UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
FACULDADE DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO ANTONIO ALVES ALENCAR

IMPLEMENTAÇÃO DE UM DRIVER DE INTERFACE DE REDE PARA O SISTEMA OPERACIONAL MINIX 3

Belém/PA
2018

MARCELO ANTONIO ALVES ALENCAR

IMPLEMENTAÇÃO DE UM DRIVER DE INTERFACE DE REDE PARA O SISTEMA OPERACIONAL MINIX 3

Trabalho de Conclusão de Curso apresentado para obtenção do título de Bacharel em Ciência da Computação. Instituto de Ciências Exatas e Naturais. Faculdade de Computação. Universidade Federal do Pará.

Orientador: Prof. Dr. Alexandre Beletti Ferreira
Coorientadora: Profa. Dra. Regiane Silva Kawasaki Francês

Belém/PA
2018

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da Universidade Federal do Pará
Gerada automaticamente pelo módulo Ficat, mediante os dados fornecidos pelo(a) autor(a)

A368i Alencar, Marcelo Antonio Alves

Implementação de um driver de interface de rede para o sistema operacional MINIX 3 / Marcelo Antonio Alves Alencar. – 2018.

97f. : il.

Trabalho de Conclusão de Curso (Graduação) - Faculdade de Computação. Instituto de Ciências Exatas e Naturais. Universidade Federal do Pará, Belém, 2018.

Orientação: Prof. Dr. Alexandre Beletti Ferreira

Coorientação: Profa. Dra. Regiane Silva Kawasaki Francês.

1. MINIX. 2. Driver de dispositivo. 3. Redes de computadores. I. Ferreira, Alexandre Beletti, *orient*. II. Título

---

CDD 005.268

MARCELO ANTONIO ALVES ALENCAR

IMPLEMENTAÇÃO DE UM DRIVER DE INTERFACE DE REDE PARA O SISTEMA OPERACIONAL MINIX 3

Trabalho de Conclusão de Curso apresentado para obtenção do título de Bacharel em Ciência da Computação. Instituto de Ciências Exatas e Naturais. Faculdade de Computação. Universidade Federal do Pará.

Orientador: Prof. Dr. Alexandre Beletti Ferreira
Coorientadora: Profa. Dra. Regiane Silva Kawasaki Francês

Data de aprovação: 21 de fevereiro de 2018
Conceito: _____

Banca examinadora:

Prof. Dr. Alexandre Beletti Ferreira
Campus São Paulo / Instituto Federal de São Paulo – Orientador

Profa. Dra. Regiane Silva Kawasaki Francês
Instituto de Ciências Exatas e Naturais / Universidade Federal do Pará – Coorientadora

Prof. Dr. Josivaldo de Souza Araújo
Instituto de Ciências Exatas e Naturais / Universidade Federal do Pará – Avaliador

Prof. Dr. Raimundo Viégas Junior
Instituto de Ciências Exatas e Naturais / Universidade Federal do Pará – Avaliador

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Dr. Alexandre Beletti Ferreira, por se dispor a me orientar mesmo que a distância, e também agradeço à minha coorientadora, Prof.ª Dra. Regiane Silva Kawasaki Francês, pelo apoio desde os semestres iniciais do curso.

Agradeço à Prof.ª Ma. Cristina de Barros Nunes, que desde o ensino fundamental tem me ajudado tanto, contribuindo com minha formação acadêmica.

Agradeço aos meus familiares, por permitir que pudesse me dedicar aos estudos, pelo constante incentivo, acompanhamento e apoio a minha vida acadêmica, e por serem meus principais exemplos de vida.

Agradeço aos servidores técnicos-administrativos, docentes e estagiários da Faculdade de Computação por fazerem o curso funcionar.

Agradeço a cada pessoa que de forma direta ou indireta tornou possível a realização deste trabalho.

RESUMO

A gerência dos dispositivos de entrada/saída é uma das principais funções realizadas por um sistema operacional, permitindo que *softwares* aplicativos utilizem os dispositivos por meio de uma interface padronizada. Em decorrência da grande variedade de *hardware* disponível no mercado, é necessário a utilização dos *drivers* de dispositivo, que são programas que recebem comandos genéricos do sistema operacional e executam as rotinas requeridas para cada dispositivo, conforme especificação do fabricante, reduzindo a complexidade da comunicação entre processos de usuário e dispositivos de entrada/saída. Este trabalho tem como objetivo produzir documentação sobre o processo de desenvolvimento de um *driver* de interface de rede para o sistema operacional MINIX 3, servindo como material de apoio para estudo. São apresentadas informações sobre o sistema operacional e também a implementação de um *driver* para interface de rede, como exemplo para explicar seu desenvolvimento, com código-fonte e funcionamento descritos.

