Module XVI

DMA Devices And DMA Device Drivers

Direct Memory Access (DMA)

- A hardware technology
- Allows device hardware to communicate directly with memory
- Purpose: provide high-speed transfer of data

How DMA works (Output)

- A device driver
  - Places outgoing data in memory
  - Tells a DMA device the address of the data (and possibly the size)
  - Continues with other processing
- DMA hardware in the device
  - Extracts data from memory and sends it
  - Interrupts when the transfer is complete

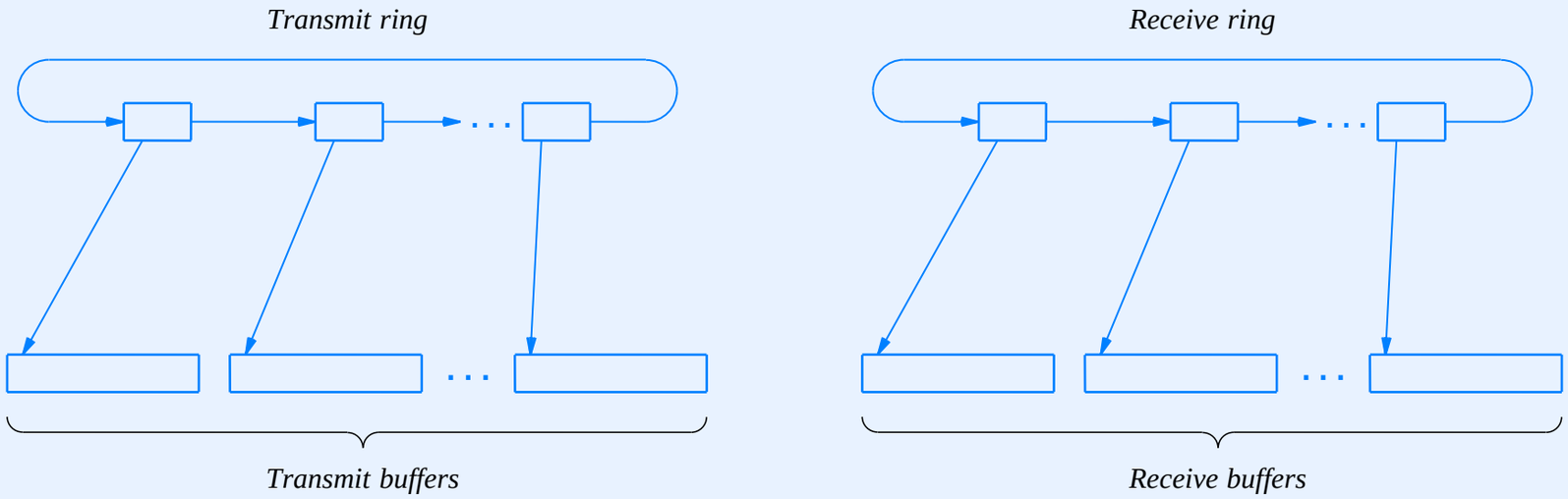
How DMA works (Input)

- A device driver
  - Creates a buffer in memory
  - Tells a DMA device the address of the buffer (and possibly the size)
  - Continues with other processing
- DMA hardware in the device
  - Waits for data to arrive and places the data in the buffer
  - Interrupts when the transfer is complete

More Advanced DMA

- Uses multiple buffers arranged in a ring
- Each buffer has a header that contains
  - A bit that specifies whether the buffer is full
  - The address of the next buffer in the ring
- DMA hardware
  - Either sends data and marks the buffer empty (output) or fills a buffer and marks the buffer full (input)
  - Moves to the next buffer in the ring
  - Only stops when it reaches an empty output buffer or a full input buffer

Illustration Of Buffers In A Ring



Primary Uses Of DMA

- DMA works best for “block” transfer
- Examples:
  - A disk driver that transfers 512-byte disk blocks
  - A network driver that transfers network packets

Optimizing DMA Transfers

- Goal: keep a device busy
- Technique: avoid waiting until an interrupt to set up the next transfer
  - For output, continue to place outgoing data in output buffers before the DMA device reaches the buffer
  - For input, consume incoming data and make the buffer empty before the DMA device reaches the buffer

