Davicom Ethernet Software Solution

Version 1.00

Davicom Semiconductor Inc.

2009/01/07

Content

❖ Davicom Ethernet Products

- ✓ Ethernet controller
  - ◆ DM9000E
  - ◆ DM9000A/B
  - ◆ DM9102H
- ✓ Ethernet Switch controller
  - ◆ DM9003
  - ◆ DM9013
  - ◆ DM9103
  - ◆ DM8203
- ✓ USB to Ethernet controller
  - ◆ DM9601
- ✓ PHY single chip transceiver
  - ◆ DM9161

❖ Chip function & Registers

- ■ Ethernet controller
  - ◆ Ethernet
    - ● Supports processor interface of byte/word/dword mode to internal memory accesses. DM9000E/A/B for local bus interface, and DM9102H for PCI bus.

- Integrated 10/100M transceiver with HP Auto-MDIX
- The auto-negotiation function will automatically configure to take the maximum advantage of its ability.
- IEE802.3x flow control and back pressure mode

- Registers:
  - Software reset, PHY power down/up, PHY reset
  - PHY HP Auto-MDIX enable
  - PHY Auto-Negotiation enable
  - Network Control/Status registers
  - MAC address and Hash table
  - Interrupt Mask/Status registers
  - Internal memory data read/write registers
  - TX control/status Register
  - RX control/status Register

- Ethernet Switch controller
  - SWITCH
  - Registers:
    - Software reset, PHY reset
    - PHY0/1/2 by PHY address bit[1:0]
    - PHY HP Auto-MDIX enable
    - PHY Auto-Negotiation enable
    - Network Control/Status registers
    - MAC address and Hash table
    - Interrupt Mask/Status registers
    - Internal memory data read/write registers
    - TX control Register
    - RX control/status Register
    - Monitor Registers
    - Switch Control Register
    - VLAN Control Register (Reg53H)
    - VLAN mapping table Registers
    - Per Port
      - Per Port Index Register
      - Per Port Control Register

- Per Port Status Register
- MIB counter Index Register
- MIB counter Data Registers
- VLAN ID (vid[11:0])

- USB to Ethernet controller
  - USB to Ether
    - USB 1.1 compliant
    - Support 12Mb/s Full-Speed operation
    - Support USB standard commands
    - Support vendor specific commands
    - Efficient TX/RX FIFO auto management
    - The auto-negotiation function will automatically configure to take the maximum advantage of its ability.
    - IEE802.3x flow control and back pressure mode
  - Descriptor
    - Interface 0 Configuration
      - Endpoint 1, Bulk In, Read EP1 for packet from device
      - Endpoint 2, Bulk Out, Write EP2 for packet to device
      - Endpoint 3, Interrupt In, Read EP3 for content from device (8-bytes= NSR,TSR1,TSR2, RSR,ROCR,RXC,TXC(bit[2:0]),GPR)
      - Descriptor Values
        - Device Descriptor, 18bytes // lpUsbDevice
        - Configuration0 Descriptor, 9bytes
        - Interface0 Descriptor, 9bytes // lpInterface
        - Endpoint1 Descriptor, 6bytes // lpUsbFuncs
        - Endpoint3 Descriptor, 6bytes
        - String0 Descriptor, Code array, 4bytes
          - String1/2/3 Descriptor/UNICODE String (Loaded from EEPROM)
    - WinCE USB driver
      - The USB driver function table:
        - lpRegisterNotificationRoutine

