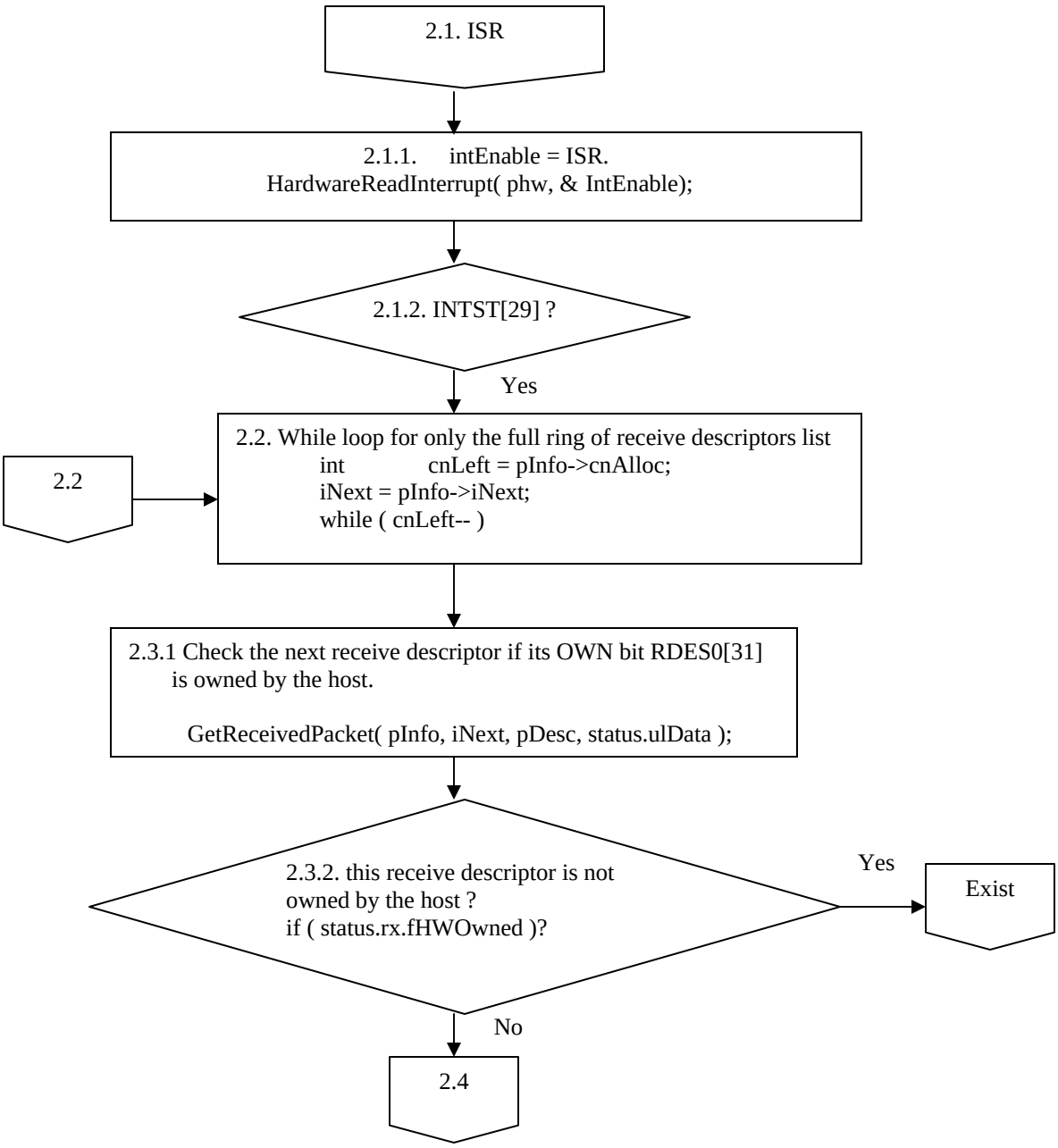
2.5	Get the received frame length from RDES0[10-0], subtract 4-byte CRC packet_len = status.rx.wFrameLen - 4;	interrupt.c / dev_rcv_packets()

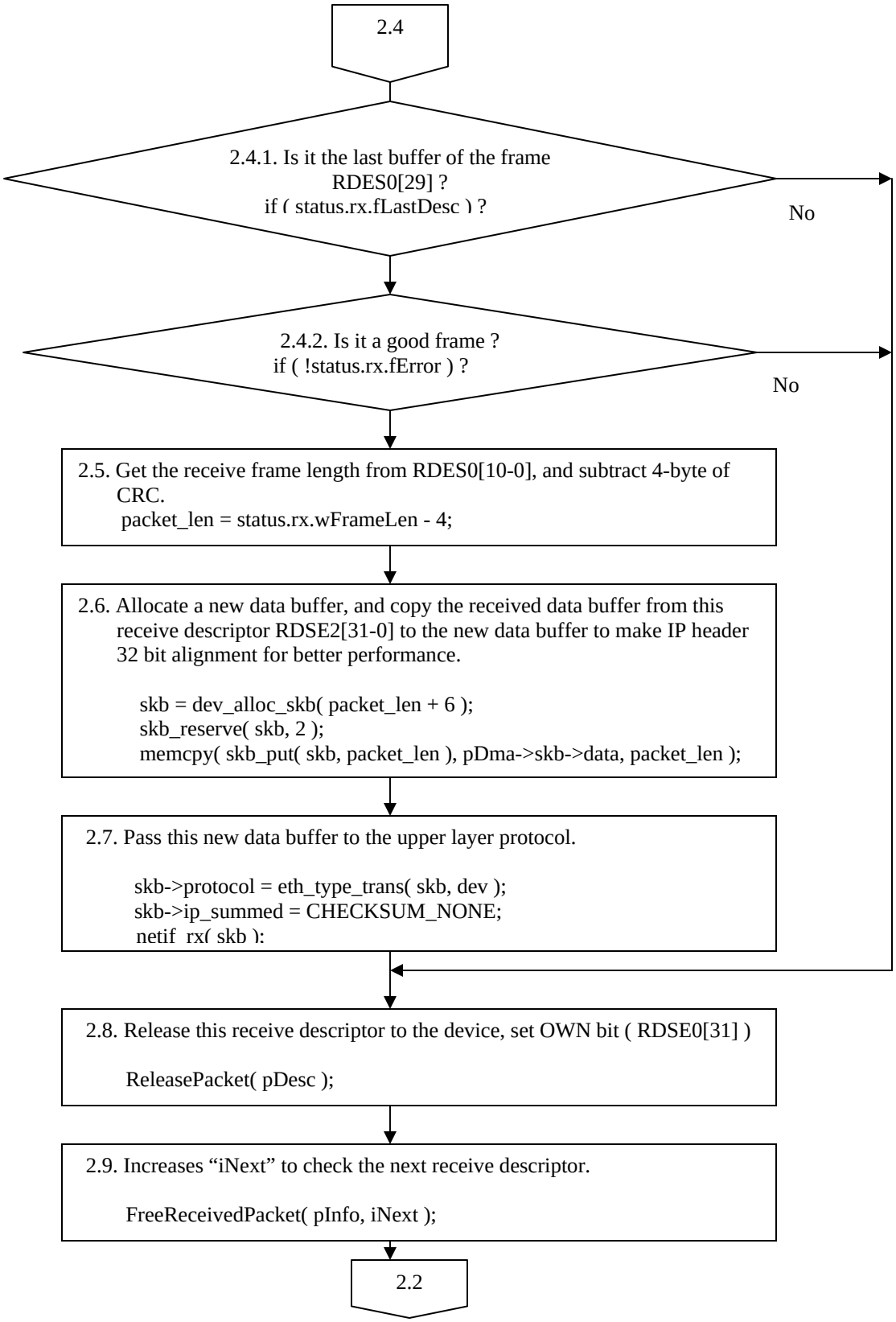
2.6	Allocate a new data buffer, and copy the received data buffer from this receive descriptor RDSE2[31-0] to the new data buffer to make IP header 32 bit alignment for better performance.
	