An Example Of Optimized DMA Processing (Input)

- Xinu has a DMA Ethernet driver
- The device is given a set of DMA buffers
- A high-priority network input process reads and handles each incoming network packet
- The driver switches to the input process immediately when the device interrupts to specify that a packet has been placed in a buffer
- Note: a disk driver uses the same general approach

The Complexity Of DMA Drivers

- The driver must understand
  - Exactly what the DMA device expects in the memory buffer
  - How to coordinate to avoid changing a buffer that the device is using
- Consequence: writing a DMA driver can be incredibly complex

Example Code For A DMA Driver

- To show the complexity of a DMA interface, we will examine code for the am335x_eth device used on BeagleBone Black
  - Definitions of constants and CSR registers
  - Initialization code
  - The input function (read)
  - The output function (write)
- Note: just look at the overall size and complexity, not the details

Example Definitions For A DMA Ethernet Device (am335x)

Definitions For An am335x Driver (Part 1)

```
/* am335x_eth.h - Ethernet device definitions for AM335X SoC */

struct  eth_a_ale {
        uint32  idver;
        uint32  res1;
        uint32  ctrl;
        uint32  res2;
        uint32  prescale;
        uint32  res3;
        uint32  unknown_vlan;
        uint32  res4;
        uint32  tblctl;
        uint32  res5[4];
        uint32  tblw2;
        uint32  tblw1;
        uint32  tblw0;
        uint32  portctl[6];
};
        uint32  tx_intmask_set;
        uint32  tx_intmask_clear;
        uint32  in_vector;
        uint32  eoi_vector;
        byte    res2[8];
        uint32  rx_intstat_raw;
        uint32  rx_intstat_masked;
        uint32  rx_intmask_set;
        uint32  rx_intmask_clear;
        uint32  dma_intstat_raw;
        uint32  dma_intstat_masket;
        uint32  dma_intmask_set;
        uint32  dma_intmask_clear;
```

Definitions For An am335x Driver (Part 2)

```
        uint32  in_vector;
        uint32  eoi_vector;
        byte    res2[8];
        uint32  rx_intstat_raw;
        uint32  rx_intstat_masked;
        uint32  rx_intmask_set;
        uint32  rx_intmask_clear;
        uint32  dma_intstat_raw;
        uint32  dma_intstat_masket;
        uint32  dma_intmask_set;
        uint32  dma_intmask_clear;
        uint32  rx_pendthresh[8];
        uint32  rx_freebuffer[8];
};

struct  eth_a_stateram {
        uint32  tx_hdp[8];
        uint32  rx_hdp[8];
        uint32  tx_cp[8];
        uint32  rx_cp[8];
};
```

Definitions For An am335x Driver (Part 3)

```
struct  eth_a_sl {
        uint32  idver;
        uint32  macctrl;
        uint32  macstat;
        uint32  reset;
        uint32  rx_maxlen;
        uint32  bofftest;
        uint32  rx_pause;
        uint32  tx_pause;
        uint32  emctrl;
        uint32  rx_pri_map;
        uint32  tx_gap;
};

#define ETH_AM335X_SLCTL_FD     0x00000001      /* Full Duplex  */
#define ETH_AM335X_SLCTL_LB     0x00000002      /* Loopback     */
#define ETH_AM335X_SLCTL_EN     0x00000020      /* Rx/Tx Enable */
#define ETH_AM335X_SLCTL_GIG    0x00000080      /* Gigabit mode */

struct  eth_a_ss {
        uint32  idver;
        uint32  ctrl;
        uint32  reset;
        uint32  stat_port_en;
        uint32  ptype;
```

