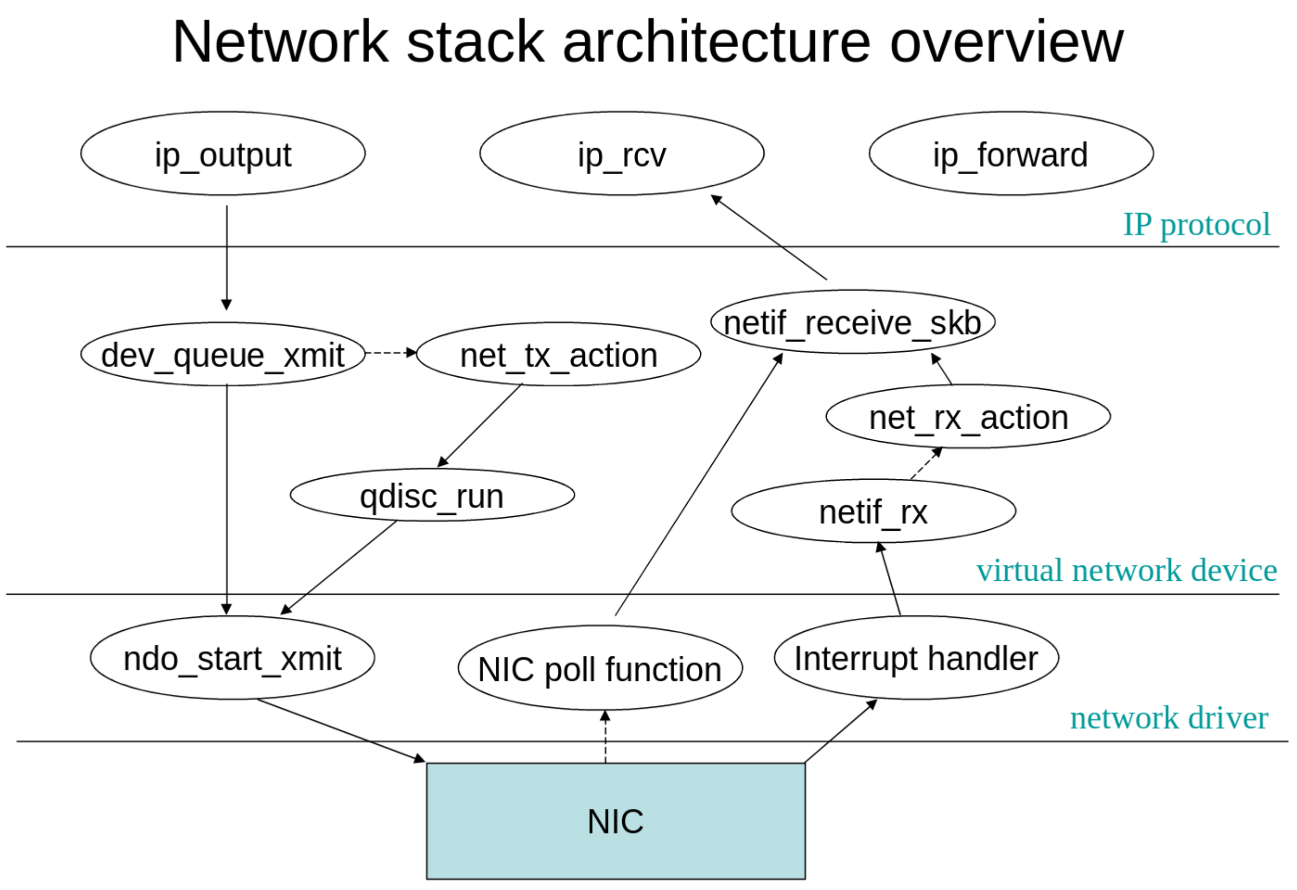
網路驅動程式實驗

使用ＱＥＭＵ

一、網路驅動程式概述

驅動程式與網路層



傳送：如有 Qdisc, 封包會 queue 在那裡並由 NET_TX_SOFTIRQ 來處理，否則就直接傳送。最後都會呼叫到驅動程式 net_device_ops 的 ndo_start_xmit()。

接收：可由系統主動 polling 或採 interrupt 方式處理。interrupt 函式從網路卡上將資料收下，包裝成 skbuff 後，呼叫 netif_rx() 往上送。netif_rx() 會將封包排入 backlog 交給 NET_RX_SOFTIRQ 處理。

網路裝置描述 net_device

```
struct net_device {
	char			name[IFNAMSIZ]; // 裝置名如 eth%d, register_netdev()時會換為數字
	struct list_head	dev_list;		// 所有網路裝置的串列
	unsigned short		hard_header_len;	// 乙太網路 header 長度 14 bytes (ETH_HLEN)
	unsigned int		mtu;		// 乙太網路 MTU 為  1500 bytes (ETH_DATA_LEN)
	unsigned long		tx_queue_len;	// queue 裡最多  packet 數, ether_setup() → 1000
	unsigned short		type;	// 提供  ARP 決定硬體位址種類，如  ARPHDR_ETHER
	unsigned char		addr_len;  // MAC 位址長度
	unsigned char		broadcast[MAX_ADDR_LEN]; // 0xffffffffffff
	unsigned char		*dev_addr;  // MAC address
	unsigned int		flags;	// 如下 IFF_  prefix
	netdev_features_t	features;
	int			watchdog_timeo; // 傳送超時  → ndo_tx_timeout 會被呼叫
	const struct net_device_ops *netdev_ops; // 網路介面操作函式
	const struct header_ops *header_ops; // operation for packet header
	struct net_device_stats		stats; // 統計資訊
};

/**
 * enum net_device_flags - &struct net_device flags
 *
 * These are the &struct net_device flags, they can be set by drivers, the
 * kernel and some can be triggered by userspace. Userspace can query and
 * set these flags using userspace utilities but there is also a sysfs
 * entry available for all dev flags which can be queried and set. These flags
 * are shared for all types of net_devices. The sysfs entries are available
 * via /sys/class/net/<dev>/flags. Flags which can be toggled through sysfs
 * are annotated below, note that only a few flags can be toggled and some
 * other flags are always always preserved from the original net_device flags
 * even if you try to set them via sysfs. Flags which are always preserved
 * are kept under the flag grouping @IFF_VOLATILE. Flags which are volatile
 * are annotated below as such.
 *
 * You should have a pretty good reason to be extending these flags.
 *
 * @IFF_UP: interface is up. Can be toggled through sysfs.
 * @IFF_BROADCAST: broadcast address valid. Volatile.
 * @IFF_DEBUG: turn on debugging. Can be toggled through sysfs.
 * @IFF_LOOPBACK: is a loopback net. Volatile.
 * @IFF_POINTOPOINT: interface is has p-p link. Volatile.
 * @IFF_NOTRAILERS: avoid use of trailers. Can be toggled through sysfs.
 *	Volatile.
 * @IFF_RUNNING: interface RFC2863 OPER_UP. Volatile.
 * @IFF_NOARP: no ARP protocol. Can be toggled through sysfs. Volatile.
 * @IFF_PROMISC: receive all packets. Can be toggled through sysfs.
 * @IFF_ALLMULTI: receive all multicast packets. Can be toggled through
 *	sysfs.
 * @IFF_MASTER: master of a load balancer. Volatile.
 * @IFF_SLAVE: slave of a load balancer. Volatile.
 * @IFF_MULTICAST: Supports multicast. Can be toggled through sysfs.
 * @IFF_PORTSEL: can set media type. Can be toggled through sysfs.
 * @IFF_AUTOMEDIA: auto media select active. Can be toggled through sysfs.
 * @IFF_DYNAMIC: dialup device with changing addresses. Can be toggled
 *       through sysfs.
 * @IFF_LOWER_UP: driver signals L1 up. Volatile.
 * @IFF_DORMANT: driver signals dormant. Volatile.
 * @IFF_ECHO: echo sent packets. Volatile.
 */
enum net_device_flags {
        IFF_UP                          = 1<<0,  /* sysfs */
        IFF_BROADCAST                   = 1<<1,  /* volatile */
        IFF_DEBUG               = 1<<2,  /* sysfs */
        IFF_LOOPBACK                    = 1<<3,  /* volatile */
        IFF_POINTOPOINT                 = 1<<4,  /* volatile */
        IFF_NOTRAILERS                  = 1<<5,  /* sysfs */
        IFF_RUNNING                     = 1<<6,  /* volatile */
        IFF_NOARP               = 1<<7,  /* sysfs */
        IFF_PROMISC                     = 1<<8,  /* sysfs */
        IFF_ALLMULTI                    = 1<<9,  /* sysfs */
        IFF_MASTER                      = 1<<10, /* volatile */
        IFF_SLAVE               = 1<<11, /* volatile */
        IFF_MULTICAST                   = 1<<12, /* sysfs */
        IFF_PORTSEL                     = 1<<13, /* sysfs */
        IFF_AUTOMEDIA                   = 1<<14, /* sysfs */
        IFF_DYNAMIC                     = 1<<15, /* sysfs */
        IFF_LOWER_UP                    = 1<<16, /* volatile */
        IFF_DORMANT                     = 1<<17, /* volatile */
        IFF_ECHO                = 1<<18, /* volatile */
};
```

網路裝置函式 net_device_ops

```
/*
 * This structure defines the management hooks for network devices.
 * The following hooks can be defined; unless noted otherwise, they are
 * optional and can be filled with a null pointer.
 *
 * int (*ndo_init)(struct net_device *dev);
 *     This function is called once when network device is registered.
 *     The network device can use this to any late stage initializaton
 *     or semantic validattion. It can fail with an error code which will
 *     be propogated back to register_netdev
 * void (*ndo_uninit)(struct net_device *dev);
 *     This function is called when device is unregistered or when registration
 *     fails. It is not called if init fails.
 * int (*ndo_open)(struct net_device *dev);
 *     This function is called when network device transistions to the up
 *     state.
 * int (*ndo_stop)(struct net_device *dev);
 *     This function is called when network device transistions to the down
 *     state.
 * netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
 *                               struct net_device *dev);
 *       Called when a packet needs to be transmitted.
 *       Must return NETDEV_TX_OK , NETDEV_TX_BUSY.
 *       (can also return NETDEV_TX_LOCKED iff NETIF_F_LLTX)
 *       Required can not be NULL.
 * void (*ndo_set_rx_mode)(struct net_device *dev);
 *       This function is called device changes address list filtering.
 *       If driver handles unicast address filtering, it should set
 *       IFF_UNICAST_FLT to its priv_flags.
 * int (*ndo_set_mac_address)(struct net_device *dev, void *addr);
 *       This function  is called when the Media Access Control address
 *       needs to be changed. If this interface is not defined, the
 *       mac address can not be changed.
 * int (*ndo_validate_addr)(struct net_device *dev);
 *       Test if Media Access Control address is valid for the device.
 * int (*ndo_do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
 *       Called when a user request an ioctl which can't be handled by
 *       the generic interface code. If not defined ioctl's return
 *       not supported error code.
 * int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
 *       Used to set network devices bus interface parameters. This interface
 *       is retained for legacy reason, new devices should use the bus
 *       interface (PCI) for low level management.
 * int (*ndo_change_mtu)(struct net_device *dev, int new_mtu);
 *       Called when a user wants to change the Maximum Transfer Unit
 *       of a device. If not defined, any request to change MTU will
 *       will return an error.
 * void (*ndo_tx_timeout)(struct net_device *dev);
 *       Callback uses when the transmitter has not made any progress
 *       for dev->watchdog ticks.
 * struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
 *       Called when a user wants to get the network device usage
 *       statistics. Drivers must do one of the following:
 *       1. Define @ndo_get_stats64 to fill in a zero-initialised
 *          rtnl_link_stats64 structure passed by the caller.
 *       2. Define @ndo_get_stats to update a net_device_stats structure
 *          (which should normally be dev->stats) and return a pointer to
 *          it. The structure may be changed asynchronously only if each
 *          field is written atomically.
 *       3. Update dev->stats asynchronously and atomically, and define
 *          neither operation.
 *
 */
struct net_device_ops {
        int                     (*ndo_init)(struct net_device *dev);
        void                    (*ndo_uninit)(struct net_device *dev);
        int                     (*ndo_open)(struct net_device *dev);
        int                     (*ndo_stop)(struct net_device *dev);
        netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
                                                   struct net_device *dev);
	void			(*ndo_set_rx_mode)(struct net_device *dev);
	int			(*ndo_set_mac_address)(struct net_device *dev,
						       void *addr);
	int			(*ndo_validate_addr)(struct net_device *dev);
	int			(*ndo_do_ioctl)(struct net_device *dev,
					        struct ifreq *ifr, int cmd);
	int			(*ndo_set_config)(struct net_device *dev,
					          struct ifmap *map);
	int			(*ndo_change_mtu)(struct net_device *dev,
						  int new_mtu);
	struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
};
```