Palavras-chave: MINIX. Driver de dispositivo. Redes de computadores.

ABSTRACT

Management of input/output devices is one of the main roles performed by an operating system, allowing application software to use devices through a standardized interface. Due to a wide variety of hardware available on the market, it is necessary to use device drivers, which are programs that receive generic commands from the operating system and execute the required routines for each device, according to the manufacturer's specifications, reducing complexity of communication between user processes and input/output devices. This paper aims to produce documentation on the process of network interface driver development for the MINIX 3 operating system, serving as study support material. Information about the operating system is presented and also the implementation of a network interface driver is presented, as an example to explain its development, with source code and its operation described.

Keywords: MINIX. Device driver. Computer networks

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E SÍMBOLOS

0xN	Número em base hexadecimal
BDF	*Bus/Device/Function* (Barramento/Dispositivo/Função)
BIOS	*Basic Input/Output System* (Sistema Básico de Entrada/Saída)
DMA	*Direct Memory Access* (Acesso Direto à Memória)
E/S	Entrada/Saída
EEPROM	*Electrically Erasable Programmable Read-Only Memory* (Memória Eletricamente Programável e Apagável Somente Leitura)
EISA	*Extended Industry Standard Achitecture* (Arquitetura Padrão da Indústria Estendido)
FTP	*File Transfer Protocol* (Protocolo de Transferencia de Arquivos)
IRQ	*Interrupt Request* (Requisiçao de Interrupção)
PC	*Personal Computer* (Computador Pessoal)
MAC	*Media Access Control* (Controle de Acesso ao Meio)
MII	*Media-Independent Interface* (Interface Independente de Mídia)
PCI	*Peripheral Component Interconnect* (Interconexão para Componentes Periféricos)
POSIX	*Portable Operating System Interface* (Interface Portável para Sistemas Operacionais)
RAM	*Random Access Memory* (Memória de Acesso Aleatório)
RGMII	*Reduced Gigabit Media-Independent Interface* (Interface Independente de Mídia Gigabit Reduzido)
SGE	*SiS Gigabit Ethernet*
TCP	*Trasmission Control Protocol* (Protocolo de Controle de Transmissão)
UDP	*User Datagram Protocol* (Protocolo de Datagramas de Usuário)

SUMÁRIO

1 INTRODUÇÃO

1.1 Sistema Operacional MINIX

O sistema operacional MINIX tem sua origem na década de 1970, quando na época, o código-fonte do sistema operacional UNIX V6, da AT&T, era utilizado para estudos em cursos de sistemas operacionais. Em 1979, com o lançamento do UNIX V7, a nova licença da AT&T impediu o uso de seu código-fonte para estudos. Para resolver este problema, Andrew Tanenbaum escreveu um novo sistema operacional, compatível com o UNIX V7, que pudesse ser usado para o ensino, disponibilizando o código-fonte para estudos e modificações. Lançado em 1987, o MINIX tem como diferencial sua estrutura modular, e é baseado na arquitetura de micronúcleo. (TANENBAUM, 2016)

Em 1997 o MINIX 2 foi lançado, agora reformulado para ser compatível com o padrão POSIX (*Portable Operating System Interface*), aumentando sua portabilidade para computadores além dos baseados na arquitetura IBM PC.

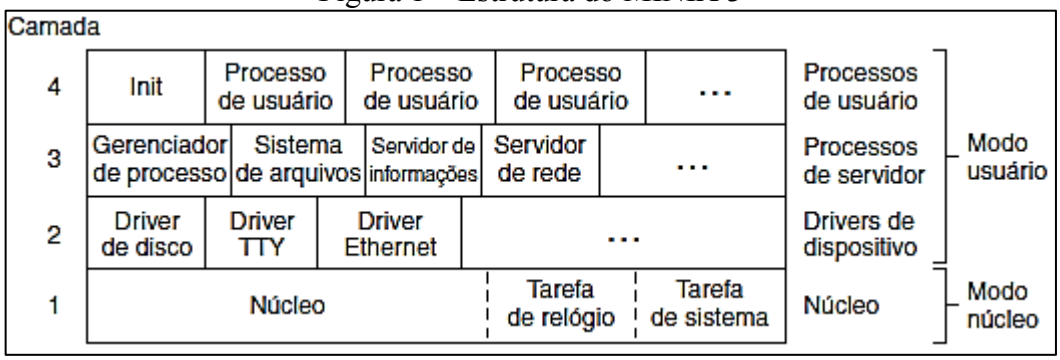
De acordo com Machado e Maia (2007, p. 61) o conceito de arquitetura micronúcleo surgiu no sistema operacional Mach, onde o núcleo oferece somente os serviços essenciais executando em modo núcleo, enquanto que o restante dos serviços é executado em modo usuário, em módulos, isolando cada função do sistema e aumentando sua confiabilidade. Caso um problema ocorra em um dos módulos, este não afetará o sistema inteiro, diferente de sistemas operacionais de arquitetura monolítica, onde todo o sistema operacional é composto por vários módulos ligados a um único executável operando em modo núcleo, de tal forma que a ocorrência de um problema em um módulo pode afetar o núcleo inteiro.

