DM9000A

16 / 8 Bit Ethernet Controller with General Processor Interface

Programming Guide V1. 12

Technical Reference Manual

Davicom Semiconductor, Inc

Contents:

1 How to Read/ Write DM9000A Register

There are only two addressing ports through the access of the host interface. One is INDEX port and the other is DATA port. INDEX port is decoded by low the pin 32 CMD = 0, and DATA port is decoded by high the pin 32 CMD = 1.

All of the control and status registers in the DM9000A are accessed indirectly by the INDEX/ DATA ports. The command sequence to access these specified control/ status register is: firstly, to write the register address into INDEX port; then, to read/ write their data through DATA port.

The content of INDEX port is the address of the register to access DATA port later. Before accessing the 8-bit data in any register or the 8/ 16-bit data in the RX/ TX FIFO SRAM (total 16K Bytes: 13K Byte RX SRAM, 3K Byte TX FIFO), the address of the register must be written into INDEX port.
Please refer to the DM9000A data sheet chapter 9.1 about host interface.

For example, to read and write the DM9000A register:

(Where CMD pin is connected to Processor SA2)

```
UINT16 IOaddr;      /* UINT32 IOaddr=0x19000000; for example defined in ARM-base HPI BANK3*/

void iow ( UINT16 register, UINT8 dataB )
{
     outb (register, IOaddr); /* I/O output a Byte to INDEX port, select the register */
     outb (dataB, IOaddr+ 4); /* I/O output a Byte to DATA port, WRITE the register data */
}

UINT8 ior ( UINT16 register )
{
     outb (register, IOaddr); /* I/O output a Byte to INDEX port, select the register */
     return inb (IOaddr + 4); /* I/O input a Byte from DATA port, READ the register data */
}
```

1-1. How to Read I/O Mode

There are two operation modes of DATA Bus width, 8-bit or 16-bit, when access to the internal memory of the DM9000A MAC for RX/ TX packets.

These two modes are selected by the strap pin (21) EECS shown the following table:

EECS (pin 21)	I/O DATA width
0	16-bit
1	8-bit

Where, "1" means pull-high with the 10K Ohm resistor, and "0" means floating (default*).

And, the status of DATA width operation mode can be examined from the Bit [7] of ISR REG. FEH,

ISR (REG. FEH) Bit [7]	IOMODE (DATA I/O mode) Operation
0	16-bit mode
1	8-bit mode

For example, to check the I/O mode of the DM9000A NIC chip:

```
(UINT8) IO_mode = ior ( 0xFE ) >> 7;    /* ISR Bit [7] IOMODE indicating DATA I/O mode */
```

*Note: the pins EECS, EECK, EEDIO, WAKE and processor parallel interface are all have a pull-low resistor about 60K Ohm internally. When floating, the pin is default "0".
The EECS pin (21) is also used as a strap pin to set the 8/ 16-bit DATA width of the internal memory, the RX/ TX FIFO SRAM, for access the RX/ TX packets.

2 Driver Initializing Steps

1. To power up the internal PHY:

```
iow ( 0x1F, 0x00 );                    /* set PHYPD Bit [0] = 0 in GPR REG. 1FH */
```

   The PHY is powered down at default. The power-up procedure needs to enable it by writing low "0" to PHYPD, the Bit [0] in GPR REG.1FH. Please refer to the chapter 2-6 about the PHY setting.

2. To do a software-reset for the DM9000A initial (see the chapter 2-2):

```
iow ( 0x00, 0x01 );      /* set RST Bit [0] = 1 in NCR REG. 00, for a period time 10 us */
udelay (10);   iow ( 0x00, 0x00 );      /* wait 10us then clear it, or let it auto-clear */
```

3. Program the NCR register to choose normal mode by setting NCR (REG. 00) LBK Bit [2:1] = "00"b. The system designer can choose the network operations such as setting the internal MAC/ PHY loopback mode, forced the external PHY Full/ half duplex mode or to force collisions, and the wakeup events enable. Please refer to the data sheet chapter 6.1 about setting the NCR register.

4. To set the IMR register REG. FFH Bit [7] = 1 to enable the Pointer Auto Return function, which is the memory read/ write address pointer of the RX/ TX FIFO SRAM.

5. Read the EEPROM data 3 words, for the individual Ethernet node address (if necessary).

6. Write 6-byte Ethernet node address into the Physical Address Registers (PAR REG. 10H~15H).

7. Write Hash table 8 bytes into the Multicast Address Registers (MAR REG. 16H ~ REG. 1DH).

8. Clear TX & INTR status by R/W1 the NSR register REG. 01 and ISR REG. FEH. The 3 bits of Bit [2] TX1END, Bit [3] TX2END and WAKEST Bit [5], will be cleared automatically by reading it or writing "1". Please refer to the data sheet chapter 6.2& 6.33 about setting the NSR& ISR registers.

9. To handle the NIC interrupts or polling service routines.

10. Program the IMR register (REG. FFH) PRI Bit [0]/ PTI Bit [1] to enable the RX/ TX interrupt.

11. Program the RCR register to enable RX. The RX function is enabled by setting the RXEN Bit [0]= 1 in the RX control register, RCR REG. 05. Please refer to the data sheet ch.6.6 about setting RCR.

12. The DM9000A NIC is being activated and ready RX/ TX now.

For example, to initialize the DM9000A NIC:

```
static int DM9000A_init (void)
{
     UINT16    i;
     iow ( 0x1F, 0x00 );      /* GPR REG. 1FH PHYPD Bit[0] = 0 to activate internal PHY */
     udelay (2000);           /* wait >2 ms (2~60 ms normal) for PHY power-up ready */

     /* software-RESET NIC */
     iow ( 0x00 , 0x3 );      /* NCR REG. 00 RST Bit[0]=1 RESET on, LBK=1 MAC loopback on */
     udelay (10);             /* wait >10us for software-RESET ok, then let it auto-clear */

     phy_write ( 0, 0x8000 ); /* PHY software RESET: registers to the default states */
     i = 0;    do { udelay (10); i++; }   while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
     phy_write ( 4, 0x01E1 | 0x0400 );  /* operating PHY media mode with H/W Flow-control */
     phy_write ( 0, 0x1200 ); /* PHY Auto-negotiation: Auto sense and recovery registers */
     i = 0;    do { udelay (200); i++; }   while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
     udelay (2000);           /* wait >2 ms for PHY Auto-sense linking to partner */

     /* set other registers depending on applications */
     iow ( 0xFF, 0x80 );      /* IMR enable only Pointer Auto Return for RX/TX FIFO SRAM */
     iow ( 0x08, 0x3F );      /* BPTR REG.08(if necessary) High Water Overflow 3KB, 600us */
     iow ( 0x09, 0x5A );      /* FCTR REG.09 (if necessary) High/ Low Threshold 5KB/ 10KB */
     iow ( 0x0A, 0x29 );      /* FCR REG.0AH enable TXPEN, BKPM (for TX_half), FLCE (RX) */

     /* setup the receive configuration for Broadcast/ Multicast Hash table */
     dm9000_hash_table (dev);

     /* program operating registers~ */
     iow ( 0x00, 0x00 );      /* enable chip functions and disable loopback back to normal*/

     /* clear any pending interrupt */
     iow ( 0x01, 0x2c );      /* clear NSR 3bits status: TX1END, TX2END, WAKEST by RW/C1 */
     iow ( 0xFE, 0x3f );      /* clear ISR status: PR,PT, ROS,ROO, UDRUN, LNKCHG by RW/C1 */
```

```
    /* enable interrupts ~go */
    iow ( 0xFF, 0Xbf );      /* IMR REG. FFH enable PAR+ PTI+PRI+ROI+ROOI+UDRUNI+LNKCHGI */

    /* enable RX Broadcast/ ALL MULTICAST to activate DM9000A ~go */
    iow ( 0x05, 0x30 | 1 );  /* RCR REG. 05 RXEN Bit[0] = 1 to enable RX machine/ filter */

    /* initialize the driver variables or the user passed arguments */
    (UINT8) IO_mode = ior ( 0xFE )>>7; /* ISR Bit [7] IOMODE indicating DATA I/O mode */
    /* #define TRUE 1   #define FALSE 0     #define FULL 1      #define HALF 0 */
    LINK = ( ior ( 0X01 )& 0x40 ) ? TRUE: FALSE; /* if NSR Bit [6] = 1: LINK ok, else failed*/
    SPEED = ( ior ( 0X01 ) & 0x80 ) ? 10: 100;   /* if NSR Bit [7] = 0: SPEED = 100Mbps */
    DUPLEX = ( ior ( 0x00 ) & 8 ) ? FULL: HALF;  /* if NCR Bit [3] = 1: Full Duplex mode */

    return (0);              /* RETURN "NU_SUCCESS" */
}   /* end DM9000A_init I/O routine */
```

2-1. How to Identify DM9000A

For example, to search the DM9000A NIC:

```
int search9000 (void)
{
    int CardFound = 0;
    UINT16 VID_value;

    /* read REG. 28H & 29H for DM9000A Vendor ID */
    outb ( 0x28, IOaddr );        /* UINT16 IOaddr for I/O INDEX port (I/O base address) */
    VID_value = inb ( IOaddr+4 ); /* UINT16 IOdata for I/O DATA port (= IOaddr + 4) */
    outb ( 0x29, IOaddr );
    VID_value |= ( inb ( IOaddr + 4 ) << 8 );
    if ( VID_value == 0x0A46 ) {  /* check Vendor ID if DM9000A NIC found ? */
        if ( ior (0x2c)== 0x18 ) CardFound=1;/* TRUE, found the DM9000A NIC; else FALSE */
        /* system variables VID & PID (if necessary), or just discard below */
```

```
          (UINT16) VID = 0x0A46;   /* got Vendor ID */
          (UINT16) PID = ior ( 0x2A ) | ( ior ( 0x2B ) << 8 );   /* got Product ID 9000A */
     }
     return CardFound;
}    /* end search9000 function */
```

Because the Vendor ID and the Product ID might be changed by the EEPROM 93C46/ LC46 setting, there is another way for reference as follows,

```
int search9000 (void)
{
     int CardFound = 0;

     iow ( 0x00, 0x01 );          /* software-RESET ON for recovery registers */
     udelay (10);                 /* wait >10 us for software-RESET ok */
     if ( ( ior (0x2c) == 0x18 ) && ( ior ( 0x08 ) == 0x37 ) && ( ior ( 0x09 ) == 0x38 ) )
     {
          CardFound = 1;          /* TRUE, success found the DM9000A NIC; else FALSE */
          (UINT16) VID = ior ( 0x28 ) | ( ior ( 0x29 ) << 8 );   /* got Vendor ID 0A46 */
          (UINT16) PID = ior ( 0x2A ) | ( ior ( 0x2B ) << 8 );   /* got Product ID 9000 */
     }
     return    CardFound;
}    /* end search9000 function */
```

2-2. Software Reset

The DM9000A can be reset by either hardware-reset or software-reset. A hardware-reset can be accomplished by asserting the power-on PWRST# pin 40. A software-reset can be accomplished by setting the network control register (NCR REG. 00) Bit [0] = 1, then wait 10 us for auto-clear RST=0:

```
iow ( 0x00, 0x01 );               /* NCR REG. 00 RST Bit [0] = 1, software-RESET ON */
udelay (10);                      /* wait >10 us for RESET ok, then let it auto-clear or */
iow ( 0x00, 0x00 );               /* NCR REG. 00 RST Bit [0] = 0, RESET OFF, chip normal */
```