	skb = dev_alloc_skb(packet_len + 6);
	skb_reserve(skb, 2);
	memcpy(skb_put(skb, packet_len), pDma->skb->data, packet_len);
		interrupt.c / dev_rcv_packets()

2.7	Pass this new data buffer to the upper layer protocol.
	
	skb->protocol = eth_type_trans(skb, dev);
	skb->ip_summed = CHECKSUM_NONE;
	netif_rx(skb);
		interrupt.c / dev_rcv_packets()

2.8	Release this receive descriptor to the device, set OWN bit (RDSE0[31]) by call, ReleasePacket(pDesc);	interrupt.c / dev_rcv_packets() hardware.h/ ReleasePacket ()
2.9	Increases "iNext" to check the next receive descriptor by call, FreeReceivedPacket(pInfo, iNext);	interrupt.c / dev_rcv_packets() hardware.h/ ReleasePacket ()

Table 6-2. dev_rcv_packets () Receive Routine


2.1. ISR
2.1.1. intEnable = ISR.
HardwareReadInterrupt(phw, & IntEnable);
2.1.2. INTST[29] ?
Yes
2.2
2.2. While loop for only the full ring of receive descriptors list
int cnLeft = pInfo->cnAlloc;
iNext = pInfo->iNext;
while (cnLeft--)
2.3.1 Check the next receive descriptor if its OWN bit RDES0[31]
is owned by the host.
GetReceivedPacket(pInfo, iNext, pDesc, status.ulData);
2.3.2. this receive descriptor is not
owned by the host ?
if (status.rx.fHWOwned)?
Yes
Exist
No
2.4


2.4
2.4.1. Is it the last buffer of the frame
RDES0[29] ?
if (status.rx.fLastDesc) ?
No
2.4.2. Is it a good frame ?
if (!status.rx.fError) ?
No
2.5. Get the receive frame length from RDES0[10-0], and subtract 4-byte of CRC.
packet_len = status.rx.wFrameLen - 4;
2.6. Allocate a new data buffer, and copy the received data buffer from this receive descriptor RDSE2[31-0] to the new data buffer to make IP header 32 bit alignment for better performance.
skb = dev_alloc_skb(packet_len + 6);
skb_reserve(skb, 2);
memcpy(skb_put(skb, packet_len), pDma->skb->data, packet_len);
2.7. Pass this new data buffer to the upper layer protocol.
skb->protocol = eth_type_trans(skb, dev);
skb->ip_summed = CHECKSUM_NONE;
netif_rx(skb);
2.8. Release this receive descriptor to the device, set OWN bit (RDSE0[31])
ReleasePacket(pDesc);
2.9. Increases “iNext” to check the next receive descriptor.
FreeReceivedPacket(pInfo, iNext);
2.2

7 KSZ88xx Driver API Reference

The KS88xx driver contains a rich set of API functions that are used to configure all the KSZ88xx device features.

Functions within the API are organized within logical groups and alphabetized within each group. A description of each API group is shown in Table 7.

Device Accesses APIs	Provides functions and macros to read /write device registers.
Device Initialization APIs	For device initializations.
Device Interrupt APIs	To handle the device interrupts.
Device Transmit APIs	For target host CPU transmits the packet from host system memory to device.
Device Receive APIs	For target host CPU receives the packet from device to the host system memory.
Set Device PHY APIs	Provides functions to configure device PHY.
Set Device Ports APIs	Provides functions to configure device port function.
Set Device LinkMD APIs	Provides functions to configure device LinkMD.
Set Wake-on-LAN APIs [20]	Provides functions to configure device Wake-on-LAN function.
Set Device STP APIs [21]	Provides functions to set Spanning Tree states on the device.
Set Device VLAN APIs [21]	Provides functions to configure VLAN function on the device.
Set Device Rate Limiting APIs [21]	Provides functions to configure broadcast storm protection and rate limiting control on the device.
Set Device QoS APIs [21]	Provides functions to configure QoS function on the device.
Set Device Mirror APIs [21]	Provides functions to configure Port Mirroring function on the device.
Device Table Accesses APIs [21]	Provides functions to read/write device indirect registers.

Table 7. KS88xx Driver API Groups

7.1 Device Accesses APIs

[20] This function is only available on KS8841/KSZ8861 device.
[21] This function is only available on KS8842/KSZ8862 device.

This section describes functions and macros that access the KSZ88xx device registers. These APIs are listed in the Table 7-1.

HardwareReadBuffer	Reads the packet data from the device QMU RXQ to the target system memory.
HardwareReadRegByte	Reads a BYTE from the specified bank and register.
HardwareReadRegDWord	Reads a LONG (32-bit) from the specified bank and register.
HardwareReadRegWord	Reads a WORD (16-bit) from the specified bank and register.
HardwareSelectBank	Changes the bank of registers.
HardwareWriteRegByte	Writes a BYTE to the specified bank and register.
HardwareWriteRegDWord	Writes a LONG (32-bit) to the specified bank and register.
HardwareWriteRegWord	Writes a WORD (16-bit) to the specified bank and register.
HW_READ_BYTE	Reads a BYTE from the specified register at current bank if device is generic bus interface.
HW_READ_DWORD	Reads a LONG (32-bit) from the specified register at current bank if device is generic bus interface.
HW_READ_WORD	Reads a WORD (16-bit) from the specified register at current bank if device is generic bus interface.
HW_WRITE_BYTE	Writes a BYTE to the specified register at current bank if device is generic bus interface.
HW_WRITE_DWORD	Writes a LONG (32-bit) to the specified register at current bank if device is generic bus interface.
HW_WRITE_WORD	Writes a WORD (16-bit) to the specified register at current bank if device is generic bus interface.

Table 7-1. Device Accesses APIs

Synopsis

```
void HardwareReadBuffer
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bOffset,    /* The register offset */
    PULONG    pdwData,    /* Pointer to a buffer to store the packet
                             data */
    int       length      /* The length of the buffer to read */
)
```

Description

This routine is used to read the LONG (32-bit) of packet data from the device QMU RXQ to the target system memory pointer by ‘pdwData’ up to the ‘length’ bytes.

Return

None.

Synopsis

```
void HardwareReadRegByte
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    PUCHAR    pbData      /* Pointer to BYTE to store the data */
)
```

Description

This routine reads a BYTE from specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
void HardwareReadRegDWord
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    PULONG    pwData      /* Pointer to LONG to store the data */
)
```

Description

This routine reads a LONG (32-bit) from specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
void HardwareReadRegWord
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    PULONG    pwData      /* Pointer to WORD to store the data */
)
```

Description

This routine reads a WORD (16-bit) from specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
void HardwareSelectBank
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank       /* The new bank of registers */
)
```

Description

This routine changes the bank of registers and keeps track of current bank.

Return

None.

Synopsis

```
void HardwareWriteRegByte
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    UCHAR     bValue      /* The data value */
)
```

Description

This routine writes a 'bValue' BYTE to a specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
void HardwareReadRegDWord
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    ULONG     dwValue     /* The data value */
)
```

Description

This routine writes a ‘dwValue’ LONG (32-bit) to a specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
void HardwareReadRegWord
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bBank,      /* The bank of registers */
    UCHAR     bOffset,    /* The register offset */
    ULONG     wValue      /* The data value */
)
```

Description

This routine writes a ‘wValue’ WORD (16-bit) to a specified bank and register. It calls HardwareSelectBank if the bank is different than the current bank.

Return

None.

Synopsis

```
#define HW_READ_BYTE( phwi, addr, data ) \
*data = *((volatile UINT8 *)(( phwi )->m_ulVIoAddr + addr ))
```

Description

This macro reads a BYTE to ‘data’ from a specified register ‘addr’.

Note: It reads at current bank if device is generic bus interface.

Synopsis

