RTL8169

From OSDev Wiki

This page or section is a stub. You can help the wiki by *accurately* contributing (http://wiki.osdev.org/index.php?title=RTL8169&action=edit) to it.

Programming Guide for the RTL8169(S)-32/64 Network Interface Chipsets

The RTL8169 is Realtek's next generation of high-performance network cards. This particular chipset is designed to operate at 10/100/1000 Mbps speeds.

Contents

Basic Startup

Get the MAC Address

First things first, we need to get the NIC's physical MAC address. Normally we should go through the EEPROM to access the MAC address, but in this case we will take a shortcut and obtain the address from the MAC0-6 registers. The MAC registers start at offset 0x00 and should be accessed in 8-bit segments.

ex:

```
char i;
for (i = 0; i < 6; i++)
```

```
{
    mac_address[i] = inportb(ioaddr + i); /*ioaddr is the base address ot
}
```

Reset the Chip

It is necessary to reset the RTL8169 to put the registers in a known default state, as well as begin powering up the chip. We need to send the reset command to Chip Command register at offset 0x37. The reset bit is at offset 0x10, therefore we need to do an 'outportb' using those offsets to tell the chip to begin the resetting process. Once we have sent that command, it will take some time for the chip to complete the operation, and because it is altering registers we should stop all further writes/reads to registers until it has completed. The reset bit in the Command Register will self-clear once the operation is finished, that's what we should check for before proceeding setting up the chip.

ex:

```
outportb(ioaddr + 0x37, 0x10); /*set the Reset bit (0x10) to the Commanc
while(inportb(ioaddr + 0x37) & 0x10)
  ;/*setting a timeout could be useful if the card is problematic*/
```

Setting up the Rx Descriptors

One of the new features introduced with this new-gen chip is that it is completely descriptor-based. The chip supports up to 3 unique descriptor sets with 1024 descriptors available per set; 1024 for receiving packets, 1024 for normal transmission, and 1024 for high-priority transmission packets. It is not required to use all of the descriptors per set, or even use all three sets. Each descriptor is 16-bytes wide and contains a 64-bit buffer address (to store data for transmission or to tell the reception process where to dump the data) and Command uint32_t.

Rx Descriptors are used to tell the NIC where to put packets once received. Because this is an asynchronous operation, we must setup the descriptors before we enable the reception of packets so that the NIC knows where to put things. A major function of descriptors is to tell the NIC which descriptors are owned by the OS (host) and which ones are owned by the NIC. If the descriptor is owned by the NIC, the NIC can then read it and use it for the next packet it receives, if not, it either skips over it or sends a RDU (Rx Descriptor Unavailable) interrupt and halts all further Rx operations. Because there can be multiple descriptors used, there is also a EOR (End of Rx descriptor Ring) bit used when the NIC has reached the end of the descriptors and needs to loop back to the beginning of the descriptors. There is one more big 'gotcha' with all of this, the beginning of the descriptor arrays must be aligned on a 256-byte boundary.

eg:

```
struct Descriptor
{
    unsigned int command,  /* command/status uint32_t */
                 vlan,     /* currently unused */
```

```
                    low_buf,  /* low 32-bits of physical buffer address */
                    high_buf; /* high 32-bits of physical buffer address */
};

struct Descriptor *Rx_Descriptors = (struct Descriptor *)0x100000; /* 1M

void setup_rx_descriptors()
{
    /* rx_buffer_len is the size (in bytes) that is reserved for incomir
    unsigned int OWN = 0x80000000, EOR = 0x40000000; /* Bit offsets */
    int i;
    for(i = 0; i < num_of_rx_descriptors; i++) /* num_of_rx_descriptors
    {
        if(i == (num_of_rx_descriptors - 1)) /* Last descriptor? if so,
          Rx_Descriptors[i].command = (OWN | EOR | (rx_buffer_len & 0x3F
        else
          Rx_Descriptors[i].command = (OWN | (rx_buffer_len & 0x3FFF));
        /* VLAN adjustments are not part of this guide at the moment - l
        Rx_Descriptors[i].low_buf = (unsigned int)&packet_buffer_address
        /* If you are programming for a 64-bit OS, put the high memory l
    }
}
```

Configuring RxConfig and TxConfig

(Un)locking Registers

In order to use some critical registers, we must 'unlock' them to alter their states and then 'lock' them after we are done to prevent accidental alterations to those registers during operation. The 9346CR register is at offset 0x50 and is used as a key to those critical registers. In order to 'unlock' them, we must set the card in "Config register write enable" mode. This is done by setting both the EEM (operating mode) bits. Once that has been done, it is then ok to finish setting up the card. Once configuration is done and the OS begins proceeding as normal, the EEM bits should be set low to set the card in "normal" mode and lock the config registers.

