EPOS Driver para a placa Am79C970A

Eberle Andrey Rambo
Khristian Alexander Schönrock
Leandro Santos Grapiuna
Rogerio Bodemuller Junior

1 Introdução

Este trabalho tem como objetivo relatar o funcionamento do driver da placa de rede Am79C970A da AMD feito para o EPOS. O enunciado do trabalho era fazer um relatório sobre o funcionamento do driver de rede do VMWare feito para o EPOS. A partir do código-fonte do EPOS foi concluído que o driver é da placa Am79C970A. E, com o manual desta placa, foi possível a realização do trabalho.

No restante do relatório são apresentados os códigos dos três arquivos que constituem o driver. pcnet32.h, pcnet32.cc e pcnet32_init.cc.

2 pcnet32.h

No arquivo pcnt32.h é definido a classe Am79C970A com seus registradores, o bloco de inicialização, os descritores utilizados "ring buffer" e os métodos de manipulação da placa.

Aqui será mostrado apenas 3 métodos desta classe. O s_reset(), onde a leitura do registrador S_RESET causa um pulso interno de reset de software, mas não afeta as configurações do BCR(Bus Configuration Register) ou do T-MAU(Twisted Pair Transeiver com as funções do Medium Attachment Unit operando em half-duplex).

```
void s_reset(){
   // seta o S_RESET.
   IA32::in16(_io_port + WIO_RESET);
   // espera pelo CSR0_STOP = 0x0004.
   for(int i = 0; (i < 100 ) && !(csr(CSC) & 0x0004); i++);
}
```

Os outros dois métods são para leitura e escrita dos registradores de controle e status.

```
// lê um registrador da placa de rede.
Reg16 csr(int a) volatile{
   // primeiro coloca o endereço para efetuar a leitura.
   IA32:out16(_io_port + WIO_RAP, a);
   // depois lê o dado daquele endereço.
   return IA32::in16(_io_port + WIO_RDP);
}

// escreve um registrador na placa de rede.
void csr(int a, Reg16 v){
   // primeiro coloca o endereço para efetuar a escrita.
   IA32:out16(_io_port + WIO_RAP, a);
   // depois escreve o dado no endereço fornecido
   // previamente.
   IA32:out16(_io_port + WIO_RDP, v);
}
```

Ainda no arquivo pcnet32.h há a definição da classe pcnet32. Esta classe herda das classes Am79C970A e Ethernet_NIC. Para a classe pcnet32 é declarado seus atributos e métodos. Enfatizamos a presença de dois atributos. Device, para controle compartilhado e sobre informações das interrupções. E um array de devices que é a quantidade máxima de pcnet32 que podem ser utilizados simultaneamente.

```
struct Device {
   PCNet32 * device;
   unsigned int interrupt;
   bool in_use;
}

static Device _devices[UNITS];
```

3 pcnet32_init.cc

No arquivo pcnet32_init.cc é feito a implementação do método de classe init(). Neste método é instanciado um Locator com o valor de retorno da função scan(). loc terá um valor inválido caso não seja encontrado nenhum barramento PCI válido. scan() tem a função de verificar se este barramento é

válido onde recebe como parâmetros os valores 0x1022 (identificador do fabricante do dispositivo, neste caso da AMD), 0x2000 (identificador único do PCNet-PCI II) e unit("valor" do dispositivo, que é passado como parâmetro na chamada da função init()). Caso não tenha encontrado um barramento PCI válido, o método é finalzado.

Caso o barramento PCI seja válido, tenta habilitar as regiões de IO e o "bus matering". Para isto chama a função command(), passando como parâmetros o Locator instanciado anteriormente em um ou em três registradores de 16 bits. O primeiro registrador é o retorno da chamada da função command(), passando como parâmetro o próprio locator. O segundo é o valor 0x1, que habilita as regiões de IO. E o terceiro é o valor 0x4, que habilita o bus mastering.

Instancia o cabeçalho Header, uma struct definida em /include/pci.h, e inicializa os valores deste cabeçalho configurando para o controlador PCNet-PCI II. Caso o identificador do fabricante for inválido (0x0 ou 0xffff), o método é fnalizado.

Caso o identificador do fabricante seja válido, verifica se a região de IO está acessível, através da negação da comparação de command (registrador de 16 bits, parte da struct do cabeçalho) com a constante COMMAND_IO (0x1). Verifica se é possível utilizar o bus mastering, através da negação da comparação de command com a constante COMMAND_MASTER (0x4).

Seguindo o método, um IO_Port (registrador de 16 bits) é instanciado, inicializando com o valor do endereço físico da região de IO (região 0). Um IO_Irq (registrador de 16 bits) é instanciado, inicializando com o valor de interrupt_line (8 bits).

Aloca um buffer de acesso direto a memória para o bloco inicial, rx e tx rings (buffers circulares). Aloca espaço para o PCNet32 e chama seu método construtor passando como parâmetro os valores unit, io_port, irq e dma_buf inicializando o dispositivo.

Registra o dispositivo recém instanciado, sendo _devices[unit] um atributo de PCNet32 que identifica este dispositivo. O valor de in_use indica que ele não está em uso (só mudará quando for chamado o construtor), device indica o prórpio dispositivo e interrupt é o valor de irq transformado para inteiro.