Iniciou-se o desenvolvimento do MINIX 3 em 2004, reestruturando o núcleo com a intenção de se produzir um sistema operacional modular tolerante a falhas e seguro para PCs e sistemas embarcados. Uma das mudanças importantes para o MINIX 3 foi mover os *drivers* de dispositivo para fora do núcleo, que agora são tratados como processos em modo usuário.

A comunicação entre processos é realizada por meio de mensagens, gerenciadas pelo núcleo, com formato padronizado de acordo com os tipos predefinidos no código-fonte. O processo de envio/recebimento de mensagens usa funções bloqueantes, dispensando a necessidade de *buffers* de mensagens.

A Figura 1 ilustra a estrutura atual do MINIX, composta pela arquitetura de micronúcleo em conjunto com a arquitetura de camadas.

Figura 1 – Estrutura do MINIX 3



Fonte: TANENBAUM e WOODHULL, 2008 (p. 120)

Na camada 4 estão os programas de usuário, que utilizam as bibliotecas do sistema para acessar as funcionalidades disponibilizadas pelo sistema operacional fornecidas por meio da interface comum, e o processo *init* (o primeiro processo de usuário iniciado pelo sistema operacional). A camada 3 contém os servidores que implementam as chamadas de alto nível usadas pelos programas de usuário, recebem as requisições para uso das funcionalidades do sistema, e por não terem permissão para acessar diretamente o *hardware*, comunicam-se com os *drivers* para realizar as operações. Os *drivers* de dispositivo estão na camada 2, e podem se comunicar com a tarefa de sistema para realizar leituras e escritas no *hardware* e na memória por meio das chamadas de núcleo. Na camada 1, o núcleo gerencia a comunicação por mensagens e o escalonamento de processos, além de tratar interrupções e acesso às portas de E/S. A tarefa de sistema implementa as chamadas de núcleo usadas pelos servidores e *drivers* e realiza operações de E/S em nome deles. A tarefa de relógio trata as interrupções causadas pelo relógio do sistema.

Os processos de usuário não se comunicam diretamente com os dispositivos para realizar as operações de entrada e saída. Quando um processo de usuário utiliza as funções expostas pela biblioteca padrão, de forma transparente está se comunicando por meio de mensagens com outros processos responsáveis pela operação em cada camada (servidores, *drivers* de dispositivo, tarefa do sistema), que retornam os dados solicitados às camadas superiores.

1.2 *Drivers* de dispositivos

Um *driver* de dispositivo é o programa responsável pela comunicação do subsistema de E/S de um sistema operacional com o controlador do dispositivo. (TANENBAUM, 2009)

Por controlador de dispositivo, entende-se como o componente eletrônico que opera o *hardware* a baixo nível, realizando as operações físicas (mecânicas ou não). É função do *driver* de dispositivo inicializar o dispositivo, receber requisições de operações provenientes do sistema operacional, comandar a execução das operações no controlador, receber o resultado da operação realizada pelo controlador e retornar os dados solicitados de forma que o sistema operacional entenda.

1.3 Trabalhos relacionados

O trabalho de Beach (2003) apresenta a implementação de um *driver* para a interface de rede 3Com 3c905C no sistema operacional Minix-vmd. O Minix-vmd é derivado do MINIX 2, e contém modificações para adicionar novas funcionalidades que ainda não estavam disponíveis no MINIX 2, como o suporte ao barramento PCI (inexistente até a versão 2.0.4). Algumas informações descritas no trabalho ainda são válidas para o MINIX 3, mas outras sofreram alterações com as atualizações do sistema e precisam ser documentadas novamente.

O trabalho de Linnenbank (2009) apresenta a implementação e testes de performance de um *driver* para a interface de rede Intel Pro/1000 no sistema operacional MINIX 3. A interação do *driver* com o sistema operacional não é explicada no texto, que tem foco nos testes de performance, mas o código-fonte é bem organizado e foi absorvido no repositório do MINIX 3, podendo ser usado para entender a estrutura de um *driver* de interface de rede.

O livro “Sistemas Operacionais: Projeto e Implementação” (TANENBAUM; WOODHULL, 2008) trata da implementação de *drivers* de dispositivo no MINIX 3 limitando-se somente a poucos exemplos de dispositivos de bloco (disco de RAM, disco rígido) e caractere (driver de terminal, teclado e vídeo em modo texto). Não há explicações sobre o funcionamento do serviço de rede ou de *drivers* de interface de rede.