TxConfig

The Transmit Configuration register is at offset 0x40 and contains the configuration data for things such as DMA thresholds and the loopback status. There are only two bits that we should be primarily concerned with, the MXDMA and the IFG bits. The MXDMA bits are used for setting the DMA transfer threshold for transmit bursts. I usually set this to unlimited burst data (one full packet). This can be set to lower if you have proper handling through the host OS. The IFG (Inter-Frame Gap) is the amount of time that elapses between each packet operation. This can be set to the standard (96/960/9600ns) or you may increase the time (for reasons unknown to me) by increments of 8ns.

eg:

```
outportl(ioaddr + 0x40, 0x03000700); /* IFG: normal, MXDMA: unlimited */
```

RxConfig

The Receive Configuration register is at offset 0x44 and contains the configuration data for things such as DMA thresholds, FIFO thresholds, and packet reception rules. The RXFTH bits are used for setting the Rx Fifo Threshold. This threshold determines how much data is required in the buffers to begin telling the host OS that there is data ready. I normally set this to unlimited data as it means that the NIC will tell the OS that data is ready as soon as a full packet is in the on-card FIFO buffer. The MXDMA is based off the same explanation given for the Tx MXDMA. I set the MXDMA to unlimited. The AB/AM/APM/AAP bits are used for setting packet filters. I usually set this to a promiscuous mode with all of the previous bits set to allow the card to receive and process any packets that physically reach the NIC.

eg:

```
outportl(ioaddr + 0x44, 0x0000E70F) /* RXFTH: unlimited, MXDMA: unlimite
```

Max Packet Sizes

Max Transmit Packet Size

The MTPS register is located at offset 0xEC and is used for setting the maximum packet size that can be transmitted from the RTL8169. The maximum value that we can provide here is 0x3B. That value is enough to cover 'jumbo' packets and still allow for a stable FIFO operation (no underruns). This value can be set lower if need be, but for normal usage the maximum value is fine.

eg:

```
outportb(ioaddr + 0xEC, 0x3B); /* Maximum tx packet size */
```

Receive Packet Maximum Size

The RMS register is located at offset 0xDA and is used for setting the maximum packet size that can be received into the NIC. If set to zero, the NIC will accept no packets (must be set prior to normal operation!). The maximum size that can be set is 0x3FFF (16k - 1), but if a packet larger than 8k is received error bits are set and a lengthier process must be started in order to determine if the packet is actually good or not. I keep this value to 0x1FFF (8k - 1) as it is rare to receive such large packets and it also ensures that error packets are actually error packets and not false positives.

eg:

```
outportw(ioaddr + 0xDA, 0x1FFF); /* Maximum rx packet size */
```

Full Reset Example

This is a barebones example of how to reset the RTL8169 without anything like auto-negotiation (explained later) or (G)MII interactions.

eg:

```
struct Descriptor
{
    unsigned int command,  /* command/status uint32_t */
                 vlan,     /* currently unused */
                 low_buf,  /* low 32-bits of physical buffer address */
                 high_buf; /* high 32-bits of physical buffer address */
};

/**
 * Note that this assumes 16*1024=16KB (4 pages) of physical memory at 1
 * the same linear address range
 */
struct Descriptor *Rx_Descriptors = (struct Descriptor *)0x100000; /* 1M
struct Descriptor *Tx_Descriptors = (struct Descriptor *)0x200000; /* 2M

unsigned long num_of_rx_descriptors = 1024, num_of_tx_descriptors = 1024

void setup_rx_descriptors()
{
    /* rx_buffer_len is the size (in bytes) that is reserved for incomir
    unsigned int OWN = 0x80000000, EOR = 0x40000000; /* bit offsets */
    int i;
    for(i = 0; i < num_of_rx_descriptors; i++) /* num_of_rx_descriptors
    {
        if(i == (num_of_rx_descriptors - 1)) /* Last descriptor? if so,
          Rx_Descriptors[i].command = (OWN | EOR | (rx_buffer_len & 0x3F
        else
          Rx_Descriptors[i].command = (OWN | (rx_buffer_len & 0x3FFF));

        /** packet_buffer_address is the *physical* address for the buff
        Rx_Descriptors[i].low_buf = (unsigned int)&packet_buffer_address
        Rx_Descriptors[i].high_buf = 0;
        /* If you are programming for a 64-bit OS, put the high memory l
    }
}

void reset()
{
    unsigned int i;
    unsigned char mac_address[6];

    outportb(ioaddr + 0x37, 0x10); /* Send the Reset bit to the Command
    while(inportb(ioaddr + 0x37) & 0x10){} /* Wait for the chip to finis
```

```
    for (i = 0; i < 6; i++)
      mac_address[i] = inportb(ioaddr + i);

    setup_rx_descriptors();
    outportb(ioaddr + 0x50, 0xC0); /* Unlock config registers */
    outportl(ioaddr + 0x44, 0x0000E70F); /* RxConfig = RXFTH: unlimited,
    outportb(ioaddr + 0x37, 0x04); /* Enable Tx in the Command register,
    outportl(ioaddr + 0x40, 0x03000700); /* TxConfig = IFG: normal, MXDM
    outportw(ioaddr + 0xDA, 0x1FFF); /* Max rx packet size */
    outportb(ioaddr + 0xEC, 0x3B); /* max tx packet size */

    /* offset 0x20 == Transmit Descriptor Start Address Register
       offset 0xE4 == Receive Descriptor Start Address Register

       Again, these are *physical* addresses. This code assumes physical
       typically not the case in real world kernels
    */
    outportl(ioaddr + 0x20, (unsigned long)&Tx_Descriptors[0]; /* Tell t
    outportl(ioaddr + 0xE4, (unsigned long)&Rx_Descriptors[0]; /* Tell t

    outportb(ioaddr + 0x37, 0x0C); /* Enable Rx/Tx in the Command regist
    outportb(ioaddr + 0x50, 0x00); /* Lock config registers */
}
```

External Links

- http://www.magnesium.net/~wpaul/rt/RTL8110S_8169S_DataSheet_1.3.pdf, Datasheet for the RTL8169S and RTL8110S chipsets.