封包 skbuff 描述及操作

```
/**
 *	struct sk_buff - socket buffer
 *	@dev: Device we arrived on/are leaving by
 *	@len: Length of actual data
 *	@data_len: Data length stored in separated fragments
 *	@csum: Checksum (must include start/offset pair)
 *	@csum_start: Offset from skb->head where checksumming should start
 *	@csum_offset: Offset from csum_start where checksum should be stored
 *	@ip_summed: Driver fed us an IP checksum, read checksumming description below
 *	@pkt_type: Filled by eth_type_trans(): PACKET_HOST, PACKET_OTHERHOST, ...
 *	@protocol: Packet protocol from driver. Value from eth_type_trans()
 *	@transport_header: Transport layer header
 *	@network_header: Network layer header
 *	@mac_header: Link layer header
 *	@tail: Data tail pointer
 *	@end: End of buffer
 *	@head: Head of buffer
 *	@data: Data head pointer
 */

struct sk_buff {
	struct net_device	*dev;
	unsigned int		len,
				data_len;
	__u8			pkt_type:3;
	__u8			ip_summed:2;

	union {
		__wsum			csum;
		struct {
			__u16	csum_start;
			__u16	csum_offset;
		};
	};
	__be16			protocol;
	__u16			transport_header;
	__u16			network_header;
	__u16			mac_header;
	/* These elements must be at the end, see alloc_skb() for details.  */
	sk_buff_data_t		tail;
	sk_buff_data_t		end;
	unsigned char		*head,
				*data;
};

/* A. Checksumming of received packets by device.
 *
 * CHECKSUM_NONE:
 *
 *   Device failed to checksum this packet e.g. due to lack of capabilities.
 *   The packet contains full (though not verified) checksum in packet but
 *   not in skb->csum. Thus, skb->csum is undefined in this case.
 *
 * CHECKSUM_UNNECESSARY:
 *
 *   The hardware you're dealing with doesn't calculate the full checksum
 *   (as in CHECKSUM_COMPLETE), but it does parse headers and verify checksums
 *   for specific protocols. For such packets it will set CHECKSUM_UNNECESSARY
 *   if their checksums are okay. skb->csum is still undefined in this case
 *   though. It is a bad option, but, unfortunately, nowadays most vendors do
 *   this. Apparently with the secret goal to sell you new devices, when you
 *   will add new protocol to your host, f.e. IPv6 8)
 *
 *   CHECKSUM_UNNECESSARY is applicable to following protocols:
 *     TCP: IPv6 and IPv4.
 *     UDP: IPv4 and IPv6. A device may apply CHECKSUM_UNNECESSARY to a
 *       zero UDP checksum for either IPv4 or IPv6, the networking stack
 *       may perform further validation in this case.
 *     GRE: only if the checksum is present in the header.
 *     SCTP: indicates the CRC in SCTP header has been validated.
 *
 *   skb->csum_level indicates the number of consecutive checksums found in
 *   the packet minus one that have been verified as CHECKSUM_UNNECESSARY.
 *   For instance if a device receives an IPv6->UDP->GRE->IPv4->TCP packet
 *   and a device is able to verify the checksums for UDP (possibly zero),
 *   GRE (checksum flag is set), and TCP-- skb->csum_level would be set to
 *   two. If the device were only able to verify the UDP checksum and not
 *   GRE, either because it doesn't support GRE checksum of because GRE
 *   checksum is bad, skb->csum_level would be set to zero (TCP checksum is
 *   not considered in this case).
 *
 * CHECKSUM_COMPLETE:
 *
 *   This is the most generic way. The device supplied checksum of the _whole_
 *   packet as seen by netif_rx() and fills out in skb->csum. Meaning, the
 *   hardware doesn't need to parse L3/L4 headers to implement this.
 *
 *   Note: Even if device supports only some protocols, but is able to produce
 *   skb->csum, it MUST use CHECKSUM_COMPLETE, not CHECKSUM_UNNECESSARY.
 *
 * CHECKSUM_PARTIAL:
 *
 *   This is identical to the case for output below. This may occur on a packet
 *   received directly from another Linux OS, e.g., a virtualized Linux kernel
 *   on the same host. The packet can be treated in the same way as
 *   CHECKSUM_UNNECESSARY, except that on output (i.e., forwarding) the
 *   checksum must be filled in by the OS or the hardware.
 *
 * B. Checksumming on output.
 *
 * CHECKSUM_NONE:
 *
 *   The skb was already checksummed by the protocol, or a checksum is not
 *   required.
 *
 * CHECKSUM_PARTIAL:
 *
 *   The device is required to checksum the packet as seen by hard_start_xmit()
 *   from skb->csum_start up to the end, and to record/write the checksum at
 *   offset skb->csum_start + skb->csum_offset.
 *
 *   The device must show its capabilities in dev->features, set up at device
 *   setup time, e.g. netdev_features.h:
 *
 *	NETIF_F_HW_CSUM	- It's a clever device, it's able to checksum everything.
 *	NETIF_F_IP_CSUM - Device is dumb, it's able to checksum only TCP/UDP over
 *			  IPv4. Sigh. Vendors like this way for an unknown reason.
 *			  Though, see comment above about CHECKSUM_UNNECESSARY. 8)
 *	NETIF_F_IPV6_CSUM - About as dumb as the last one but does IPv6 instead.
 *	NETIF_F_...	- Well, you get the picture.
 *
 * CHECKSUM_UNNECESSARY:
 *
 *   Normally, the device will do per protocol specific checksumming. Protocol
 *   implementations that do not want the NIC to perform the checksum
 *   calculation should use this flag in their outgoing skbs.
 *
 *	NETIF_F_FCOE_CRC - This indicates that the device can do FCoE FC CRC
 *			   offload. Correspondingly, the FCoE protocol driver
 *			   stack should use CHECKSUM_UNNECESSARY.
 *
 * Any questions? No questions, good.		--ANK
 */

/**
 *	netdev_alloc_skb - allocate an skbuff for rx on a specific device
 *	@dev: network device to receive on
 *	@length: length to allocate
 *
 *	Allocate a new &sk_buff and assign it a usage count of one. The
 *	buffer has unspecified headroom built in. Users should allocate
 *	the headroom they think they need without accounting for the
 *	built in space. The built in space is used for optimisations.
 *
 *	%NULL is returned if there is no free memory. Although this function
 *	allocates memory it can be called from an interrupt.
 */
static inline struct sk_buff *netdev_alloc_skb(struct net_device *dev,
					       unsigned int length)
{
	return __netdev_alloc_skb(dev, length, GFP_ATOMIC);
}

/*
 * It is not allowed to call kfree_skb() or consume_skb() from hardware
 * interrupt context or with hardware interrupts being disabled.
 * (in_irq() || irqs_disabled())
 *
 * We provide four helpers that can be used in following contexts :
 *
 * dev_kfree_skb_irq(skb) when caller drops a packet from irq context,
 *  replacing kfree_skb(skb)
 *
 * dev_consume_skb_irq(skb) when caller consumes a packet from irq context.
 *  Typically used in place of consume_skb(skb) in TX completion path
 *
 * dev_kfree_skb_any(skb) when caller doesn't know its current irq context,
 *  replacing kfree_skb(skb)
 *
 * dev_consume_skb_any(skb) when caller doesn't know its current irq context,
 *  and consumed a packet. Used in place of consume_skb(skb)
 */

/**
 *	skb_put - add data to a buffer
 *	@skb: buffer to use
 *	@len: amount of data to add
 *
 *	This function extends the used data area of the buffer. If this would
 *	exceed the total buffer size the kernel will panic. A pointer to the
 *	first byte of the extra data is returned.
 */
unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
{
	unsigned char *tmp = skb_tail_pointer(skb);
	SKB_LINEAR_ASSERT(skb);
	skb->tail += len;
	skb->len  += len;
	if (unlikely(skb->tail > skb->end))
		skb_over_panic(skb, len, __builtin_return_address(0));
	return tmp;
}
```

```
/**
 *      skb_push - add data to the start of a buffer
 *      @skb: buffer to use
 *      @len: amount of data to add
 *
 *      This function extends the used data area of the buffer at the buffer
 *      start. If this would exceed the total buffer headroom the kernel will
 *      panic. A pointer to the first byte of the extra data is returned.
 */
unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
{
        skb->data -= len;
        skb->len  += len;
        if (unlikely(skb->data<skb->head))
                skb_under_panic(skb, len, __builtin_return_address(0));
        return skb->data;
}
/**
 *      skb_reserve - adjust headroom
 *      @skb: buffer to alter
 *      @len: bytes to move
 *
 *      Increase the headroom of an empty &sk_buff by reducing the tail
 *      room. This is only allowed for an empty buffer.
 */
static inline void skb_reserve(struct sk_buff *skb, int len)
{
        skb->data += len;
        skb->tail += len;
}
/**
 *      skb_pull - remove data from the start of a buffer
 *      @skb: buffer to use
 *      @len: amount of data to remove
 *
 *      This function removes data from the start of a buffer, returning
 *      the memory to the headroom. A pointer to the next data in the buffer
 *      is returned. Once the data has been pulled future pushes will overwrite
 *      the old data.
 */
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
{
        return skb_pull_inline(skb, len);
}
/**
 *      skb_trim - remove end from a buffer
 *      @skb: buffer to alter
 *      @len: new length
 *
 *      Cut the length of a buffer down by removing data from the tail. If
 *      the buffer is already under the length specified it is not modified.
 *      The skb must be linear.
 */
void skb_trim(struct sk_buff *skb, unsigned int len)
{
	if (skb->len > len)
		__skb_trim(skb, len);
}
/**
 *	skb_headroom - bytes at buffer head
 *	@skb: buffer to check
 *
 *	Return the number of bytes of free space at the head of an &sk_buff.
 */
static inline unsigned int skb_headroom(const struct sk_buff *skb)
{
	return skb->data - skb->head;
}
/**
 *	skb_tailroom - bytes at buffer end
 *	@skb: buffer to check
 *
 *	Return the number of bytes of free space at the tail of an sk_buff
 */
static inline int skb_tailroom(const struct sk_buff *skb)
{
	return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
}
```

二、實驗環境設定（實作）

開發環境為 Ubuntu 14.04.2 64 bit。需要的原始碼有：

linux-3.18.14.tar.xz
busybox-1.23.2.tar.bz2

編譯 Linux kernel 及 BusyBox

BusyBox 及 Linux kernel 的 config 檔已提供: busybox.config, linux.config. 它們只是'defconfig' 再加上一些小改變。BusyBox 改為 static linking；移除幾個實險中用不到的 kernel module。

如要帶入提供的這兩個檔案，需要把檔案複製到各別原始碼目錄下，並改名為 .config。然後在各別目錄下以 make oldconfig 確認無誤後，即可使用 make 指令編譯。

建立 root filesystem

在實驗目錄下，建立一子目錄名 rootfs。並執行下列 scripts 安裝 BusyBox, 系統啓動設定及 kernel module 至 root filesystem 裡。

```
./install-busybox-to-rootfs.sh
./install-config-to-rootfs.sh
./install-kmod-to-rootfs.sh
```

最後用這個 script 將 root filesystem 打包為 rootfs.cpio.gz。這個檔案將提供 QEMU 使用。

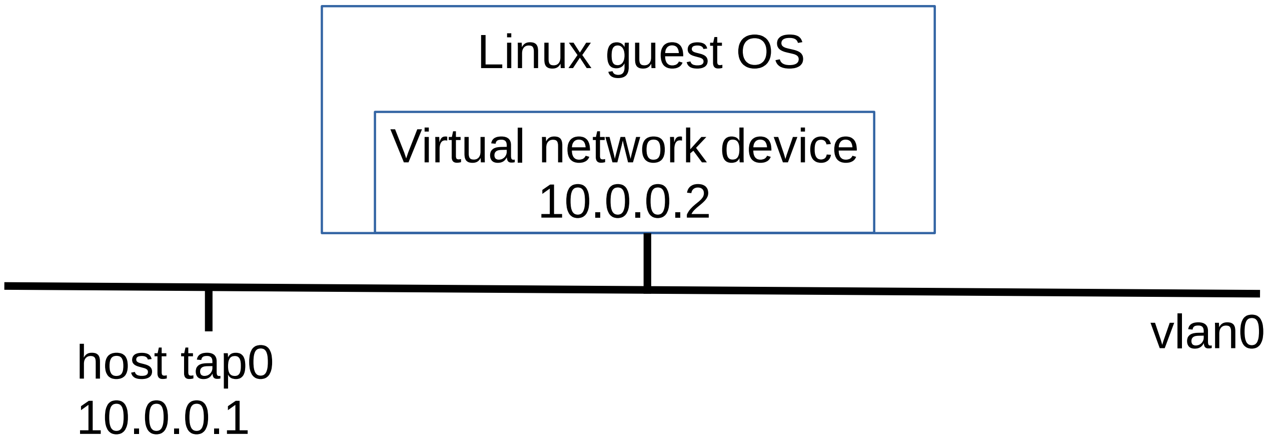
```
./pack-rootfs.sh
```

啓動 QEMU

以 root 權限執行 start-qemu.sh

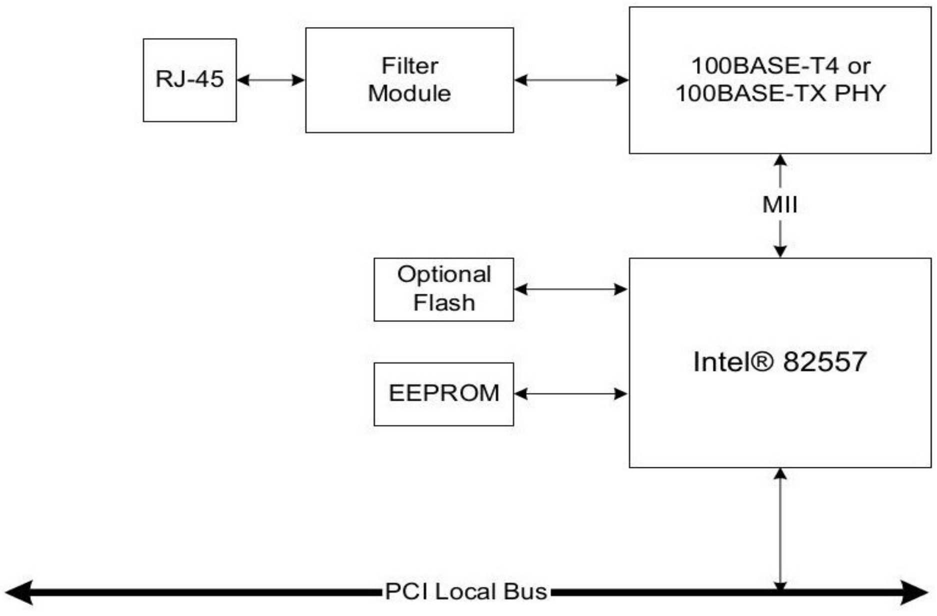
```
sudo ./start-qemu.sh
```

這會在 host (開發環境) 端建立一 tap0 虛擬網路介面，其 IP 位址為 10.0.0.1；而 guest (QEMU 虛擬機器) 端的虛擬網路介面模擬為 Intel PRO/100 (i82559ER)，其 IP 位址為 10.0.0.2。二個網路介面都處於相同的虛擬區域網路 vlan0 中。試著在 host 及 guest 用 ping 檢查網路是否如規畫運作。