1.4 Justificativa

Uma das necessidades para que um sistema operacional tenha sucesso é a capacidade de se adaptar e funcionar em diferentes combinações de dispositivos. Com o constante lançamento de novo *hardware* no mercado, a tarefa de suportar o uso de cada *hardware* específico deixa de ser realizada pelo sistema operacional, cabendo a este somente a gerência de recursos e a disponibilização de uma interface padronizada entre os programas de usuário e os *drivers* de dispositivo (normalmente fornecidos pelo fabricante). É possível que ainda não

exista suporte a um dispositivo específico em um sistema operacional (fabricante não disponibilizou *drivers* por algum motivo), mas com informações suficientes (obtidas em manuais técnicos, kits de desenvolvimento ou engenharia reversa, por exemplo), terceiros podem desenvolver *drivers* de dispositivo, permitindo seu uso.

A realização deste trabalho se justifica pela quantidade reduzida de documentação disponível sobre o desenvolvimento de *drivers* para interface de rede no MINIX 3. A implementação de um novo *driver* de interface de rede permite que o processo de desenvolvimento seja documentado para posterior disponibilização aos interessados, possibilitando a modificação ou criação de novos *drivers* para seus dispositivos, caso ainda não sejam suportados pelo MINIX 3.

1.5 Objetivo geral

O objetivo geral deste trabalho é produzir documentação para o processo de desenvolvimento de um *driver* de interface de rede para o sistema operacional MINIX 3, servindo como material de apoio para estudo.

1.6 Objetivos específicos

a) projetar e implementar um *driver* de interface de rede para um dispositivo real, desta forma o processo poderá ser documentado na prática;
b) realizar testes de desempenho no *driver* implementado e apresentar os resultados obtidos, para validar a implementação;
c) disponibilizar código-fonte e documentação para estudo e realização de novos trabalhos.

1.7 Escopo e organização do trabalho

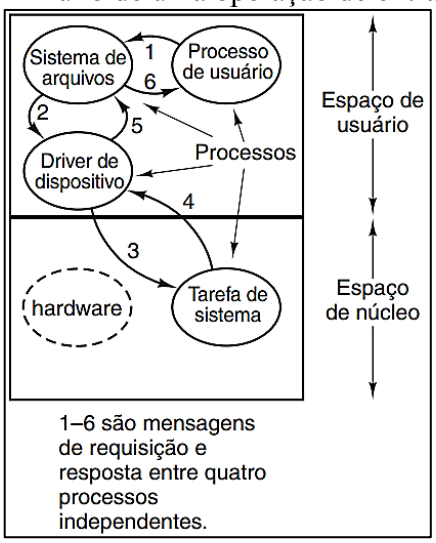
O primeiro capítulo apresenta o trabalho e seu objeto de pesquisa. No segundo capítulo são apresentadas informações sobre o sistema de entrada/saída do MINIX 3.3, sobre o barramento PCI e sua implementação no MINIX 3.3, e sobre o funcionamento do serviço de rede INET no MINIX 3.3. O terceiro capítulo aborda o desenvolvimento de um *driver* de interface de rede, que tem seu funcionamento descrito no código-fonte. Por fim, na conclusão é feita uma avaliação sobre a realização do trabalho e possibilidade de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Entrada e saída no MINIX 3

No MINIX 3 o software de entrada e saída é dividido em vários processos modulares que se comunicam por meio de mensagens. O micronúcleo realiza o tratamento das interrupções de *hardware*, gerencia de memória, escalonamento de processos e comunicação entre processos. Em seguida, temos os *drivers* de dispositivo, que são processos em modo usuário com permissão de uso das chamadas de sistema para ler ou escrever valores nas portas de entrada/saída do *hardware*, quando o acesso ao dispositivo é autorizado pelo micronúcleo. A interface para os processos de usuário é fornecida pelos processos servidores (sistema de arquivos e serviço de rede, por exemplo). Estes recebem as requisições provenientes dos processos de usuário e solicitam aos *drivers* sua execução. A Figura 2 mostra o fluxo de uma operação de entrada/saída no MINIX 3.