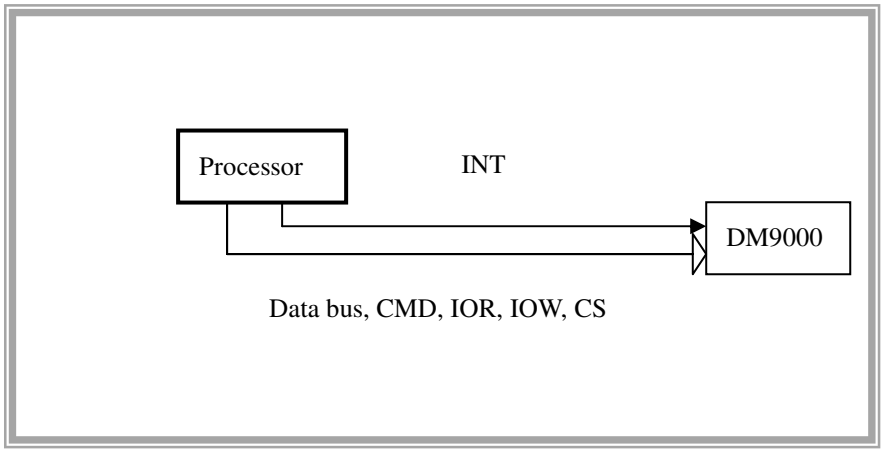
- lpGetDeviceInfo
- lpFindInterface
- lpOpenPipe, m_pipeIn, m_pipeOut, m_pipeInt
- lpIssueVendorTransfer, In/Out (hUsbHandle)
- lpIssueBulkTransfer, In (m_pipeIn),Out (m_pipeOut)
- lpGetTransferStatus
- lpCloseTransfer

- Registers:
  - Software reset, PHY power down/up, PHY reset
  - PHY Auto-Negotiation enable
  - Network Control/Status registers
  - USB Device Address Register
  - Receive Packet Counter Register(RXC)
  - Transmit Packet Counter Register(TXC)
  - USB Control
  - Status Registers

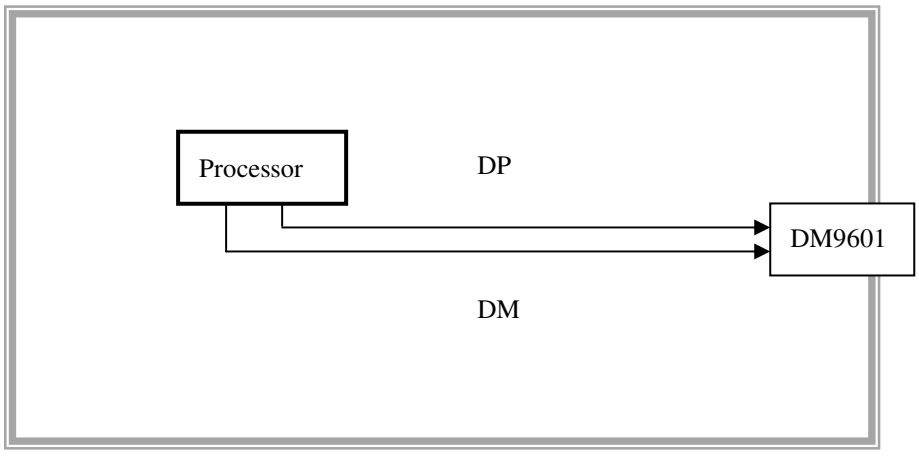
- PHY single chip transceiver
  - PHY
  - Registers:
    - PHY Reset
    - PHY HP Auto-MDIX enable
    - PHY Auto-Negotiation enable
    - Reduced MII Enable
    - Auto-Negotiation Monitor

❖ System Block Diagram

■ DM9000



■ DM9601



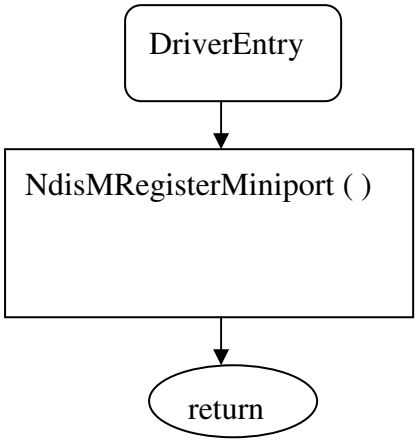
❖ Standard Network Layer (OSI Model Stack)