2-3. PHY Reset

Output one word data 0x8000 into the MII register offset 0, by writing "1" to the Reset Bit [15] in the basic mode control register (BMCR REG. 00):

```
phy_write ( 0x00, 0x8000 );        /* PHY software RESET: registers to the default states */
i = 0;    do { udelay (10); i++; }      while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
phy_write ( 0x00, 0x1200 );        /* PHY Auto-negotiation: Auto sense and recovery register*/
i = 0;    do { udelay (200); i++; }     while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
udelay (2000);                     /* wait >2 ms for PHY Auto-sense linking to partner */
```

And the function call of phy_write () is as follows,

```
static void phy_write (int offset, UINT16 value)  /* refer to chapter 4-3 about PHY WRITE */
{
     /* fill the phyxcer register address into EPAR REG. 0CH */
     iow ( 0x0c, offset | 0x40 );  /* issue the PHY address + 40H (EPAR PHY_ADR = "01"b) */

     /* fill the written data into EPDRH & EPDRL */
     iow ( 0x0E, (value >> 8) & 0xff ); /* put the high-byte into EPDRH REG.0EH */
     iow ( 0x0D, value & 0xff );   /* put the low-byte into EPDRL REG.0DH */

     iow ( 0x0B, 0x08 );           /* clear the command = 0x08 firstly */
     outb ( 0x0A, IOaddr + 4 );    /* issue the "PHY + WRITE" command = 0AH */
     udelay (5);                   /* wait 1~5us for the "PHY + WRITE" command completion */
     outb ( 0x08, IOaddr + 4 );    /* clear this PHY "WRITE" command */
}
```

This Reset bit, which is self-clearing to be "0", will keep returning a value of "1" until the reset process is completed by checking the Reset Bit [15] of the PHY REG. 00 value is equal to "0" ok:

```
while ( phy_read ( 0 ) & 0x8000 ) {; }; /* PHY RESET finished ? Time wait <3 seconds */
```

And the function call of phy_read () is as follows,

```
static UINT16 phy_read (int offset) /* refer to the chapter 4-2 about PHY READ setting */
{
     /* fill the phyxcer register address into EPAR REG. 0CH */
     iow ( 0x0c, offset | 0x40 );  /* issue the PHY address + 40H (EPAR PHY_ADR = "01"b) */

     iow ( 0x0B, 0x08 );           /* clear the command = 0x08 firstly */
     outb ( 0x0c, IOaddr + 4 );    /* issue the "PHY + READ" command = 0CH */
     udelay (5);                   /* wait 1~5 us for the "PHY + READ" command completion */
     outb ( 0x08, IOaddr + 4 );    /* clear this PHY "READ" command */

     /* the read data is in EPDRH REG. 0EH for high-byte, EPDRL REG. 0DH for low-byte */
     return ( ( ior ( 0x0E ) << 8 ) | ior ( 0x0D ) );
}
```

2-4. PHY Auto-Negotiation

Output one word data 0x1200 into the MII register offset 0, by writing "1" to the Auto-negotiation enable Bit [12] and the Restart auto-negotiation Bit [9] in the PHY REG. 00 BMCR:

```
phy_write ( 0x00, 0x1200 );        /* PHY Auto-negotiation: Auto sense and recovery register*/
i = 0;    do { udelay (200); i++; }     while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
udelay (2000);                     /* wait >2 ms for PHY Auto-sense linking to partner */
```

This Restart auto-NEGO bit, which is self-clearing to be "0", will keep returning a value of "1" until the PHY Auto-negotiation is initiated. While completed, the Bit [5] of the PHY REG. 01 is equal to "1":

```
while ( !(phy_read ( 1 ) & 0x20) ) {; }; /* PHY Auto-negotiation finished? Time wait <6 s */
```

2-5. GPIO Pins Setting

If the DM9000A is operated in 8-bit mode, there are 6 general purpose pins, GP1 ~ GP6, can be used. Their IO types are controlled by GPCR REG. 1EH. And their IO data are presented by GPR REG. 1FH. The default values of GPCR Bit [3:1] are all "0"s for the GP3 ~ GP1 pins as the input mode respectively. But, the GPCR Bit [6:4] GPC64 are all forced to "1"s only for the GP6 ~ GP4 pins as the output mode. And the GPCR Bit [0] is forced to "1" too, for powering down PHY. Then set PHYPD = 0 to turn it on.

For example, to active GP1 pin 31 outputting high "1" (in 8-bit mode only):

```
iow ( 0x1E, 0x02 );          /* GPCR REG.1EH GPC1 Bit[1] = 1 output port (in 8-bit mode) */
iow ( 0x1F, 0x02 );          /* GPR REG. 1FH GPIO1 Bit[1]= 1 to activate GP1 pin 31 high */
```

2-6. PHY Power-down Setting

In the power-down (power-saving) mode, the internal PHY of the DM9000A will disable all transmit, receive and MII interface functions except the MDC/ MDIO management interface.

There are two ways to set the PHY power-down mode:

One is to set-bit the PHYPD Bit [0] = 1 in the MAC GPR REG. 1FH, the other is to write "1" to the Power-down Bit [11] in the PHY BMCR REG. 00:

```
iow ( 0x1F, 0x01 );          /* 1. Power-down PHY by MAC register GPR REG. 1FH setting */
```

Or it is,

```
phy_write ( 0x00, 0x800 );   /* 2. PHY powered down by PHY register BMCR REG. 00 setting */
```

The DM9000A status default is the GPR PHYPD Bit [0] = 1 to cut down the power of the internal PHY. Once easy, the internal PHY is to be active, the system driver needs to clear this power-down bit PHYPD in GPR REG. 1FH by writing "0" and wait 2 ms at least for the internal PHY ready:

```
iow ( 0x1F, 0x00 );          /* GPR REG. 1FH PHYPD Bit [0] = 0 to activate internal PHY */
udelay (2000);               /* wait >2 ms (2~60 ms normal) for PHY power-up ready */
```

3 How to access EEPROM

The DM9000A supports the serial EEPROM interface and provides a very easy method to access it. It is only to read/ write the EEPROM&PHY control/ address/ data register (REG. 0BH ~ REG. 0EH), which are used to control and read/ write the EEPROM address or data.

For example, IC 93C46 is a 64-word (1024-bit) EEPROM and the word addresses are mapped to the EROA Bit [5:0] in the EEPROM&PHY address register, EPAR REG. 0CH. Besides, the EPAR PHY_ADR Bit [7:6] must be set "00"b to enable the word address for the EEPROM decoded.
And, to set the PHY_ADR Bit [7:6] = "01"b in EPAR is to select the internal PHY address.
Please refer to the data sheet chapter 6.12 ~ 6.14 about setting the EEPROM&PHY registers.

The EEPROM of the DM9000A holds the following parameters:

1. Ethernet node address.
2. Vendor ID and Product ID auto load.
3. Processor control bus pins polarity setting.
4. Wake-up mode control.
5. PHY power-up or PHY power-down while powered on.
6. AUTO-MDIX enable/ disable setting.
7. LED1 & LED2 pins act as IO16 pin and IOWAIT/ WAKE pin in 16-bit mode.
8. LED mode 0 or mode 1 setting.

All of the above mentioned settings are read from the EEPROM upon hardware power-on reset. The specific target device is IC 93C46, the 1024-bit serial EEPROM.
(EEPROM is 93C46/ LC46 such as ATmel, Micro-chip, and CSI.)
All of accesses to the EEPROM are done in words. All of the EEPROM addresses in the specification are specified as word addresses.

The default setting of the DM9000A can be changed by the I/O strap pins or the EEPROM with higher priority at the EEPROM setting.

3-1. EEPROM Format

Name	Address (Word)	Programming Value (Hex)	Description
Ethernet Node Address	0 ~ 2	XX:XX: XX:XX: XX:XX	Auto-download 6-byte MAC Address to PAR REG. 10H ~ REG. 15H.
Auto Load Control	3	5445	Bit [1:0] = 00: Disable vendor ID and product ID programming. *01: Accept vendor ID and product ID programming. Bit [3:2] = 00: Disable setting of Word 6 Bit [8:0]. *01: Accept setting of Word 6 Bit [8:0]. Bit [5:4]: Reserved = 0. Bit [7:6] = 00: Disable setting of Word 7 Bit [3:0]. *01: Accept setting of Word 7 Bit [3:0]. Bit [9:8]: Reserved = 0. Bit [11:10] = 00: Disable setting of Word 7 Bit [7]. *01: Accept setting of Word 7 Bit [7]. Bit [13:12] = 00: Disable setting of Word 7 Bit [8]. *01: Accept setting of Word 7 Bit [8]. Bit [15:14] = 00: Disable setting of Word 7 Bit [15:12]. *01: Accept setting of Word 7 Bit [15:12]. Note: The remark * is now programming value.
Vendor ID	4	0A46	2-byte vendor ID.
Product ID	5	9000	2-byte product ID.
Pin Control	6	01E7	When Word 3 Bit [3:2] = 01, these bits can control the CS, IOR, IOW , INT, IOWAIT and IO16 pins polarity: Bit [0] = 0: CS pin is active high. *1: CS pin is active low. Bit [1] = 0: IOR pin is active high. *1: IOR pin is active low. Bit [2] = 0: IOW pin is active high. *1: IOW pin is active low. Bit [3] = *0: INT pin is active high. 1: INT pin is active low. Bit [4] = *0: INT pin is force output. 1: INT pin is force open-collected. Bit [5] = 0: IOWAIT is active high. *1: IOWAIT is active low. Bit [6] = 0: IOWAIT is force output. *1: IOWAIT is force open-collected. Bit [7] = 0: IO16 is active high. *1: IO16 is active low. Bit [8] = 0: IO16 is force output. *1: IO16 is force open-collected. Bit [15:9]: Reserved = 0.
Wake-up Mode Control	7	4180	Depend on the setting of Word 3 Bit [15:6] to accept auto load control: Bit [0] = *0: WAKE pin is active high. 1: WAKE pin is active low. Bit [1] = *0: WAKE pin is in level mode.

			1: WAKE pin is in pulse mode. Bit [2] = *0: Magic packet wake-up event is disabled. 1: Magic packet wake-up event is enabled. Bit [3] = *0: Link_change wake-up event is disabled. 1: Link_change wake-up event is enabled. Bit [6:4]: Reserved = 0. Bit [7] = 0: LED mode 0. *1: LED mode 1. Bit [8] = 0: The internal PHY is disabled after power-ON. *1: The internal PHY is enabled after power-ON. The GPR REG. 1FH Bit [0] is modified from this Bit [8]. Bit [13:9]: Reserved = 0. Bit [14] = 0: AUTO-MDIX OFF, *1: AUTO-MDIX ON. Bit [15]: Reserved = 0.

Note: The programming value of the EEPROM is only for reference to list 16-bit values, "XX:XX:XX:XX:XX:XX, 5445, 0A46, 9000, 01E7, 4180".

Table 1. EEPROM Format in 8-bit mode

Name	Address (Word)	Programming Value (Hex)	Description
Ethernet Node Address	0 ~ 2	XX:XX: XX:XX: XX:XX	Auto-download 6-byte MAC Address to PAR REG. 10H ~ REG. 15H.
Auto Load Control	3	5405	Bit [1:0] = 00: Disable vendor ID and product ID programming. *01: Accept vendor ID and product ID programming. Bit [3:2] = 00: Disable setting of Word 6 Bit [8:0]. *01: Accept setting of Word 6 Bit [8:0]. Bit [7:6] = 00: Disable setting of Word 7 Bit [3:0]. *01: Accept setting of Word 7 Bit [3:0] and Word 7 Bit [13:12] = 10 to let LED2 act as WAKE in 16-bit mode only. Bit [9:8]: Reserved = 0. Bit [11:10] = 00: Disable setting of Word 7 Bit [7]. *01: Accept setting of Word 7 Bit [7]. Bit [13:12] = 00: Disable setting of Word 7 Bit [8]. *01: Accept setting of Word 7 Bit [8]. Bit [15:14] = 00: Disable setting of Word 7 Bit [15:12]. *01: Accept setting of Word 7 Bit [15:12]. Note: The remark * is now programming value.
Vendor ID	4	0A46	2-byte vendor ID.
Product ID	5	9000	2-byte product ID.
Pin Control	6	01E7	When Word 3 Bit [3:2] = 01, these bits can control the CS, IOR, IOW , INT, IOWAIT and IO16 pins polarity: Bit [0] = 0: CS pin is active high. *1: CS pin is active low. Bit [1] = 0: IOR pin is active high. *1: IOR pin is active low. Bit [2] = 0: IOW pin is active high.