Definitions For An am335x Driver (Part 4)

```
        uint32  soft_idle;
        uint32  thru_rate;
        uint32  gap_thresh;
        uint32  tx_start_wds;
        uint32  flow_ctrl;
        uint32  vlan_type;
        uint32  ts_ltype;
        uint32  dlr_ltype;
};

struct  eth_a_wr {
        uint32  idver;
        uint32  reset;
        uint32  ctrl;
        uint32  int_ctrl;
        uint32  c0_rx_thresh_en;
        uint32  c0_rx_en;
        uint32  c0_tx_en;
        uint32  c0_misc_en;
        uint32  res1[8];
        uint32  c0_rx_thresh_stat;
        uint32  c0_rx_stat;
        uint32  c0_tx_stat;
        uint32  c0_misc_stat;
};
```

Definitions For An am335x Driver (Part 5)

```
struct  eth_a_mdio {
        uint32  ver;
        uint32  ctrl;
        uint32  alive;
        uint32  link;
        uint32  linkintraw;
        uint32  linkintmasked;
        byte    res1[8];
        uint32  userintraw;
        uint32  userintmasked;
        uint32  userintmaskset;
        uint32  userintmaskclr;
        byte    res2[80];
        uint32  useraccess0;
        uint32  userphysel0;
        uint32  useraccess1;
        uint32  userphysel1;
};

#define ETH_AM335X_MDIOCTL_EN   0x40000000

#define ETH_AM335X_MDIOUA_GO    0x80000000 /* Perorm MDIO access*/
#define ETH_AM335X_MDIOUA_WR    0x40000000 /* Write access      */
#define ETH_AM335X_MDIOUA_ACK   0x20000000 /* Read Ack          */
#define ETH_AM335X_MDIOUA_DM    0x0000ffff /* MDIO Data Mask    */
```

Definitions For An am335x Driver (Part 6)

```
struct  eth_a_csreg {
        volatile struct eth_a_ale         *ale;
        volatile struct eth_a_cpdma       *cpdma;
        volatile struct eth_a_stateram    *stateram;
        volatile struct eth_a_sl          *sl;
        volatile struct eth_a_ss          *ss;
        volatile struct eth_a_wr          *wr;
        volatile struct eth_a_mdio        *mdio;
};

struct  eth_a_rx_desc {
        struct  eth_a_rx_desc *next;
        uint32  buffer;
        uint16  buflen;
        uint16  bufoff;
        uint16  packlen;
        uint16  stat;
};

#define ETH_AM335X_RDS_SOP      0x8000  /* Start of packet      */
#define ETH_AM335X_RDS_EOP      0x4000
#define ETH_AM335X_RDS_OWN      0x2000
#define ETH_AM335X_RDS_EOQ      0x1000

#define ETH_AM335X_RX_RING_SIZE 32
```

Definitions For An am335x Driver (Part 7)

```
struct  eth_a_tx_desc {
        struct  eth_a_tx_desc *next;
        uint32  buffer;
        uint16  buflen;
        uint16  bufoff;
        uint16  packlen;
        uint16  stat;
};

#define ETH_AM335X_TDS_SOP      0x8000
#define ETH_AM335X_TDS_EOP      0x4000
#define ETH_AM335X_TDS_OWN      0x2000
#define ETH_AM335X_TDS_EOQ      0x1000
#define ETH_AM335X_TDS_DIR      0x0010
#define ETH_AM335X_TDS_P1       0x0001

#define ETH_AM335X_TX_RING_SIZE 16

#define ETH_AM335X_ALE_ADDR             0x4A100D00
#define ETH_AM335X_CPDMA_ADDR           0x4A100800
#define ETH_AM335X_STATERAM_ADDR        0x4A100A00
#define ETH_AM335X_SL1_ADDR             0x4A100D80
#define ETH_AM335X_MDIO_ADDR            0x4A101000
#define ETH_AM335X_SS_ADDR              0x4A100000
#define ETH_AM335X_WR_ADDR              0x4A101200

#define ETH_AM335X_RXINT                41
#define ETH_AM335X_TXINT                42

#define ETH_AM335X_INIT_DELAY   1000000
```

Initialization Of A DMA Ethernet Device (am335x)