- **Overview**
  - **Upper layers**
    - **7.Application Layer** // HTTP, FTP
    - **6.Presentation Layer**
    - **5.Session Layer** // WinSock
  - **Lower layers**
    - **4.Transport Layer** // TCP, UDP
    - **3.Network Layer** // IP,ICMP
    - **2.Data Link Layer** // Ethernet (switch, bridge)
    - **1.Physical Layer** // Ethernet (hub, repeater)

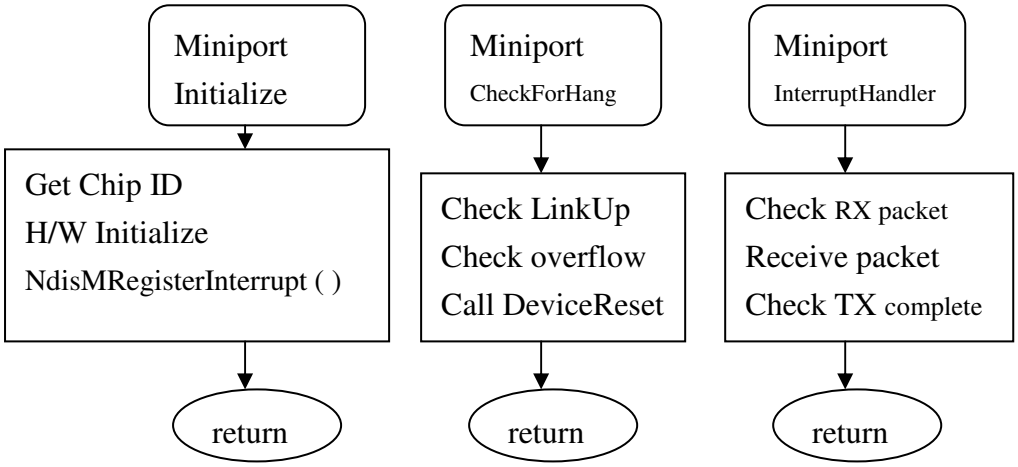
- **DM9000 works for L1 & L2**
- **Embedded system kernel works for L3~L7**

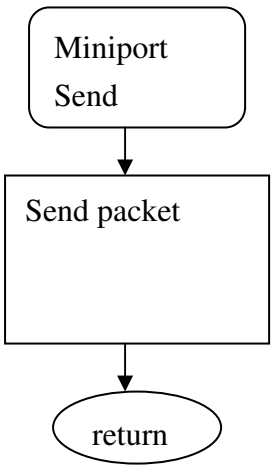
❖ Programming reference – flow chart

- Flow chart
- DM9000

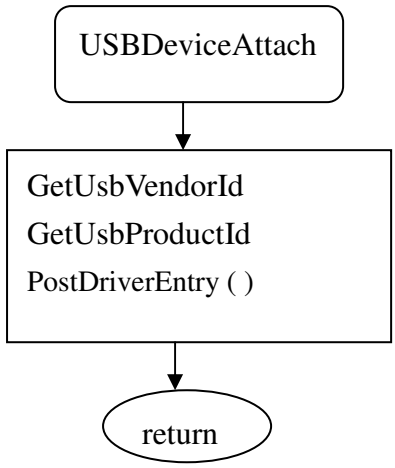


//Registers a miniport's MiniportXxx entry points:
MiniportInitialize, MiniportReset, MiniportCheckForHang, MiniportHalt, MiniportInterruptHandler, MiniportISRHandler, MiniportQueryInformation, MiniportSetInformation, MiniportSend, MiniportDisableInterrupt, MiniportEnableInterrupt





DM9601



// USBDeviceAttach ,

// Passes a USB device interface function table of the client drivers:

struct _USB_FUNCS {} ;

(This structure contains the USB driver function table.)