```
#define HW_READ_DWORD( phwi, addr, data ) \
*data = *((volatile UINT32 *)(( phwi )->m_ulVIoAddr + addr ))
```

Description

This macro reads a LONG (32-bit) to ‘data’ from a specified register ‘addr’.

Note: It reads at current bank if device is generic bus interface.

Synopsis

```
#define HW_READ_WORD( phwi, addr, data ) \
*data = *((volatile UINT16 *)(( phwi )->m_ulVIoAddr + addr ))
```

Description

This macro reads a WORD (16-bit) to ‘data’ from a specified register ‘addr’.

Note: It reads at current bank if device is generic bus interface.

Synopsis

```
#define HW_WRITE_BYTE( phwi, addr, data ) \
*((volatile UINT8 *)(( phwi )->m_ulVIoAddr + addr )) = (UINT8)( data )
```

Description

This macro writes a BYTE ‘data’ to a specified register ‘addr’.

Note: It writes at current bank if device is generic bus interface.

Synopsis

```
#define HW_WRITE_DWORD( phwi, addr, data ) \
*((volatile UINT32 *)(( phwi )->m_ulVIoAddr + addr )) = (UINT32)(data )
```

Description

This macro writes a DOUBLE WORD ‘data’ to a specified register ‘addr’.

Note: It writes at current bank if device is generic bus interface.

Synopsis

```
#define HW_WRITE_WORD( phwi, addr, data ) \
*((volatile UINT16 *)(( phwi )->m_ulVIoAddr + addr )) = (UINT16)(data )
```

Description

This macro writes a WORD ‘data’ to a specified register ‘addr’.

Note: It writes at current bank if device is generic bus interface.

7.2 Device Initialization APIs

This section describes functions that driver initialize the KSZ88xx device. These APIs are listed in the Table 7-2.

HardwareDisable	Disable the device QMU transmit/receive engine.
HardwareEnable	Enable the device QMU transmit/receive engine.
HardwareInitialize	Verify KSZ88xx device ID.
HardwareReadAddress	Retrieves the device station MAC address.
HardwareReset	Software reset the device.
HardwareSetAddress	Set the device station MAC address.
HardwareSetup	Setup the device for the default proper operation.
SwitchEnable [22]	Enable/Disable the device Switch engine.

Table 7-2. Device Initialization APIs

[22] Only valid for KS8842/KSZ8862 device.

Synopsis

```
void HardwareDisable
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine disables the device QMU transmit/receive engine.

Return

None.

Synopsis

```
void HardwareEnable
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine enables the device QMU transmit/receive engine.

Note: Call HardwareSetup to setup the device default proper operation
       before call HardwareEnable.

Return

None.

Synopsis

```
BOOLEAN HardwareInitialize
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine checks if the device ID is correct for this driver
(KSZ8841 or KSZ8842).

Return

TRUE – device ID is correct for this driver.
FALSE – device ID is incorrect for this driver.

Synopsis

```
BOOLEAN HardwareReadAddress
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine retrieves the MAC address of the device station, and stores it in pHardware->m_bPermanentAddress.

Return

TRUE – successful.

Synopsis

```
BOOLEAN HardwareReset
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine performs the device software reset.

Return

TRUE – successful.

Synopsis

```
void HardwareSetAddress
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

This routine programs the MAC address of the device when the MAC address is stored in pHardware->m_bOverrideAddress.

Return

None.

Synopsis

```
void HardwareSetup
(
    PHARDWARE pHardware   /* Pointer to hardware instance. */
)
```

Description

- This routine sets the device for proper operation, including default setting for device transmit/receive control, port control, and PHY link speed (auto-negotiation).

Return

None.

Synopsis

```
void SwitchEnable
(
    PHARDWARE pHardware,   /* Pointer to hardware instance. */
    BOOLEAN   fEnable      /* 1 - enable switch; 0 – disable switch */
)
```

Description

This routine is used to enable/disable Switch Engine.
Note: Only KSZ8842 device has switch function.

Return

None.

7.3 Device Interrupt APIs

This section describes functions set and acknowledge the KSZ88xx device interrupts. These APIs are listed in the Table 7-3.

HardwareAcknowledgeInterrupt	Acknowledges the specified device interrupts.
HardwareBlockInterrupt	Blocks all device interrupts.
HardwareDisableInterrupt	Disables the device interrupt.
HardwareDisableInterruptBit	Disables the device specified interrupt bits.
HardwareEnableInterrupt	Enables the device interrupt.
HardwareEnableInterruptBit	Enables the device specified interrupt bits.
HardwareReadInterrupt	Reads the current device interrupt mask.
HardwareSetInterrupt	Reset the device interrupt mask, and enable them.
HardwareSetupInterrupt	Setup the device interrupt mask for proper operation.

Table 7-3. Device Interrupt APIs

Synopsis

```
void HardwareAcknowledgeInterrupt
(
    PHARDWARE pHardware,   /* Pointer to hardware instance. */
    USHORT    wInterrupt   /* The interrupt masks to be acknowledged */
)
```

Description

This routine acknowledges the specified device interrupts.

Return

None.

Synopsis

```
ULONG HardwareBlockInterrupt
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine blocks all interrupts of the device and returns the current interrupt enable mask so that interrupts can be restored later.

Return

The current interrupt enable mask.

Synopsis

```
void HardwareDisableInterrupt
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine disables the device interrupt.

Return

None.

Synopsis

```
void HardwareDisableInterruptBit
(
    PHARDWARE pHardware,   /* Pointer to hardware instance. */
    USHORT    wInterrupt   /* The interrupt masks bit to be disabled */
)
```

Description

This routine disables the device specified interrupt.

Return

None.

Synopsis

```
void HardwareEnableInterrupt
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine enables the device interrupts mask from pHardware->m_wInterruptMask which is previous set by the HardwareSetupInterrupt.

Note: Call HardwareSetupInterrupt to setup the device default interrupts mask before call HardwareEnableInterrupt.

Return
None.

Synopsis

```
void HardwareReadInterrupt
(
    PHARDWARE pHardware,   /* Pointer to hardware instance. */
    PUSHORT   pwStatus   /* Pointer to USHORT to store the interrupt mask */
)
```

Description

This routine reads the current device interrupt mask and stores it in ‘pwStatus’.

Return
None.

Synopsis

```
void HardwareSetInterrupt
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    USHORT    wInterrupt    /* The interrupt mask to enable */
)
```

Description

This routine enables the device interrupt mask by ‘wInterrupt’.

Return

None.

Synopsis

```
void HardwareSetupInterrupt
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine setup the device default interrupt mask to the pHardware->m_wInterruptMask.

Return

None.

7.4 Device Transmit APIs

This section describes functions that driver transmits the packets from host CPU system memory to the KSZ88xx device. These APIs are listed in the Table 7-4.

HardwareAllocPacket	Allocates the device TXQ memory to transmit a packet.
HardwareSendPacket	Device transmits a frame from TXQ.
HardwareSetTransmitLength	Writes the transmit control and frame length to “Control Word” and “Byte Count” in the TXQ frame header.
HardwareTransmitDone	Handles transmit complete interrupt processing.
HardwareWriteBuffer	Writes a transmit frame to the device TXQ.

Table 7-4. Device Transmit APIs

Synopsis

```
int HardwareAllocPacket
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    Int       length        /* The length of the transmit frame */
)
```

Description

This routine allocates the device TXQ memory by the size of ‘length’.

Return

```
1 – Successful,
0 – Fail, Device TXQ does not have enough memory for the size of
     ‘length’.
```

Synopsis

```
BOOLEAN HardwareSendPacket
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine issues a transmit command to the device. The current

```
transmit frame prepared in the TXQ memory is queued for transmit.


Return

TRUE  – Successful,
FALSE – Fail.
```

Synopsis

```
ULONG HardwareSetTransmitLength
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    int       length        /* The length of the packet. */
)
```

Description

This routine writes the transmit control word and frame length 'length' to "Control Word", and "Byte Count" in the TXQ frame header of the TXQ.