三、Intel 8255x 軟體介面簡介

82557 Network Interface Card Block Diagram

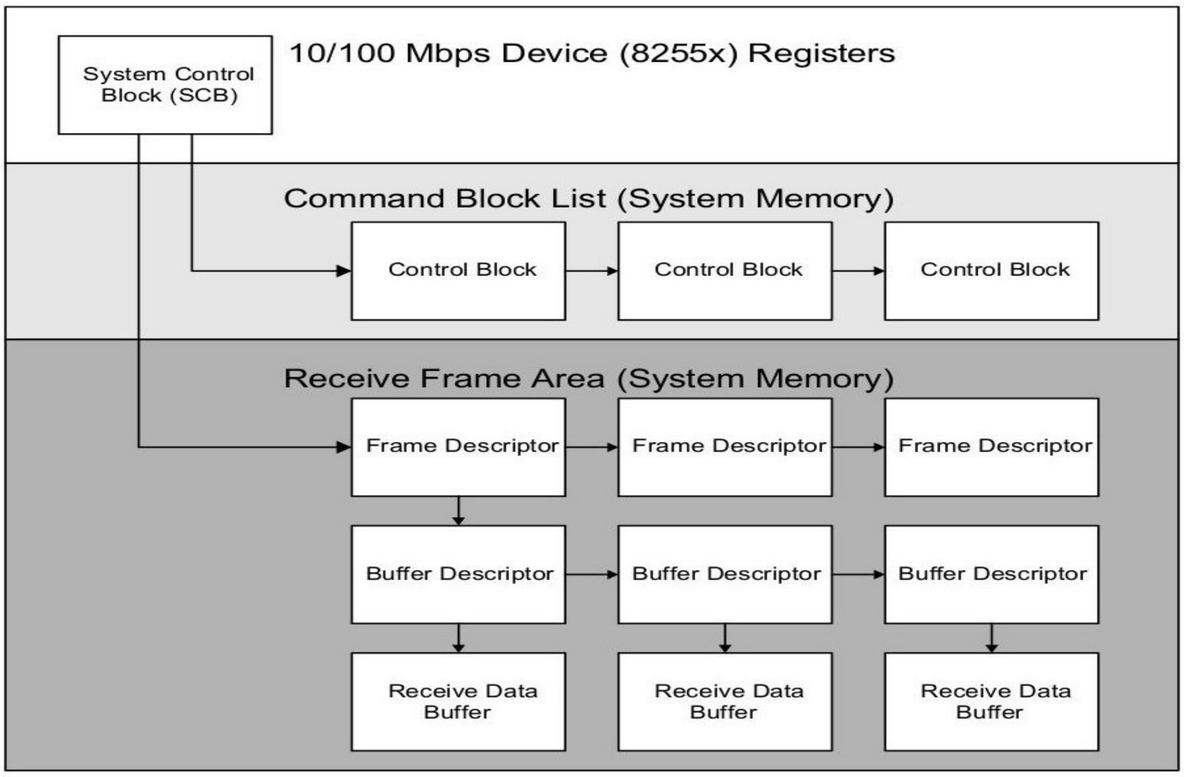


The 8255x LAN controllers establish a shared memory communication system with the host CPU. Software controls the device by writing and reading data to and from this shared memory space. All of the LAN controller functions (configuration, transmitting data, receiving data, etc.) that are software manageable are controlled through this shared memory space.

The Shared Memory Architecture

The shared memory structure is divided into three parts: the Control/Status Registers (CSR), the Command Block List (CBL), and the Receive Frame Area (RFA). The CSR physically resides on the LAN controller and can be accessed by either I/O or memory cycles, while the rest of the memory structures reside in system (host) memory. The first 8 bytes of the CSR is called the System Control Block (SCB). The SCB serves as a central communication point for exchanging control and status information between the host CPU and the 8255x. The host software controls the state of the Command Unit (CU) and Receive Unit (RU) (for example, active, suspended or idle) by writing commands to the SCB. The device posts the status of the CU and RU in the SCB Status word and indicates status changes with an interrupt. The SCB also holds pointers to a linked list of action commands called the CBL and a linked list of receive resources called the RFA. This type of structure is shown in the figure below.

8255x Memory Architecture

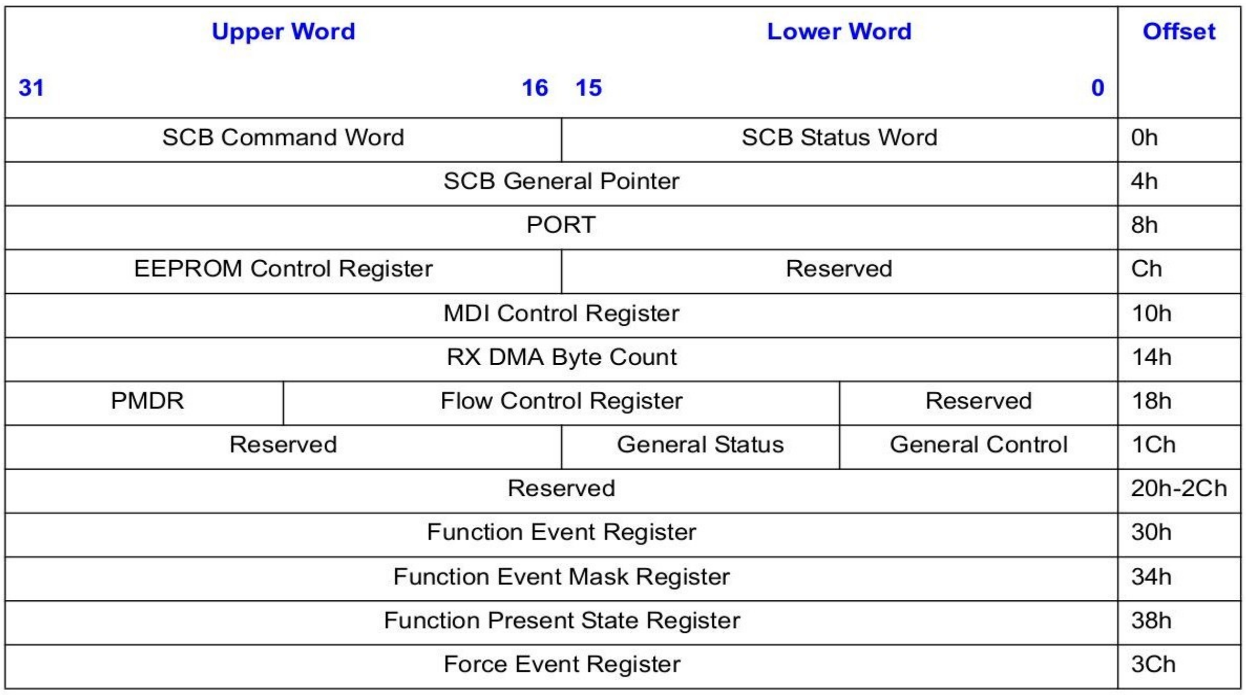


The CBL consists of a linked list of individual action commands in structures called Command Blocks (CBs). The CBs contain command parameters and status of the action commands. Action commands are categorized as follows:

- Non-transmit (non-Tx) commands: This category includes commands such as no operation (NOP), Configure, IA Setup, Multicast Setup, Dump and Diagnose.
- Transmit (Tx) command: This includes Transmit Command Blocks (TxCB).

The Receive Frame Area (RFA) consists of a list of Receive Frame Descriptors (RFDs) and a list of user-prepared or NOS provided buffers.

Control / Status Register (CSR)


. **Control / Status Register**

Upper Word		Lower Word		Offset
31	16	15	0	
SCB Command Word		SCB Status Word		0h
SCB General Pointer				4h
PORT				8h
EEPROM Control Register		Reserved		Ch
MDI Control Register				10h
RX DMA Byte Count				14h
PMDR	Flow Control Register		Reserved	18h
Reserved		General Status	General Control	1Ch
Reserved				20h-2Ch
Function Event Register				30h
Function Event Mask Register				34h
Function Present State Register				38h
Force Event Register				3Ch

- SCB Command Word. This register is where software writes commands for the CU and RU.
- SCB Status Word. The device places the CU and RU status for the CPU to read in this word.
- SCB General Pointer. The SCB General Pointer points to various data structures in main memory depending on the current SCB Command word.
- Port Interface. This special interface allows the CPU to reset the device and force it to dump information to main memory or perform an internal self test.
- EEPROM Control Register. The EEPROM Control Register allows the CPU to read and write to an external EEPROM.
- MDI Control Register. This register allows the CPU to read and write information from Physical Layer components through the Management Data Interface.

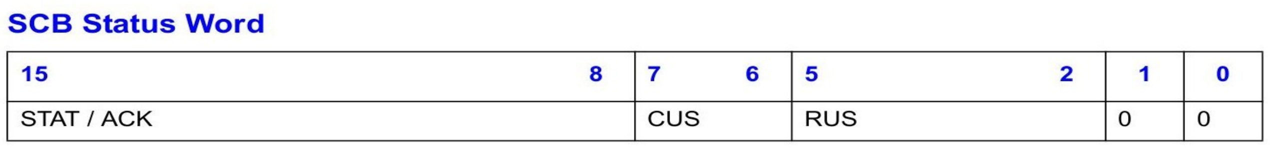
System Control Block (SCB)

Control commands are issued to the device by writing them into the SCB. This causes the device to examine the command, clear the lower byte of the SCB command word (indicating command acceptance), and perform the required action. Control commands perform the following types of tasks:

- Operate the Command Unit (CU). The SCB controls the CU by specifying the address of the Command Block List (CBL) and by starting or resuming execution of CBL commands.
- Operate the Receive Unit (RU). The SCB controls RU frame reception by specifying the address of the Receive Frame Area (RFA) and by starting, resuming, or aborting frame reception.
- Load the dump counters address.
- Command the device to dump or dump and reset its internal statistical counters.
- Indicate the cause of the current interrupt(s). In a similar manner, the CPU can send

Interrupt Acknowledgments to the device by writing them into the Interrupt Acknowledge byte (upper byte of the SCB Status word).
- The device also indicates status to the CPU through bits in the SCB Status word such as CU status and RU status.

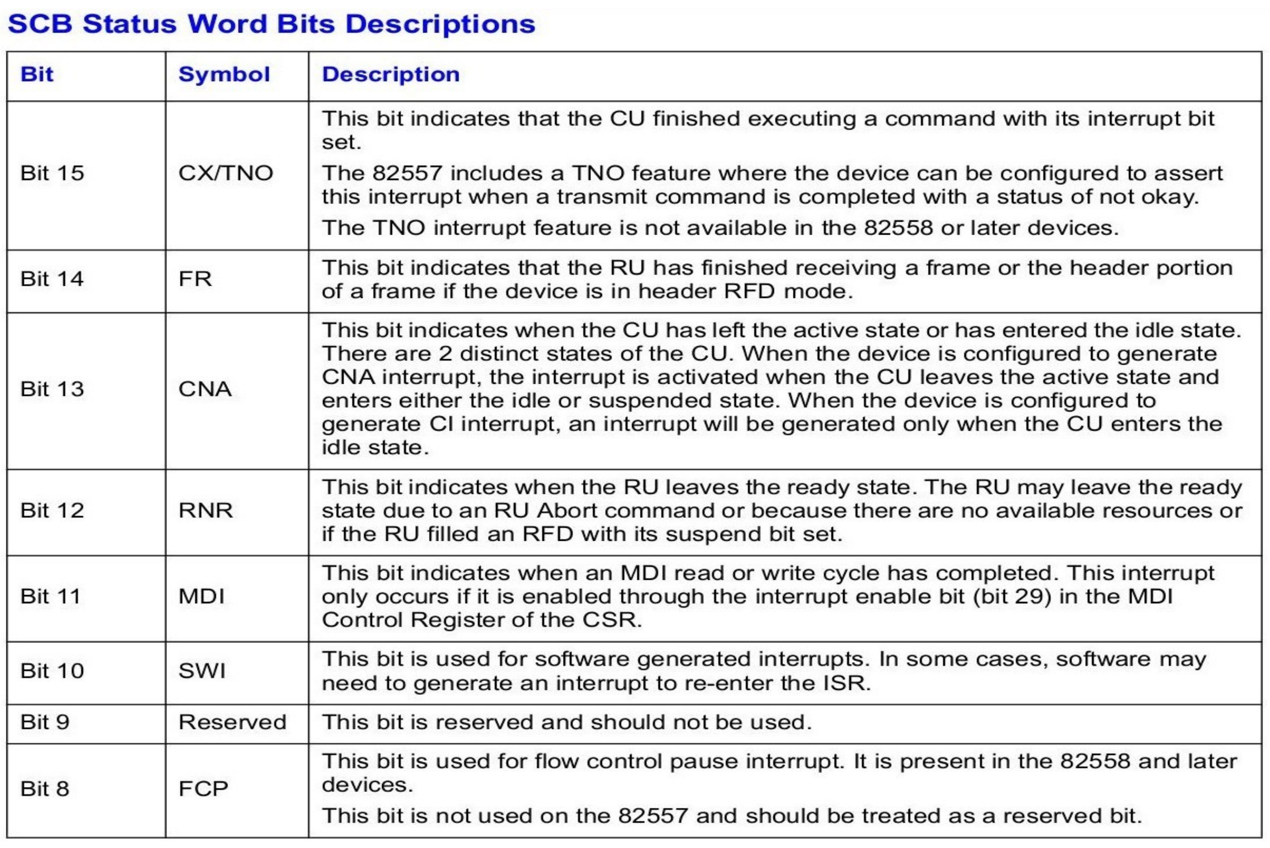
SCB Status Word



SCB Status Word

15 8	7 6	5 2	1	0
STAT / ACK	CUS	RUS	0	0

The SCB Status word is addressable as two bytes. The upper byte is called the STAT/ACK byte, and the lower, the SCB Status byte. The SCB Status byte indicates the status of the CU and RU. The STAT/ACK byte reports the device status as bits, which represent the causes of interrupts. Writing to the STAT/ACK bits will acknowledge pending interrupts. As described below, there are many different possible interrupt events. The LAN controller asserts the interrupt line to the CPU if any of these interrupt events need to be serviced. More than one STAT/ACK bits may be set at the same time. Writing 1 back to a STAT/ACK bit that was set will acknowledge that particular interrupt bit. The device will de-assert its interrupt line only when all pending interrupt STAT bits are acknowledged. All pending STAT bits do not need to be acknowledged in a single access, but it is recommended if the interrupt service routine is likely to process all pending interrupts.