Initialization Of An am335x (Part 1)

```
/* ethinit.c - ethinit, eth_phy_read, eth_phy_write */

#include <xinu.h>

struct  eth_a_csreg eth_a_regs;

struct  ethcblk ethertab[1];

/*------------------------------------------------------------------------
 * eth_phy_read - read a PHY register
 *------------------------------------------------------------------------
 */
int32   eth_phy_read (
                volatile struct eth_a_mdio *mdio,/* MDIO CSR pointer    */
                byte    regadr, /* PHY Register number  */
                byte    phyadr, /* PHY address          */
                uint32  *value  /* Pointer to value     */
        )
{

        /* Ethernet PHY has only 32 registers */

        if(regadr > 31) {
                return SYSERR;
        }
```

Initialization Of An am335x (Part 2)

```
/* Only 32 possible PHY addresses */

if(phyadr > 31) {
        return SYSERR;
}

/* Wait for the previous access to complete */

while( (mdio->useraccess0 & ETH_AM335X_MDIOUA_GO) != 0 );

/* Start the access */

mdio->useraccess0 = (ETH_AM335X_MDIOUA_GO) |
                    (regadr << 21) |
                    (phyadr << 16);

/* Wait until the access is complete */

while( (mdio->useraccess0 & ETH_AM335X_MDIOUA_GO) != 0 );

/* Check if the access was successful */

if( (mdio->useraccess0 & ETH_AM335X_MDIOUA_ACK) == 0 ) {
        return SYSERR;
}
```

Initialization Of An am335x (Part 3)

```
        /* Copy the value read */

        (*value) = mdio->useraccess0 & ETH_AM335X_MDIOUA_DM;

        return OK;
}

/*------------------------------------------------------------------------
 * eth_phy_write - write a PHY register
 *------------------------------------------------------------------------
 */
int32   eth_phy_write (
                volatile struct eth_a_mdio *mdio, /* MDIO CSR pointer   */
                byte    regadr, /* PHY register number  */
                byte    phyadr, /* PHY address          */
                uint32  value   /* Value to be written  */
        )
{

        /* There are only 32 PHY registers */

        if(regadr > 31) {
                return SYSERR;
        }
```

Initialization Of An am335x (Part 4)

```
        /* There are only 32 possible PHY addresses */

        if(phyadr > 31) {
                return SYSERR;
        }

        /* Wait for the previous access to complete */

        while( (mdio->useraccess0 & ETH_AM335X_MDIOUA_GO) != 0);

        /* Start the access */

        mdio->useraccess0 = ETH_AM335X_MDIOUA_GO |
                            ETH_AM335X_MDIOUA_WR |
                            (regadr << 21) |
                            (phyadr << 16) |
                            (value & 0xffff);

        /* Wait for the access to complete */

        while( (mdio->useraccess0 & ETH_AM335X_MDIOUA_GO) != 0);

        return OK;
}
```

Initialization Of An am335x (Part 5)

```
/*------------------------------------------------------------------------
 * eth_phy_reset - Reset an Ethernet PHY
 *------------------------------------------------------------------------
 */
int32   eth_phy_reset (
                volatile struct eth_a_mdio *mdio, /* MDIO CSR pointer   */
                byte    phyadr                    /* PHY Address        */
        )
{
        uint32  phyreg; /* Variable to hold ETH PHY register value      */
        int32   retries;/* Number of retries                            */
        int32   retval; /* Return value of functions called here        */

        /* Read the PHY Control Register */

        retval = eth_phy_read(mdio, ETH_PHY_CTLREG, phyadr, &phyreg);
        if(retval == SYSERR) {
                return SYSERR;
        }

        /* Set the Reset bit and write the register */

        phyreg |= ETH_PHY_CTLREG_RESET;
        eth_phy_write(mdio, ETH_PHY_CTLREG, phyadr, phyreg);
```

Initialization Of An am335x (Part 6)

```
/* Check if Reset operation is complete */

for(retries = 0; retries < 10; retries++) {
        if(eth_phy_read(mdio, ETH_PHY_CTLREG, phyadr, &phyreg) == SYSERR) {
                return SYSERR;
        }
        if((phyreg & ETH_PHY_CTLREG_RESET) == 0) {
                break;
        }
        else {
                retries++;
                DELAY(ETH_AM335X_INIT_DELAY);
                continue;
        }
}

if(retries >= 3) {
        return SYSERR;
}
```