The transmit control word are consisted of 'pHardware->m_bPortTX' and 'pHardware->m_bTransmitPacket'.

'pHardware->m_bTransmitPacket' is for Transmit frame ID field in the "Control Word".

'pHardware->m_bPortTX' is for Transmit Destination Port Number.
If pHardware->m_bPortTX is 0, Transmit packet by loopkup mode,
If pHardware->m_bPortTX is 1, Transmit packet to Port1,
If pHardware->m_bPortTX is 2, Transmit packet to Port2,
If pHardware->m_bPortTX is 3, Transmit packet to Port1 and Port2.

Note: Call this routine before copying the Ethernet frame to the TXQ.

Return

4-byte of ""Control Word", and "Byte Count"

Synopsis

```
BOOLEAN HardwareTransmitDone
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine handles transmit done interrupt processing.

Return

TRUE if packet is sent successful; otherwise, FALSE.

Synopsis

```
void HardwareWriteBuffer
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    UCHAR     bOffset,    /* The register offset */
    PULONG    pdwData,    /* Pointer to a transmit data buffer */
    int       length      /* The length of the buffer to write */
)
```

Description

This routine is used to write the LONG (32-bit) of packet data from the target system memory pointer by ‘pdwData’ to the device QMU TXQ up to ‘length’ bytes.

Return

None.

7.5 Device Receive APIs

This section describes functions that driver receives the packets from the KSZ88xx device to host CPU system memory. These APIs are listed in the Table 7-5.

HardwareReceiveBuffer	Reads a received frame from the device RXQ.
HardwareReceiveLength	Gets the length of the received packet.
HardwareReceiveMoreDataAvailable	Gets the total length of received packets in the device RXQ.
HardwareReceiveStatus	Checks the received packet status.
HardwareReleaseReceive	Release the received packet memory in the device RXQ.

Table 7-5. Device Receive APIs

Synopsis

```
void HardwareReceiveBuffer
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    void*     pBuffer,      /* Point to a system memory buffer to store
                               the received packet */
    int       length        /* The length of the received frame */
)
```

Description

This routine reads the LONG (32-bit) of received data from the device RXQ memory by the size of ‘length’ to the system memory pointer by ‘pBuffer’.

Return
None.

Synopsis

```
int HardwareReceiveLength
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine returns the length of the received packet.

Return
The length of the received packet.
Return zero if the received packet is not a good packet.

Synopsis

```
USHORT HardwareReceiveMoreDataAvailable
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine returns the total length of the received packets data in the device RXQ. If the value is not zero, there are more packets data in the RXQ.

Return
The length of the total received packets in the RXQ.

Synopsis

```
BOOLEAN HardwareReceiveStatus
(
    PHARDWARE pHardware, /* Pointer to hardware instance. */
    PUSHORT   pwStatus,  /* Point to USHORT to store received status */
    PUSHORT   pwLength   /* point to USHORT to store received length */
)
```

Description

This routine returns the received packet status and length.

Return

TRUE if received packet is a good packet; otherwise, FALSE.

Synopsis

```
void HardwareReleaseReceive
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
)
```

Description

This routine releases the receive packet memory in the device RXQ.

Return

None.

7.6 Set Device PHY APIs

This section describes functions that driver get or set PHY of the KSZ88xx device ports. These APIs are listed in the Table 7-6.

HardwareGetLinkStatus	Sets the link speed or duplex to the device specific port.
HardwareSetCapabilities	Gets the link speed or duplex from the device specific port.
SwitchGetLinkStatus	Gets the link status of device ports.
SwitchSetLinkSpeed	Sets the link speed or duplex of the device ports.
SwitchRestartAutoNego	Restarts the link auto-negotiation of the device ports.

Table 7-6. Device PHY APIs

Synopsis

```
void HardwareGetLinkStatus
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
    UCHAR     bPhy,        /* The Port index */
    ULONG   * pBuffer      /* Point to a buffer to store the link
                              status */
)
```

Description

This routine is used to get the Link status, Link speed, Link duplex status, Link Capable status, Link auto-negotiation advertisement status, and Link partner capabilities status from a specific port 'bPhy'.
'pBuffer' is a 10 array of ULONG buffer that contains the link status after this routine is called,

```
pBuffer[0] : Link status
pBuffer[1] : Link Speed status
pBuffer[2] : Link Duplex mode status
pBuffer[3] : Link Capable status
pBuffer[4] : Link Auto-Negotiation Advertisement status
pBuffer[5] : Link Partner Capabilities status
pBuffer[6] : Reserved
pBuffer[7] : Reserved
pBuffer[8] : Reserved
pBuffer[9] : Reserved

For the Link status information 'pBuffer[0]':
0  means Link is download
1  means Link is good

For the Link Speed status information 'pBuffer[1]':
1000000  means Link Speed is 100Mbps
100000   means Link Speed is 10Mbps

For the Link Duplex mode status information 'pBuffer[2]':
0x01  means Link Duplex is full duplex
0x02  means cable is crossed
0x04  means is reversed

For the Link Capable status information 'pBuffer[3]':
0x00000001  means 10BaseT full duplex
0x00000002  means 10BaseT half duplex
0x00000004  means 100BaseTX full duplex
0x00000008  means 100BaseTX half duplex
0x00000100  means Link Pause

For the Link Auto-Negotiation Advertisement status information
'pBuffer[4]':
```

```
0x00000001  means 10MBPS full duplex
0x00000002  means 10MBPS half duplex
0x00000004  means 100MBPS full duplex
0x00000008  means 100MBPS half duplex
0x00000100  means Pause frame
0x00010000  means enable Auto MDIX
0x00020000  means Force MDIX
0x00040000  means Auto Polarity

For the Link Partner Capabilities status information ‘pBuffer[5]’:
0x00000001  means 10MBPS full duplex
0x00000002  means 10MBPS half duplex
0x00000004  means 100MBPS full duplex
0x00000008  means 100MBPS half duplex
0x00000100  means Pause frame
```

Return

None.

Synopsis

```
void HardwareSetCapabilities
(
    PHARDWARE pHardware,      /* Pointer to hardware instance. */
    UCHAR     bPhy,           /* The PHY (port) index */
    ULONG     ulCapabilities  /* A set of flags indicating different
                                 capabilities */
)
```

Description

This routine sets the link speed and duplex to a specific PHY port ‘bPhy’ by its capabilities ‘ulCapabilities’.

The PHY port’s auto-negotiation advertisement link speed, duplex, and pause frame are set according to ’ulCapabilities’ value:

```
0x00000001,   10MBPS, full duplex
0x00000002,   10MBPS, half duplex
0x00000004,   100MBPS, full duplex
0x00000008,   100MBPS, half duplex
0x00000100,   Pause frame
0x00010000,   Auto MDIX (Micrel)
0x00020000,   Force MDIX
```

Return

None.

Synopsis

```
void SwitchGetLinkStatus
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine reads PHY registers to determine the current link status Of the device ports, and updates the PHY information to pHardware->m_PortInfo[Port].

Return
None.

Synopsis

```
void SwitchSetLinkSpeed
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine sets the link speed and duplex to the device all ports. The default is auto-negotiation advertised to 100BT/10BT, full/half duplex capability.
If 'pHardware->m_bSpeed' or 'pHardware->m_bDuplex' has been defined before call this routine, then, the link speed and duplex are set according to these values:
If pHardware->m_bSpeed is 100, advertised to 100BT only.
If pHardware->m_bSpeed is 10, advertised to 10BT only.
If pHardware->m_bDuplex' is 1, advertised to full duplex only.
If pHardware->m_bDuplex' is 2, advertised to half duplex only.

Return
None.

Synopsis

```
void SwitchRestartAutoNego
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine restarts auto-negotiation of the device ports.