SCB Status Word Bits Descriptions

Bit	Symbol	Description
Bit 15	CX/TNO	This bit indicates that the CU finished executing a command with its interrupt bit set. The 82557 includes a TNO feature where the device can be configured to assert this interrupt when a transmit command is completed with a status of not okay. The TNO interrupt feature is not available in the 82558 or later devices.
Bit 14	FR	This bit indicates that the RU has finished receiving a frame or the header portion of a frame if the device is in header RFD mode.
Bit 13	CNA	This bit indicates when the CU has left the active state or has entered the idle state. There are 2 distinct states of the CU. When the device is configured to generate CNA interrupt, the interrupt is activated when the CU leaves the active state and enters either the idle or suspended state. When the device is configured to generate CI interrupt, an interrupt will be generated only when the CU enters the idle state.
Bit 12	RNR	This bit indicates when the RU leaves the ready state. The RU may leave the ready state due to an RU Abort command or because there are no available resources or if the RU filled an RFD with its suspend bit set.
Bit 11	MDI	This bit indicates when an MDI read or write cycle has completed. This interrupt only occurs if it is enabled through the interrupt enable bit (bit 29) in the MDI Control Register of the CSR.
Bit 10	SWI	This bit is used for software generated interrupts. In some cases, software may need to generate an interrupt to re-enter the ISR.
Bit 9	Reserved	This bit is reserved and should not be used.
Bit 8	FCP	This bit is used for flow control pause interrupt. It is present in the 82558 and later devices. This bit is not used on the 82557 and should be treated as a reserved bit.

SCB Status Word Bits Descriptions

Bit	Symbol	Description
Bits 7:6	CUS	This field contains the CU status (2 bits). Valid values are for this field are: 00 Idle 01 Suspended 10 LPQ Active 11 HQP Active
Bits 5:2	RUS	This field contains the RU status (4 bits). Valid values are: 0000 Idle 0001 Suspended 0010 No resources 0011 Reserved 0100 Ready 0101 Reserved 0110 Reserved 0111 Reserved 1000 Reserved 1001 Reserved 1010 Reserved 1011 Reserved 1100 Reserved 1101 Reserved 1110 Reserved 1111 Reserved
Bits 1:0	Reserved	These bits are reserved and should not be used.

The SCB Status word is not updated immediately in response to SCB commands. For example, the CU status will remain in the idle state for a period of time after the CU start command is issued. Software should not rely exclusively on the state of the SCB Status word to determine when it is appropriate to issue commands requiring the device to be in a specific state. Software may be required to keep an internal state engine on the commands recently issued to the device to insure that data read from the register is valid.

SCB Command Word

. SCB Command Word

31 26	25	24	23 20	19	18 16
Specific Interrupt Mask Bits	SI	M	CU Command	0	RU Command

SCB Command Word Bits Descriptions

Bit	Symbol	Description
Bits 31:26	Specific Interrupt Mask Bits	The mask bits range from bit 31 to 26. Writing a 1 to a mask bit disables the 8255x (except the 82557) from generating an interrupt, or asserting the INTA# pin, due to that corresponding event. The device may still generate interrupts due to other interrupt events that are not masked. The corresponding bits and their masks are: 31 - CX Mask 30 - FR Mask 29 - CNA Mask 28 - RNR Mask 27 - ER Mask 26 - FCP Mask These bits are also described in Section 6.3.2, "System Control Block (SCB)". These bits are not present in the 82557 and should be treated as reserved.
Bit 25	SI	This bit is used for the software generated interrupt. Writing a 1 to this bit causes the device to generate an interrupt, and writing a 0 has no effect. Reads from this bit always return a zero. The M bit (bit 24) has higher precedence than the SI bit. Thus, if a 1 is simultaneously written to both, no interrupts occur.
Bit 24	M	This bit is used as the interrupt mask bit. When this bit is set to 1, the device does not assert its INTA# pin (PCI interrupt pin). The M bit has higher precedence than bits 31 through 26 of this word and the SI bit (bit 25).
Bits 23:20	CUC	This field contains the CU Command. Valid values for this field are: 0000 NOP. The no operation command does not affect the current state of the unit. 0001 CU Start. CU Start begins execution of the first command on the CBL. A pointer to the first CB of the CBL should be placed in the SCB General Pointer before issuing this command. **NOTE:** The CU Start command should only be issued when the CU is in the idle or suspended states (never when the CU is in the active state) and all of the previously issued CBs have been processed and completed by the CU. Sometimes, it is only possible to determine that all CBs are completed by checking the complete bit in all previously issued Command Blocks. 0010 CU Resume. The CU Resume command resumes CU operation by executing the next command. If the CU is Idle, it ignores the CU Resume command. 0100 Load Dump Counters Address. This command directs the device where to write dump data when the Dump Statistical Counters or Dump and Reset Statistical Counters command is used. It must be executed at least once before the Dump Statistical Counters or Dump and Reset Statistical Counters command is used. The address of the dump area must be placed in the general pointer register. 0101 Dump Statistical Counters. This command directs the device to dump its statistical counters to the area designated by the Load Dump Counters Address command. 0110 Load CU Base. The internal CU Base Register is loaded with the value in the SCB General Pointer. 0111 Dump and Reset Statistical Counters. This command directs the device to first dump its statistical counters to the area designated by the Load Dump Counters Address command and then to clear these counters. 1010 CU Static Resume. It resumes CU operation by executing the next command. If the CU is idle, it will ignore the CU Resume command. This command should be used only when the device CU is in the suspended state and has no pending CU Resume commands. This command is only valid for the 82558 and later devices. It is not valid for the 82557.
Bit 19	Reserved	This bit is reserved and should be set to 0.
Bits 18:16	RUC	This field contains the RU Command. Valid values are: 000 NOP. NOP is a no operation command and does not alter current state of unit. 001 RU Start. RU Start enables the receive unit. The pointer to the RFA must be placed in the SCB General Pointer before using this command. The device pre-fetches the first RFD in preparation of receiving incoming frames that pass its address filtering. 010 RU Resume. The RU Resume command resumes frame reception (only when in suspended state). 011 Receive DMA Redirect. This command is only valid for the 82558 and later devices. The buffers are indicated by an RBD chain, which is pointed to by an offset stored in the general pointer register (in the RU base). 100 RU Abort. The RU Abort command immediately stops RU receive operation. 101 Load Header Data Size (HDS). After a load HDS command is issued, the device expects to only find header RFDs or to be used in Receive DMA mode until it is reset. This value defines the size of the header portion of the RFDs or receive buffers. The HDS value is defined by the lower 14 bits of the SCB General Pointer; thus, bits 15 through 31 should always be set to zeros when using this command. The value of HDS should be an even non-zero number. 110 Load RU Base. The internal RU Base Register is loaded with the value that was placed in the SCB General Pointer just before this command was issued.

Transmit Action Command

. Transmit Command Format

Offset	Command Word Bits 31:16								Status Word Bits 15:0				
00h	EL	S	I	CID	000	NC	SF	100	C	X	OK	U	XXXXXXXXXXX
04h	Link Address (A31:A0)												
08h	Transmit Buffer Descriptor Array Address												
	TBD Number				Transmit Threshold				EOF	0	Transmit Command Block Byte Count		

Link Address	This is the 32-bit address of the next command block. It is added to the CU base to obtain the actual address.
EL (Bit 31)	If this bit is set to one, it indicates that this command block is the last one on the CBL. The CU will go from the active to the idle state after the execution of the CB is finished. This transition will always cause an interrupt with the CNA/CI bit set in the SCB.
S (Bit 30)	If this bit is set to one, the CU will be suspended after the completion of this CB. A CNA interrupt will be generated if the device is configured for this. The CU transitions from the active to the suspended state after the execution of the CB.
I (Bit 29)	If the I bit is set to one, the device generates an interrupt after the execution of the CB is finished. If I is not set to one, the CX interrupt will not be generated.
CID (Bits 28:24)	The CNA Interrupt Delay field is only present on 82558 and later generation controllers. (It is not a valid field for the 82557, unless special microcode is downloaded to this device.) The CID indicates the length of time CNA interrupts are delayed by the device.
Bits 23:21	These bits are reserved and should all be set to 0.
NC	0: CRC and Source Address are inserted by the controller. If the "No Source Address Insertion" (NSAI) bit is set by the configure command, then only the CRC is inserted by the controller. Normally, this bit should be set because it is desirable to have the device compute and insert the CRC automatically. 1: CRC and Source Address are not inserted by the controller and are assumed to come from memory.
SF	This bit indicates whether the device is operating in simplified or flexible mode. 0 = Simplified Mode. All transmit data is in the TCB, and the TBD array address field must equal all 1s. 1 = Flexible Mode. Data is in the TCB (optional) and in a linked list of the TBDs.
CMD (Bits 18:16)	This is the transmit command, which has a value of 100b.
C (Bit 15)	The C bit indicates that the transmit DMA has completed processing the last byte of data associated with the TCB. This is not the actual completion of the transmit command as the C bit indicates in other action commands. The actual completion of a transmit command occurs when the frame is actually sent out on the wire. At the end of actual transmission, no further status is posted in the TCB, but the transmit statistical counters are updated.
OK (Bit 13)	The OK bit indicates that the command was executed without error. If it equals 1, no error occurred (command executed OK). If the OK bit is zero and the C bit is set, then an error occurred. **NOTE:** For the transmit command, the OK bit is always set when the C bit is set.
U (Bit 12)	The U bit indicates that one or more underruns were encountered by this or previously transmitted frames since the last TCB status update. Since there is no mechanism for indicating underruns during or at the end of frame transmission, this bit is set in addition to the transmit underruns statistical counter for software management purposes.
Bits 11:0	These bits must be set to all zeros.
TBD Array Address	In flexible mode, this is a 32-bit address pointing to the first TBD in a contiguous list of TBDs called the TBD array. A TBD is two Dwords, a transmit buffer pointer and buffer size data. In simplified mode this field should be set by software to a null pointer (0FFFFFFFFh).
TBD Number	In flexible mode, this represents the number of transmit buffers in the contiguous TBD array. It should have a one to one correspondence of TBDs and buffers in the array. If the device finds the TBD number equal to 0, it assumes the TBD array address is a null pointer and the EOF bit is set. The 82558 and 82559 have a special dynamic TBD mode that the 82557 does not have. If the dynamic TBD mode is enabled (in the configure command), software should write a value of FFh into this field. Software should also mark each TBD as valid or invalid. In the 82557, the TBD number is the only indication that the TBD is the last associated with a particular transmit frame.

Transmit Threshold

The transmit threshold defines the number of bytes that should be present in the controller's transmit FIFO before it starts transmitting the frame. The value is internally multiplied by 8 to give a granularity of 8 bytes. For example, a value of 1 means the 82557 will start transmitting only when it has 8 bytes in its transmit FIFO. The transmit threshold should be within a range of 1 to 0E0h. (The value 0FFh should not be used.)

EOF

The EOF bit indicates if the whole frame is in the transmit command block. For consistency, it should be set by software, although it is not checked in simplified or flexible mode.

TCB Byte Count

For either simplified or flexible mode, the controller is able to transmit data from memory immediately contiguous to the TCB itself. The amount of data to be read from this space is determined by the 14-bit TCB byte count. This counter indicates the number of bytes that will be transmitted from the transmit command block, starting with the third byte after the TCB count field (address N + 10h). The TCB count field can be any number of bytes up to a maximum of 2600, which allows the user to transmit a frame with a header having an odd number of bytes. In simplified mode, the TCB byte count indicates the total number of bytes to be transmitted and should not equal zero. In flexible mode, if the TCB byte count equals 0, then all data is taken from the buffers pointed to by the TBD array.

. Transmit Buffer Descriptor

Odd Word (Bits 31:16)		Even Word (Bits 15:0)		
Transmit Buffer #0 Address				0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	EL	0	Size (Actual Count)	4
Transmit Buffer #1 Address				8
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	EL	0	Size (Actual Count)	C
Transmit Buffer #N Address				N*8
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	EL	0	Size (Actual Count)	N*8+4

Transmit Buffer #N

This is the starting address of the memory area that contains the data to be sent. It is an absolute 32-bit address. It does not add the CU base value to determine the physical address.

EL (End of List)

The EL bit is not used by the 82557 and is only valid for 82558 and later generation devices. When it is set, the TBD is the last TBD associated with this transmit frame.