Initialization Of An am335x (Part 7)

```
        /* Check if the Link is established */

        for(retries = 0; retries < 10; retries++) {
                if(eth_phy_read(mdio, ETH_PHY_STATREG, phyadr, &phyreg) == SYSERR) {
                        return SYSERR;
                }
                if(phyreg & ETH_PHY_STATREG_LINK) {
                        break;
                }
                else {
                        retries++;
                        DELAY(ETH_AM335X_INIT_DELAY);
                        continue;
                }
        }
        if(retries >= 3) {
                return SYSERR;
        }

        return OK;
}
```

Initialization Of An am335x (Part 8)

```
/*------------------------------------------------------------------------
 * ethinit - initialize the TI AM335X ethernet hardware
 *------------------------------------------------------------------------
 */
int32   ethinit (
                struct  dentry *devptr
        )
{
        struct  ethcblk *ethptr;                /* Ethernet control blk pointer */
        struct  eth_a_tx_desc *tdescptr;/* Tx descriptor pointer        */
        struct  eth_a_rx_desc *rdescptr;/* Rx descriptor pointer        */
        struct  netpacket *pktptr;      /* Packet pointer               */
        struct  eth_a_csreg *csrptr;    /* Ethernet CSR pointer         */
        uint32  phyreg;                 /* Variable to store PHY reg val*/
        int32   retval;                 /* Return value                 */
        int32   i;                      /* Index variable               */

        /* Get the Ethernet control block address       */
        /* from the device table entry                  */

        ethptr = &ethertab[devptr->dvminor];

        /* Store the address of CSRs in the Ethernet control block      */

        csrptr = &eth_a_regs;
        ethptr->csr = csrptr;
```

Initialization Of An am335x (Part 9)

```
/* Initialize the addresses of all the submodules       */

csrptr->ale = (struct eth_a_ale *)ETH_AM335X_ALE_ADDR;
csrptr->cpdma = (struct eth_a_cpdma *)ETH_AM335X_CPDMA_ADDR;
csrptr->sl = (struct eth_a_sl *)ETH_AM335X_SL1_ADDR;
csrptr->stateram = (struct eth_a_stateram *)
                                ETH_AM335X_STATERAM_ADDR;
csrptr->ss = (struct eth_a_ss *)ETH_AM335X_SS_ADDR;
csrptr->wr = (struct eth_a_wr *)ETH_AM335X_WR_ADDR;
csrptr->mdio = (struct eth_a_mdio *)ETH_AM335X_MDIO_ADDR;

/* Reset all the submodules */

csrptr->cpdma->reset = 1;
while(csrptr->cpdma->reset == 1);

csrptr->sl->reset = 1;
while(csrptr->sl->reset == 1);

csrptr->wr->reset = 1;
while(csrptr->wr->reset == 1) ;

csrptr->ss->reset = 1;
while(csrptr->ss->reset == 1) ;

/* Enable MDIO  */

csrptr->mdio->ctrl |= ETH_AM335X_MDIOCTL_EN;
```

Initialization Of An am335x (Part 10)

```
/* Reset the PHY */

retval = eth_phy_reset(csrptr->mdio, 0);
if(retval == SYSERR) {
        kprintf("Cannot reset Ethernet PHY\n");
        return SYSERR;
}

retval = eth_phy_read(csrptr->mdio, ETH_PHY_CTLREG, 0, &phyreg);
if(retval == SYSERR) {
        return SYSERR;
}

if( (phyreg & ETH_PHY_CTLREG_SM) == ETH_PHY_10M ) {
        kprintf("Ethernet Link is Up. Speed is 10Mbps\n");
}
else if( (phyreg & ETH_PHY_CTLREG_SM) == ETH_PHY_100M ) {
        kprintf("Ethernet Link is Up. Speed is 100Mbps\n");
}
else if( (phyreg & ETH_PHY_CTLREG_SM) == ETH_PHY_1000M ) {
        kprintf("Ethernet Link is Up. Speed is 1000Mbps\n");
}
else {
        return SYSERR;
}
```