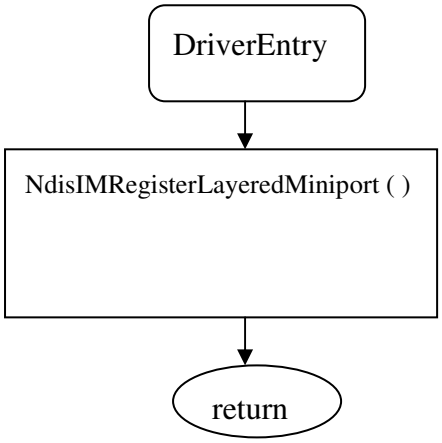
// PostDriverEntry, Save below:

g_hDevice = hDevice; // Handle to a USB device

g_lpUsbFuncs = lpUsbFuncs; // USB device interface function table

g_lpUsbDevice = lpUsbDevice; // LPGET_DEVICE_INFO function

g_lpInterface = lpInterface; // LPFIND_INTERFACE function

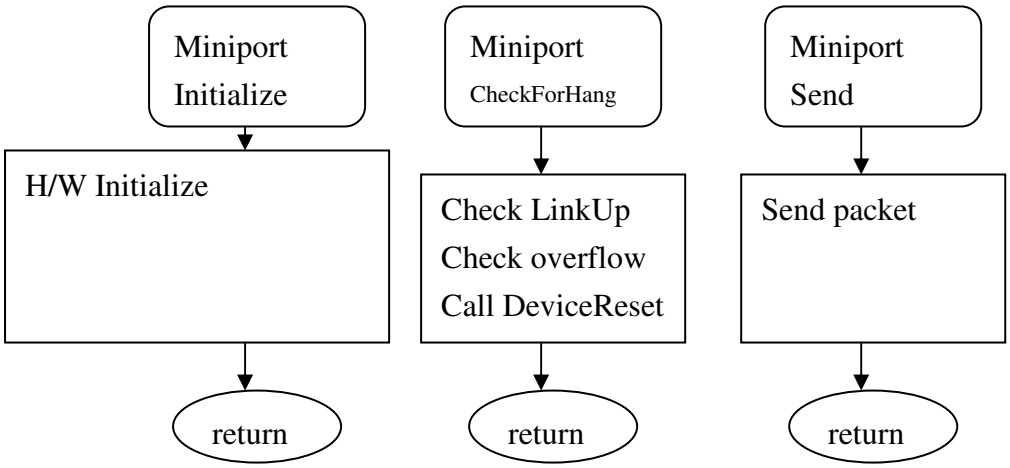


//Registers an intermediate driver's MiniportXxx entry points:

MiniportInitialize, MiniportReset, MiniportCheckForHang,

MiniportHalt, MiniportQueryInformation,

MiniportSetInformation, MiniportSend



❖ Programming reference – source program

✓ Initialize

```
void  DeviceInitialize(void)
{
        DeviceWritePort(DM9_GPCR, (1<<0));
        NdisStallExecution(1); // 1 us

        DeviceWritePort(DM9_GPR, 0x01);
        NdisStallExecution(1); // 1 us
        DeviceWritePort(DM9_GPR, 0x00);
        NdisStallExecution(1000); // 1000 us
        NdisStallExecution(1000); // 1000 us

        //reset PHY
        DeviceWritePhy(0, 0, 0x8000);
        NdisStallExecution(20); // 20 us

    //DeviceWritePort(0x38, 0x01); // 2mA
    //DeviceWritePort(0x38, 0x21); // 4mA
        DeviceWritePort(0x38, 0x41); // 6mA

        //Software reset
        DeviceWritePort(DM9_NCR, NCR_RST);
        NdisStallExecution(200);

        val = DeviceReadPort(DM9_ISR);
        if(val & MAKE_MASK(6))
        {
            WP(WSTR("IOMode :: 32 bit\r\n"));
            m_nIoMode = DWORD_MODE;
            m_nIoMaxPad = 3;
        }
        else if(!(val & MAKE_MASK(7)))
        {
```

```
                WP(WSTR("IOMode :: 16 bit\r\n"));
                m_nIoMode = WORD_MODE;
                m_nIoMaxPad = 1;
        }
        else
        {
                WP(WSTR("IOMode :: 8 bit\r\n"));
                m_nIoMode = BYTE_MODE;
                m_nIoMaxPad = 0;
        }
        //Program operating register
        DeviceWritePortB(DM9_NCR,    0x00);
        DeviceWritePortB(DM9_TXCR,    0x00);                //TX Polling clear
        DeviceWritePortB(DM9_BACKTH,    0x3f);    //Less 3Kb, 200us
        DeviceWritePortB(DM9_PAUSETH,    0x38);  //Flow Control : High/Low Water
        DeviceWritePortB(DM9_FLOW,    0xff);          //Flow Control
        DeviceWritePortB(DM9_SMCR,    0x00);          //Special Mode
        DeviceWritePortB(DM9_NSR,    0x2c);           //clear TX status
        DeviceWritePortB(DM9_ISR,    ISR_CLR_STATUS); //Clear interrupt status
        NdisStallExecution(1000); // 1000 us

        WNextLine();
        ipcr= (WORD) DeviceReadPort(0x39);
        if (ipcr&1)
          WP(_T("[#] REG39 %x, INT Polarity= Low Active\r\n"), ipcr);
        else
          WP(_T("[#] REG39 %x, INT Polarity= High Active\r\n"), ipcr);
        //ipcr |= 0x01; // Cast to be lo-active (Low)
           ipcr &= 0xfe; // Cast to be hi-active
        DeviceWritePort(0x39, ipcr);
        ipcr= (WORD) DeviceReadPort(0x39);
        if (ipcr&1)
          WP(WSTR ("[#] REG39 %x, INT Polarity= Low Active\r\n"), ipcr);
        else
          WP(WSTR ("[#] REG39 %x, INT Polarity= High Active\r\n"), ipcr);
        WNextLine();
}
```

✓ Timer

```
BOOL DeviceCheckForHang (void)
{
	U32   cr;

	U16 wLinkState;
	wLinkState= DeviceReadPhy(0, 0x1);
	wLinkState= DeviceReadPhy(0, 0x1);

	if (!(wLinkState&0x4)){

		WP(WSTR("[dm9XXX:MAC soft-rst \r\n"));

		//[software reset the device-MAC]
		//[&MAC register clear]
		DeviceWritePort(DM9_NCR, 0x03); NdisStallExecution(20); // 20 ms
		DeviceWritePort(DM9_NCR, 0x00);
		DeviceWritePort(DM9_NCR, 0x03); NdisStallExecution(20);
		DeviceWritePort(DM9_NCR, 0x00);
		//[&MAC register clear]
		DeviceWritePort(DM9_NCR, 0x03); NdisStallExecution(20);
		DeviceWritePort(DM9_NSR, 0x2C); //_RPT_WREG(RN_NSR, DM9_NSR, 0x2C);

		//[Enable memory chain][;Enable pointer auto-return for Tx & Rx FIFO]
		DeviceWritePort(DM9_IMR, (1<<7));

		DeviceWritePort(DM9_NCR, 0x00);
	}

	cr = DeviceGetReceiveStatus();
	U32   rxps,rxci;

	rxps = cr >> 31;
	rxci = cr & 0x7FFFFFFF;
```

```
        REPORT(TID_NIC_RXPS,rxps);
        REPORT(TID_NIC_RXCI,rxci);

        U32   lastread = m_szLastStatistics[TID_GEN_RCV_OK];
        U32   lastsent = m_szLastStatistics[TID_GEN_XMIT_OK];

        memcpy(
                (void*)&m_szLastStatistics,
                (void*)&m_szStatistics,
                sizeof(m_szStatistics));

        // report hang if
        // 1. receive count stalled but overflow, or
        if((m_szStatistics[TID_GEN_RCV_OK] == lastread) && cr)
        {
                WP(WSTR("dm9000:NIC_DEVICE_OBJECT::DeviceCheckForHang    1. receive count
stalled but overflow\r\n"));
                return TRUE;
        }
        // 2. tx idle while tqueue out of resource
        if(m_pUpper->DriverIsOutOfResource() &&
                (DeviceHardwareStatus() & NIC_HW_TX_IDLE) )
        {
                WP(WSTR("dm9000:NIC_DEVICE_OBJECT::DeviceCheckForHang    2. tx idle while
tqueue out of resource\r\n"));
                return TRUE;
        }
        return FALSE;
}
```