Instala o tratador de interrupções, um vetor de interrupções onde a interrupção número irq (transformado para inteiro e menor que 64) será o endereço de int_handler. E, por último, habilita as interrupções para o controlador PCNet-PCI II.

A seguir é apresentado a implementação do método init() que acabamos de explicar.

```
void PCNet32::init(unsigned int unit){
   PC_PCI::Locator loc =
      PC_PCI::scan(PCI_VENDOR_ID, PCI_DEVICE_ID, unit);
   if(!loc) {
   db<Init, PCNet32>(WRN) <<
      "PCNet32::init: PCI scan failed!\n";
   return;
   }
   PC_PCI::command(loc, PC_PCI::command(loc)
         | PC_PCI::COMMAND_IO | PC_PCI::COMMAND_MASTER);
   PC_PCI::Header hdr;
   PCI::header(loc, &hdr);

   if(!hdr) {
      db<Init, PCNet32>(WRN) <<
         "PCNet32::init: PCI header failed!\n";
      return;
   }

   if(!(hdr.command & PC_PCI::COMMAND_IO))
      db<Init, PCNet32>(WRN) <<
         "PCNet32::init: I/O unaccessible!\n";
   if(!(hdr.command & PC_PCI::COMMAND_MASTER))
      db<Init, PCNet32>(WRN)
         << "PCNet32::init: not master capable!\n";

   IO_Port io_port = hdr.region[PCI_REG_IO].phy_addr;
   IO_Irq irq = hdr.interrupt_line;
   DMA_Buffer * dma_buf =
      new(kmalloc(sizeof(MMU::DMA_Buffer)))
   DMA_Buffer(DMA_BUFFER_SIZE);
   PCNet32 * dev = new (kmalloc(sizeof(PCNet32)))
   PCNet32(unit, io_port, irq, dma_buf);

   _devices[unit].in_use = false;
   _devices[unit].device = dev;
   _devices[unit].interrupt = Machine::irq2int(irq);

   Machine::int_vector(Machine::irq2int(irq), &int_handler);
   IC::enable(irq);
}
```

4 pcnet32.cc

No arquivo pcnet32.cc é feita a implementação dos métodos da classe pcnet32. Iremos abordar os construtores, os métodos send e receive, os tratadores de interrupção e o destrutor.

4.1 Construtores e Destrutor

```
PCNet32(unsigned int unit,
   IO_Port io_port, IO_Irq irq, DMA_Buffer * dma_buf):
```

É o construtor chamado em init(). Distribui o DMA_Buffer alocado por init(), configura o bloco de inicialização, os anéis de descritores Rx_Desc Ring e Tx_Desc Ring, os buffers Rx e Tx e chama o reset().

```
PCNet32(unsigned int unit):
```

É o construtor que é chamado pelo usuário. Nele é verificado se o número da unidade é válida, e se o dispositivo não está em uso. Caso passe nos testes, o dispositivo que está no array de dispositivos é atribuído a *this e atribui true à variável in_use.

```
~PCNet32():
```

No construtor padrão, a inicialização do dispositivo está restrito ao número de unidades possíveis e ao estado do mesmo, ou seja, se está em uso ou não. Portanto, no destrutor é apenas modificado o estado do dispositivo, desbloqueando para uma nova chamada do construtor.

4.2 reset

No reset(), primiramente é chamada a método s_reset(), causando um reset de software e espera pelo STOP (CSR0, bit 2). Em seguida chama o método bcr() para selecionar o estilo, ou seja, o modo como os registradores são utilizados pelo software. O método bcr() configura o registrador RAP (Register Address Port) para entrada/saída do BDP (BCR Data Port). Neste caso, é gravado o valor SWSTYLE2 e aguarda a leitura do BDP. E pega o enderesso do MAC na PROM.

Habilita o autoselect (ASEL) usando o método bcr(). Novamente pelo método bcr(), habilita full-duplex (FDEN). Desabilita a transmissão do stop no overflow (DXSUFLO). Os bits TINTM (Transmit Interrupt Mask

), RINTM (Receive Interrupt Mask) e IDONDM (Init Done Mask) são usados em conjunto com DXSUFLO.

Habilita o preenchimento de frames. APAD_XMT faz com que a placa "encha"os frames transmitidos até 64 bytes, depois adicionando o FCS. O bit DMAPLUS é usado quando a placa de rede está ligada em um barramento PCI. Este bit é usado está sendo usado em conjunto como APAD_XMT.

Ajusta as interrupções. Alguns bits do CRS5 indicam as causas das interrupções. Esses registradores são feitos de tal modo que eles são limpados (cleared) ao gravar-se um "1" neles. Ou seja, uma aplicação pode ler CSR5 e escrever de volta o mesmo valor para limpar a condição de interrupção. Os bits utilizados para ajustar as interrupções são: TOKINTD (desabilita a interrupção de quando a transmissão é bem-sucedida), SINTE (habilita interrupções de sistema), SLPINTE (habilita interrupção de sleep), EXDINTE (habilita interrupção de "Excessive Deferral") e MPINTE (habilita interrupção por magic packet).