Initialization Of An am335x (Part 11)

```
if(phyreg & ETH_PHY_CTLREG_FD) {
        kprintf("Link is Full Duplex\n");
        csrptr->sl->macctrl |= ETH_AM335X_SLCTL_FD;
}
else {
        kprintf("Link is Half Duplex\n");
}

/* Read the device MAC address */
for(i = 0; i < 2; i++) {
        ethptr->devAddress[4+i] = *((byte *)(0x44e10630+i));
}
for(i = 0; i < 4; i++) {
        ethptr->devAddress[i] = *((byte *)(0x44e10634+i));
}

kprintf("MAC Address is: ");
for(i = 0; i < 5; i++) {
        kprintf("%02X:", ethptr->devAddress[i]);
}
kprintf("%02X\n", ethptr->devAddress[5]);

/* Initialize the rx ring size field */
ethptr->rxRingSize = ETH_AM335X_RX_RING_SIZE;
```

Initialization Of An am335x (Part 12)

```
/* Allocate memory for the rx ring */
ethptr->rxRing = (void*)getmem(sizeof(struct eth_a_rx_desc)*
                                ethptr->rxRingSize);
if((int32)ethptr->rxRing == SYSERR) {
        return SYSERR;
}

/* Zero out the rx ring */
memset((char*)ethptr->rxRing, NULLCH,
        sizeof(struct eth_a_rx_desc)*ethptr->rxRingSize);

/* Allocate memory for rx buffers */
ethptr->rxBufs = (void*)getmem(ETH_BUF_SIZE *
                                ethptr->rxRingSize);
if((int32)ethptr->rxBufs == SYSERR) {
        return SYSERR;
}

/* Zero out the rx buffers */
memset((char *)ethptr->rxBufs, NULLCH, ETH_BUF_SIZE *
                                        ethptr->rxRingSize);
```

Initialization Of An am335x (Part 13)

```
/* Initialize the rx ring */

rdescptr = (struct eth_a_rx_desc *)ethptr->rxRing;
pktptr = (struct netpacket *)ethptr->rxBufs;

for(i = 0; i < ethptr->rxRingSize; i++) {
        rdescptr->next = rdescptr + 1;
        rdescptr->buffer = (uint32)pktptr->net_ethdst;
        rdescptr->buflen = ETH_BUF_SIZE;
        rdescptr->bufoff = 0;
        rdescptr->stat = ETH_AM335X_RDS_OWN;
        rdescptr++;
        pktptr++;
}
(--rdescptr)->next = NULL;

ethptr->rxHead = 0;
ethptr->rxTail = 0;
ethptr->isem = semcreate(0);
if((int32)ethptr->isem == SYSERR) {
        return SYSERR;
}

/* initialize the tx ring size */
ethptr->txRingSize = ETH_AM335X_TX_RING_SIZE;
```

Initialization Of An am335x (Part 14)

```
/* Allocate memory for tx ring */
ethptr->txRing = (void*)getmem(sizeof(struct eth_a_tx_desc)*
                                ethptr->txRingSize);
if((int32)ethptr->txRing == SYSERR) {
        return SYSERR;
}

/* Zero out the tx ring */
memset((char*)ethptr->txRing, NULLCH,
        sizeof(struct eth_a_tx_desc)*ethptr->txRingSize);

/* Allocate memory for tx buffers */
ethptr->txBufs = (void*)getmem(ETH_BUF_SIZE *
                                ethptr->txRingSize);
if((int32)ethptr->txBufs == SYSERR) {
        return SYSERR;
}

/* Zero out the tx buffers */
memset((char*)ethptr->txBufs, NULLCH, ETH_BUF_SIZE *
                                      ethptr->txRingSize);

/* Initialize the tx ring */

tdescptr = (struct eth_a_tx_desc *)ethptr->txRing;
pktptr = (struct netpacket *)ethptr->txBufs;
```