Figura 2 – Fluxo de uma operação de entrada/saída



Fonte: TANENBAUM e WOODHULL, 2008 (p. 248)

Por exemplo, quando um processo de usuário solicita a abertura de um arquivo armazenado em disco usando a função `open()`, a implementação da função se encarrega de enviar uma mensagem ao sistema de arquivos (1), contendo os atributos passados pelo programa (caminho do arquivo, *flags*, modo), que então envia uma mensagem ao *driver* (2), com a posição que deverá ser lida. Como o *driver* não tem permissão de acessar diretamente o *hardware*, mas tem permissão para usar as chamadas do sistema para operar o hardware, solicita à tarefa do sistema (3) a realização da operação de leitura a baixo nível. Caso a leitura tenha

sucesso (ou não), a tarefa do sistema retorna ao *driver* uma mensagem com o resultado da operação realizada no *hardware* (4), o *driver* retornará ao sistema de arquivos uma mensagem com o conteúdo solicitado (ou informações sobre o erro) (5), e o sistema de arquivos retornará a resposta conforme especificado pela biblioteca padrão ao programa de usuário (6).

Esta estrutura modularizada tem como objetivo aumentar a confiabilidade do sistema operacional, pois as restrições impostas aos processos executados em modo usuário impedem o uso indevido dos recursos. Isso também permite que os *drivers* sejam atualizados ou reiniciados, em caso de falha, sem parar completamente o sistema operacional. O servidor responsável pela tarefa de verificar se cada *driver* de dispositivo está funcionando corretamente é o *Reincarnation Server* (Servidor de Reencarnação), que periodicamente consulta se um *driver* foi finalizado incorretamente, e os reinicia caso necessário.

2.2 Estrutura dos *drivers* de dispositivo no MINIX

O MINIX 3 oferece suporte para duas classes de dispositivo: dispositivos de bloco e dispositivos de caractere. Um *driver* de dispositivo, independente de sua classe, possui no mínimo uma função principal, que é executada desde a inicialização do sistema e é mantida enquanto o sistema estiver operante. Esta função é responsável por receber as solicitações em forma de mensagens provenientes de outros processos, coordenar a execução da solicitação e enviar a resposta ao solicitante. Além das funções de operação do *hardware*, opcionalmente é possível adicionar funções responsáveis por encerrar ou reiniciar o *driver*. A Figura 3 mostra um exemplo de função principal.

Figura 3 – Esboço da função principal de um *driver* de dispositivo de E/S

```
message mess;                                   /* buffer de mensagem */

void io_driver() {
   initialize();                                /* feito apenas uma vez, durante a inicialização do
                                                   sistema. */
   while (TRUE) {
         receive(ANY, &mess);                   /* espera uma requisição para atender */
         caller = mess.source;                  /* processo de quem veio a mensagem */
         switch(mess.type) {
              case READ:    rcode = dev_read(&mess); break;
              case WRITE:   rcode = dev_write(&mess); break;
              /* Demais casos entram aqui, incluindo OPEN, CLOSE e IOCTL */
               default:     rcode = ERROR;
         }
         mess.type = DRIVER_REPLY;
         mess.status = rcode;                   /* código do resultado */
         send(caller, &mess);                   /* retorna mensagem de resposta para o processo
                                                   que fez a chamada */
   }
}
```

Fonte: TANENBAUM e WOODHULL, 2008 (p. 250)

Após a inicialização específica do *hardware*, o *driver* entra em um laço de repetição, aguardando novas mensagens por meio da função *receive* da classe de dispositivo. Quando chamada, essa função bloqueia a execução do *driver* até que uma nova mensagem esteja disponível para recebimento, continuando a execução para processar a mensagem recebida.

Durante a implementação do *driver* pode ser conveniente importar a biblioteca do sistema contendo o código comum correspondente à classe de dispositivo. Para os dispositivos genéricos de bloco e de caractere, o processo de comunicação entre processos é simplificado para o desenvolvedor, pois a biblioteca do sistema executa os procedimentos para recepção da mensagem, e ao receber uma mensagem, o sistema executa a função correspondente ao tipo de mensagem recebida, de acordo com o que for configurado pelo *driver* em uma estrutura contendo os ponteiros para tais funções. A Figura 4 mostra um exemplo de uso desta estrutura simplificada. No caso dos dispositivos de rede, ainda não existe uma biblioteca de código comum para simplificar a comunicação com o driver, então é necessário tratar as mensagens e executar as funções necessárias manualmente.

Figura 4 – Esboço da função principal para um *driver* de dispositivo de bloco

```
static struct blockdriver dev_tab = {
  .bdr_open = do_open,
  .bdr_close = do_close
};

int main(void) {
  sef_startup();
  blockdriver_task(&dev_tab);
  return(OK);
}
```

Fonte: Próprio autor.

As mensagens utilizadas na comunicação entre processos têm tamanho e formato padronizados (TANENBAUM; WOODHULL, 2008, p. 146), e cada mensagem contém: identificador do processo remetente da mensagem, identificador do tipo de mensagem, e carga útil com estrutura variável de acordo com o tipo da mensagem (com tamanho limitado a 56 bytes). A transferência de quantidades maiores de dados é possível utilizando funções de sistema para cópia segura de memória por meio de concessões (*grants*). Cada concessão define uma área de memória e dá a outro processo específico permissões para leitura/escrita nesta área (HERDER et al., 2009).