Return
None.

7.7 Set Device Ports APIs

This section describes functions that driver gets or sets port configuration of the KSZ88xx device ports. These APIs are listed in the Table 7-7.

PortConfigBackPressure	Enable/disable Back Pressure on a specific port.
PortConfigForceFlowCtrl	Enable/disable Force Flow Control on a specific port.
PortGetBackPressure	Gets Back Pressure setting status on a specific port.
PortGetForceFlowCtrl	Gets Force Flow Control setting status on a specific port.

Table 7-7. Device Ports APIs

Synopsis

```
#define PortConfigBackPressure(pHardware, port, enable )
```

Description

This macro ‘enable’ (enables or disables) Back Pressure on a specific port ‘port’.

Return
None.

Synopsis

```
#define PortConfigForceFlowCtrl (pHardware, port, enable )
```

Description

This macro ‘enable’ (enables or disables) Force Flow Control on a specific port ‘port’. If enabled, it is regardless of AN result.

Return
None.

Synopsis

```
#define PortGetBackPressure (pHardware, port )
```

Description

This macro gets Back Pressure setting status on a specific port ‘port’.

Return

TRUE - Back Pressure is enabled.
FALSE - Back Pressure is disabled.

Synopsis

```
#define PortGetForceFlowCtrl (pHardware, port )
```

Description

This macro gets Force Flow Control setting status on a specific port ‘port’.

Return
TRUE - Force Flow Control is enabled.
FALSE - Force Flow Control is disabled.

7.8 Set Device LinkMD™ APIs

This section describes functions that driver uses device LinkMD™ feature to test cable diagnostics capabilities to determine cable length, diagnose faulty cables, and determine distance to fault. These APIs are listed in the Table 7-8.

HardwareGetCableStatus	Gets the cable status through device LinkMD™ feature on a specific port.

Table 7-8. Device LinkMD APIs

Synopsis

```
void HardwareGetCableStatus
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
    UCHAR     bPhy,        /* The Port index */
    ULONG   * pBuffer      /* Point to a buffer to store the cable status
*/
)
```

Description

This routine is used to get the cable status through the device LinkMD™ feature on a specific port ‘bPhy’.
‘pBuffer’ is a 10 array of ULONG buffer that contains the tested cable status after this routine is called,

```
pBuffer[0] : cable length
pBuffer[1] : cable status
pBuffer[2] : cable twister pair 1-2 length
pBuffer[3] : cable twister pair 1-2 status
pBuffer[4] : cable twister pair 3-6 length
pBuffer[5] : cable twister pair 3-6 status
pBuffer[6] : cable twister pair 4-5 length (unavailable)
pBuffer[7] : cable twister pair 4-5 status (unavailable)
pBuffer[8] : cable twister pair 7-9 length (unavailable)
pBuffer[9] : cable twister pair 7-9 status (unavailable)

For the cable status information,
0 means cable is unknown
1 means cable is good
2 means cable is crossed
3 means cable is reversed
4 means cable is crossed and reversed
5 means cable is open
6 means cable is short
```

Return

None.

7.9 Set Wake-on-LAN APIs

This section describes functions that configure device Wake-on-LAN (WOL) event by receipt of a Magic Packet or a network wake up frame. When device detects a wake up event, it asserts the PMEN pin to low which could connect to host system to put the system into a powered state (working state).

User can configure up to four ‘wake up’ frames. The ‘wake up’ frames are certain types of packets with specific CRC values that device recognizes as a ‘wake up’ frame. The specific CRC values are calculated by byte specified (mask) within 64 byte of the ‘wake up’ packet.

These APIs are listed in the Table 7-9.

HardwareClearWolPMEStatus	Clear PME_Status to dessert PMEN pin.
HardwareEnableWolMagicPacket	Enable WOL event caused by receipting of a Magic Packet.
HardwareEnableWolFrame	Enable WOL event caused by receipting of a network ‘wake up’ frame.
HardwareSetWolFrameByteMask	Configure the byte mask within 64 byte of the ‘wake up’ frame pattern to calculate CRC value.
HardwareSetWolFrameCRC	Configure the expected 32bit CRC value of the ‘wake up’ frame pattern.

Table 7-9. Device Wake-on-LAN APIs

Synopsis

```
void HardwareClearWolPMEStatus
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine is used to clear PME_Status to dessert PMEN pin.

Return

None.

Synopsis

```
void HardwareEnableWolMagicPacket
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine is used to enable the Wake-on-LAN that wake-up signal is caused by receipting of a Magic Packet.

KSZ8841 device can support D1, D2, or D3 power state by EEPROM setting. By default, device supports D3 power state without EEPROM setting. The example here is by default D3 power state.

Return

None.

Synopsis

```
void HardwareEnableWolFrame
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    UINT32    dwFrame       /* whose wake up frame to be enabled (0 -
                               Frame 0, 1 - Frame 1, 2 - Frame 2, 3 -
                               Frame 3). */
)
```

Description

This routine is used to enable the Wake-on-LAN that wake-up signal is caused by receipting of a network 'wake-up' packet. The device can support up to four different 'wake-up' frames.

KSZ8841 device can support D1, D2, or D3 power state by EEPROM setting. By default, device supports D3 power state without EEPROM setting. The example here is by default D3 power state.

Note: Call HardwareSetWolFrameByteMask and HardwareSetWolFrameCRC to setup the device proper operation before call HardwareEnableWolFrame.

Return

None.

Synopsis

```
void HardwareSetWolFrameByteMask
(
    PHARDWARE pHardware    /* Pointer to hardware instance. */
    UINT32    dwFrame,     /* whose wake up frame to be enabled (0 -
                              Frame 0, 1 - Frame 1, 2 - Frame 2, 3 -
                              Frame 3). */
    UINT8     bByteMask    /* byte number to mask to calculate CRC
                              value (0 - byte 0, 63 - byte 63). */
)
```

Description

This routine is used set the byte mask within 64 byte of the 'Wake up' frame pattern to calculate CRC value.
The device can support up to four different 'wake-up' frames.

Return

None.

Synopsis

```
void HardwareSetWolFrameCRC
(
    PHARDWARE pHardware,   /* Pointer to hardware instance. */
    UINT32    dwFrame,     /* whose wake up frame to be enabled (0 -
                              Frame 0, 1 - Frame 1, 2 - Frame 2, 3 -
                              Frame 3). */
    UINT32    dwCRC        /* Expected 32bit CRC value. */
)
```

Description

This routine is used set expected 32-bit CRC value of the 'Wake up' frame pattern.
The device can support up to four different 'wake-up' frames.

Return

None.

7.10 Set Device STP APIs

This section describes functions that configure device Spanning Tree function by Spanning Tree states. These APIs are listed in the Table 7-10.

HardwareInit_STP	Initialize the STP support on the device.
PortSet_STP_State	Configures the STP state on a specific port.

Table 7-10. Device STP APIs

Synopsis

```
void HardwareInit_STP
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
)
```

Description

This routine initializes the spanning tree support on the device by write STP multicast address “01:80:C2:00:00:00” to device static MAC table and set Forward port to host CPU port, Override is TRUE.

Return

None.

Synopsis

```
void PortSet_STP_State
(
    PHARDWARE pHardware,      /* Pointer to hardware instance. */
    UCHAR     bPort,          /* The port index */
    int       nState          /* The spanning tree state */
)
```

Description

This routine configures the STP state ‘nState’ on a specific port ‘bPort’.
The ‘nState’ can be Blocking state, Disable state, Listening state, Learning state, and Forwarding state.

Return
None.