✓ Interrupt

```
void DeviceInterruptEventHandler(U32 uValue)
{
        // check RX activities
        if(uValue & 0x01)
        {
```

```
            Dm9LookupRxBuffers();
        }

        U32   nsr, txs;
        nsr = DeviceReadPort(DM9_NSR);
        txs = nsr = (nsr>>2)&0x03;

        for(;txs;txs>>=1)
        {
            if(!(txs & 1)) continue;
            m_nTx--;
            DeviceSendCompleted(m_TQWaiting.Dequeue());
        } // of check tx-end loop

        if(nsr)
        {
            DeviceSend(NULL); //C_DM9000::DeviceSend null
        }
}

int     Dm9LookupRxBuffers(void)
{
        if(!m_mutexRxValidate.TryLock()) return 0;

        int             counts=0;
        int             errors=0;

        U32             desc;
        PDM9_RX_DESCRIPTOR    pdesc;

        U8 szbuffer[DRIVER_BUFFER_SIZE];

        for(pdesc=(PDM9_RX_DESCRIPTOR)&desc;;)
        {
            CHECK_SHUTDOWN();

            // probe first byte
```

```
desc = DeviceReadDataWithoutIncrement();

// check if packet available, 01h means available, 00h means no data
if(pdesc->bState != 0x01) break;

// get the data descriptor again
desc = DeviceReadData();

// read out the data to buffer
// Performance issue: maybe we may discard the data
// just add the rx address.
// if the length is greater than buffer size, ...
if((pdesc->nLength > DRIVER_BUFFER_SIZE))
{
    DeviceIndication(AID_LARGE_INCOME_PACKET);
    break;
}

DeviceReadString((PU8)&szbuffer,pdesc->nLength);

WP(WSTR("!RC=%d!%d!"), ++nRxCnt, pdesc->nLength); //nRxCnt++;

// check status, as specified in DM9_RXSR,
// the following bits are error
// <3> PLE
// <2> AE
// <1> CE
// <0> FOE
if(pdesc->bStatus & MAKE_MASK4(3,2,1,0))
{
    errors++;
    WP(WSTR(" Err RXStatus\r\n"));
    continue;
} // of error happens

counts++;
```

```
            DeviceReceiveIndication(
                0,
                (PVOID)&szbuffer,
                pdesc->nLength);
        } // of forever read loop

        REPORT(TID_GEN_RCV_OK, counts);
        REPORT(TID_GEN_RCV_ERROR, errors);

        m_mutexRxValidate.Release();
        return counts;
}
```