			*1: IOW pin is active low. Bit [3] = *0: INT pin is active high. 1: INT pin is active low. Bit [4] = *0: INT pin is force output. 1: INT pin is force open-collected. Bit [5] = 0: IOWAIT is active high. *1: IOWAIT is active low. Bit [6] = 0: IOWAIT is force output. *1: IOWAIT is force open-collected. Bit [7] = 0: IO16 is active high. *1: IO16 is active low. Bit [8] = 0: IO16 is force output. *1: IO16 is force open-collected. Bit [15:9]: Reserved = 0.
Wake-up Mode Control	7	4180	Depend on the setting of Word 3 Bit[15:6] to accept auto load control: Bit [0] = *0: The WAKE pin is active high. 1: The WAKE pin is active low. Bit [1] = *0: The WAKE pin is in level mode. 1: The WAKE pin is in pulse mode. Bit [2] = *0: Magic packet wake-up event is disabled. 1: Magic packet wake-up event is enabled. Bit [3] = *0: Link_change wake-up event is disabled. 1: Link_change wake-up event is enabled. Bit [6:4]: Reserved = 0. Bit [7] = 0: LED mode 0. *1: LED mode 1. Bit [8] = 0: The internal PHY is disabled after power-ON. *1: The internal PHY is enabled after power-ON. The GPR REG. 1FH Bit [0] is modified from this Bit [8]. Bit [11:9]: Reserved = 0. Bit [13:12] = *00: LED2 pin 38 act normal. 01: LED2 pin 38 act as IOWAIT pin used in 16-bit mode. 10: LED2 pin 38 act as WAKE pin used in 16-bit mode. Bit [14] = 0: AUTO-MDIX OFF, *1: AUTO-MDIX ON. Bit [15] = *0: LED1 pin 39 act normal, 1: LED1 pin 39 act as IO16 pin in 16-bit mode only.

Note: List the programming values of the EEPROM in 16-bit mode,

"XX:XX:XX:XX:XX:XX, 5405, 0A46, 9000, 01E7, C180" if LED1 pin 39 act as "IO16" pin used.

"XX:XX:XX:XX:XX:XX, 5405, 0A46, 9000, 01E7, 5180" if LED2 pin 38 act as "IOWAIT" pin used.

"XX:XX:XX:XX:XX:XX, 5445, 0A46, 9000, 01E7, 6180" if LED2 pin 38 act as "WAKE" pin used.

"XX:XX:XX:XX:XX:XX, 5405, 0A46, 9000, 01E7, D180" if LED1 pin 39 act as "IO16" and LED2 pin 38 act as "IOWAIT" pin used.

"XX:XX:XX:XX:XX:XX, 5445, 0A46, 9000, 01E7, E180" if LED1 pin 39 act as "IO16" and LED2 pin 38 act as "WAKE" pin used.

Table 2. EEPROM Format in 16-bit mode

3-2. How to Read SROM

The following steps are shown to read data from the serial EEPROM (SROM):

Step 1: write the SROM word address into EPAR REG. 0CH.

Step 2: write command = 0x04 into the EEPROM&PHY Control Register (EPCR REG. 0BH) to start the "SROM + READ" operation:

i. EPCR (REG. 0BH) EPOS Bit [3] = 0: select SROM mode (this is default: 0).

ii. EPCR (REG. 0BH) ERPRR Bit [2] = 1: issue READ command.

Step 3: read EPCR REG. 0BH and wait until the ERRE Bit [0] = 0 ok, or just following Step 4.

Step 4: wait 40 us at least, then write "0" into EPCR REG. 0BH to clear this command.

Step 5: read the SROM data high-byte from EPDRH REG. 0EH, and the low-byte from EPDRL REG. 0DH in the EEPROM&PHY Data registers.

HOWTO read SROM Word 3 for Auto Load Control shown as follows:

1. write word address = 0x03 into EPAR REG. 0CH

```
iow ( 0x0C, 0x03 );
```

2. write the "SROM + READ" command = 0x04 into EPCR REG. 0BH

```
iow ( 0x0B, 0x04 );
```

3. wait until EPCR (REG. 0BH) ERRE Bit [0] = 0 ok, or just following Step 4

```
do { udelay ( 10 ); i++; }      while ( ( i < 500 ) && ( inb ( IOaddr + 4 ) & 1 ) );
```

4. delay 40 us at least, then write "0" into EPCR REG. 0BH to clear it

```
udelay ( 40 );                  /* wait >40us for the "EEPROM+READ" command completion */

iow ( 0x0B, 0 );                                    /* clear this "EEPROM + READ" command */
```

5. got EPDRH REG. 0EH for the high-byte of the SROM data, and

```
(UINT8) e2prom[i*2+1] = ior ( 0x0E );           /* the high byte of this 16-bit word */
```

got EPDRL REG. 0DH for the low-byte

```
(UINT8) e2prom[i*2] = ior ( 0x0D );             /* the low  byte of this 16-bit word */
```

For example, to read the MAC address from the SROM Word 0& 1& 2:

```
for ( i = 0; i < 3; i++ ) {                         /* read Ethernet node address from SROM */
     iow ( 0x0C, i );
     iow ( 0x0B, 0x04 );
     udelay ( 40 );
     iow ( 0x0B, 0 );
```

```
    /* read the SROM word data from EPDRH & EPDRL, to put into the device address */
    (UINT8) dev_addr[i*2+1] = ior ( 0x0E );      /* the high byte of this 16-bit word */
    (UINT8) dev_addr[i*2] = ior ( 0x0D );        /* the low  byte of this 16-bit word */
}
```

3-3. How to Write SROM

The following steps are to write data into the SROM:

Step 1: write the SROM word address into EPAR REG. 0CH.

Step 2: write the SROM data high-byte into EPDRH REG. 0EH, and the low-byte into EPDRL.

Step 3: write command = 0x12 into EPCR REG. 0BH to start the "SROM + WRITE" operation:

i. EPCR (REG. 0BH) WEP Bit [4] = 1: enable SROM WRITE operation.

ii. EPCR (REG. 0BH) EPOS Bit [3] = 0: select SROM mode (this is default: 0).

iii. EPCR (REG. 0BH) ERPRW Bit [1] = 1: issue WRITE command.

Step 4: read EPCR REG. 0BH and wait until the ERRE Bit [0] = 0 ok, or just following Step 5.

Step 5: wait 500 us at least, then write "0" into EPCR REG. 0BH to clear this command.

HOWTO write 0x1E7 into SROM Word 6 for Pin Control shown as follows:

1. write word address = 0x06 to EPAR REG. 0CH

```
iow ( 0x0C, 0x06 );
```

2. write the SROM data high-byte 0x01 into EPDRH, and the low-byte 0xE7 into EPDRL

```
iow ( 0x0E, 0x01 );                          /* the high byte of this 16-bit word */
iow ( 0x0D, 0xE7 );                          /* the low  byte of this 16-bit word */
```

3. write the "SROM + WRITE" command = 0x12 into EPCR REG. 0BH

```
iow ( 0x0B, ( 0x02 | 0x10 ) );
```

4. wait until EPCR (REG. 0BH) ERRE Bit [0] = 0 ok, or just following Step 5

```
do { udelay ( 10 ); i++; }      while ( ( i < 500 ) && ( inb ( IOaddr + 4 ) & 1 ) );
```

5. delay 500 us at least, then write "0" into EPCR REG. 0BH to clear it

```
udelay ( 60000 );               /* wait >500 us for "EEPROM+WRITE" command completion */
iow ( 0x0B, 0 );                             /* clear this "EEPROM + WRITE" command */
```

For example, to write the MAC address 00:E0:63:0E:56:78 into SROM Word 0& 1& 2:

```
/* write Ethernet node address into the serial EEPROM 93C46/ LC46 */
srom_write ( 0x00, 0xE000 );                /* Word 0: low-byte = 0x00, high-byte = 0xE0 */
srom_write ( 0x01, 0x0E63 );                /* Word 1: low-byte = 0x63, high-byte = 0x0E */
srom_write ( 0x02, 0x7856 );                /* Word 2: low-byte = 0x56, high-byte = 0x78 */

iow ( 0x0B, 0x20 );                         /* REEP Bit [5] = 1 to re-load EEPROM value */
udelay (400);                               /* wait 365 us at least, then clear it */
iow ( 0x0B, 0x00 );                         /* REEP Bit [5] = 0 cleared for new config. */



void srom_write ( int offset, UINT16 dataW )
{
     UINT16    i, tmpv;
     /* write the SROM word address into EPAR REG. 0CH */
     iow ( 0x0C, offset );

     /* write the SROM Word data into EPDRH & EPDRL */
     iow ( 0x0D, dataW & 0xff );            /* put the low  byte of this 16-bit word */
     iow ( 0x0E, ( dataW >> 8 ) & 0xff );   /* put the high byte of this 16-bit word */

     /* issue the "SROM + WRITE" command = 0x12 into EPCR REG. 0BH */
     iow ( 0x0B, 0x02 | 0x10 );

     /* wait >500 us for "SROM + WRITE" completed then write "0" to clear command */
     do { udelay (10); tmpv= inb (IOaddr+ 4); i++; } while ( (i < 500) && ( tmpv & 1 ) );
     udelay ( 60000 );
     iow( 0x0B, 0 );
}
```

4 How to access PHY

The DM9000A provides a very easy method to access the PHY data, and it's very similar to access the SROM data above. To read/ write the EEPROM&PHY control/ address/ data registers, which map to the PHY register offset address to access it, and then the NIC takes care all detail steps to get it.

4-1. PHY Register Offset Read/ Write

The DM9000A PHY supports 32 registers, which are mapped to the Bit [4:0] in EPAR REG. 0CH. The default value of the PHY_ADR Bit [7:6] in EPAR REG. 0CH is "01"b to select (decoder) the PHY mode. Please refer to the data sheet chapter 6.12 ~ 6.14 about setting the EEPROM&PHY registers.

4-2. How to Read PHY

The following steps are shown to read data from the PHY register:

Step 1: write the PHY register offset into EPAR (REG. 0CH) EROA Bit [4:0], and the PHY address "01"b into EPAR (REG. 0CH) PHY_ADR Bit [7:6].

Step 2: write command = 0x0c into EPCR REG. 0BH to start the "PHY + READ" operation:

i. EPCR (REG. 0BH) EPOS Bit [3] = 1: select PHY mode (0: select SROM, see the chapter 3-2)

ii. EPCR (REG. 0BH) ERPRR Bit [2] = 1: issue READ command.

Step 3: read EPCR REG. 0BH and wait until ERRE Bit [0] = 0 ok, or just following Step 4.

Step 4: wait 5 us maximum, then write "0x08" into EPCR REG. 0BH to clear this READ command.

Step 5: read the PHY data high-byte from EPDRH REG. 0EH, and the low-byte from EPDRL REG. 0DH in the EEPROM&PHY Data registers.