7.11 Set Device VLAN APIs

This section describes functions that configure device VLAN function. These APIs are listed in the Table 7-11.

HardwareConfigDefaultVID	Configures the device default VLAN id.
HardwareConfigPortBaseVlan	Configures the device port-base VLAN membership.
PortGetDefaultVID	Retrieves the device default VLAN id.
SwitchDisableVlan	Disables the device VLAN function.
SwitchEnableVlan	Enables the device VLAN function.
SwitchInitVlan	Sets the device for the proper VLAN parameters. [23]
SwitchVlanConfigDiscardNonVID	Configures the device Discard Non PVID packets function.
SwitchVlanConfigIngressFiltering	Configures the device Ingress VLAN filtering function.
SwitchVlanConfigInsertTag	Configures the device 802.1q Tag insertion function
SwitchVlanConfigRemoveTag	Configures the device 802.1q Tag removal function.

Table 7-11. Device VLAN APIs

[23] This functions must been called before other VLAN APIs.

Synopsis

```
void HardwareConfigDefaultVID
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
    UCHAR     bPort,         /* the port index */
    USHORT    wVID           /* VLAN id value */
)
```

Description

This routine configures the port-base default VLAN ID ‘wVID’ to a specific port ‘bPort’.

Return
None.

Synopsis

```
void HardwareConfigPortBaseVlan
(
    PHARDWARE pHardware      /* Pointer to hardware instance. */
    UCHAR     bPort,         /* The port index */
    UCHAR     bMember        /* The port-base VLAN membership */
)
```

Description

This routine configures the port-base VLAN membership ‘bMember’ to a specific port ‘bPort’.

Return
None.

Synopsis

```
void PortGetDefaultVID
(
    PHARDWARE pHardware,      /* Pointer to hardware instance. */
    UCHAR     bPort,          /* the port index */
    USHORT    pwVID           /* Pointer to USHORT to store VLAN id */
)
```

Description

This routine retrieves the port-base default VLAN ID from a specific port ‘bPort’.

Return
None.

Synopsis

```
void SwitchDisableVlan
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
)
```

Description

This routine disables the device 802.1q VLAN mode.

Return

None.

Synopsis

```
void SwitchEnableVlan
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
)
```

Description

This routine enables all the device VLAN functions to all the ports, including

- Create VLAN entry for port-base VLAN id in the VLAN table.
- Enables Unicast port-VLAN mismatch discard (all packets can not cross VLAN boundary)(SGCR2), the device will discard packets whose incoming port and outgoing port is not in the Port VLAN Membership (PnCR2[24]).
- The last step enables the device 802.1q VLAN mode (SGCR2).

Return

None.

[24] n is 1, 2, 3.

Synopsis

```
void SwitchInitVlan
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
)
```

Description

This routine initializes the proper VLAN parameters, like port-base default VLAN id, and port-base Port VLAN Membership to the default value.

Return

None.

Synopsis

```
void SwitchVlanConfigDiscardNonVID
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
    UCHAR     bPort,          /* the port index */
    BOOLEAN   fSet            /* TRUE, enable; FALSE, disable. */
)
```

Description

This routine configures the Discard Non PVID packets on a specific port ‘bPort.
If enabled, the device will discard packets whose VLAN id does not match ingress port-base default VLAN.

Return

None.

Synopsis

```
void SwitchVlanConfigIngressFiltering
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
    UCHAR     bPort,          /* the port index */
    BOOLEAN   fSet            /* TRUE, enable; FALSE, disable. */
)
```

Description

This routine configures the Ingress VLAN filtering function on a specific port ‘bPort.
If enabled, the device will discard packets whose VLAN id membership in the VLAN table bits [18:16] does not include the ingress port that received this packet.

Return

None.

Synopsis

```
void SwitchVlanConfigInsertTag
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
    UCHAR     bPort,          /* the port index */
    BOOLEAN   fSet            /* TRUE, enable; FALSE, disable. */
)
```

Description

This routine configures the 802.1q Tag insertion function on a specific port ‘bPort.
If enabled, the device will insert 802.1q tag to the transmit packet if received packet is an untagged packet. The device will not insert 802.1q tag if received packet is tagged packet.

Return

None.

Synopsis

```
void SwitchVlanConfigRemoveTag
(
    PHARDWARE pHardware       /* Pointer to hardware instance. */
    UCHAR     bPort,          /* the port index */
    BOOLEAN   fSet            /* TRUE, enable; FALSE, disable. */
)
```

Description

This routine configures the 802.1q Tag removal function on a specific port ‘bPort.
If enabled, the device will removed 802.1q tag from the transmit packet if received packet is a tagged packet. The device will not remove 802.1q tag if received packet is untagged packet.

Return

None.

7.12 Set Device Rate Limiting APIs

This section describes functions that configure the device rate limiting at the ingress and egress ports including broadcast storm protection. These APIs are listed in the Table 7-12.

HardwareConfigBroadcastStorm	Configures the device broadcast storm threshold.
HardwareConfigRxPriorityRate	Configures the device ingress rate limiting at a specific priority frame on a specific port.
HardwareConfigTxPriorityRate	Configures the device egress rate limiting at a specific priority frame on a specific port.
SwitchDisableBroadcastStorm	Disables the device broadcast storm protection on a specific port.
SwitchDisablePriorityRate	Disables the device ingress and egress rate limiting at all priority frames on a specific port.
SwitchEnableBroadcastStorm	Enables the device broadcast storm protection on a specific port.
SwitchEnablePriorityRate	Enables the device ingress and egress rate limiting for all the priority frames on a specific port.
SwitchInitBroadcastStorm	Initializes the device broadcast storm threshold at 1 percent of line rate, and disables the broadcast storm protection on all the ports.
SwitchInitPriorityRate	Initializes the device ingress and egress with no rate limiting to all priority frames, and disables the device ingress and egress rate limiting at all priority frames on all the ports.

Table 7-12. Device Rate Limiting APIs

Synopsis

```
void HardwareConfigBroadcastStorm
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    ULONG     percent          /* The Broadcast storm threshold in
                                  percent of transmit rate */
)
```

Description

This routine configures the broadcast storm protection rate at 'percent' percent of line rate.

For example, to set broadcast storm protection 1% of line rate, the calculation is

148,800 frames/sec * 67 ms/interval X 1% = 99 frames/interval (approx.) = 0x63

Then, set 0x6300 to SGCR3.

Return

None.

Synopsis

```
void HardwareConfigRxPriorityRate
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort,           /* The port index. */
    UCHAR     bPriority,       /* The priority index to configure */
    ULONG     dwRate           /* The rate limit in number of Kbps */
)
```

Description

This routine configures the Ingress data rate limit in the 'dwRate' unit of Kbps for specific priority 'bPriority' frame to the 'bPort' port.

For example, to set data rate 512 Kbps for priority 2 frames to the port 1, call:

HardwareConfigRxPriorityRate (pHardware, 0, 2, 512);

This routine will configure 0x0400 to P1IRCR.

Return

None.

Synopsis

```
void HardwareConfigTxPriorityRate
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort,           /* The port index. */
    UCHAR     bPriority,       /* The priority index to configure */
    ULONG     dwRate           /* The rate limit in number of Kbps */
)
```

Description

This routine configures the Egress data rate limit in the ‘dwRate’ unit of Kbps for specific priority ‘bPriority’ frame to the ‘bPort’ port.

For example, to set data rate 2 Mbps for priority 0 frames to the port 2, call:
HardwareConfigTxPriorityRate (pHardware, 1, 0, 2048);

This routine will configure 0x0006 to P2ERCR.

Return

None.

Synopsis

```
void SwitchDisableBroadcastStorm
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort            /* The port index. */
)
```

Description

This routine disables the broadcast storm protection function on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisablePriorityRate
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort            /* The port index. */
)
```

Description

This routine disables the Ingress and Egress rate limiting control to all the priority frames on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchEnableBroadcastStorm
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort            /* The port index. */
)
```

Description

This routine enables the broadcast storm protection function on the ‘bPort’ port.