O servidor de reencarnação é parte importante do mecanismo de tolerância a falhas do MINIX 3. Ele é usado para iniciar servidores e *drivers*, tornando-se o processo pai destes. Os processos são monitorados periodicamente para detectar quando uma falha ocorre durante a

execução, finalizando e reiniciando os processos automaticamente caso necessário. A comunicação entre o servidor de reencarnação e os *drivers*/servidores é realizada por meio do componente *System Event Framework* (SEF). Os *drivers* (e outros servidores) devem se registrar durante a inicialização configurando o SEF com as funções que respondem às solicitações do servidor de reencarnação (inicialização, verificação de status, atualização em tempo real, recuperação de falhas) e que realizam tratamento dos sinais (notificações de eventos). O usuário pode utilizar a ferramenta "`service`" para interagir com o servidor de reencarnação, gerenciando os *drivers* e serviços.

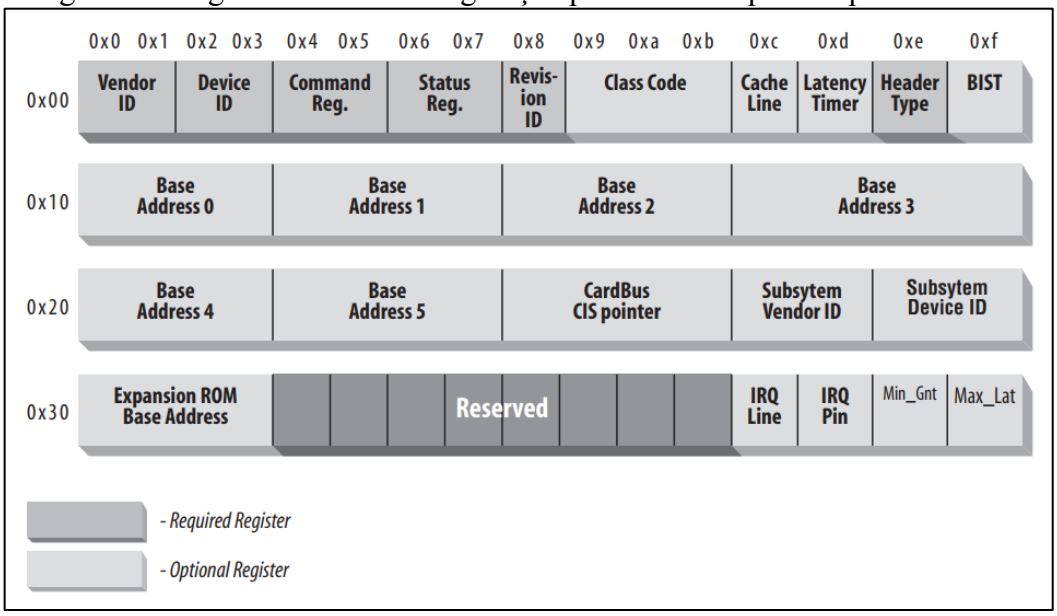
No código-fonte do MINIX 3.3, os drivers estão localizados no diretório "`/minix/drivers/`". Dentro do diretório de cada driver, um arquivo "Makefile" deve ser criado, indicando as configurações para processamento do compilador. Os *drivers* de rede também possuem um arquivo de configuração, onde são definidos os identificadores dos dispositivos que cada *driver* é capaz de controlar, para que somente o *driver* correto seja carregado durante a inicialização do sistema operacional.

2.3 O Barramento PCI

O barramento PCI (*Peripheral Component Interconnect*) é um barramento de alta velocidade para E/S, introduzido em 1992 para substituir o barramento EISA (*Extended Industry Standard Architecture*), e é utilizado como interface de E/S, permitindo a conexão de placas de expansão e periféricos a um sistema computacional (CORBET et al., 2005, p. 302). Uma das vantagens do barramento PCI é a de permitir a detecção e configuração automática dos dispositivos pelo BIOS (*Basic Input/Output System* – Sistema Básico de Entrada/Saída), enquanto que no barramento ISA era necessário definir manualmente configurações como linhas IRQ (*Interrupt Request* – Requisição de Interrupção), canais DMA (*Direct Memory Access* – Acesso Direto à Memória) e endereçamento das portas de E/S.

Cada dispositivo PCI no sistema possui um espaço mapeado em memória com tamanho fixo (256 bytes) para acesso aos registradores de configuração do dispositivo. Esta área contém informações necessárias para identificar e configurar o dispositivo dentro do sistema operacional. A Figura 5 mostra a composição desta área.