For example, to read the PHY register of BMCR at address offset = 0x00:

1. write offset = 0x00 into EPAR REG. 0CH Bit [4:0], and "01"b into EPAR REG. 0CH Bit [7:6]

```
iow ( 0x0C, ( 0x00 | 0x40 ) );        /* issue PHY address+ 40H (EPAR PHY_ADR = "01"b) */
```

2. write the "PHY + READ" command = 0x0c into EPCR REG. 0BH

```
iow ( 0x0B, ( 0x04 | 0x08 ) );
```

3. wait until EPCR (REG. 0B) ERRE Bit [0] = 0 ok, or just following Step 4

```
do { udelay ( 1 ); i++; }        while ( ( i < 50 ) && ( inb ( IOaddr + 4 ) & 1 ) );
```

4. delay 5 us maximum, then write "0x8" into EPCR REG. 0BH to clear it but keep PHY selected

```
udelay ( 5 );                   /* wait 1~5 us for the "PHY + READ" command completion */

iow ( 0x0B, 0x08 );                             /* clear this PHY "READ" command */
```

5. read the PHY data high-byte from EPDRH REG. 0EH, and read the low-byte from EPDRL

```
(UINT8) phy[offset*2+1] = ior ( 0x0E );         /* the high byte of this 16-bit word */

(UINT8) phy[offset*2] = ior ( 0x0D );           /* the low  byte of this 16-bit word */
```

And it's the same as the function call of phy_read (0x00) at Page 11 in the chapter 2-3.

4-3. How to Write PHY

The following steps are to write data into the PHY register:

Step 1: write the PHY register offset into EPAR (REG. 0CH) EROA Bit [4:0], and the PHY address Bit [1:0] = "01"b into EPAR (REG. 0CH) PHY_ADR Bit [7:6].

Step 2: write the PHY data high-byte into EPDRH REG. 0EH, and the low-byte into EPDRL.

Step 3: write command = 0x0A into EPCR REG. 0BH to start the "PHY + WRITE" operation:

i. EPCR (REG. 0B) EPOS Bit [3] = 1: select PHY mode (0: select SROM, see the chapter 3-3)

ii. EPCR (REG. 0B) ERPRW Bit [1] = 1: issue WRITE command.

Step 4: read EPCR REG. 0BH and wait until ERRE Bit [0] = 0 ok, or just following Step 5.

Step 5: wait 5 us maximum, then write "0x8" into EPCR REG. 0BH to clear this WRITE command.

For example, to write data = 0x101 into ANAR REG. 04 for the 100M Full-duplex support:

1. write offset = 0x04 into EPAR REG. 0CH Bit [4:0], and "01"b into EPAR REG. 0CH Bit [7:6]

```
iow ( 0x0C, ( 0x04 | 0x40 ) );  /* issue PHY address+ 40H (EPAR PHY_ADR = "01"b) */
```

2. write the PHY data high-byte 0x01 into EPDRH, and write the low-byte 0x01 into EPDRL

```
iow ( 0x0E, 0x01 );             /* put the high byte of this 16-bit word */

iow ( 0x0D, 0x01 );             /* put the low  byte  of this 16-bit word */
```

3. write the "PHY + WRITE" command = 0x0A into EPCR REG. 0BH

```
iow ( 0x0B, ( 0x02 | 0x08 ) );
```

4. wait until EPCR (REG. 0BH) ERRE Bit [0] = 0 ok, or just following Step 5

```
do { udelay ( 1 ); i++; }  while ( ( i < 50 ) && ( inb ( IOaddr + 4 ) & 1 ) );
```

5. delay 5 us maximum, then write "8" into EPCR REG. 0BH to clear it but keep PHY selected

```
udelay ( 5 );                   /* wait 1~5us for the "PHY + WRITE" command completion */

iow ( 0x0B, 0x08 );             /* clear this PHY "WRITE" command */
```

And it's the same as the function call of phy_write (0x04, 0x101) at Page 10 in the chapter 2-3.

4-4. How to select PHY Mode for Media Type

In MII registers, to program the Bit [15:7] of BMCR REG. 00 for the PHY mode setting, and Bit [15:0] of ANAR REG. 04 for the PHY capability setting. And, the linking result is shown in BMSR REG.01 and in ANLPAR REG. 05. While power-on, BMCR REG. 00 is default 0x3100 with Auto-negotiation to auto sense linking to 10/100M switching hub, and ANAR REG. 04 is default 0x01E1 with all TX capability. Please refer to the data sheet chapter 8.1 & 8.5 about setting the PHY BMCR and ANAR registers.

1. 10BASE-T half duplex fixed mode (i.e. forced 10M speed & half duplex):

```
PHY_reg4 = 0x0021; /* set 10_HDX is supported and the device supports 802.3 CSMA/ CD */
PHY_reg0 = 0x0000; /* Speed selection: 10Mbps; Duplex mode: half duplex operation */
```

2. 10BASE-T Full duplex fixed mode (i.e. forced 10M speed & Full duplex):

```
PHY_reg4 = 0x0041; /* set 10_FDX is supported and the device supports 802.3 CSMA/ CD */
PHY_reg0 = 0x0100; /* Speed selection: 10Mbps; Duplex mode: Full duplex operation */
```

3. 100BASE-TX half duplex fixed mode (i.e. forced 100M speed & half duplex):

```
PHY_reg4 = 0x0081; /* set TX_HDX is supported and the device supports 802.3 CSMA/ CD */
PHY_reg0 = 0x2000; /* Speed selection: 100Mbps; Duplex mode: half duplex operation */
```

4. 100BASE-TX Full duplex fixed mode (i.e. forced 100M speed & Full duplex):

```
PHY_reg4 = 0x0101; /* set TX_FDX is supported and the device supports 802.3 CSMA/ CD */
PHY_reg0 = 0x2100; /* Speed selection: 100Mbps; Duplex mode: Full duplex operation */
```

5. Auto-negotiation mode:

```
PHY_reg4 = 0x01E1; /* set all TX_* are supported and the device supports 802.3 CSMA/ CD */
PHY_reg0 = 0x1200; /* detect Speed& Duplex, shown command in BMCR and status in BMSR */
```

For example, to force the PHY in 100Mbps speed and Full Duplex mode:

```
(UINT16) PHY_reg4 = 0x0101;        /* set 100BASE-TX, TX_FDX, and 802.3 CSMA/CD supported */
(UNIT16) PHY_reg0 = 0x2100;        /* Speed selection: 100Mbps; Duplex mode: Full Duplex */
phy_write ( 0x04, PHY_reg4 | 0x0400 );  /* operating PHY media mode with H/W Flow-control */
phy_write ( 0x00, 0x1200 );        /* PHY Auto-negotiation: Auto sense and recovery register*/

i = 0;    do { udelay (200); i++; }     while ( (i < 30000) && !( ior ( 0x01 ) & 0x40) );
udelay (2000);                     /* wait >2 ms for PHY Auto-sense linking to partner */
iow ( 0x1F, 0x00 );                /* GPR REG. 1FH PHYPD Bit[0] = 0 activate the int. PHY */
```

4-5. How to read PHY Link Status

For detecting the link status of the internal PHY, easily reading the LINKST Bit [6] in the network status register (MAC NSR REG. 01) to check the link status is either 1: link ok, or 0: link failed.
Please refer to the DM9000A data sheet chapter 6.2 about setting the NSR register.

For detecting the link status of the external PHY, firstly setting the EXT_PHY Bit [7] = 1 in the network control register (MAC NCR REG. 00) to enable the external MII interface. Then, reading the LINKST Bit [6] of NSR REG. 01 to check the link status is either 1: link ok, or 0: link failed.
Please refer to the data sheet chapter 6.1 about the NCR register setting.

To command for the external PHY, the commands of phy_write() and phy_read() are used and they are the same for the internal PHY. Please refer to chapter 2-3 about the PHY read/ write function calls.

For example, to detect the link status of the external PHY:

```
iow ( 0x00, 0x00 | 0x80 );         /* enable the ext. PHY by setting EXT_PHY Bit [7] = 1 */
/* detect the PHY Link status ok? until <6 seconds, then wait >2 ms for the PHY LINK ok */
i = 30000; while (!LINK && i) { udelay (200); LINK = ior (1) & 0x40; i--; }; udelay (2000);
/* or, to check the Link status Bit [2] of PHY BMSR REG. 01, until <6s, then wait >2ms ok */
i = 30000; while (!LINK && i) { udelay (200); LINK = phy_read (1) & 4; i--; }; udelay (2000);
```

5 How to set Node Address and Hash Table

The DM9000A needs the individual MAC address, as Ethernet node address which must be put on the Physical Address Registers (PAB0~5, REG. 10H~15H). Generally, this MAC address is the first 3-word data auto-loaded from the word address 0&1&2 in the EEPROM 93C46 (see chapter 3-1, Table 1 & 2). Note: For the MAC address, the first byte is PAB0 REG. 10H, and the last byte is PAB5 REG. 15H.

The DM9000A provides a 64-bit Hash table, as the hardware packet filter put in the Multicast Address Registers (MAB0 ~ MAB7, REG. 16H ~ REG. 1DH), to accept the valid packet incoming with the Multicast address in the frame header.