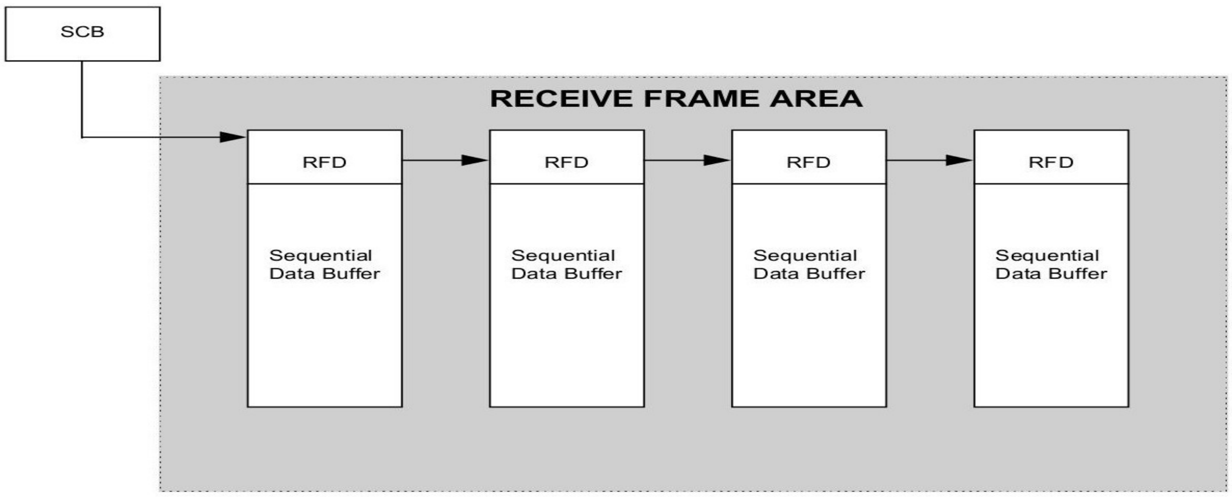
Size (Actual Count)

This 14-bit quantity specifies the number of bytes that hold information for the current buffer. It is set by the CPU before transmission.

Receive Operation

In the simplified RFA structure, the data portion of the received frame (including the Ethernet header) is part of the RFD and is located in contiguous memory immediately after the size field in the RFD. The simplified memory structure is shown in the figure below.

Simplified Memory Structure



. **Receive Frame Descriptor Format**

Offset	Command Word Bits 31:16						Status Word Bits 15:0			
00h	EL	S	000000000	H	SF	000	C	0	OK	Status Bits
04h	Link Address (A31:A0)									
08h	Reserved									
0Ch	0	0	Size				EOF	F	Actual Count	

EL (Bit 31)	The EL bit indicates that this RFD is the last one in the RFA.
S (Bit 30)	The S bit suspends the RU after receiving the frame.
H (Bit 20)	The H bit indicates if the current RFD is a header RFD. If it equals 1, the current RFD is a header RFD, and if it is 0, it is not a header RFD. **NOTE:** If a load HDS command was not previously issued, the device disregards this bit.
SF (Bit 19)	The SF bit equals 0 for simplified mode.
C (Bit 15)	This bit indicates the completion of frame reception. It is set by the device.
OK (Bit 13)	The OK bit indicates whether the frame was received without any errors and stored in memory. If the last frame was received with sufficient memory space, the OK bit will be set, even if it was the last RFD in the RFA with the EL bit set. After receiving the frame, the device enters the no resource condition, generates an RNR interrupt, and starts discarding frames until the RU is restarted with sufficient resources.
Status Bits (Bits 12:0)	This field contains the results of the receive operation:
Link Address	The link address is a 32-bit offset to the next RFD. It is added to the RU base. The link address of the last frame can be used to form a cyclical link to the first RFD.
Size	This field is used in the simplified mode and represents the data buffer size. In the header RFD, the size field identifies the data buffer size excluding the header area. The size value should be an even number.
EOF	This bit is set by the device when it has completed placing data in the data area. Before a new RFD can be included in the RFA, the EOF bit must be cleared by software.
F	This bit is set by the device when it updates the actual count field. Before a new RFD can be included in the RFA, the F bit must be cleared by software.
Actual Count	The number of bytes written into the data area.

Initial Receive Frame Area Structure

To enable the device to receive frames, software must setup the following structure:
1. The SCB general pointer in the SCB should point to the first RFD on the list.
2. The link offset of each RFD in the list should point to the next RFD.
3. The EL bit in the last RFD should be set.

More About Command Unit and Receive Unit

Software can issue control commands by writing to the RUC and CUC fields of the SCB command word. The SCB CU and RU command fields are two fields in the lower byte of the SCB command word, called the SCB command byte. Since the 8255x clears the SCB command byte when the control command is accepted:

- Software must wait for this byte to be cleared before the next control command can be issued.
- CU and RU control commands must never be issued together in the same SCB write cycle.

States Of Command Unit

The CU can be modeled as a logical machine that exists in one of the following states at any given time:

- Idle. The CU is currently not executing an action command and is not associated with a CB in the CBL. This is the initial state. It is also the state reached after the CU finishes executing a CBL where the last CB had an EL bit set. A CU start command must be issued to begin execution on a new CBL.
- Suspended. The CU is not executing a CB but has read a next link pointer in the last CB that it executed before it suspended execution. A CU resume command forces the 8255x to continue execution from the CB at the next link address.
- Active. The CU is currently executing an action command.

States Of Receive Unit

The RU is modeled as a logical machine that takes one of the following states at any given time. Software can determine the current RU status by reading the SCB status word in the CSR (bits 5:2).

- Idle (0000). The RU has no memory resources and is discarding incoming frames. This is the initial RU state after reset.
- No Resources Due to No More RFDs (0010). The RU has no memory resources due to a lack of RFDs and is discarding incoming frames. This state differs from the idle state in that the RU accumulates statistics on the number of frames it has to discard. The 8255x enters this state after it processes an RFD that its EL bit set.
- Suspended (0001). The RU discards all incoming frames even though free memory resources exist to store incoming frames. The 8255x enters this state after it processes an RFD with its S bit set.
- Ready (0100). The RU has free memory resources and is ready to store incoming frames.

四、e100.c 操作說明

Driver Operation

Memory-mapped mode is used exclusively to access the device's shared-memory structure, the Control/Status Registers (CSR). All setup, configuration, and control of the device, including queuing of Tx, Rx, and configuration commands is through the CSR. cmd_lock serializes accesses to the CSR command register. cb_lock protects the shared Command Block List (CBL).

Transmit

A Tx skb is mapped and hangs off of a TCB. TCBs are linked together in a fixed-size ring (CBL) thus forming the flexible mode memory structure. A TCB marked with the suspend-bit indicates the end of the ring. The last TCB processed suspends the controller, and the controller can be restarted by issue a CU resume command to continue from the suspend point, or a CU start command to start at a given position in the ring.

Non-Tx commands (config, multicast setup, etc) are linked into the CBL ring along with Tx commands. The common structure used for both Tx and non-Tx commands is the Command Block (CB).

cb_to_use is the next CB to use for queuing a command; cb_to_clean is the next CB to check for completion; cb_to_send is the first CB to start on in case of a previous failure to resume. CB clean up happens in interrupt context in response to a CU interrupt. cbs_avail keeps track of number of free CB resources available.

Receive

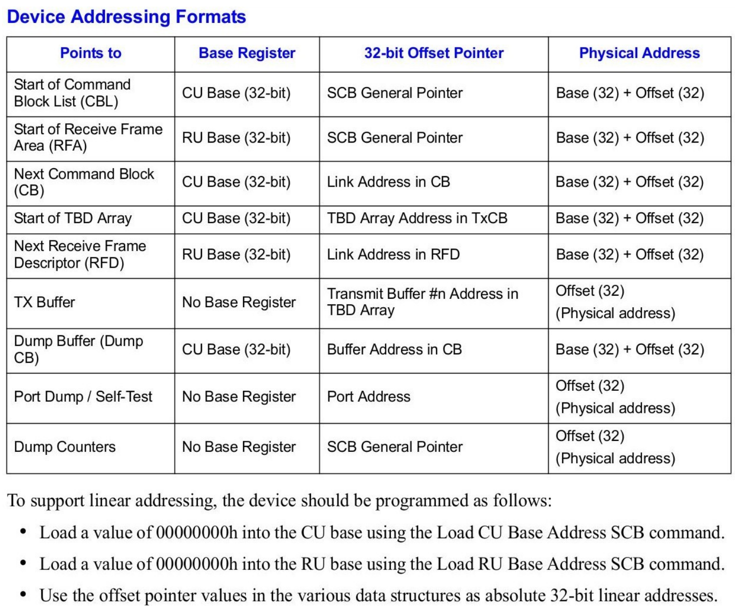
The Receive Frame Area (RFA) comprises a ring of Receive Frame Descriptors (RFD) + data buffer, thus forming the simplified mode memory structure. Rx skbs are allocated to contain both the RFD and the data buffer, but the RFD is pulled off before the skb is indicated. The data buffer is aligned such that encapsulated protocol headers are u32-aligned. Since the RFD is part of the mapped shared memory, and completion status is contained within the RFD, the RFD must be dma_sync'ed to maintain a consistent view from software and hardware.

In order to keep updates to the RFD link field from colliding with hardware writes to mark packets complete, we use the feature that hardware will not write to a size 0 descriptor and mark the previous packet as end-of-list (EL).   After updating the link, we remove EL and only then restore the size such that hardware may use the previous-to-end RFD.

Under typical operation, the receive unit (RU) is start once, and the controller happily fills RFDs as frames arrive. If replacement RFDs cannot be allocated, or the RU goes non-active, the RU must be restarted. Frame arrival generates an interrupt, and Rx indication and re-allocation happen in the same context, therefore no locking is required. A software-generated interrupt is generated from the watchdog to recover from a failed allocation scenario where all Rx resources have been indicated and none replaced.

五、e100.c 之資料結構宣告及基礎操作

CSR (Control/Status Registers) declarations



Device Addressing Formats

Points to	Base Register	32-bit Offset Pointer	Physical Address
Start of Command Block List (CBL)	CU Base (32-bit)	SCB General Pointer	Base (32) + Offset (32)
Start of Receive Frame Area (RFA)	RU Base (32-bit)	SCB General Pointer	Base (32) + Offset (32)
Next Command Block (CB)	CU Base (32-bit)	Link Address in CB	Base (32) + Offset (32)
Start of TBD Array	CU Base (32-bit)	TBD Array Address in TxCB	Base (32) + Offset (32)
Next Receive Frame Descriptor (RFD)	RU Base (32-bit)	Link Address in RFD	Base (32) + Offset (32)
TX Buffer	No Base Register	Transmit Buffer #n Address in TBD Array	Offset (32) (Physical address)
Dump Buffer (Dump CB)	CU Base (32-bit)	Buffer Address in CB	Base (32) + Offset (32)
Port Dump / Self-Test	No Base Register	Port Address	Offset (32) (Physical address)
Dump Counters	No Base Register	SCB General Pointer	Offset (32) (Physical address)

To support linear addressing, the device should be programmed as follows:

- Load a value of 00000000h into the CU base using the Load CU Base Address SCB command.
- Load a value of 00000000h into the RU base using the Load RU Base Address SCB command.
- Use the offset pointer values in the various data structures as absolute 32-bit linear addresses.

```
struct csr {
        struct {
                u8 status;       // CU, RU 狀態
                u8 stat_ack;     // 收到中斷的原因
                u8 cmd_lo;       // CU, RU 命令
                u8 cmd_hi;       // 中斷控制 & 遮罩
                u32 gen_ptr;     // CU, RU 參數
        } scb;
        u32 port;
        u16 flash_ctrl;
        u8 eeprom_ctrl_lo;
        u8 eeprom_ctrl_hi;
        u32 mdi_ctrl;
        u32 rx_dma_count;
};

enum scb_status {                // RU 的幾個可能狀態
        rus_no_res      = 0x08,
        rus_ready       = 0x10,
        rus_mask        = 0x3C,
};

enum scb_stat_ack {              // 中斷的幾個可能情況
        stat_ack_not_ours    = 0x00,
        stat_ack_sw_gen      = 0x04,        // 軟體觸發中斷
        stat_ack_rnr         = 0x10,        // 接收資源不足
        stat_ack_cu_idle     = 0x20,        // CU 進入 idle 或 suspended 狀態(CNA interrupt)
        stat_ack_frame_rx    = 0x40,        // RU 收到一個 frame
        stat_ack_cu_cmd_done = 0x80,        // CU 完成了一個要求中斷(I bit set)的 cb
        stat_ack_not_present = 0xFF,
        stat_ack_rx = (stat_ack_sw_gen | stat_ack_rnr | stat_ack_frame_rx),
        stat_ack_tx = (stat_ack_cu_idle | stat_ack_cu_cmd_done),
        // 上面 2 個是 RX 或 TX 時的幾個中斷可能總集合，但程式中未用到
};

enum scb_cmd_hi {
        irq_mask_none = 0x00,
        irq_mask_all  = 0x01,               // 遮罩所有中斷
        irq_sw_gen    = 0x02,               // 產生軟體中斷
};

enum scb_cmd_lo {                           // SCB 各種可能命令
        cuc_nop       = 0x00,
        ruc_start     = 0x01,
        ruc_load_base  = 0x06,
        cuc_start     = 0x10,
        cuc_resume     = 0x20,
        cuc_dump_addr  = 0x40,
        cuc_dump_stats = 0x50,
        cuc_load_base  = 0x60,
        cuc_dump_reset = 0x70,
};
```