Figura 5 – Registradores de configuração padronizados para dispositivos PCI



Fonte: CORBET et al., 2005 (p. 308)

Dos campos obrigatórios mostrados, para identificação do dispositivo são utilizados os campos somente leitura: *Vendor ID* (ID do Fabricante), *Device ID* (ID do Dispositivo) e *Class Code* (Classe). Os campos *Base Address* (Endereços Base) podem conter endereços de regiões para E/S atribuídas pelo BIOS na inicialização, caso indicado pelo dispositivo, onde é possível acessar os registradores de controle. O acesso aos registradores de configuração deve ser restrito somente à inicialização do dispositivo e ao tratamento de erros do barramento, enquanto que a operação do dispositivo é feita nos registradores de controle.

2.4 Arquitetura PCI no MINIX

A biblioteca do sistema dispõe de uma interface para uso da arquitetura PCI. Para utilizar as funções disponíveis, é necessário importar o arquivo de cabeçalho "`minix/syslib.h`", que contém a declaração destas funções, além do arquivo de cabeçalho "`machine/pci.h`", que contém constantes que serão utilizadas nestas funções. A Figura 6 mostra algumas das funções declaradas em "`minix/syslib.h`" para realização de operações com dispositivos PCI.

Figura 6 – Funções para operação de dispositivos PCI

```
void pci_init(void);
int pci_first_dev(int *devindp, u16_t *vidp, u16_t *didp);
int pci_next_dev(int *devindp, u16_t *vidp, u16_t *didp);
void pci_reserve(int devind);
void pci_ids(int devind, u16_t *vidp, u16_t *didp);
u8_t pci_attr_r8(int devind, int port);
u16_t pci_attr_r16(int devind, int port);
u32_t pci_attr_r32(int devind, int port);
void pci_attr_w8(int devind, int port, u8_t value);
void pci_attr_w16(int devind, int port, u16_t value);
void pci_attr_w32(int devind, int port, u32_t value);
char *pci_dev_name(u16_t vid, u16_t did);
char *pci_slot_name(int devind);
int pci_get_bar(int devind, int port, u32_t *base,
      u32_t *size, int *ioflag);
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A função "`pci_init`" é responsável pela inicialização do subsistema PCI no MINIX, executando detecção e configuração dos dispositivos para uso dentro do sistema operacional, caso ainda não tenham sido configurados. Esta função deve ser executada antes de usar qualquer função para manipulação de dispositivos PCI. A função "`pci_first_dev`" é utilizada para obter o primeiro dispositivo PCI disponível (que ainda não foi reservado para uso), retornando como parâmetros: o índice (variável "`devindp`") que será utilizado nas outras funções e identificadores do fabricante e dispositivo (variáveis "`vidp`" e "`didp`"). A função "`pci_next_dev`" é utilizada para obter o próximo dispositivo disponível, a partir do índice especificado (variável "`devindp`"), e retorna como parâmetros os identificadores do fabricante e dispositivo (variáveis "`vidp`" e "`didp`"). A função "`pci_reserve`" marca o dispositivo (identificado pelo índice obtido nas funções de busca) como em uso, reservando o controle somente para o *driver* que o solicitou. A função "`pci_ids`" retorna os identificadores de fabricante e de dispositivo, quando fornecido o índice do dispositivo. As funções "`pci_attr_r8`", "`pci_attr_r16`" e "`pci_attr_r32`" realizam a leitura dos valores de 8, 16 e 32 bits, respectivamente, do espaço de configuração do dispositivo, recebendo como parâmetro o índice do dispositivo e o endereço para leitura, e retorna o valor obtido. Da mesma forma, as funções "`pci_attr_w8`", "`pci_attr_w16`" e "`pci_attr_w32`" realizam a escrita de valores para os endereços especificados, recebendo como parâmetro o índice do dispositivo, o endereço para escrita e o valor a ser escrito. Para estas funções de leitura e escrita nos registradores de configuração PCI é cômodo utilizar as constantes declaradas em "`machine/pci.h`" em vez de especificar manualmente o endereço dos registradores. A Tabela 1 mostra algumas das constantes disponíveis para uso. A função "`pci_dev_name`" obtém o