Note: This routine also sets “Broadcast Storm Protection” that does not include multicast packets.

Return

None.

Synopsis

```
void SwitchEnablePriorityRate
(
    PHARDWARE pHardware,       /* Pointer to hardware instance. */
    UCHAR     bPort            /* The port index. */
)
```

Description

This routine enables the Ingress and Egress rate limiting control to all the priority frames on the ‘bPort’ port.

Note: Call HardwareConfigRxPriorityRate to setup Egress rate limiting, and HardwareConfigTxPriorityRate to setup Ingress rate limiting before call SwitchEnablePriorityRate.

Return
None.

Synopsis

```
void SwitchInitBroadcastStorm
(
    PHARDWARE pHardware        /* Pointer to hardware instance. */
)
```

Description

This routine initializes the device broadcast storm threshold at 1 percent of line rate, and disables the broadcast storm protection on all the ports

Return
None.

Synopsis

```
void SwitchInitPriorityRate
(
    PHARDWARE pHardware        /* Pointer to hardware instance. */
)
```

Description

This routine initializes the ingress and egress with no rate limiting to all priority frames, and disables the device ingress and egress rate limiting at all priority frames on all the ports.

Return
None.

7.13 Set Device QoS APIs

This section describes functions that configure the device QoS priority support, including Port-Based priority, 802.1p based priority, and DiffServ based priority. These APIs are listed in the Table 7-13.

HardwareConfig_TOS_Priority	Configures the device TOS of DiffServ based priority.
HardwareConfig802_1P_Priority	Configures the device 802.1p based priority .
SwitchConfigPortBased	Configures the device Port based priority on a specific port.
SwitchDisable802_1P	Disables the device 802.1p priority of QoS on a specific port.
SwitchDisableDiffServ	Disables the device DiffServ priority of QoS on a specific port.
SwitchDisableDot1pRemapping	Disables the device 802.1p priority re-mapping function on a specific port.
SwitchDisableMultiQueue	Disables the device transmit multiple queues selection on a specific port.
SwitchEnable802_1P	Enables the device 802.1p priority of QoS on a specific port.
SwitchEnableDot1pRemapping	Enables the device 802.1p priority re-mapping function on a specific port.
SwitchEnableDiffServ	Enables the device DiffServ priority of QoS on a specific port.
SwitchEnableMultiQueue	Enables the device transmit multiple queues selection on a specific port.
SwitchInitPriority	Disables all the QoS (Port based, 802.1p based, DiffServ based) functions.

Table 7-13. Device QoS APIs

Synopsis

```
void HardwareConfig_TOS_Priority
(
    PHARDWARE pHardware,    /* Pointer to hardware instance. */
    UCHAR     bTosValue,    /* ToS value from 6-bit (bit7 ~ bit2) of ToS
                               field, range from 0 to 63. */
    USHORT    wPriority     /* Priority to be assigned */
)
```

Description

This routine configures the TOS of DiffServ based priority.
The ‘bTosValue’ ToS value (from bit7 ~ bit2 of ToS field, range from 0 to 63) is mapped to transmit priority queue ‘wPriority’ (queue 0 to queue 3).

Ingress packets with be classified to which transmit priority queue according to the 6-bit of TOS value if DiffServ priority of QoS is enabled.

Return

None.

Synopsis

```
void HardwareConfig802_1P_Priority
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bTagPriorityValue,  /* The 802.1p tag priority value, range
                                     from 0 to 7 */
    USHORT    wPriority           /* Priority to be assigned */
)
```

Description

This routine configures the 802.1p based priority.
The ‘bTagPriorityValue’ of 802.1p tag priority value (range from 0 to 7) is mapped to transmit priority queue ‘wPriority’ (queue 0 to queue 3).

Ingress packets with be classified to which transmit priority queue according to the 3-bit of 802.1p Tag priority value if 802.1p priority of QoS is enabled.

Return

None.

Synopsis

```
void SwitchConfigPortBased
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPriority,         /* Priority to be assigned */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine configures the Port based priority on the ‘bPort’ port. Assign this port to transmit priority queue ‘bPriority’ (queue 0 to queue 3).

All the packets received at this port are sent to transmit priority queue ‘bPriority’.

Note: Call SwitchEnableMultiQueue to split single transmit queue into four priority queues before call SwitchConfigPortBased.

Return

None.

Synopsis

```
void SwitchDisable802_1P
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the 802.1p based priority QoS on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisableDiffServ
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the DiffServ based priority QoS on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisableDot1pRemapping
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the 802.1p priority re-mapping function on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisableMultiQueue
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables transmit multiple queues selection on the ‘bPort’ port. Only single transmit queue on the port.

Return

None.

Synopsis

```
void SwitchEnable802_1P
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine enables the 802.1p based priority QoS on the ‘bPort’ port.
Note: Call HardwareConfig802_1P_Priority to setup 802.1p priority value mapping to transmit priority queue, and SwitchEnableMultiQueue to split single transmit queue into four priority queues before call SwitchEnable802_1P.

Return
None.

Synopsis

```
void SwitchDisableDot1pRemapping
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

The routine disables the 802.1p priority re-mapping function on bPort’ port.

Return
None.

Synopsis

```
void SwitchEnableDot1pRemapping
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine enables the 802.1p priority re-mapping function on the ‘bPort’ port. That allows 802.1p priority field is replaced with the Port's default tag's priority value if the ingress packet's 802.1p priority has a higher priority than Port's default tag's priority

Return
None.

Synopsis

```
void SwitchEnableMultiQueue
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bPort               /* The port index */
)
```

Description

This routine enables the transmitting multiple queues selection on the ‘bPort’ port. The port transmit queue is split into four priority queues.

Return

None.

Synopsis

```
void SwitchInitPriority
(
    PHARDWARE pHardware           /* Pointer to hardware instance. */
)
```

Description

This routine disables all the QoS (Port based, 802.1p based, DiffServ based) functions on all the ports.

Return

None.

7.14 Set Device Mirror APIs

This section describes functions that configure the device port mirroring/monitoring/sniffing functions. These APIs are listed in the Table 7-14.

SwitchDisableMirrorReceive	Disables the device Receive Only mirror on a specific port.
SwitchDisableMirrorRxAndTx	Disables the device Receive And Transmit mirror.
SwitchDisableMirrorTransmit	Disables the device Transmit Only mirror on a specific port.
SwitchDisableMirrorSniffer	Disables the device mirror sniffer on a specific port.
SwitchEnableMirrorReceive	Enables the device Receive Only mirror on a specific port.
SwitchEnableMirrorRxAndTx	Enables the device Receive And Transmit mirror.
SwitchEnaableMirrorTransmit	Enables the device Transmit Only mirror on a specific port.
SwitchEnableMirrorSniffer	Enables the device mirror sniffer on a specific port.
SwitchInitMirror	Disables all the mirroring functions.

Table 7-14. Device Mirror APIs

Synopsis

```
void SwitchDisableMirrorReceive
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the "receive only" mirror on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisableMirrorRxAndTx
(
    PHARDWARE pHardware          /* Pointer to hardware instance. */
)
```

Description

This routine disables the "receive AND transmi" mirror function.

Return

None.

Synopsis

```
void SwitchDisableMirrorTransmit
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the "transmit only" mirror on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchDisableMirrorSniffer
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine disables the mirror sniffer on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchEnableMirrorReceive
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine enables the "receive only" mirror on the ‘bPort’ port. All the packets received on this port are mirrored to the sniffer port.

Return

None.