Initialization Of An am335x (Part 15)

```
for(i = 0; i < ethptr->txRingSize; i++) {
        tdescptr->next = NULL;
        tdescptr->buffer = (uint32)pktptr->net_ethdst;
        tdescptr->buflen = ETH_BUF_SIZE;
        tdescptr->bufoff = 0;
        tdescptr->stat = (ETH_AM335X_TDS_SOP |
                          ETH_AM335X_TDS_EOP |
                          ETH_AM335X_TDS_DIR |
                          ETH_AM335X_TDS_P1);
        tdescptr++;
        pktptr++;
}

ethptr->txHead = 0;
ethptr->txTail = 0;
ethptr->osem = semcreate(ethptr->txRingSize);
if((int32)ethptr->osem == SYSERR) {
        return SYSERR;
}

/* Enable the ALE and put it into bypass mode */
csrptr->ale->ctrl = (ETH_AM335X_ALECTL_EN |
                     ETH_AM335X_ALECTL_BY);
```

Initialization Of An am335x (Part 16)

```
        /* Put the ports 0, 1 in forwarding state */
        csrptr->ale->portctl[0] = ETH_AM335X_ALEPCTL_FWD;
        csrptr->ale->portctl[1] = ETH_AM335X_ALEPCTL_FWD;

        /* Start the rx and tx processes in DMA */
        csrptr->cpdma->tx_ctrl = 1;
        csrptr->cpdma->rx_ctrl = 1;

        /* Initialize the head desc pointers for tx and rx */
        csrptr->stateram->tx_hdp[0] = 0;
        csrptr->stateram->rx_hdp[0] = (uint32)ethptr->rxRing;

        /* Enable Rx and Tx in MAC */
        csrptr->sl->macctrl |= ETH_AM335X_SLCTL_EN;

        /* Set interrupt vectors */
        set_evec(ETH_AM335X_TXINT, (uint32)devptr->dvintr);
        set_evec(ETH_AM335X_RXINT, (uint32)devptr->dvintr);

        /* Enable the CPDMA interrupts */
        csrptr->cpdma->tx_intmask_set = 0x1;
        csrptr->cpdma->rx_intmask_set = 0x1;

        /* Route the interrupts to core 0 */
        csrptr->wr->c0_tx_en = 0x1;
        csrptr->wr->c0_rx_en = 0x1;

        return OK;
}
```

Reading From A DMA Ethernet Device (am335x)

Ethread For An am335x (Part 1)

```
/* ethread.c - ethread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethread - read an incoming packet on TI AM335X Ethernet
 *------------------------------------------------------------------------
 */
devcall ethread (
                struct  dentry *devptr,
                char    *buf,
                int32   count
        )
{
        struct  ethcblk *ethptr;        /* Ethernet ctl blk ptr */
        struct  eth_a_csreg *csrptr;    /* Ethernet CSR pointer */
        struct  eth_a_rx_desc *rdescptr;/* Rx Desc. pointer     */
        struct  eth_a_rx_desc *prev;    /* Prev Rx desc pointer */
        uint32  retval;                 /* Num of bytes returned*/

        ethptr = &ethertab[devptr->dvminor];

        /* Get the pointer to Ethernet CSR */
        csrptr = (struct eth_a_csreg *)ethptr->csr;
```

Ethread For An am335x (Part 2)

```
/* Wait for a packet */
wait(ethptr->isem);
/* Get pointer to the descriptor */
rdescptr = (struct eth_a_rx_desc *)ethptr->rxRing +
                                        ethptr->rxHead;

/* Read the packet length */
retval = rdescptr->packlen;
if(retval > count) {
        retval = count;
}

/* Copy the packet into user provided buffer */
memcpy((char *)buf, (char *)rdescptr->buffer, retval);

/* Initialize the descriptor for next packet */
rdescptr->stat = ETH_AM335X_RDS_OWN;
rdescptr->bufoff = 0;
rdescptr->buflen = ETH_BUF_SIZE;
rdescptr->packlen = 0;
rdescptr->next = NULL;
```