To program the unicast (MAC) address and to map the Multicast address into the 64-bit Hash table (i.e. 64 Ethernet groups), following the useful sample codes with the CRC-checksum function:

```
/* dev_addr[6]: Ethernet node address , map to Physical Address Register (PAB0~PAB5 REG. 10H~15H)
  mc_list[8]: Multicast address, map to Multicast Address Register (MAB0~MAB7 REG. 16H~1DH)*/

static void dm9000_hash_table (struct DEVICE *dev)
{
     struct dev_mc_list *mcptr = dev->mc_list;
     int mc_cnt = dev->mc_count;
     UINT32 hash_val;
     UINT16 i, offset, hash_table[4];

     for (i = 0; i < 3; i++) {     /* read Ethernet node address from SROM Word 0& 1& 2 */
          iow ( 0x0c, i );
          iow ( 0x0B, 0 );
          outb ( 0x04, IOaddr + 4 );
          udelay (100);
          outb ( 0x00, IOaddr + 4 );
          /* read the SROM data Word 0 & 1 & 2, then copy them into the DEVICE address */
          dev->dev_addr[i*2] =   ior ( 0x0D );     /* the low-byte  of this 16-bit word */
          dev->dev_addr[i*2+1] = ior ( 0x0E );     /* the high-byte of this 16-bit word */
     }
```

```
    for (i = 0, offset = 0x10; i < 6; i++, offset++)  /* set Ethernet node address */
         iow ( offset, dev->dev_addr[i] );  /* write the MAC address into PAB0~ PAB5 */

    for (i = 0; i < 4; i++)  hash_table[i] = 0;       /* clear the 4-word Hash table */

    hash_table[3] = 0x8000;                /* set the Broadcast packet passing */

    /* calculating the 64-bit Hash table for the Multicast addresses */
    /* compute the CRC32 of the destination address, firstly, and
       the six most significant bits are the hash value:
       the upper 3 bits determine the multicast register MT0~MT7,
       the next 3 bits determine the bit within the register. */
    for (i = 0; i < mc_cnt; i++, mcptr = mcptr->next) {
         hash_val = cal_CRC( (char *)mcptr->dmi_addr, 6, 0 ) & 0x3f;
         hash_table[hash_val / 16] |= (UINT16) 1 << (hash_val % 16);
    }

    /* write the Hash table 8 bytes into MAB0~MAB7 (REG. 16H~1DH) for the MAC MD table */
    for (i = 0, offset = 0x16; i < 4; i++)  {
         iow ( offset++, hash_table[i] & 0xff );
         iow ( offset++, (hash_table[i] >> 8) & 0xff );
    }
}    /* end dm9000_hash_table function */
```

```
/*   calculating the CRC-checksum value of the RX packet incoming
     flag = 1 : return the reverse CRC (for the received packet CRC)
            0 : return the normal CRC (for the Hash-table index)      */

unsigned long cal_CRC (unsigned char * Data, unsigned int Len, UINT8 flag)
{
     // UINT32 Crc = ether_crc_le(Len, Data);/* #include <linux/crc32.h> Linux OS support */
     unsigned long Crc = 0xFFFFFFFF;

     while (Len--) {
```

```
          Crc = CrcTable[(Crc ^ *Data++) & 0xFF] ^ (Crc >> 8);
     }
     if (flag) return ~Crc;
     else      return Crc;
}    /* end cal_CRC function */
```

```
/* CRC32 Table to output one parameter for the fast cal_CRC () module immediately */
UINT32 CrcTable[256] = {
     0x00000000L, 0x77073096L, 0xEE0E612CL, 0x990951BAL,
     0x076DC419L, 0x706AF48FL, 0xE963A535L, 0x9E6495A3L,
     0x0EDB8832L, 0x79DCB8A4L, 0xE0D5E91EL, 0x97D2D988L,
     0x09B64C2BL, 0x7EB17CBDL, 0xE7B82D07L, 0x90BF1D91L,
     0x1DB71064L, 0x6AB020F2L, 0xF3B97148L, 0x84BE41DEL,
     0x1ADAD47DL, 0x6DDDE4EBL, 0xF4D4B551L, 0x83D385C7L,
     0x136C9856L, 0x646BA8C0L, 0xFD62F97AL, 0x8A65C9ECL,
     0x14015C4FL, 0x63066CD9L, 0xFA0F3D63L, 0x8D080DF5L,
     0x3B6E20C8L, 0x4C69105EL, 0xD56041E4L, 0xA2677172L,
     0x3C03E4D1L, 0x4B04D447L, 0xD20D85FDL, 0xA50AB56BL,
     0x35B5A8FAL, 0x42B2986CL, 0xDBBBC9D6L, 0xACBCF940L,
     0x32D86CE3L, 0x45DF5C75L, 0xDCD60DCFL, 0xABD13D59L,
     0x26D930ACL, 0x51DE003AL, 0xC8D75180L, 0xBFD06116L,
     0x21B4F4B5L, 0x56B3C423L, 0xCFBA9599L, 0xB8BDA50FL,
     0x2802B89EL, 0x5F058808L, 0xC60CD9B2L, 0xB10BE924L,
     0x2F6F7C87L, 0x58684C11L, 0xC1611DABL, 0xB6662D3DL,
     0x76DC4190L, 0x01DB7106L, 0x98D220BCL, 0xEFD5102AL,
     0x71B18589L, 0x06B6B51FL, 0x9FBFE4A5L, 0xE8B8D433L,
     0x7807C9A2L, 0x0F00F934L, 0x9609A88EL, 0xE10E9818L,
     0x7F6A0DBBL, 0x086D3D2DL, 0x91646C97L, 0xE6635C01L,
     0x6B6B51F4L, 0x1C6C6162L, 0x856530D8L, 0xF262004EL,
     0x6C0695EDL, 0x1B01A57BL, 0x8208F4C1L, 0xF50FC457L,
     0x65B0D9C6L, 0x12B7E950L, 0x8BBEB8EAL, 0xFCB9887CL,
     0x62DD1DDFL, 0x15DA2D49L, 0x8CD37CF3L, 0xFBD44C65L,
     0x4DB26158L, 0x3AB551CEL, 0xA3BC0074L, 0xD4BB30E2L,
     0x4ADFA541L, 0x3DD895D7L, 0xA4D1C46DL, 0xD3D6F4FBL,
     0x4369E96AL, 0x346ED9FCL, 0xAD678846L, 0xDA60B8D0L,
```

```
0x44042D73L, 0x33031DE5L, 0xAA0A4C5FL, 0xDD0D7CC9L,
0x5005713CL, 0x270241AAL, 0xBE0B1010L, 0xC90C2086L,
0x5768B525L, 0x206F85B3L, 0xB966D409L, 0xCE61E49FL,
0x5EDEF90EL, 0x29D9C998L, 0xB0D09822L, 0xC7D7A8B4L,
0x59B33D17L, 0x2EB40D81L, 0xB7BD5C3BL, 0xC0BA6CADL,
0xEDB88320L, 0x9ABFB3B6L, 0x03B6E20CL, 0x74B1D29AL,
0xEAD54739L, 0x9DD277AFL, 0x04DB2615L, 0x73DC1683L,
0xE3630B12L, 0x94643B84L, 0x0D6D6A3EL, 0x7A6A5AA8L,
0xE40ECF0BL, 0x9309FF9DL, 0x0A00AE27L, 0x7D079EB1L,
0xF00F9344L, 0x8708A3D2L, 0x1E01F268L, 0x6906C2FEL,
0xF762575DL, 0x806567CBL, 0x196C3671L, 0x6E6B06E7L,
0xFED41B76L, 0x89D32BE0L, 0x10DA7A5AL, 0x67DD4ACCL,
0xF9B9DF6FL, 0x8EBEEFF9L, 0x17B7BE43L, 0x60B08ED5L,
0xD6D6A3E8L, 0xA1D1937EL, 0x38D8C2C4L, 0x4FDFF252L,
0xD1BB67F1L, 0xA6BC5767L, 0x3FB506DDL, 0x48B2364BL,
0xD80D2BDAL, 0xAF0A1B4CL, 0x36034AF6L, 0x41047A60L,
0xDF60EFC3L, 0xA867DF55L, 0x316E8EEFL, 0x4669BE79L,
0xCB61B38CL, 0xBC66831AL, 0x256FD2A0L, 0x5268E236L,
0xCC0C7795L, 0xBB0B4703L, 0x220216B9L, 0x5505262FL,
0xC5BA3BBEL, 0xB2BD0B28L, 0x2BB45A92L, 0x5CB36A04L,
0xC2D7FFA7L, 0xB5D0CF31L, 0x2CD99E8BL, 0x5BDEAE1DL,
0x9B64C2B0L, 0xEC63F226L, 0x756AA39CL, 0x026D930AL,
0x9C0906A9L, 0xEB0E363FL, 0x72076785L, 0x05005713L,
0x95BF4A82L, 0xE2B87A14L, 0x7BB12BAEL, 0x0CB61B38L,
0x92D28E9BL, 0xE5D5BE0DL, 0x7CDCEFB7L, 0x0BDBDF21L,
0x86D3D2D4L, 0xF1D4E242L, 0x68DDB3F8L, 0x1FDA836EL,
0x81BE16CDL, 0xF6B9265BL, 0x6FB077E1L, 0x18B74777L,
0x88085AE6L, 0xFF0F6A70L, 0x66063BCAL, 0x11010B5CL,
0x8F659EFFL, 0xF862AE69L, 0x616BFFD3L, 0x166CCF45L,
0xA00AE278L, 0xD70DD2EEL, 0x4E048354L, 0x3903B3C2L,
0xA7672661L, 0xD06016F7L, 0x4969474DL, 0x3E6E77DBL,
0xAED16A4AL, 0xD9D65ADCL, 0x40DF0B66L, 0x37D83BF0L,
0xA9BCAE53L, 0xDEBB9EC5L, 0x47B2CF7FL, 0x30B5FFE9L,
0xBDBDF21CL, 0xCABAC28AL, 0x53B39330L, 0x24B4A3A6L,
0xBAD03605L, 0xCDD70693L, 0x54DE5729L, 0x23D967BFL,
```

```
     0xB3667A2EL, 0xC4614AB8L, 0x5D681B02L, 0x2A6F2B94L,
     0xB40BBE37L, 0xC30C8EA1L, 0x5A05DF1BL, 0x2D02EF8DL
};   /* end CrcTable[256] table */
```

```
#define ETHERNET_POLYNOMIAL_LE 0xedb88320U;
static unsigned long ether_crc_le(int length, unsigned char *data) /* for little-endian */
{
     UINT32 crc = 0xffffffff;      /* initial value */

     while(--length >= 0)
     {
          UINT8 current_octet = *data++;
          int bit;

          for (bit = 8; --bit >= 0; current_octet >>= 1)
          {
               if ((crc ^ current_octet) & 1)
               {
                    crc >>= 1;
                    crc ^= ETHERNET_POLYNOMIAL_LE;
               }
               else
               {
                    crc >>= 1;
               }
          }
     }
     return crc;
}
```

5-1. RX Perfect-Filter with Hash Table

Besides, the RX machine supports Perfect-Filter by setting the RCR Bit [7] = 1 with Hash table:

```
iow ( 0x05, ior (5) | 0x80 );      /* HASHALL Bit[7]=1 in REG. 05, Hash-table filter all */
```

While the MAC address of the RX packet incoming is matched to 64-bit Hash table (or it's just Ethernet node address itself), this packet would be filtered and put into the RX SRAM as one received packet. Please refer to the data sheet chapter 6.6 about RCR REG. 05 setting.

```
/* 8-byte MARs: 3 node address is 00:60:6e:66:00:17, 00:60:6e:66:00:18, 00:60:6e:66:00:19 */
unsigned char  perfect_f_addr[3][6] = { {0x00, 0x60, 0x6e, 0x66, 0x00, 0x19},
 {0x00, 0x60, 0x6e, 0x66, 0x00, 0x17},  {0x00, 0x60, 0x6e, 0x66, 0x00, 0x18}    };
unsigned int   i, hash_table[4];
unsigned long  hash_val;

for (i = 0; i < 4; i++)       hash_table[i] = 0;

for (i = 0; i < 3; i++) {
     hash_val = cal_CRC( (char *)perfect_f_addr[i], 6 ) & 0x3f;
     hash_table[hash_val / 16] |= (unsigned int) 1 << (hash_val % 16);
}

for (i = 0; i < 4; i++) {
     iow ( 0x16 + 2*i, hash_table[i] & 0xff );
     iow ( 0x16 + 2*i+1, (hash_table[i] >> 8) & 0xff );
}

iow ( 0x05, ior (5) | 0x80 ); /* REG. 05 HASHALL Bit[7] = 1 */
```

6 How to Transmit Packets

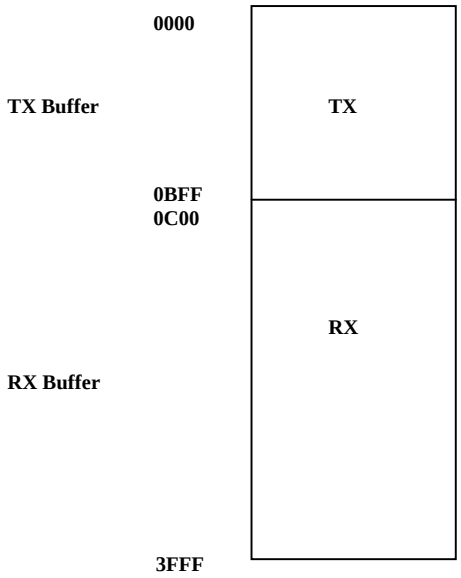


Figure 1. Packet Transmitting Buffer

Before transmitting a packet, the data of the packet must save into the TX FIFO, which is the internal SRAM address 0 ~ 0xBFF in the DM9000A MAC and to write F8H into INDEX port firstly (it means MWCMD REG. F8H port number latched). Then, the length of the packet is put in MDRAH REG. FDH for the high-byte and MDRAL REG. FCH for the low-byte. The final step is to set the TXREQ (Transmit Request) Bit [0] = 1 in TCR REG. 02 for transmitting this packet.