- ✓ Send

```
NDIS_STATUS   DriverSend(
        PNDIS_PACKETpPacket,
        UINT            uFlags)
{
        PERF_PROBE_ON(PROBE_SEND);

        PCQUEUE_GEN_HEADER pobj;

        if(!(pobj = m_TQueue.Dequeue()))
        {
            m_bOutofResources = 1;
            WP(WSTR("<DM9:m_bOutofResources\n"));
            return NDIS_STATUS_RESOURCES;
        }

        PNDIS_BUFFERpndisFirstBuffer;
        UINT            uPhysicalBufferCount;
        UINT            uBufferCount;
        UINT            uTotalPacketLength;

        PNDIS_BUFFERpndisCurrBuffer;
        PU8         pcurr = (PU8)CQueueGetUserPointer(pobj);
```

```
        PVOID       ptrBuffer;
        UINTnBuffer;
        U32         idx,check;

        NdisQueryPacket(
                pPacket,
                &uPhysicalBufferCount,
                &uBufferCount,
                &pndisFirstBuffer,
               &uTotalPacketLength);

       if (uTotalPacketLength > ETH_MAX_FRAME_SIZE)
       {
        WP(WSTR("dm9000:NIC_DRIVER_OBJECT::DriverSend uTotalPacketLength >
ETH_MAX_FRAME_SIZE\r\n"));
             return NDIS_STATUS_FAILURE;
       }

        uPhysicalBufferCount &= 0xFFFF;

        PERF_PROBE_ON(PROBE_SEND+1);

        for(idx=0,check=0,pndisCurrBuffer=pndisFirstBuffer;
                idx < uBufferCount;
                idx++, pndisCurrBuffer = pndisCurrBuffer->Next)
        {
                NdisQueryBuffer(
                        pndisCurrBuffer,
                        &ptrBuffer,
                        &nBuffer);

                if(!nBuffer) continue;

                NdisMoveMemory(pcurr, ptrBuffer, nBuffer);
                pcurr += nBuffer;
                check += nBuffer;
```

```
        } // of for gathering buffer

        if(uTotalPacketLength != check)
        {
                WP(WSTR("dm9000:NIC_DRIVER_OBJECT::DriverSend uTotalPacketLength !=
check\r\n"));
                return NDIS_STATUS_FAILURE;
        }

        PERF_PROBE_OFF(PROBE_SEND+1);

        pobj->pPacket = (PVOID)pPacket;
        pobj->uFlags = uFlags;
        pobj->nLength = uTotalPacketLength;
        m_pLower->DeviceSend(pobj);

        PERF_PROBE_OFF(PROBE_SEND);

        return NDIS_STATUS_SUCCESS;
}

int     DeviceSend(PCQUEUE_GEN_HEADER pObject)
{
        PCQUEUE_GEN_HEADER pcurr;

        PROCESSOR_LanEnableInterrupt(); //enable int

        if(pObject) m_TQStandby.Enqueue(pObject);

       if(m_nTx >=1)
       {
                U32   nsr, txs;

                nsr = DeviceReadPort(DM9_NSR);
                txs = nsr = (nsr>>2)&0x03;

                for(;txs;txs>>=1)
```

```
            {
                if(!(txs & 1)) continue;
                m_nTx--;
                DeviceSendCompleted(m_TQWaiting.Dequeue());
            } // of check txend loop
        }

        WP(WSTR("!mTx!%d"),m_nTx);
        for(; m_nTx < 1;)
        {
            /* if there is no more standby packet, claim IDLE */
            if(!(pcurr = m_TQStandby.GetHead()))
            {
                DeviceIndication(NIC_IND_TX_IDLE); // Claim Tx IDLE
                break;
            }

            /* increment counter */
            m_nTx++;
            WP(WSTR("!%d"),m_nTx);

            /* get first pkt in queue */
            m_TQWaiting.Enqueue(pcurr=m_TQStandby.Dequeue());

            /* fill data */
            DeviceWriteString((PU8)CQueueGetUserPointer(pcurr),pcurr->nLength);
            DeviceWritePort(DM9_TXLENH,HIGH_BYTE(pcurr->nLength));
            DeviceWritePort(DM9_TXLENL,LOW_BYTE(pcurr->nLength));

            // TXCR<0>, issue TX request
            DeviceWritePort(DM9_TXCR, MAKE_MASK(0));
        } // of for feeding loop

        PROCESSOR_LanEnableInterrupt(); //enable int

        return 0;
}
```