CB (Command Block) declarations

```
struct cb {
        __le16 status;                  // 命令執行結果
        __le16 command;                 // 命令及 CU 控制(suspend, interrupt, end of list, etc)
        __le32 link;                    // 指向下個 CB
        union {
                u8 iaaddr[ETH_ALEN];            // 設網卡 MAC address 命令專用
                struct config config;           // 設定命令專用，內容可參考 i8255x 手冊 62 頁
                struct multi multi;             // multicast 設定命令專用
                struct {                        // 傳送命令專用
                        u32 tbd_array;          // TBD 陣列在實體記憶體位址
                        u16 tcb_byte_count;     // skb 資料如跟在 cb 後,要傳送的 bytes(不用,設 0)
                        u8 threshold;           // 開始傳送條件：transmit FIFO 最少可用空間
                        u8 tbd_count;           // TBD 陣列元素個數
                        struct {                // 就一個 TBD 直接放在 CB 裡
                                __le32 buf_addr;        // 被傳送資料(skb->data)的實體記憶體位址
                                __le16 size;            // 被傳送資料長度(skb->len)
                                u16 eol;                // 此 CB 中最後一個 TBD?
                        } tbd;
                } tcb;
                __le32 dump_buffer_addr;
        } u;
                                                // 以下與 i8255x 無關,單純 e100.c 內部使用
        struct cb *next, *prev;                 // 串接所有的 CB
        dma_addr_t dma_addr;                    // CB 頭的實體記憶體位址
        struct sk_buff *skb;                    // 被傳送的 skb(非 TxCB 時為 NULL)
};

enum cb_status {                                // CB 執行結果
        cb_complete = 0x8000,                   // 傳送成功時這 2 個 flags 都會設起來
        cb_ok       = 0x2000,
};

enum cb_command {                               // CB command word
        cb_nop    = 0x0000,                     // 無作用命令
        cb_iaaddr = 0x0001,                     // 設定本網卡 MAC address
        cb_config = 0x0002,                     // 設定各種參數
        cb_multi  = 0x0003,                     // 設定 multicast addresses
        cb_tx     = 0x0004,                     // 傳送
        cb_ucode  = 0x0005,                     // 載入 microcode
        cb_dump   = 0x0006,                     // dump 內部 registers 值到 memory
                                                // 此行以下(含)為 CB 命令可帶的 flags
        cb_tx_sf  = 0x0008,                     // 0: 傳送資料在 TCB 裡(simplified mode)
        cb_tx_nc  = 0x0010,                  // 0: controller does CRC (normal)
        cb_cid    = 0x1f00,                  // CNA 中斷延遲
        cb_i      = 0x2000,                  // 命令執行後產生中斷
        cb_s      = 0x4000,                  // 命令執行後,CU 進入 suspended 狀態
                                             // (根據 config,可產生 CNA interrupt)
        cb_el     = 0x8000,                  // 表示是 CBL 裡最後一個 CB,命令執行後,
                                             // CU 進入 idle 狀態,並發 CNA/CI interrupt
};
```

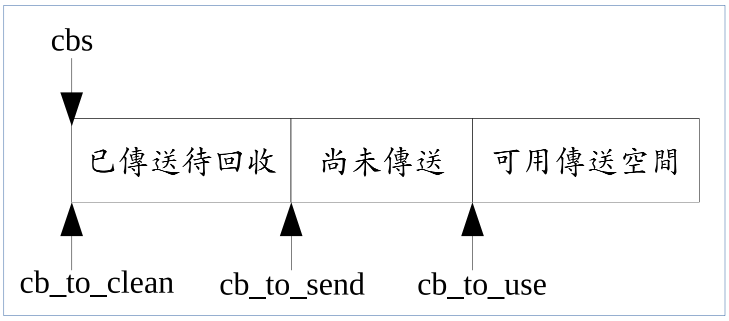
RFD (Receive Frame Descriptor) declarations

```
struct rfd {
        __le16 status;                       // 接收資料狀態  (cb_complete & cb_ok)
        __le16 command;                      // RU 控制(suspend, end of list)
        __le32 link;                         // 下一個  RFD 實體記憶體位址
        __le32 rbd;                          // Reserved
        __le16 actual_size;                  // 實際收到的資料 bytes 數
        __le16 size;                         // 接在 RFD 後之 data buffer size
};

struct rx {                                  // 包裝並串起所有 Rx-skb
        struct rx *next, *prev;
        struct sk_buff *skb;                 // 存放 rfd+ethernet frame
        dma_addr_t dma_addr;                 // skb->data 的實體記憶體位址
};
```

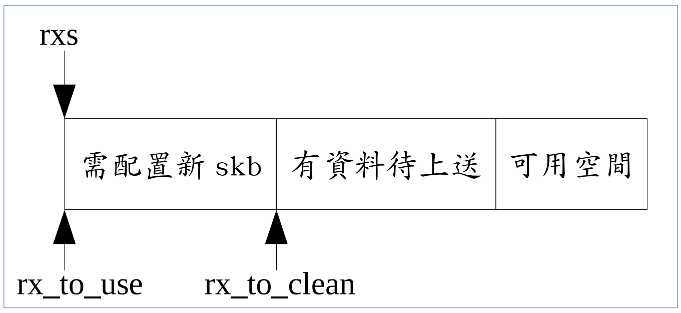
執行 SCB Commands 及 Action Commands

```
#define E100_WAIT_SCB_TIMEOUT 20000 /* we might have to wait 100ms!!! */
#define E100_WAIT_SCB_FAST 20       /* delay like the old code */
static int e100_exec_cmd(struct nic *nic, u8 cmd, dma_addr_t dma_addr)
{
        unsigned long flags;
        unsigned int i;
        int err = 0;

        spin_lock_irqsave(&nic->cmd_lock, flags);    //序列化 SCB 存取(禁止中斷,存下 IRQ 設定)

        /* Previous command is accepted when SCB clears */
        for (i = 0; i < E100_WAIT_SCB_TIMEOUT; i++) {
                if (likely(!ioread8(&nic->csr->scb.cmd_lo)))
                        break;
                cpu_relax();
                if (unlikely(i > E100_WAIT_SCB_FAST))
                        udelay(5);
        }
        if (unlikely(i == E100_WAIT_SCB_TIMEOUT)) {
                err = -EAGAIN;
                goto err_unlock;
```

```
	}

	if (unlikely(cmd != cuc_resume))
		iowrite32(dma_addr, &nic->csr->scb.gen_ptr);
	iowrite8(cmd, &nic->csr->scb.cmd_lo);

err_unlock:
	spin_unlock_irqrestore(&nic->cmd_lock, flags);

	return err;
}

static int e100_exec_cb(struct nic *nic, struct sk_buff *skb,
	int (*cb_prepare)(struct nic *, struct cb *, struct sk_buff *))
{
	struct cb *cb;
	unsigned long flags;
	int err = 0;

	spin_lock_irqsave(&nic->cb_lock, flags);

	if (unlikely(!nic->cbs_avail)) {
		err = -ENOMEM;
		goto err_unlock;
	}

	cb = nic->cb_to_use;
	nic->cb_to_use = cb->next;
	nic->cbs_avail--;
	cb->skb = skb;
```



```
	err = cb_prepare(nic, cb, skb);		// 準備好 cb 內容(如:cb->command, cb->tcb)
	if (err)
		goto err_unlock;

	if (unlikely(!nic->cbs_avail))
		err = -ENOSPC;

	/* Order is important otherwise we'll be in a race with h/w:
	 * set S-bit in current first, then clear S-bit in previous. */
	cb->command |= cpu_to_le16(cb_s);
	wmb();
	cb->prev->command &= cpu_to_le16(~cb_s);

	while (nic->cb_to_send != nic->cb_to_use) {
		if (unlikely(e100_exec_cmd(nic, nic->cuc_cmd,
			nic->cb_to_send->dma_addr))) {
			/* Ok, here's where things get sticky.  It's
			 * possible that we can't schedule the command
			 * because the controller is too busy, so
			 * let's just queue the command and try again
			 * when another command is scheduled. */
				if (err == -ENOSPC) {
					//request a reset
					schedule_work(&nic->tx_timeout_task);
				}
				break;
			} else {
				nic->cuc_cmd = cuc_resume;
				nic->cb_to_send = nic->cb_to_send->next;
			}
		}

err_unlock:
		spin_unlock_irqrestore(&nic->cb_lock, flags);

		return err;
}
```



nic: netdev_priv of e100

```
struct nic {
	struct net_device *netdev;
	struct pci_dev *pdev;
	struct rx *rxs;			// pointer to an array of struct rx
	struct rx *rx_to_use;
	struct rx *rx_to_clean;
	struct rfd blank_rfd;		// rfd 初始化範本
	enum ru_state ru_running;	// 目前 RU 狀態
	spinlock_t cb_lock;
	spinlock_t cmd_lock;
	struct csr __iomem *csr;	// CSR 在虛擬定址空間的記憶體位址
	enum scb_cmd_lo cuc_cmd; // 下回發送的 CU 命令
	unsigned int cbs_avail;		// 可用 cb 剩餘數量
	struct napi_struct napi;	// structure for NAPI scheduling
	struct cb *cbs;			// pointer to an array of CB
	struct cb *cb_to_use;		// 指向第一個可用 CB
	struct cb *cb_to_send;		// 指向第一個待執行 CB
	struct cb *cb_to_clean;		// 指向第一個待回收  CB
	__le16 tx_command;		// 傳送命令(cb_tx|cb_tx_sf),用來初始化 cb->command
	struct timer_list watchdog;	// carrier detection & statistics update
	dma_addr_t cbs_dma_addr;	// cbs 陣列實體記憶體位址
};
```

六、e100裝置偵測

```
static const struct net_device_ops e100_netdev_ops = {
	.ndo_open		= e100_open,
	.ndo_stop		= e100_close,
	.ndo_start_xmit		= e100_xmit_frame,
	.ndo_validate_addr	= eth_validate_addr,
	.ndo_set_rx_mode	= e100_set_multicast_list,
	.ndo_set_mac_address= e100_set_mac_address,
	.ndo_change_mtu		= e100_change_mtu,
	.ndo_do_ioctl		= e100_do_ioctl,
	.ndo_tx_timeout		= e100_tx_timeout,
};

static void e100_get_defaults(struct nic *nic)
{
	/* no interrupt for every tx completion, delay=256us (Manual: delayed CNA interrupt) */
	nic->tx_command = cpu_to_le16(cb_tx | cb_tx_sf | cb_cid);

	/* Template for a freshly allocated RFD */
	nic->blank_rfd.command = 0;
	nic->blank_rfd.rbd = cpu_to_le32(0xFFFFFFFF);
	nic->blank_rfd.size = cpu_to_le16(VLAN_ETH_FRAME_LEN + ETH_FCS_LEN);
}

static int e100_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
	struct net_device *netdev;
	struct nic *nic;
	int err;

	if (!(netdev = alloc_etherdev(sizeof(struct nic))))
		return -ENOMEM;

	netdev->netdev_ops = &e100_netdev_ops;
	netdev->watchdog_timeo = E100_WATCHDOG_PERIOD;
	nic = netdev_priv(netdev);
	netif_napi_add(netdev, &nic->napi, e100_poll, E100_NAPI_WEIGHT);
	nic->netdev = netdev;
	nic->pdev = pdev;

	if ((err = pci_enable_device(pdev)))
		goto err_out_free_dev;

	if (!(pci_resource_flags(pdev, 0) & IORESOURCE_MEM)) {
		err = -ENODEV;
		goto err_out_disable_pdev;
	}
	if ((err = pci_request_regions(pdev, DRV_NAME)))
		goto err_out_disable_pdev;
```

```
/* ether_setup - setup Ethernet network device
 * @dev: network device
 *
 * Fill the device structure with Ethernet-generic values.
 */
void ether_setup(struct net_device *dev)
{
    dev->header_ops      = &eth_header_ops;
    dev->type            = ARPHRD_ETHER;
    dev->hard_header_len = ETH_HLEN;
    dev->mtu             = ETH_DATA_LEN;
    dev->addr_len        = ETH_ALEN;
    dev->tx_queue_len    = 1000; /* Ethernet wants good queues */
    dev->flags           = IFF_BROADCAST|IFF_MULTICAST;
    dev->priv_flags     |= IFF_TX_SKB_SHARING;
    memset(dev->broadcast, 0xFF, ETH_ALEN);
}
```

```
netif_napi_add() must be used to initialize a napi context
prior to calling *any* of the other napi related functions.
```

```
Mark all PCI regions associated with
the PCI device as reserved
```

```
	nic->csr = pci_iomap(pdev, (use_io ? 1 : 0), sizeof(struct csr));
	if (!nic->csr) {
		err = -ENOMEM;
		goto err_out_free_res;
	}

	e100_get_defaults(nic);
```

```
pci_iomap - create a virtual mapping cookie for a PCI BAR
@dev: PCI device that owns the BAR
@bar: BAR number
@maxlen: length of the memory to map

Using this function you will get a __iomem address to your device BAR.
You can access it using ioread*() and iowrite*(). These functions hide
the details if this is a MMIO or PIO address space and will just do what
you expect from them in the correct way.

@maxlen specifies the maximum length to map. If you want to get access to
the complete BAR without checking for its length first, pass %0 here.
```

```
	/* locks must be initialized before calling hw_reset */
	spin_lock_init(&nic->cb_lock);
	spin_lock_init(&nic->cmd_lock);

	/* Reset the device before pci_set_master() in case device is in some
	 * funky state and has an interrupt pending - hint: we don't have the
	 * interrupt handler registered yet. */
	e100_hw_reset(nic);		// use SCB port interface to reset
	pci_set_master(pdev);		// enable device bus mastering

	init_timer(&nic->watchdog);
	nic->watchdog.function = e100_watchdog;
	nic->watchdog.data = (unsigned long)nic;
	INIT_WORK(&nic->tx_timeout_task, e100_tx_timeout_task);

	if ((err = e100_eeprom_load(nic)))	// 載入 EEPROM 內容
		goto err_out_iounmap;
	e100_phy_init(nic);			// 用 MII 介面初始化 PHY
	memcpy(netdev->dev_addr, nic->eeprom, ETH_ALEN); // 取得 EEPROM 上 MAC address

	if ((err = register_netdev(netdev)))
		goto err_out_iounmap;

	return 0;

err_out_iounmap:
	pci_iounmap(pdev, nic->csr);
err_out_free_res:
	pci_release_regions(pdev);
err_out_disable_pdev:
	pci_disable_device(pdev);
err_out_free_dev:
	free_netdev(netdev);
	return err;
}
```