The DM9000A will generate an interrupt at the PT Bit [1] = 1 in ISR REG. FEH, if setting the PTI Bit [1] = 1 in IMR REG. FFH, and also to set a completion flag to either the TX1END Bit [2] = 1 or the TX2END Bit [3] = 1 in NSR REG. 01 in toggle to indicate that the packet is transmitted completely.

6-1. Packet Transmission

Step 1: check the memory 8/ 16 DATA width,

```
(UINT8) IO_mode = ior ( 0xFE ) >> 7;    /* ISR Bit [7] IOMODE indicating DATA I/O mode */
```

Step 2: write the packet's data into TX FIFO SRAM,

```
outb ( IOaddr, 0xF8 );                  /* trigger MWCMD REG. F8H with write_ptr increase */
```

```
/* UINT8 TX_data[]: the transmitting data, int TX_length: the length of TX_data[] */
if ( IO_mode == 1 ) {         /* I/O 8-bit Byte mode if ( IO_mode == DM9000A_BYTE_MODE ) */
for ( i = 0; i < TX_length; i++ )       /* loop to write a Byte data into TX 3K Byte FIFO */
     outb ( TX_data[i], IOaddr + 4 );        }

else if ( IO_mode == 0 ) {    /* IO 16-bit Word mode if( IO_mode == DM9000A_WORD_MODE ) */
(int) length_tmp = ( TX_length + 1 ) / 2;    /* depended on Word mode for loop */
for ( i = 0; i < length_tmp; i++ ) /* loop to write a Word data into TX 3K Byte FIFO */
     outw ( ( (UINT16 *) TX_data)[i], IOaddr + 4 );    }
```

Step 3: write the transmitting length into MDRAL REG. FCH and MDRAH REG. FDH,

```
/* write the high-byte of the TX data length into MDRAH REG. FDH */
iow ( 0xFD, (TX_length >> 8) & 0xff );
/* write the low-byte of the TX data length into MDRAL REG. FCH */
iow ( 0xFC, TX_length & 0xff );
```

Step 4: start to transmit one packet out,

```
iow ( 0x02, 0x01 );           /* set the TX request command, TXREQ  Bit [0] in TCR REG. 02 */
```

6-2. To Check a Completion Flag

If the driver is used the polling/ interrupt method, the program segment can be inserted into the TX routine for detecting a packet transmission completed or even to double check the TX interrupt:

```
(UINT8) TX_status = ior ( 0x01 );                /* read NSR REG. 01 for TX completed */
if ( TX_status & (4 | 8) ) { /* TX success */ }   /* check if TX1END or TX2END = 1, TX ok */
```

The program segment shown as follows can be inserted into the interrupt handler, if the driver is used the interrupt driven and setting the IMR PTI Bit [1] = 1 enable:

```
(UINT8) INT_status = ior ( 0xFE );      /* got DM9000A interrupt status in ISR REG. FEH */
if ( INT_status & 0x02 ) {    /* TX packet done */     /* check if PT Bit [1]= 1, TX done */
     iow ( 0xFE, 0x02 );      }         /* clear PT Bit [1] latched in ISR REG. FEH */
// if ( INT_status & 0x01 ) { /* RX INTR routine */    /* check if PR Bit [0]= 1, RX INTR */
//   iow ( 0xFE, 0x01 );      }         /* clear PR Bit [0] latched in ISR REG. FEH */
```

7 How to Receive Packets

The received data is stored in the RX SRAM, which is the internal RX memory and the address is at 0C00~3FFF in the DM9000A FIFO. There are four bytes for the header data of each packet stored in the RX SRAM and used MRCMDX REG. F0H and MRCMD REG. F2H registers to get the information.

The first byte is used to check whether a package is received in the RX SRAM. If Bit [1:0] of the first byte is "01", it means there is a packet received. If Bit [1:0] is "00", it means there is no packet received in the RX SRAM. Before reading the other bytes, it must make sure the Bit [1:0] is "01". The second byte is stored the status of the received packet. The format of the status byte is the same as RSR REG. 06. According to this status format, the received packet can be verified as a correct packet or an error packet. Please refer to the DM9000A data sheet chapter 6.7 about the RSR register. The third and fourth bytes are the length of the received packet. The others bytes are the received packet data, the frame payload and 4-byte CRC. The following diagram is shown the format of the received package:

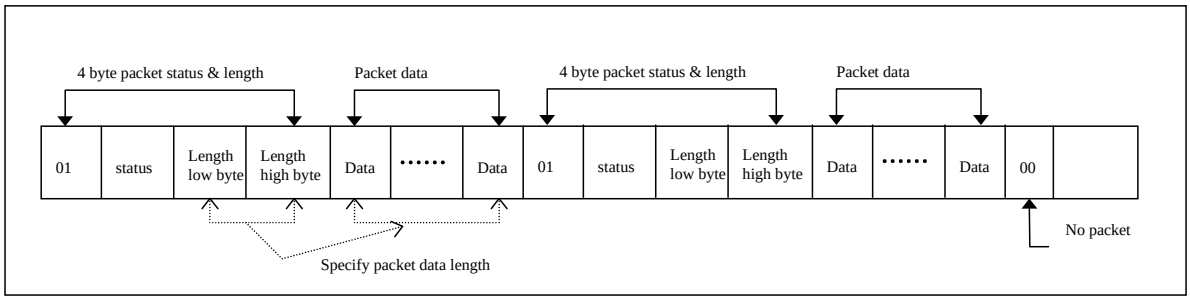


Figure 2. Block Diagram of the Packets Received

7-1. How to identify the RX Packet Ready

The following program segment can be inserted to the interrupt handler if the driver is designed as the interrupt driven or the polling method for waiting packets received ready:

```
(UINT8) INT_status = ior ( 0xFE );      /* got DM9000A interrupt status in ISR REG. FEH */
if (INT_status & 0x01) { /* follow chapter 7-2 & 7-3 */ /* check if PR Bit[0]=1, RX INTR */
     iow ( 0xFE, 0x01 ); }              /* clear the PR Bit [0] latched in ISR REG. FEH */
```

7-2. Get the RX Packet Status and Length

MRCMD (REG. F2H): memory data read command with the increment of the RX read pointer. The read pointer will be increased after reading the memory read command MRCMD REG. F2H. The size, the increment of the RX read pointer, is depended on the system application, which the I/O Bus width is Byte/ Word to increase one or two bytes respectively. MRCMD REG. F2H is only used to read the RX packet's status, length and the packets' data from the 13K-Byte RX SRAM.

For example, to get the RX packet's status and length:

```
(UINT8) IO_mode = ior ( 0xFE ) >> 7;    /* ISR Bit [7] IOMODE indicating DATA I/O mode */
outb ( 0xF2, IOaddr );                  /* trigger MRCMD REG. F2H with read_ptr increase */
/* (int) RX_status: the RX packet status, (int) RX_length: the length of the RX packet */
if ( IO_mode == 1 ) {         /* I/O 8-bit Byte mode if ( IO_mode == DM9000A_BYTE_MODE ) */
     RX_status = inb ( IOaddr + 4 ) + ( inb ( IOaddr + 4 ) << 8 );
     RX_length = inb ( IOaddr + 4 ) + ( inb ( IOaddr + 4 ) << 8 );    }

else if ( IO_mode == 0 ) {    /* I/O 16-bit Word mode if ( IO_mode == DM9000A_WORD_MODE ) */
     RX_status = inw ( IOaddr + 4 ); /* the high-byte is RX status as RSR REG. 06 format */
     RX_length = inw ( IOaddr + 4 );         }
```

7-3. Get the RX Packet Data

The read pointer will be increased after reading the memory read command MRCMD REG. F2H. According to the length of the received packet, dump out all of the RX payload and the 4-byte CRC.

For example, to get the RX packet frame:

```
/* UINT8 RX_data[]: the data of the received packet with the 4-byte CRC checksum */
if ( IO_mode == 1 ) {         /* I/O 8-bit Byte mode if ( IO_mode == DM9000A_BYTE_MODE ) */
for ( i = 0 ; i < RX_length ; i++ )     /* loop to READ a Byte data from 13K-Byte RX SRAM */
     RX_data[ i ] = (UINT8) inb ( IOaddr + 4 );             }
else if ( IO_mode== 0 ) {     /* I/O 16-bit Word mode if ( IO_mode == DM9000A_WORD_MODE ) */
(int) length_tmp = ( RX_length + 1 ) / 2;    /* Word mode */
for ( i = 0 ; i < length_tmp ; i++ )    /* loop to READ a Word data from 13K-Byte RX SRAM */
     ( (UINT16 *)RX_data)[ i ] = inw ( IOaddr + 4 );        }
```

8 Flow Control

8-1. How to setup Flow Control

For the RX flow control mode, to set the FLCE Bit [0] = 1 in the RX/ TX Flow control register (FCR REG. 0AH) to enable the DM9000A RX Flow control function of the. Then, the NIC automatically responds the TX pause packet with the pause time = 0000H / FFFFH from the link partner, in order to prevent the RX SRAM of the link partner overrunning if the NIC sending too fast, and to improve the RX/ TX performance. Please refer to the data sheet chapter 6.11 about the FCR register settings.

For example, to enable the flow control mode:

```
iow ( 0x0A, 0x21 );      /* set-bit FLCE Bit [0] & TXPEN Bit [5] to enable Flow-control */
iow ( 0x0A, ior ( 0x0A ) | 0x08 ); /* set-bit BKPM Bit [3] for Flow-control half duplex */
```

8-2. How to force Flow Control Pause Packet

Setting the TXPEN Bit [5] = 1 in the RX/ TX flow control register (FCR REG. 0AH) to force TX pause packet enable. Then, the NIC automatically sends the TX pause packet with the pause time = FFFFH / 0000H while the RX SRAM free space less/ larger than the high/ low water threshold in FCTR REG. 09. The trigger condition to send the pause packet is depending on the value of the Bit [3:0] LWOT and the Bit [7:4] HWOT of FCTR REG. 09, for checking the free space in the RX SRAM. Or for test, manually extra setting the TXP0 Bit [7] = 1 or the TXPF Bit [6] = 1 in FCR to force the TX pause packet with the pause time = 0000H / FFFFH. Please refer to the data sheet chapter 6.10 about the FCTR register.

For example, to force TX pause packet enabled:

```
iow ( 0x08, 0x3F );      /* BPTR REG. 08 (if necessary) High_water Overflow 3KB, 600us */
iow ( 0x09, 0x5A );      /* FCTR REG. 09 (if necessary) High/ Low Threshold 5KB/ 10KB */
iow ( 0x0A, 0x29 );      /* TXPEN Bit[5]= 1 in FCR REG. 0AH: force TX pause packet enable */
iow ( 0x0A, 0xA9 );      /* TXP0 Bit[7]= 1: force one TX pause packet= 0000H to hurry up! */
```

Note: the back pressure mode is only for the half duplex mode, and just set-bit the BKPM Bit [3] or the BKPA Bit [4] in FCR REG. 0AH to enable it (see the data sheet chapter 6.9 about the BPTR register).

9 Receive Watch-dog Timer and Transmit Jabber Timer

To check the RWTO Bit [4] = 1 in RSR REG. 06 for the Receive Watch-dog Timer to indicate that the received packet frame is more than 2048 Bytes, and the WTDIS Bit [6] in RCR REG. 05 must be "1" (it's default "0") to disable RX Watch-dog Timer. Please refer to the DM9000A data sheet chapter 6.6 about the RCR register setting and the chapter 6.7 about the status bits of the RSR register.