Habilita leitura e escrita em burst(rajada) sem alterar o resto do BCR18. Para isto são utilizados os bits BREADE que habilita modo de burst nas leituras e BWRITE que habilita modo de burst nas escritas.

Configura ponto de inicio de transmição para um frame cheio. Quando o número de bytes na FIFO de envio chega ao valor de XMTSP, a placa começa a tentar enviar. No caso, XMTSP = 11 $->$ full frame (248 bytes escritos).

Inicializa o bloco de inicialização de endereços lógicos. Onde _iblock é o trecho de endereços lógicos do buffer DMA passado ao construtor. Inicializa o dispositivo onde IENA (habilita interupções) e INIT (ativa a inicialização do dispositivo - leitura do bloco de inicialização da memória). Se após o loop, o IDON não estiver setado, significa que a inicialização falhou. Sinaliza o fim da inicialização e pára toda a atividade da placa ao setar o bit STOP. O STOP também desabilita o IENA.

Habilita recebimento e transmissão com os bits STRT e IENA.

4.3 Tratamento de Interrupções

`int_handler(unsigned int interrupt):`

Neste método, primeiro ele tenta pegar a interface de rede que gerou a interrupção. Se não encontrou nenhuma interface PCNet32 que gerou a interrupção apenas registra no log. Caso contrário, chama handle_int() para tratar cada instância com o seu tratador específico. Sendo que pode ter mais de uma interface de rede conectada.

`handle_int():`

Este método é chamado pelo tratador de interrupções, não sendo chamado diretamente pelo SO. Neste método, inicialmente, as interrupções da cpu são desabilitadas para o tratador não ser interrompido no meio.

Lê o "registrador 0" e usa uma mácara para ver se ocorreu um interrupção e repete enquanto sim. Lê os valores dos registradores que contêm informação sobre as interrupções 0, 4 e 5 e limpa estes valores. Checa se a placa está no estado ON (ligada).

Checa se a interrupção para recebimento está ativada. Se estiver, verifica se a interrupção que ocorreu foi de recebimento. Caso seja, aloca as variáveis na pilha, realiza a carga dos dados do frame no buffer temporário, re-ativa as interrupções e notifica ao Observador correto, de acordo com o protocolo. Ao sair as variáveis alocadas na pilha são destruídas.

Caso a interrupção não seja de recebimento, verifica se é de erro. Caso seja, verifica o erro específico. Os bits de erro estão no CSR0 e são ERR para erro genérico, MERR para erro de memória, MISS para perda de frame, CERR para erro de colisão e BABL para "bable transmiter time-out".

Caso não seja nem de recebimento nem de erro, é verificado as outras interrupções. MFCO para "missed frame counter overflow", UINT para interrupção do usuário, RCVCCO para "receive collision counter overflow", TXSTRT para início de transmição, JAB para erro de Jabber, SINT para interrupção do sistema, SLPINT para interrupção de sleep, EXDINT para "excessive deferral interrupt" e MPINT para "magic packet interrupt".

E, por fim, reabilita as intreeupções.

4.4 send e receive

Tanto o send quanto o receive utilizam os seguinte parâmetros:

Address endereço destino ou fonte.

Protocol protocolo que é tratado na camada superior.

data endereço da mensagem.

size tamanho máximo do buffer.

Com os parâmetros definidos vamos aos métodos:

```
int send(const Address & dst, const Protocol & prot,
  const void * data, unsigned int size):
```

O método começa esperando por um buffer livre. O bit OWN representa um "relação de posse" sobre o anle de descritores. Quando OWN é ZERO

o descritor pertence ao host, quando OWN é UM o descritor pertence ao usuário.

Após a posse do anel, o frame é montado e o tamanho do buffer é atualizado.

Registra no descritor os bits OWN (ownership), STP (start of packet) e ENP (end of packet). Isto significa que quando a controladora ler o descritor, deverá enviar o frame (OWN e STP) e que o frame cabe em um único buffer (STP e ENP).

Após, é configurado no registrador CSR0 o bit TDMD (Transmit Demand). Este bit força o BMU (Buffer Managament Unit) a acessar o transmit descriptor ring sem esperar pelo contador do polling. E incrementa a posição atual no buffer circular.

O retorno do método é o tamanho do buffer passado como parâmetro.

```
int receive(Address * src, Protocol * prot,
     void * data, unsigned int size):
```

O receive começa esperando que um frame esteja disponível no buffer. Em seguida, ele invoca o método receive_common re retorna o número de bytes recebidos.

```
receive_common(Address &src, Protocol &prot,
      unsigned int &size, void *buffer,
      unsigned int buf_len):
```

Neste método, o frame Ethernet é desmontado, retirando o Emissor, os dados, o protocolo a ser entregue na camada superior e o tamanho dos dados. Copia os dados para o endereço de memória fornecido. Libera a posição do buffer de frames recebidos para utilização pelo NIC. E incrementa a posição atual no buffer circular.

Referências

[1] EPOS - - source code.

[2] Am79C970A datasheet.