七、網路介面開啓和關閉（實作）

實驗根目錄下有下列 3 個 e100.c:

```
e100.c→linux-3.18.14/drivers/net/ethernet/intel/e100.c   // 方便你直接修改 kernel 内的 e100.c
e100.lab.c                // 實作使用的 e100.c, 其中需要實作的函式内容已移除
e100.orig.c               // 原始 e100.c (實作時請不要打開來看)
```

修改完 e100.c，請執行以下指令，便能重啓 QEMU 驗證你的程式。

```
1. make -j4 -C linux-3.18.14/          // 重新 compile e100.c
2. ./install-kmod-to-rootfs.sh         // 將 e100.ko copy 到 rootfs 下
3. ./pack-rootfs.sh                    // 重新打包 rootfs.cpio.gz
4. sudo ./start-qemu.sh                // 請先關掉已開始的 QEMU
```

參考下列原始碼及註解，完成 e100_up()及 e100_down()這兩個實際負責介面開關的函式。

```
// 配置 rxs 陣列(=Rx ring)來追蹤所有作為接收封包的空間
static int e100_rx_alloc_list(struct nic *nic)
{
	struct rx *rx;
	unsigned int i, count = 256;
	struct rfd *before_last;

	nic->rx_to_use = nic->rx_to_clean = NULL;
	nic->ru_running = RU_UNINITIALIZED;

	if (!(nic->rxs = kcalloc(count, sizeof(struct rx), GFP_ATOMIC)))
		return -ENOMEM;
	for (rx = nic->rxs, i = 0; i < count; rx++, i++) {
		rx->next = (i + 1 < count) ? rx + 1 : nic->rxs;
		rx->prev = (i == 0) ? nic->rxs + count - 1 : rx - 1;
		if (e100_rx_alloc_skb(nic, rx)) {
			e100_rx_clean_list(nic);
			return -ENOMEM;
		}
	}
	/* Set the el-bit on the buffer that is before the last buffer.
	 * This lets us update the next pointer on the last buffer without
	 * worrying about hardware touching it.
	 * We set the size to 0 to prevent hardware from touching this buffer.
	 * When the hardware hits the before last buffer with el-bit and size
	 * of 0, it will RNR interrupt, the RU will go into the No Resources
	 * state.  It will not complete nor write to this buffer. */
	rx = nic->rxs->prev->prev;
	before_last = (struct rfd *)rx->skb->data;
	before_last->command |= cpu_to_le16(cb_el);
	before_last->size = 0;
	pci_dma_sync_single_for_device(nic->pdev, rx->dma_addr,
		sizeof(struct rfd), PCI_DMA_BIDIRECTIONAL);

	nic->rx_to_use = nic->rx_to_clean = nic->rxs;
	nic->ru_running = RU_SUSPENDED;
```

這函式在 thread context 執行，應用 GFP_KERNEL 就夠了

確保 before_last 的更新寫入記憶體

```
        return 0;
}
```

1518 + 4

```
#define RFD_BUF_LEN (sizeof(struct rfd) + VLAN_ETH_FRAME_LEN + ETH_FCS_LEN)
static int e100_rx_alloc_skb(struct nic *nic, struct rx *rx)
{
        if (!(rx->skb = netdev_alloc_skb_ip_align(nic->netdev, RFD_BUF_LEN)))
                return -ENOMEM;
```

Since an ethernet header is 14 bytes, network drivers often end up with the IP header at an unaligned offset. The IP header can be aligned by shifting the start of the packet by 2 bytes.

```
        /* Init, and map the RFD. */
        skb_copy_to_linear_data(rx->skb, &nic->blank_rfd, sizeof(struct rfd));
        rx->dma_addr = pci_map_single(nic->pdev, rx->skb->data,
                RFD_BUF_LEN, PCI_DMA_BIDIRECTIONAL);

        /* Link the RFD to end of RFA by linking previous RFD to
         * this one.  We are safe to touch the previous RFD because
         * it is protected by the before last buffer's el bit being set */
        if (rx->prev->skb) {
                struct rfd *prev_rfd = (struct rfd *)rx->prev->skb->data;
                put_unaligned_le32(rx->dma_addr, &prev_rfd->link);
                pci_dma_sync_single_for_device(nic->pdev, rx->prev->dma_addr,
                        sizeof(struct rfd), PCI_DMA_BIDIRECTIONAL);
        }

        return 0;
}
```

Copy rx->dma_addr 到 prev_rfd->link;使用 put_unaligned_le32()是因爲 rx->skb->data 起點向後推 2 bytes,使得 prev_rfd->link 沒有 align 在 32-bit boundary 上

```
// 配置 cb 陣列(=Tx ring)來儲存待處理的命令
static int e100_alloc_cbs(struct nic *nic)
{
        struct cb *cb;
        unsigned int i, count = 128;

        nic->cuc_cmd = cuc_start;     // 下次 e100_exec_cb()的命令
        nic->cb_to_use = nic->cb_to_send = nic->cb_to_clean = NULL;
        nic->cbs_avail = 0;
```

底層使用 dma_alloc_coherent()

```
        nic->cbs = pci_pool_alloc(nic->cbs_pool, GFP_KERNEL, &nic->cbs_dma_addr);
        if (!nic->cbs)
                return -ENOMEM;
        memset(nic->cbs, 0, count * sizeof(struct cb));

        for (cb = nic->cbs, i = 0; i < count; cb++, i++) {
                cb->next = (i + 1 < count) ? cb + 1 : nic->cbs;
                cb->prev = (i == 0) ? nic->cbs + count - 1 : cb - 1;

                cb->dma_addr = nic->cbs_dma_addr + i * sizeof(struct cb);
                cb->link = cpu_to_le32(nic->cbs_dma_addr +
                        ((i+1) % count) * sizeof(struct cb));
        }
        nic->cb_to_use = nic->cb_to_send = nic->cb_to_clean = nic->cbs;
        nic->cbs_avail = count;

        return 0;
}

static void e100_disable_irq(struct nic *nic)
{
        unsigned long flags;
        spin_lock_irqsave(&nic->cmd_lock, flags);
        iowrite8(irq_mask_all, &nic->csr->scb.cmd_hi);
        e100_write_flush(nic);
        spin_unlock_irqrestore(&nic->cmd_lock, flags);
}
```

```
static void e100_disable_irq(struct nic *nic)
{
        unsigned long flags;
        spin_lock_irqsave(&nic->cmd_lock, flags);
        iowrite8(irq_mask_all, &nic->csr->scb.cmd_hi);
        e100_write_flush(nic);
        spin_unlock_irqrestore(&nic->cmd_lock, flags);
}
```

```
static int e100_hw_init(struct nic *nic)
{
        int err = 0;

        e100_hw_reset(nic);

        if (!in_interrupt() && (err = e100_self_test(nic)))  // no need to check in_interrupt()?
                return err;

        if ((err = e100_phy_init(nic)))
                return err;
        if ((err = e100_exec_cmd(nic, cuc_load_base, 0)))
                return err;
        if ((err = e100_exec_cmd(nic, ruc_load_base, 0)))
                return err;
        if ((err = e100_exec_cb(nic, NULL, e100_configure)))
                return err;
        if ((err = e100_exec_cb(nic, NULL, e100_setup_iaaddr))) // 設定 i8255x 的 MAC address
                return err;

        e100_disable_irq(nic);          // 暫時禁止  e100 發中斷
        return 0;
}
```

大部份 CU, RU 命令資料(如:TxCB, RFD)中的 link/pointer 只是 offset,需加上 base 後才是它實體記憶體中的位址.這裡把 base 設爲 0, link/pointer 就可以直接使用實體記憶體位址.

各種參數設定,如:promiscuous mode, checksum offloading.

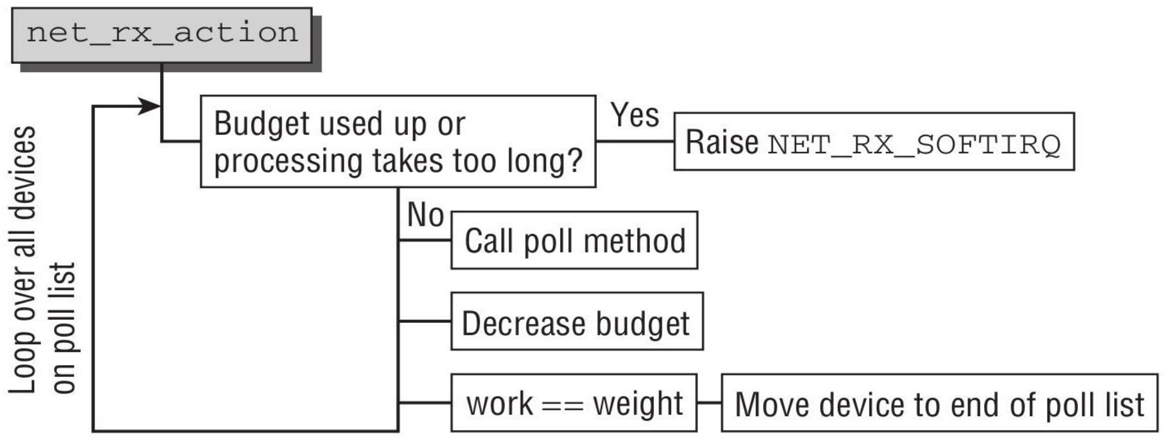
```
// 啓動或回復 RU 運作:開始接收封包
static inline void e100_start_receiver(struct nic *nic, struct rx *rx)
{
        if (!nic->rxs) return;
        if (RU_SUSPENDED != nic->ru_running) return;

        /* handle init time starts */
        if (!rx) rx = nic->rxs;

        /* (Re)start RU if suspended or idle and RFA is non-NULL */
        if (rx->skb) {
                e100_exec_cmd(nic, ruc_start, rx->dma_addr);
                nic->ru_running = RU_RUNNING;
        }
}

static int e100_up(struct nic *nic)
{
        /* 你來完成這部份的程式
         *
         * 1. 配置空間給 Rx ring & Tx ring
         * 2. e100 硬體初始化
         * 3. 啓動 RU
         * 4. 讓 nic->watchdog timer 開始(mod_timer) // 它負責 carrier detection
         * 5. 向核心登記(request_irq())中斷處理函式 e100_intr()
         * 6. 啓動網路介面的傳送佇列(netif_wake_queue()):通知上層可以開始向 e100 傳送資料了
         * 7. 啓用 NAPI (napi_enable())
         * 8. 啓用 e100 中斷(放最後，避免 NAPI 還未啓用中斷就觸發了)
         */
}

static void e100_down(struct nic *nic)
{
        /* 你來完成這部份的程式
         * 關閉介面順序基本上與啓動相反，優先解除所有非同步的背景活動  (注意 race condition)
         *
         * 1. 禁用 NAPI(napi_disable(),它會等待 poll 完成)
         * 2. 停止網路介面的傳送佇列(netif_stop_queue())
         * 3. 取消 nic->watchdog timer 並將 carrier 設為 off(netif_carrier_off())
         * 4. 重置 e100 並禁止其中斷(e100_hw_reset())
         * 5. 向核心取消登記中斷處理函式(free_irq())
         * 6. 回收 Tx ring & Rx ring 空間
         */
}
```

完成後，在 QEMU 裡執行看看，如果中斷函式有正確掛載，你應該會看到許多“e100_intr”出現在畫面上。如果 nic->watchdog timer 有在運作，dmesg 裡應找得到“NIC Link is Up...”。如果畫面出現太多 kernel messages，可 echo 0 > /proc/sys/kernel/printk，禁止 kernel messages 寫到 console 上，但你還是可以透過 dmesg 查看。

八、中斷處理與輪詢（實作）

傳統的網路驅動程式，是網路介面在收到每個封包後，就發一個中斷通知 CPU 來處理。這樣的做法，在高速且繁忙的網路環境下，對 CPU 效能影響很大；因為 CPU 被迫高頻率的暫停正在執行的工作，去處理這些中斷。如果網路介面有能力暫存一定數量的封包且允許禁用 Rx 中斷，採輪詢(polling)方式可以有效改善網路繁忙時的 CPU 效率。輪詢傳統上被認為沒有效率，那只限於高頻的輪詢且沒有資料可以處理。現今許多 Linux 網路驅動程式，都採用中斷搭配輪詢的方式。

做法上，網路中斷處理函式在收到 Rx 中斷後，即啓動輪詢模式，並禁止 Rx 中斷再發生。核心會不斷呼叫驅動程式的輪詢函式，直到所有 Rx 封包都處理完畢(如下圖)。輪詢函式在處理完最後一個 Rx 封包後，須通知核心此時不再需要輪詢，並重新允許 Rx 中斷發生。


net_rx_action
Loop over all devices on poll list
Budget used up or processing takes too long?
Yes
Raise NET_RX_SOFTIRQ
No
Call poll method
Decrease budget
work == weight
Move device to end of poll list