For example, to check the Receive Watch-dog Timer:

```
/* UINT16 RX_status: the RX packet status, int RX_length: the length of the RX packet */
If (RX_status & 0x1000)  {    /* REG. 06 RWTO Bit[4]= 1 indicated RX length >= 2048 Bytes */ }
```

To check the TJTO Bit [7] = 1 in TSR1/ 2 REG. 03/ 04 for the Transmit Jabber Timer to indicate that the transmitted packet frame is more than 2048 Bytes, and the TJDIS Bit [6] in TCR REG. 02 must be "1" (it's default "0") to disable TX Jabber Timer. Please refer to the DM9000A data sheet chapter 6.3 about setting the TCR register and the chapter 6.4 & 6.5 about the TSR1 and TSR2 status registers.

For example, to check the Transmit Jabber Timer:

```
/* UINT8 TX_data[]: the transmitting data, int TX_length: the length of TX_data[] */
If (TX_length > 2048)    {    /* TSR1 /2 TJTO Bit[7] = 1 indicated TX length > 2048 Bytes */ }
```

9-1. How to Transmit and Receive more than 2048-Byte Packets

If a system transmits or receives more than 2048 Byte packets, which size is although over 802.3 spec., the TJDIS Bit [6] in TCR REG. 02 and the WTDIS Bit [6] in RCR REG. 05 should be set to "1"b firstly. For example,

```
#define   TCR  0x02
#define   RCR  0x05
iow ( TCR, ior ( TCR ) | 0x40 );   /* TCR Bit [6] TJDIS enable for TX length > 2048 Bytes */
iow ( RCR, ior ( RCR ) | 0x40 );   /* RCR Bit [6] WTDIS enable for RX length > 2048 Bytes */
```

10 WOL (Wake-up on LAN)

The DM9000A supports the WOL function and the pin 22 WAKE (or the pin 38 LED2, if setting the EEPROM Word 7 Bit [13:12] = "10" in 16-bit mode) output the wake-up signal to the embedded system. There are three methods to generate WOL signals: the Magic Packet, the Link Change, and the Sample Frame types. This section will discuss how to implement the WOL function in the DM9000A.

(1) Magic Packet

If the DM9000A receives a broadcast packet which content is "FF:FF:FF:FF:FF:FF" + "16 times of Ethernet node address", then the wake-up pin will be activated. Please note that the individual Ethernet node address must be the same as the unicast address in the PAR REG. 10H ~ REG. 15H.

For example:

```
#define   NCR  0x00
#define   WCR  0x0F
iow ( WCR, ior ( WCR ) | 0x08 );        /* Magic Packet event enable */
iow ( NCR, ior ( NCR ) | 0x40 );        /* Wake-up remote enable */
/* If Ethernet node address in PAR REG. 10H ~ REG. 15H is 00:60:6e:90:00:01 */
/* DM9000A WOL would be active, while received the Magic packet as follows, */
/* FF FF FF FF FF FF 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 */
/* 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 */
/* 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 */
/* 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 00 60 6E 90 00 01 */
/* 00 60 6E 90 00 01 (+ another DM9000A TX 4-byte CRC auto appends)        */
```

(2) Link Change

If the link status of the internal PHY has been changed, the wake-up pin will be active.

For example,

```
#define   NCR  0x00
#define   WCR  0x0F
iow ( WCR, ior ( WCR ) | 0x20 );        /* Link Change event enable */
iow ( NCR, ior ( NCR ) | 0x40 );        /* Wake-up function enable */
```

(3) Sample Frame

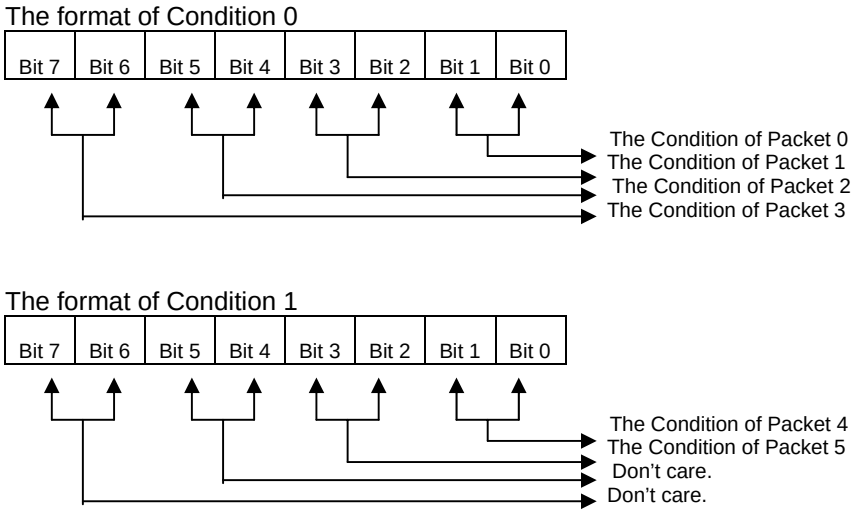
The Sample Frame would be made up of 6 packets and the maximum size of each packet can be 2048-byte. The wake-up pin will be active if the DM9000A receives any one set of the sample frames.

The following description will show how to set the Sample Frame in detail,

- Firstly close the DM9000A receiver function and clear-bit the RXEN Bit [0] in RCR REG. 05.
- Stop the self recover function: clear-bit the PAR Bit [7] in IMR REG. FFH.
- Save the packet (up to 6 max packets) into the FIFO SRAM. The format is shown below,

m-byte3 / **Packet 3**	m-byte2 / **Packet 2**	m-byte1 / **Packet 1**	m-byte0 / **Packet 0**	**DM9000A SRAM Memory location**
Packet 3-byte 0	Packet 2-byte 0	Packet 1-byte 0	Packet 0-byte 0	**0x0000**
Packet 3-byte 1	Packet 2-byte 1	Packet 1-byte 1	Packet 0-byte 1	0x0004
:	:	:	:	**0x0008**
:	:	:	:	0x000C
:	:	:	:	**0x0010**
:	:	:	:	:
Packet 3-byte 2047	Packet 2-byte 2047	Packet 1-byte 2047	Packet 0-byte 2047	**0x1FFC**

m-byte3 / **Packet 5**	m-byte2 / **Packet 4**	m-byte1/ **Condition 1**	m-byte0 / **Condition 0**	**DM9000A SRAM Memory location**
Packet 5-byte 0	Packet 4-byte 0	Cond. 1-byte 0	Cond. 0-byte 0	**0x2000**
Packet 5-byte 1	Packet 4-byte 1	Cond. 1-byte 1	Cond. 0-byte 1	0x2004
:	:	:	:	**0x2008**
:	:	:	:	0x200C
:	:	:	:	**0x2010**
:	:	:	:	:
Packet 5-byte 2047	Packet 4-byte 2047	Cond. 1-byte 2047	Cond. 0-byte 2047	**0x3FFC**



If the condition is "00"b or "01"b, don't care the byte; if "10"b, check the byte is matched or not; and if "11"b, the DM9000A MAC stops checking the sample frame.

- Then, restart the Sample Frame function by setting the SAMPLEEN Bit [4] in WCR REG. 0FH to "1" and enable the WOL function by setting the WAKEEN Bit [6] = 1 in NCR REG. 00.
- Restart the DM9000A receiver function by setting the RXEN Bit [0] in RCR REG. 05 to be "1".

For example,

```
#define   NCR  0x00
#define   RCR  0x05
#define   WCR  0x0F
#define   MWRL 0xFA
#define   MWRH 0xFB
#define   IMR  0xFF

iow ( RCR, ior ( RCR ) & 0xfe ); /* REG. 05 RXEN Bit[0]= 0 to disable RX machine/ filter */
iow ( IMR, ior ( IMR ) & 0x3f );/* REG. FFH disable PAR, Pointer Auto Return for TX& RX */

/* the Sample Frame put in the 16K Byte RX/ TX FIFO SRAM: 0x0000~ 0x3FFF */
for (i = 0, sample_ptr = 0; i < sample_length; i++, sample_ptr += 4)
{
     /* set sample_ptr into MWRH & MWRL in the DM9000A SRAM */
```

```
     iow ( MWRH, ( sample_ptr >> 8 ) & 0xff );    /* set sample_ptr high-byte into MWRH */
     iow ( MWRL, sample_ptr & 0xff );             /* set sample_ptr low-byte into MWRL */
     outb ( 0xF8, IOaddr );   /* outputting port number MWCMD=0xF8 TX FIFO WRITE locking */
     if (i % 2) outw ( ( ((UINT16 *) sample_frame)[( i+1 )/ 2] >> 8), IOaddr + 4 );
     else outw ( ((UINT16 *) sample_frame)[( i+1 )/ 2], IOaddr + 4 ); /* in 16-bit mode */
}

/* write the condition Byte [0] & Byte [1] of the Sample Frame into 0x2000~ 0x3FFD */
for (i = 0, sample_ptr = 0x2000; i < sample_length; i++, sample_ptr += 4)
{
     /* set sample_ptr into MWRH & MWRL in the DM9000A SRAM */
     iow ( MWRH, ( sample_ptr >> 8 ) & 0xff );    /* set sample_ptr high-byte into MWRH */
     iow ( MWRL, sample_ptr & 0xff );             /* set sample_ptr low-byte into MWRL */
     outb ( 0xF8, IOaddr );   /* outputting port number MWCMD=0xF8 TX FIFO WRITE locking */
     outw ( 0x0002, IOaddr + 4 );       /* fill Bit [1:0]="10" of condition 0 in packet 0 */
}
// iow ( MWRH, ( (sample_ptr+ 4) >> 8) & 0xff );
// iow ( MWRL, ( sample_ptr + 4 ) & 0xff);
// outw ( 0x0003, IOaddr+ 4 );          /* fill the Bit [1:0]="11"b to stop checking */

iow ( WCR, ior ( WCR) | 0x10 );         /* Sample Frame enable */
iow ( NCR, ior ( NCR) | 0x40 );         /* Wake-up remote enable */
iow ( RCR, ior ( RCR) | 0x01 ); /* REG. 05 RXEN Bit[0] = 1 to enable RX machine/ filter */

/* Then, the wake-up pin active if the DM9000A receives any one set of the sample frames */
```

11 Early Transmit

Early-Transmit and Two-packet-mode are mutually exclusive for the DM9000A/ 9010 chip.

For Early-Transmit feature, it's very easy to set ETXCSR REG. 30H: enable the ETE Bit [7] = 1 and setting the ETT Bit [1:0] value for the Threshold percentage %. Then, to fill the TX packet length into MDRAH REG. FDH (the high byte) and MDRAL REG. FCH (the low byte), and it's un-necessary to issue the TX polling command, the TXREQ Bit [0] in TCR REG. 02.
But, make sure to disable the function of Transmit IP/TCP/UDP Checksum generation in TCSCR REG. 31H at the same time.
Finally, the ETS1 Bit [5] and the ETS2 Bit [6] are indicating Early-Transmit packet I/ II either status under-run or not while the packet I/ II transmitting. "1": Early TX under-run event; "0": none. And, the UDRUN Bit [4] in ISR REG. FEH is also latched "1" if TX under-run event happened.
Please refer to the data sheet chapter 6.26 about ETXCSR REG. 30H setting.