✓ Header File

```
/**************************************************************************
 * Print Out function define
 **************************************************************************/

#define WSTR(str)       _T(str)

#define      WP                  NKDbgPrintfW
#define WNextLine() NKDbgPrintfW(WSTR("\r\n"))



/**************************************************************************
 * define reg addr: FOR DM9000
 **************************************************************************/
typedef      enum {
        DM9_NCR = 0,
        DM9_NSR,
        DM9_TXCR,
        DM9_TXSR1,
        DM9_TXSR2,
        DM9_RXCR,
        DM9_RXSR,
        DM9_ROCR,
        DM9_BACKTH,
        DM9_PAUSETH,
        DM9_FLOW,
        DM9_EPCNTL,  /* 0x0B */
        DM9_EPADDR,
        DM9_EPLOW,
        DM9_EPHIGH,
        DM9_WCR,

        DM9_PAR0 = 0x10,
        DM9_PAR1,
```

```
    DM9_PAR2,
    DM9_PAR3,
    DM9_PAR4,
    DM9_PAR5,

    DM9_MAR0 = 0x16,
    DM9_MAR1,
    DM9_MAR2,
    DM9_MAR3,
    DM9_MAR4,
    DM9_MAR5,
    DM9_MAR6,
    DM9_MAR7,

    DM9_GPCR = 0x1E,
    DM9_GPR,

    DM9_TRAL = 0x22,
    DM9_TRAH = 0x23,
    DM9_RWPAL,
    DM9_RWPAH,


    DM9_VIDL = 0x28,
    DM9_VIDH,
    DM9_PIDL,
    DM9_PIDH,
    DM9_CHIPR,

    DM9_SMCR=0x2F,


    DM9_MRCMDX = 0xF0,
    DM9_MRCMD = 0xF2,
    DM9_MDRAH = 0xF4,
    DM9_MDRAL,
```

```
        DM9_MWCMDX = 0xF6,
        DM9_MWCMD = 0xF8,
        DM9_MDWAL = 0xFA,
        DM9_MDWAH = 0xFB,

        DM9_TXLENL = 0xFC,
        DM9_TXLENH,

        DM9_ISR = 0xFE,
        DM9_IMR

} DM9000_REGISTER_TYPE;
```

❖ Programming reference – Linux driver program interface

✓ Driver export functions:

- 1. dm9013_probe
- 2. dm9013_init
- 3. dm9013_open
- 4. dm9013_stop
- 5. dm9013_start_xmit
- 6. dm9013_timeout
- 7. dm9013_interrupt
- 8. dm9013_packet_receive
- 9. dm9013_timer

❖ Davicom Drivers List

✓ WinCE

- ce_dm9000.rar
- ce_dm90xx_eboot.rar
- ce_dm9003.rar
- ce_dm9008.rar
- ce_dm9013.rar
- ce_dm9102.rar
- ce_dm9103.rar
- ce_dm9601.rar

✓ Linux

- lnx_dm9000.rar
- lnx_dm90xx_uboot.rar
- lnx_dm9003.rar
- lnx_dm9008.rar
- lnx_dm9013.rar
- lnx_dm9102.rar
- lnx_dm9103.rar
- lnx_dm9601.rar

✓ Others

- dos_odi_driver_dm9102.rar
- qnx_dm9102.rar
- Nucleus_dm9000.zip
- Vxworks_dm9ks_1.02.071115.rar
- sampl_dm9161.rar
- sampl_sw_auto_mdix.rar

❖ Reference data

✓ Data Sheet

- ❖ DM9000-DS-F03-041906.pdf
- ❖ DM9000A-DS-F01-101906.pdf
- ❖ DM9000B-DS-F01-020108.pdf
- ❖ DM9102H-DS-F01-021508.pdf
- ❖ DM9003-DS-P04-062708.pdf
- ❖ DM9013-DS-P03-070708.pdf
- ❖ DM9103-DS-P02_092607.pdf
- ❖ DM8203-DS-P05-110608.pdf
- ❖ DM9601-DS-P03-102908.pdf
- ❖ DM9161EP-DS-F04-121907.pdf

✓ More Information

- ❖ DM9000A Application Notes Ver 1_21_112707.pdf
- ❖ DM9000 NIC Program Guide Ver 1.12.pdf
- ❖ DM9000A Driver for WinCE Installation.pdf
- ❖ How to Install WinCE6.0 DM9003 Driver.pdf

❖ Trouble shooting