Ethread For An am335x (Part 4)

```
        /* Insert the descriptor into Rx queue */
        prev = (struct eth_a_rx_desc *)csrptr->stateram->rx_hdp[0];
        if(prev == NULL) {
                kprintf("hdp 0, adding %x\n", rdescptr);
                csrptr->stateram->rx_hdp[0] = (uint32)rdescptr;
        }
        else {
                while(prev->next != NULL) {
                        prev = prev->next;
                }
                prev->next = rdescptr;
        }

        /* Increment the head index of rx ring */
        ethptr->rxHead++;
        if(ethptr->rxHead >= ethptr->rxRingSize) {
                ethptr->rxHead = 0;
        }

        return retval;
}
```

Writing To A DMA Ethernet Device (am335x)

Ethwrite For An am335x (Part 1)

```
/* ethwrite.c - ethwrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethwrite - enqueue a packet for transmission on TI AM335X Ethernet
 *------------------------------------------------------------------------
 */
int32   ethwrite (
                struct  dentry *devptr,
                char    *buf,
                int32   count
        )
{
        struct  ethcblk *ethptr;        /* Ether entry pointer  */
        struct  eth_a_csreg *csrptr;    /* Ethernet CSR pointer */
        struct  eth_a_tx_desc *tdescptr;/* Tx Desc. pointer     */
        struct  eth_a_tx_desc *prev;    /* Prev. Desc. pointer  */

        ethptr = &ethertab[devptr->dvminor];

        /* Get the pointer to the Ethernet CSR */
        csrptr = (struct eth_a_csreg *)ethptr->csr;
```

Ethwrite For An am335x (Part 2)

```
/* Wait for an empty slot in the queue */
wait(ethptr->osem);

/* Get the pointer to the next descriptor */
tdescptr = (struct eth_a_tx_desc *)ethptr->txRing +
                                            ethptr->txTail;

/* Adjust count if greater than max. possible packet size */
if(count > PACKLEN) {
        count = PACKLEN;
}

/* Initialize the descriptor */
tdescptr->next = NULL;
tdescptr->buflen = count;
tdescptr->bufoff = 0;
tdescptr->packlen = count;
tdescptr->stat = (ETH_AM335X_TDS_SOP |  /* Start of packet      */
                  ETH_AM335X_TDS_EOP |  /* End of packet        */
                  ETH_AM335X_TDS_OWN |  /* Own flag set for DMA */
                  ETH_AM335X_TDS_DIR |  /* Directed packet      */
                  ETH_AM335X_TDS_P1);   /* Output port is port1 */

/* Copy the packet into the Tx buffer */
memcpy((char *)tdescptr->buffer, buf, count);
```

Ethwrite For An am335x (Part 3)

```
/* TODO Figure out why we need this hack        */
/* This ethernet device does not send packets smaller than 60   */
/* bytes; So pad a small packet to make it 60 bytes long        */

if(count < 60) {
        memset((char *)tdescptr->buffer+count, 0, 60-count);
        tdescptr->buflen = 60;
        tdescptr->packlen = 60;
}

/* Insert the descriptor into Tx queue */

if(csrptr->stateram->tx_hdp[0] == 0) {
        /* Tx queue is empty, this desc. will be the first */
        csrptr->stateram->tx_hdp[0] = (uint32)tdescptr;
}
else {
        /* Tx queue not empty, insert at end */
        prev = (struct eth_a_tx_desc *)
                        csrptr->stateram->tx_hdp[0];
        while(prev->next != NULL) {
                prev = prev->next;
        }
        prev->next = tdescptr;
}
```

Ethwrite For An am335x (Part 4)

```
        /* Increment the tail index of the Tx ring */
        ethptr->txTail++;
        if(ethptr->txTail >= ethptr->txRingSize) {
                ethptr->txTail = 0;
        }

        return count;
}
```

Summary

- Direct Memory Access makes it possible for a device to transfer large blocks of data to or from memory
- DMA is most useful for devices such as disks or network interfaces where each transfer is hundreds or thousands of bytes
- A device driver for a device that uses DMA contains many details and can be difficult to write

Questions?