Synopsis

```
void SwitchEnableMirrorRxAndTx
(
    PHARDWARE pHardware,         /* Pointer to hardware instance. */
    UCHAR     bPort              /* The port index */
)
```

Description

This routine enables the "receive AND transmit" mirror function. All the packets received on port A AND transmitted on port B are mirrored to the sniffer port.

Return

None.

Synopsis

```
void SwitchEnableMirrorTransmit
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bPort               /* The port index */
)
```

Description

This routine enables the "transmit only" mirror on the ‘bPort’ port. All the packets transmitted on this port are mirrored to the sniffer port.

Return

None.

Synopsis

```
void SwitchEnableMirrorSniffer
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bPort               /* The port index */
)
```

Description

This routine enables the mirror sniffer on the ‘bPort’ port.

Return

None.

Synopsis

```
void SwitchInitMirror
(
    PHARDWARE pHardware           /* Pointer to hardware instance. */
)
```

Description

This routine disables all the mirror functions on all the port.

Return

None.

7.15 Set Device Table Accesses APIs

This section describes functions that create or read all kind of device table function support, including up to 16 group 802.1Q VLAN table, up to 8 entries of static MAC table, up to 1K entries of dynamic MAC table, and a fully compliant statistics management information base (MIB) counters per port. All read/write device tables are through KSZ88xx Indirect Access registers. These APIs are listed in the Table 7-15.

PortInitCounters	Reset the device MIB counters to zero on a specific port.
PortReadCounters	Reads the device MIB counters on a specific port.
SwitchReadDynMacTable	Reads the device dynamic MAC table on a specific entry.
SwitchReadStaticMacTable	Reads the device static MAC table on a specific entry.
SwitchReadVlanTable	Reads the device VLAN table on a specific entry.
SwitchWriteStaticMacTable	Create a MAC entry to the device static MAC table on a specific entry.
SwitchWriteVlanTable	Create a VLAN entry to the device VLAN table on a specific entry.

Table 7-15. Device Table Accesses APIs

Synopsis

```
void PortInitCounters
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bPort               /* The port index */
)
```

Description

This routine resets all the device MIB counters to zero from a specific port ‘bPort’.

Return

None.

Synopsis

```
void PortReadCounters
(
    PHARDWARE pHardware,          /* Pointer to hardware instance. */
    UCHAR     bPort               /* The port index */
)
```

Description

This routine is used to read the 32 MIB counters for the port ‘bPort’. This routine reads the MIB counters from device registers and stores them in the
pHardware->m_Port[bPort]-> cnCounter[ ] database.

The application program may call this routine periodically to avoid device MIB counter overflow.

Return

None.

Synopsis

```
BOOLEAN SwitchReadDynMacTable
(
    PHARDWARE pHardware,     /* Pointer to hardware instance. */
    USHORT    wAddr,         /* The address (index) of the entry */
    PUCHAR    MacAddr,       /* Buffer to store the retrieved MAC address
                                */
    PUCHAR    pbFID,         /* Buffer to store the retrieved FID */
    PUCHAR    pbSrcPort,     /* Buffer to store the retrieved source port
                                (b0=1, indicate port 1; b1=1, indicate
                                port 2; b2=1, indicate host port) */
    PUCHAR    pbTimestamp,   /* Buffer to store the timestamp */
    PUSHORT   pwEntries      /* Buffer to store the number of entries.
                                If this is zero, the table is empty and
                                so this function should not be called
                                again until             later. */
)
```

Description

This routine reads an entry of the dynamic MAC table from the device by table index ‘wAddr’. It gets MAC address associate with FID, timestamp, and the source port where the MAC is learned from this entry. Also it reports number of valid entries ‘pwEntries’ in the dynamic MAC table.

Return

TRUE if this entry is valid; otherwise, FALSE.

Synopsis

```
BOOLEAN SwitchReadStaticMacTable
(
    PHARDWARE pHardware,  /* Pointer to hardware instance. */
    USHORT    wAddr,      /* The address (index) of the entry */
    PUCHAR    MacAddr,    /* Buffer to store the retrieved MAC address */
    PUCHAR    pbPorts,    /* Buffer to store the retrieved port members
                             (b0=1, indicate port 1; b1=1, indicate port
                             2; b2=1, indicate host port) */
    PBOOLEAN  pfOverride, /*Buffer to store the retrieved override flag*/
    PBOOLEAN  pfUserFID,  /* Buffer to store the use FID flag which
                                indicates the FID is valid */
    PUCHAR    pbFID       /* Buffer to store the FID */
)
```

Description

This routine reads an entry of the static MAC table from the device by table index ‘wAddr’. It gets MAC address associate with FID, override flag, use FID flag, and forwarding port from this entry.

Return

TRUE if this entry is valid; otherwise, FALSE.

Synopsis

```
BOOLEAN SwitchReadVlanTable
(
    PHARDWARE pHardware,     /* Pointer to hardware instance. */
    USHORT    wAddr,         /* The address (index) of the entry */
    PUSHORT   pwVID,         /* Buffer to store the retrieved VID */
    PUCHAR    pbFID,         /* Buffer to store the retrieved FID */
    PUCHAR    pbMember       /* Buffer to store the retrieved port
                                membership (b0=1, indicate port 1; b1=1,
                                indicate port 2; b2=1, indicate host
                                port)*/
)
```

Description

This routine reads an entry of the VLAN table from the device by table index ‘wAddr’. It gets VLAN id associate with FID, and port membership (which ports are members of this VLAN).

Return

TRUE if this entry is valid; otherwise, FALSE.

Synopsis

```
void SwitchWriteStaticMacTable
(
    PHARDWARE pHardware, /* Pointer to hardware instance. */
    USHORT    wAddr,     /* The address (index) of the entry */
    PUCHAR    MacAddr,   /* Point to a buffer contains MAC address */
    UCHAR     bPorts,    /* The port members  (b0=1, indicate port 1;
                            b1=1, indicate port 2; b2=1, indicate host
                            port) */
    BOOLEAN   fOverride, /* The override flag to override the port
                            receive/transmit settings */
    BOOLEAN   fValid,    /* The valid flag to indicate entry is valid */
    BOOLEAN   fUserFID,  /* The use FID flag to indicate the FID is valid
                               */
    UCHAR     bFID       /* The FID value */
)
```

Description

This routine writes a static MAC entry to the device static MAC table by table index ‘wAddr’. It creates a valid ‘fValid’ entry with MAC address associate with FID, override flag, use FID flag, and forwarding port to this entry.

Return

None.

Synopsis

```
void SwitchWriteVlanTable
(
    PHARDWARE pHardware,     /* Pointer to hardware instance. */
    USHORT    wAddr,         /* The address (index) of the entry */
    USHORT    wVID,          /* The VLAN ID value */
    UCHAR     bFID,          /* The FID value */
    UCHAR     bMember,       /* The port membership (b0=1, indicate port
                                1; b1=1, indicate port 2; b2=1, indicate
                                host port)*/
    BOOLEAN   fValid         /* The valid flag to indicate entry is valid
                                */
)
```

Description

This routine writes a VLAN entry to the device VLAN table by table index ‘wAddr’. It creates a valid ‘fValid’ entry with VLAN id associate with FID, and port membership (which ports are members of this VLAN).

Return

None.

7.16EEPROM Access APIs

This section describes functions that read or write the serial EEPROM contents if one exists. The EEPROM contains MAC address, System ID, SystemSubID, and Configuration parameters.

Synopsis

```
USHORT EepromReadWord
(
PHARDWARE pHardware,           /* Pointer to hardware instance */
UCHAR Reg                      /* The register offset */
)
```

Description
This function reads a word from the AT93C46 EEPROM.

Return
The data value.

Synopsis

```
void EepromWriteWord
```

```
(
PHARDWARE pHardware,            /* Pointer to hardware instance */
UCHAR Reg,                      /* The register offset */
USHORT Data                     /* The data value */
)
```

Description
This function writes a word to the AT93C46 EEPROM.

Return
None