The Early TX Threshold percentage table is shown as follows,

ETXCSR Bit [1]	ETXCSR Bit [0]	Early TX Threshold %
0	0	12.5%
0	1	25%
1	0	50%
1	1	75%

For example, to set the Early TX Threshold 75%:

```
#define   ETXCSR    0x30
#define   TCSCR     0x31
static int EarlyTxFlag = 1;


iow ( TCSCR, 0 );                 /* TX IP/TCP/UDP Checksum generation disable firstly */
iow ( ETXCSR, 0x80 | 0x03 );      /* Early TX Threshold 75% ("11"b) in ETXCSR REG. 30H */


/* Then, the ETS1/ETS2 bit indicating Early TX status, while packet I/II under-run or not */
```

```
static int dmfe_start_xmit (struct sk_buff *skb, struct net_device *dev)
{
     /* .. */
     db->tx_pkt_cnt ++;                /* TX-packet-counter + 1: 0 -> 1, or 1 -> 2 */
     iow ( 0xFD, (skb->len >> 8) & 0xff ); /* set TX length into MDRAH REG. FDH */
     iow ( 0xFC, skb->len & 0xff );     /* set TX length into MDRAL REG. FCH */
     outb ( 0xF8, IOaddr );             /* trigger TX MWCMD port number 0xF8 latched */
     /* .. */
     /* move TX data into the DM9000A TX FIFO SRAM */
     for ( i = 0; i < ( skb->len + 1 ) / 2; i++ )
          outw ( ( (u16 *)data_ptr)[i], IOaddr + 4 );
     /* .. */
     /* issue TX request command, the TXREQ Bit [0] in TCR REG. 02, if Early TX disable */
     if ( !EarlyTxFlag )      iow ( 0x02, 0x01 );
     /* .. */
}
```

11-1. Transmit Two-Packet-Mode

The DM9000A could automatically send out two TX packets with the maximum length 3072 Byte. As the first packet is sending and the second packet is queued in the TX 3K Byte FIFO, the MAC will automatically send the second packet out immediately when the first packet transmitted completion.

TX Two-packet-mode is to fill two packets into the TX FIFO SRAM continuously. And, it’s the DM9000A default mode to latch the two TX packets’ lengths and two TX request (TXREQ) commands in the MAC for transmitting high performance.

The DM9000A can issue TX Two-packet-mode or One-packet-mode, by setting the ONEPM Bit [4] either "1" or "0" in TX Control Register 2 (TCR2 REG. 2DH). When this ONEPM bit cleared to be "0" at most, it’s the default Two-packet-mode for filling two packets transmission continuously.

While to set-bit the ONEPM Bit [4] of TCR2 REG. 2DH put in the DM9000A initial function, it is enable TX One-packet-mode and disable the default Two-packet-mode.

For example, to set TX One-packet-mode:

```
iow ( 0x2D, 0x10 );                          /* TCR2 REG. 2DH TX One-packet-mode setting */

if ( db->tx_pkt_cnt > 1 )     return 1;      /* too many TX packets to sending */

data_ptr = (char *)skb->data;                /* set TX pointer to move data into TX FIFO */
outb ( 0xF8, IOaddr );                       /* trigger TX MWCMD port number 0xF8 latched */

db->sent_pkt_len = skb->len;
tmplen = ( skb->len + 1 ) / 2;               /* TX length changed for Word DATA I/O mode */
for ( i = 0; i < tmplen; i++ )     outw ( ( (u16 *)data_ptr )[i], IOaddr + 4 );

/* TX control: let first packet immediately send, or second packet queue */
if ( db->tx_pkt_cnt < 1 ) {                  /* sending immediately if first packet */
     iow ( 0xFD, (skb->len >> 8) & 0xff );   /* set TX length into MDRAH REG. FDH */
     iow ( 0xFC, skb->len & 0xff );          /* set TX length into MDRAL REG. FCH */

     /* issue TX polling command */
     iow ( 0x02, 0x01 );                     /* TXREQ bit auto cleared after TX completed */
     db->tx_pkt_cnt = 1;                     /* TX-packet-counter + 1: let value 0 to be 1*/
} else {                                     /* second packet queued in TX FIFO */
     db->queue_pkt_len = skb->len;
     db->tx_pkt_cnt = 2;                     /* TX-packet-counter + 1: let value 1 to be 2*/
}
return 0;                                    /* TX packets sending */



/* the driver interrupt handler or polling routine to detect TX packet completion */
(UINT8) TX_status = ior ( 0x01 );            /* got TX1END & TX2END status bits from NSR */
if ( TX_status & 0xc ) {                     /* TX event if one packet sent completed */
     db->tx_pkt_cnt --;                      /* TX-packet-counter - 1: 1 -> 0, or 2 -> 1 */
     if ( db->tx_pkt_cnt > 0 ) {             /* checking if queued TX packet, 1:yes to send*/
          iow ( 0xFD, (db->queue_pkt_len >> 8) & 0xff );    /* set TX length into MDRAH */
```

```
        iow ( 0xFC, db->queue_pkt_len & 0xff );          /* set TX length into MDRAL */
        iow ( 0x02, 0x01 );      /* issue TX polling command TXREQ then auto cleared */
        db->tx_pkt_cnt = 1;      /* TX counter = 1: let second packet send now */
    }
}
```

11-2. Transceiver with AUTO-MDIX

Besides, the DM9000A supports AUTO-MDIX powerful feature while power-on chip at default.

The DM9000A is default turning on AUTO-MDIX, and there are two ways to disable it:

1. To set the SROM Word 7, Wake-up Mode Control, Bit [14] = 1 then clear-bit for re-load it,

```
srom_write (0x07, 0x180);     /* PHY power-up but disable AUTO-MDIX setting into SROM */
iow ( 0x0B, 0x20 );           /* REEP Bit [5] of EPCR REG. 0BH to re-load EEPROM value */
iow ( 0x0B, 0x00 );           /* REEP Bit [5]= 0 cleared after 400us delay for re-load */
```

2. To write high "1" to the Mdix-down Bit [4] in the PHY REG. 20,

```
phy_write(20, 0x0010);        /* Mdix_down Bit[4]= 1 with Mdix_fix_Value Bit[5]=0 for MDI */
phy_write(20, 0x0030);        /* Mdix_down Bit[4]=1 with Mdix_fix_Value Bit[5]=1 for MDIX */
```

12 IP/TCP/UDP Checksums Offload

The DM9000A chips support the IP/TCP/UDP TX checksum generations and RX status checking. And enable the IP/TCP/UDP checksums offload functions in the TCP/IP upper layers of the embedded OS, while setting the DM9000A IP/TCP/UDP checksums generation and checking for checksums offload. Then any RX IP/TCP/UDP packet incoming, the IPS/TCPS/UDPS Bit [7:5] of RCSCSR REG_32 will indicate RX IP/TCP/UDP checksum status and it means the packet frame is ok if the bit value is zero. List all function bits in TCSCR & RCSCSR registers:

TX CheckSum Control Register (TCSCR REG. 31H):

Bit	Name	Description
2	UDPCSE	UDP CheckSum Generation Enable
1	TCPCSE	TCP CheckSum Generation Enable
0	IPCSE	IP CheckSum Generation Enable

RX CheckSum Control Status Register (RCSCSR REG. 32H):

Bit	Name	Description
7	UDPS	UDP CheckSum Status --> "0": checksum ok if UDP packet received.
6	TCPS	TCP CheckSum Status --> "0": checksum ok if TCP packet received.
5	IPS	IP CheckSum Status --> "0": checksum ok if IP packet received.
4	UDPP	UDP Packet if indicating "1"
3	TCPP	TCP Packet if indicating "1"
2	IPP	IP Packet if indicating "1"
1	RCSEN	Receive CheckSum Checking Enable When set "1", the checksum status is inserted in status first byte of packet header.
0	DCSE	Discard CheckSum Error packet When set "1", this packet will be discarded if IP/TCP/UDP checksum field is error.

Step 1: Disable Early-Transmit function while initial:

Don't enable the Early-Transmit function in ETXCSR REG. 30H at the same time.

Step 2: To disable TX checksum calculation:

To set-bit TCSCR REG_31Bit [2:0] = 111b: IP/TCP/UDP TX checksums generation enable.

Step 3: To disable RX checksum calculation:

To set-bit RCSCSR REG_32 Bit [1] = 1: IP/TCP/UDP RX checksums checking enable.

For example, to enable RX/ TX IP/TCP/UDP checksums offload:

```
#define   ETXCSR         0x30
#define   TCSCR          0x31
#define   RCSCSR         0x32

/* Net device features in "linux-<version>/include/linux/netdevice.h" */
// #define NETIF_F_HW_CSUM    8    /* Can checksum all the packets. */
dev->features |= NETIF_F_HW_CSUM;  /* for TX IP checksum offload */
// #define CHECKSUM_UNNECESSARY    2    /* in "linux-<version>/include/linux/skbuff.h" */
skb->ip_summed = CHECKSUM_UNNECESSARY;  /* for RX TCP/UDP checksums offload */

iow ( ETXCSR, ior( ETXCSR ) & 0x7f );             /* ETXCSR REG. 30H Early TX disable */
iow ( TCSCR, 0x07 );               /* TX IPCSE+TCPCSE+UDPCSE checksums generations enable */
iow ( RCSCSR, 0x03 );              /* RX checksums checking and bad-packet-discard enable */
/* IPS/TCPS/UDPS Bit[7:5] indicating checksum status, while RX IP/TCP/UDP packet incoming */
```

Note: The Linux kernel must be modified as follows,

For stop the RX IP checksum calculation: disable if (ip_fast_csum((u8 *)iph, iph->ihl) != 0) goto inhdr_error; declared at the "int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)" function in the "linux-<version>/net/ipv4/ip_input.c" file.

For stop the TX IP checksum calculation: disable iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl); declared at the "int ip_build_xmit()" and "__inline__ void ip_send_check(struct iphdr *iph)" functions in the "linux-<version>/net/ipv4/ip_output.c" file.

For stop the TX UDP checksum calculation (in the "linux-<version>/net/ipv4/udp.c" file):

```
int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)   /* Linux kernel 2.4.x */
{
/* #line 566 added */
sk->no_check = UDP_CSUM_NOXMIT;
}
```

```
static int udp_push_pending_frames(struct sock *sk, struct udp_sock *up) /* kernel 2.6.x */
{
sk->sk_no_check = UDP_CSUM_NOXMIT;
/* the line above added before this:
if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
                skb->ip_summed = CHECKSUM_NONE;
                goto send;
        }
*/
}
```

For stop the TX TCP checksum calculation: disable kb->csum = csum_block_add(skb->csum, csum, off); and also to replace: csum = csum_and_copy_from_user(from, skb_put(skb, copy),copy, 0, &err); to be: csum = copy_from_user(skb_put(skb, copy),from,copy); declared at the "static inline int skb_add_data(struct sk_buff *skb, char *from, int copy)" function in "linux-<version>/net/ipv4/tcp.c" file.

And disable it: if (skb->ip_summed == CHECKSUM_NONE) skb->csum = csum_block_add(skb->csum, csum, skb->len); and also to replace: csum = csum_and_copy_from_user(from, page_address(page)+off, copy, 0, &err); to be: csum = copy_from_user(page_address(page)+off, from, copy); declared at the "static inline int tcp_copy_to_page(struct sock *sk, char *from, struct sk_buff *skb, struct page *page, int off, int copy)" function in the "linux-<version>/net/ipv4/tcp.c" file.

(Linux OS kernel 2.6.x is only this TCP file of "tcp_ipv4.c" modified as follows,)
For stop the TX IPv4 TCP checksum calculation: disable if (skb->ip_summed == CHECKSUM_HW) {} declared at the "void tcp_v4_send_check(struct sock *sk, struct tcphdr *th, int len, struct sk_buff *skb)" function; and also to disable: rg.csumoffset = offsetof(struct tcphdr, check) / 2; then to force arg.csum = 0; declared at the "static void tcp_v4_send_reset(struct sk_buff *skb)" function; and let it disable: arg.csumoffset = offsetof(struct tcphdr, check) / 2; then to force arg.csum = 0; declared at the "static void tcp_v4_send_ack(struct sk_buff *skb, u32 seq, u32 ack, u32 win, u32 ts)" function; at last, to force th->check=0; declared at the "static int tcp_v4_send_synack(struct sock *sk, struct open_request *req, struct dst_entry *dst)" function, which are all in the "linux-<version>/net/ipv4/tcp_ipv4.c" file.