你是否有察覺到，輪詢函式是在 softirq context 下執行，它本質上就是中斷處理函式的 bottom half。

下面是驅動程式會用到的輪詢相關函式：

```
// 向核心登記輪詢函式，weight 是每回 poll 處理的封包數量上限
void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
            int (*poll)(struct napi_struct *, int), int weight);
// 檢查這個輪詢單位是否已排程或被禁用
bool napi_schedule_prep(struct napi_struct *n);
// 排程這個輪詢單位。請先用 napi_schedule_prep()檢查目前是否可以呼叫 napi_schedule()
void napi_schedule(struct napi_struct *n);
// 標記這個輪詢單位已完成；當所有 Rx 封包都處理好時使用
void napi_complete(struct napi_struct *n);

static irqreturn_t e100_intr(int irq, void *dev_id)
{
        /* 你來完成這部份的程式。中斷的原因可能有：CB 完成(CX, CNA)，Rx 完成(FR)，
         * Rx 資源不足(RNR)或軟體觸發(SWI)。中斷要求的工作，都留在輪詢時來做。
         *
         * 1. e100 有發中斷嗎？ (nic->csr->scb.stat_ack)
         *    如果這個中斷不是 e100 發的，IRQ_NONE 直接離開
         * 2. 向 scb.stat_ack ack 所有的中斷，以解除中斷訊號(de-assert interrupt line)
         * 3. 如中斷原因是 RU 資源不足(stat_ack_rnr)，將 nic->ru_running 設為 RU_SUSPENDED
         * 4. 如果輪詢還未啓動，禁止 e100 再發出中斷並啓動輪詢
         * 5. 以 IRQ_HANDLED 結束
```

```
           */
}

static int e100_poll(struct napi_struct *napi, int budget)   // budget 是每回 poll 最多可處理的封包量
{
        struct nic *nic = container_of(napi, struct nic, napi);
        unsigned int work_done = 0; // 記錄接收完成的封包數

        e100_rx_clean(nic, &work_done, budget);   // 接收 RFA 上的資料
        e100_tx_clean(nic);                        // 處理已完成的 CB

        if (work_done < budget) {     // 表示 RFA 上的封包都處理完了
                napi_complete(napi);
                e100_enable_irq(nic);
        }

        return work_done;
}
```

完成後，在 QEMU 裡執行看看，如果中斷訊號有正確解除，就不會再看到如下“e100_intr”訊息太多被禁止輸出的訊息了。如果輪詢有被排程，可以看到“e100_tx_clean”訊息出現。

```
e100_intr: 314348 callbacks suppressed
```

九、封包傳送（實作）

ndo_start_xmit()是驅動程式裡負責傳送 sk_buff 的函式，它被上層 netif_tx_lock()保護，在 SMP 環境下不會被同時執行。它送出傳送指令給網路介面後即離開。當網路介面 Tx ring 沒有空間可以儲存更多的傳送資料時，必須呼叫 netif_stop_queue()通知上層不要再呼叫 ndo_start_xmit()。

```
// 準備好一個 TxCB，來傳送 skb
static int e100_xmit_prepare(struct nic *nic, struct cb *cb,
        struct sk_buff *skb)
{
        dma_addr_t dma_addr;
        cb->command = nic->tx_command; // (cb_tx | cb_tx_sf |    cb_cid)

        dma_addr = pci_map_single(nic->pdev,
                                        skb->data, skb->len, PCI_DMA_TODEVICE);
```

這個 skb 不是 e100 自己 alloc 的，它是從網路上層來的，e100 將以 DMA 方式傳送

```
        if (unlikely(skb->no_fcs))      // the last 4 bytes of the SKB payload packet as the CRC
                cb->command |= cpu_to_le16(cb_tx_nc);     // CRC from memory
        else
                cb->command &= ~cpu_to_le16(cb_tx_nc); // CRC inserted by controller

        /* interrupt every 16 packets regardless of delay */
        if ((nic->cbs_avail & ~15) == nic->cbs_avail)
                cb->command |= cpu_to_le16(cb_i);
```

避免大量的 TxCB 完成卻沒有中斷通知 CPU 做 TxCB 回收。一般情況下，只有在處理完 CBL 上最後一個 TxCB (cb_s set)，才會有中斷(CNA)

```
        cb->u.tcb.tbd_array = cb->dma_addr + offsetof(struct cb, u.tcb.tbd);
        cb->u.tcb.tcb_byte_count = 0;          // 被傳送資料沒有直接放在 TCB 裡  (cb_tx_sf)
        cb->u.tcb.tbd_count = 1;               // 只有一個  tbd array element
        cb->u.tcb.tbd.buf_addr = cpu_to_le32(dma_addr);  // dma_addr of skb->data
        cb->u.tcb.tbd.size = cpu_to_le16(skb->len);
        return 0;
}

// e100 傳送封包函式
static netdev_tx_t e100_xmit_frame(struct sk_buff *skb,
                                        struct net_device *netdev)
{
        /* 你來完成這部份的程式
         *
         * 1. 下達傳送指令給 e100(備好一個 TxCB 並呼叫 e100_exec_cb())。
         * 2. 如 e100_exec_cb()回應空間不足，通知網路上層停止傳送封包。
         * 3. 函式 return NETDEV_TX_BUSY 或 NETDEV_TX_OK 結束。
         */
}

// CB 完成後的回收處理函式  (由 e100_poll()呼叫)
static void e100_tx_clean(struct nic *nic)
{
        struct net_device *dev = nic->netdev;
        struct cb *cb;
        int tx_cleaned = 0;     // 是否曾經回收任何 TxCB
        spin_lock(&nic->cb_lock);

        /* Clean CBs marked complete */
        for (cb = nic->cb_to_clean;
            cb->status & cpu_to_le16(cb_complete);
            cb = nic->cb_to_clean = cb->next) {

                /* 你來完成這部份程式。進到這裡的 cb 都是要回收的
                 *
                 * 1. 如果這是一個 TxCB (cb->skb != NULL):
                 *       - 統計資料更新  (dev->stats.tx_packets, dev->stats.tx_bytes)
                 *       - 回收 skb 空間
                 *           - pci_unmap_single(tbd array buffer, …)
                 *           - dev_consume_skb_any()
                 *       - cb->skb 設為 NULL
                 *       - tx_cleaned 設為 1
                 * 2. cb 的 status 重置為 0 (重要! 才能重新使用)
                 * 3. nic->cbs_avail++
                 */
        }

        spin_unlock(&nic->cb_lock);

        /* Recover from running out of Tx resources in xmit_frame */
        if (unlikely(tx_cleaned && netif_queue_stopped(nic->netdev)))
                netif_wake_queue(nic->netdev);

        return;
}
```

CB 陣列空間，從 cb_to_clean 開始，到 cb_to_send 之前，是已下給硬體的 CB。有些 CB 可能尚未被執行。已完成的 CB，它的 cb_complete status 會被硬體設爲 1。

當初在 e100_xmit_prepare()對 skb->data 做了 pci_map_single()

完成後，在 QEMU 裡執行，並

在 guest 執行: ping 10.0.0.1
在 host 執行 : sudo tcpdump -i tap0

tcpdump 可以顯示網路介面接收到的封包資訊。如果 TxCB 有正確下達硬體，可以觀察到 guest 不斷送出 ARP request；但 host 回應的 ARP reply，在未實作 e100 Rx 前，guest 無法收到。

```
23:45:43.334746 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
23:45:43.334761 ARP, Reply 10.0.0.1 is-at 36:f9:c7:87:ce:dc (oui Unknown), length 28
```

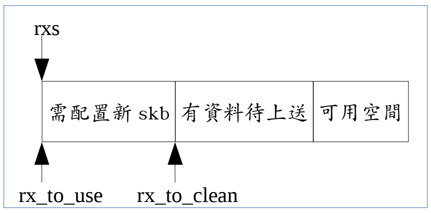
在 guest 執行 ifconfig，如果 e100_tx_clean()有正確回收 TxCB 及更新統計數據，每次 ping 時 ifconfig 的 TX packets 及 TX bytes 數值會增加。

```
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.0.2  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:1278 (1.2 KiB)
```

十、封包接收（實作）

請參考第四節 Receive 部份的說明

```
// RFA 資料回收處理函式(由 e100_poll()呼叫)
static void e100_rx_clean(struct nic *nic, unsigned int *work_done, unsigned int work_to_do)
{
    struct rx *rx;
    int restart_required = 0, err = 0;
    struct rx *old_before_last_rx, *new_before_last_rx;
    struct rfd *old_before_last_rfd, *new_before_last_rfd;
```



```
    /* Indicate newly arrived packets */
    for (rx = nic->rx_to_clean; rx->skb; rx = nic->rx_to_clean = rx->next) {
        err = e100_rx_indicate(nic, rx, work_done, work_to_do);
        /* Hit quota or no more to clean */
        if (-EAGAIN == err || -ENODATA == err)
            break;
    }
```

離開 for 迴圈，只有-EAGAIN 或-ENODATA 兩種情況。不會有 rx->skb==NULL 的情況，會先遇到有 EL bit 的 rfd，然後以-ENODATA 離開。

```
    /* On EAGAIN, hit quota so have more work to do, restart once cleanup is complete.
     * Else, are we already rnr? then pay attention!!! this ensures that the state machine
     * progression never allows a start with a partially cleaned list, avoiding a race between
     * hardware and rx_to_clean when in NAPI mode */
    if (-EAGAIN != err && RU_SUSPENDED == nic->ru_running)
        restart_required = 1;

    old_before_last_rx = nic->rx_to_use->prev->prev;
    old_before_last_rfd = (struct rfd *)old_before_last_rx->skb->data;

    /* Alloc new skbs to refill list */
    for (rx = nic->rx_to_use; !rx->skb; rx = nic->rx_to_use = rx->next) {
        if (unlikely(e100_rx_alloc_skb(nic, rx)))
            break; /* Better luck next time (see watchdog) */
    }

    new_before_last_rx = nic->rx_to_use->prev->prev;
    if (new_before_last_rx != old_before_last_rx) {    // 重新調整 EL bit 位置
        /* Set the el-bit on the buffer that is before the last buffer. This lets us update the next
         * pointer on the last buffer without worrying about hardware touching it. We set the
         * size to 0 to prevent hardware from touching this buffer. When the hardware hits
         * the before last buffer with el-bit and size of 0, it will RNR interrupt, the RUS will
         * go into the No Resources state.  It will not complete nor write to this buffer. */
        new_before_last_rfd = (struct rfd *)new_before_last_rx->skb->data;
        new_before_last_rfd->size = 0;
        new_before_last_rfd->command |= cpu_to_le16(cb_el);
        pci_dma_sync_single_for_device(nic->pdev,
            new_before_last_rx->dma_addr, sizeof(struct rfd),
            PCI_DMA_BIDIRECTIONAL);

        /* Now that we have a new stopping point, we can clear the old stopping point.  We
```

```
                 * must sync twice to get the proper ordering on the hardware side of things. */
                old_before_last_rfd->command &= ~cpu_to_le16(cb_el);
                pci_dma_sync_single_for_device(nic->pdev,
                        old_before_last_rx->dma_addr, sizeof(struct rfd),
                        PCI_DMA_BIDIRECTIONAL);
                old_before_last_rfd->size = cpu_to_le16(VLAN_ETH_FRAME_LEN
                                                        + ETH_FCS_LEN);
                pci_dma_sync_single_for_device(nic->pdev,
                        old_before_last_rx->dma_addr, sizeof(struct rfd),
                        PCI_DMA_BIDIRECTIONAL);
        }

        if (restart_required) { // skb 補充了，EL bit 也調整好了，可以重啓 RU 運作
                e100_start_receiver(nic, nic->rx_to_clean);
                if (work_done)
                        (*work_done)++;
        }
}

static int e100_rx_indicate(struct nic *nic, struct rx *rx,
        unsigned int *work_done, unsigned int work_to_do)
{
        /* 你來完成這部份程式。硬體在接收封包資料到 skb->data 後，會寫入 rfd 以下資訊:
         * status: cb_complete(完成資料接收), cb_ok(資料接收無錯誤), actual_size(實際寫入 byte 數)
         *
         * 1. 如果*work_done >= work_to_do，直接結束 -EAGAIN
         * 2. 偷看一下 rx->skb 裡的 rfd->status 的值，才能決定下一步
         *      - 未做 pci_unmap_single()前，skb->data 屬於 device，要偷看必須先做
         *        pci_dma_sync_single_for_cpu()
         *      - 放一個 rmb()在讀取 rfd status 動作之後，確保取 rfd actual size 值的動作發生在取 rfd
         *        status 值的動作之後(硬體沒下 cb_complete 前，rfd actual size 無意義)
         * 3. 如果 rfd status 沒有 cb_complete (這個 skb 裡沒資料或是代表 end of list)
         *      - 如果這個 rfd 有 EL bit，且我們認為 RU 還在執行，且硬體狀態 scb.status
         *        rus_no_res，將  nic->ru_running 設為 RU_SUSPENDED。
         *      - 以  -ENODATA 結束
         * 4. 取 rfd->actual_size(只有 LSB 14 個 bit 是代表 size)，順便檢查數值是否合理。
         * 5. pci_unmap_single(rx->dma_addr …)
         * 6. 將資料打包在 skb 裡(調整 skb 裡的 data pointer)
         *      - skb 裡封包資料起點在 rfd 之後，rfd 必須去除(用 skb_reserve())
         *      - 『放入』actual_size 的資料在 skb 尾巴  (用 skb_put())
         *      - 告知上層封包的通訊協定 skb->protocol=eth_type_trans(skb, nic->netdev)
         * 7. 如果 rfd status 沒有 cb_ok，回收封包空間(dev_kfree_skb_any())，否則
                - 統計資料更新(stats.rx_packets, stats.rx_bytes)
                - 把封包送到上層去 netif_receive_skb()
                - (*work_done)++
         * 8. rx->skb = NULL; return 0;
         */
}
```

```
/**
 * netif_receive_skb - process receive buffer from network
 * @skb: buffer to process
 *
 * netif_receive_skb() is the main receive data processing function.
 * It always succeeds. The buffer may be dropped during processing
 * for congestion control or by the protocol layers. This function may
 * only be called from softirq context and interrupts should be enabled.
 *
 * Return values (usually ignored):
 * NET_RX_SUCCESS: no congestion
 * NET_RX_DROP: packet was dropped
 */
```

完成後在 QEMU 執行，如果接收實作無誤，host 與 guest 之間的 ping 應可正常運作。