nome do dispositivo (quando existe um nome registrado na tabela de dispositivos PCI associado aos identificadores de fabricante e dispositivo em "`/minix/drivers/bus/pci/pci_table.c`"). A função "`pci_slot_name`" retorna uma cadeia de caracteres contendo o endereço em notação BDF (*Bus/Device/Function* – Barramento/Dispositivo/Função) do dispositivo com o índice informado como parâmetro. A função "`pci_get_bar`" obtém o endereço base do espaço para operações de E/S, recebendo como parâmetro o índice do dispositivo e ponteiros para variáveis onde serão armazenados: o endereço obtido, o tamanho do espaço reservado para o dispositivo e o tipo do endereço obtido.

Tabela 1 – Constantes para endereços dos registradores de configuração PCI

`PCI_VID`	ID do Fabricante
`PCI_DID`	ID do Dispositivo
`PCI_CR`	Registrador de Comando
`PCI_SR`	Registrador de Estado

`PCI_BAR`
`PCI_BAR_2`
`PCI_BAR_3`
`PCI_BAR_4`
`PCI_BAR_5`
	Registradores de Endereço Base

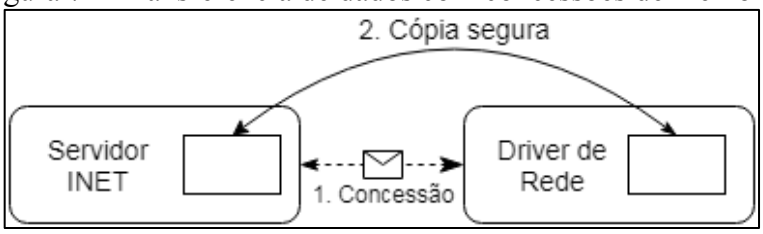
`PCI_ILR`	Registrador de Linha de Interrupção

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

2.5 O servidor de rede INET

O servidor de rede INET é responsável pelo suporte à rede no MINIX 3. Como visto anteriormente, os processos servidores estão na camada imediatamente superior aos *drivers* de dispositivo, então o servidor de rede recebe as solicitações de comunicação realizadas pelos demais processos de usuário e solicita sua execução no *hardware* por meio dos *drivers*. Um *driver* de rede tem como função realizar a recepção dos pacotes Ethernet preparados pelo INET e enviá-los para o *hardware*, assim como encaminhar pacotes recebidos pelo *hardware* para processamento pelo INET, e posterior retorno dos dados aos programas de usuário. A comunicação entre o *driver* e o INET é realizada por meio de mensagens com tamanho fixo do sistema operacional, e quando há necessidade de transferência de quantidades maiores de dados (carga útil dos pacotes enviados/recebidos, por exemplo), as mensagens contem concessões para que um processo possa acessar espaços de memória de outro processo, onde é permitido a leitura ou escrita usando as chamadas de sistema para cópia segura de memória, como mostrado na Figura 7:

Figura 7 – Transferência de dados com concessões de memória



Fonte: Próprio autor.

Um dos processos age como concedente, cria uma concessão para um intervalo de memória de tamanho determinado dentro de seu espaço de memória do processo, e fornece ao outro processo esta concessão (1). O processo que a recebe pode então copiar dados de seu próprio espaço de memória para a posição indicada na concessão, ou então copiar dados para o espaço concedido (2).

A implementação dos *drivers* de rede requer a importação do arquivo de cabeçalho "`minix/netdriver.h`", contendo as funções para comunicação com o INET:

Tabela 2 – Funções declaradas no arquivo *minix/netdriver.h*

Função	**Descrição**
`netdriver_announce`	Anuncia que o driver está pronto para uso.
`netdriver_receive`	Recebe novas mensagens do INET.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A função `netdriver_announce` é chamada ao fim do processo de inicialização do *driver* para avisar o INET que está pronto para receber solicitações.

As mensagens no MINIX são compostas por um cabeçalho, contendo o identificador do processo (campo `m_source`) e o tipo de mensagem (`m_type`), e a carga útil (de até 56 bytes) que varia de acordo com o tipo de mensagem.

Figura 8 – Estrutura de uma mensagem genérica

`m_source` (4 bytes)	`m_type` (4 bytes)	<DADOS> (56 bytes)

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A função `netdriver_receive` é utilizada dentro do laço de repetição na função principal do *driver* para receber as requisições, que podem ser de quatro tipos:

Tabela 3 – Tipos de mensagens de requisição do INET

Tipo	**Descrição**
DL_CONF	Inicialização do dispositivo.
DL_GETSTAT_S	Obter estatísticas do dispositivo.
DL_WRITEV_S	Realizar transmissão de dados.
DL_READV_S	Realizar recepção de dados.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

As mensagens do tipo DL_CONF são solicitações para configuração do *hardware*. Ao receber essa requisição, o *driver* deve inicializar o dispositivo, caso ainda não o tenha feito, definir o modo de recepção de pacotes e o endereço físico da interface. As mensagens do tipo DL_GETSTAT_S são solicitações para obter estatísticas (erros de transmissão/recepção, colisões, entre outros). As mensagens dos tipos DL_WRITEV_S e DL_READV_S são, respectivamente, solicitações para escrita (transmissão) e leitura (recepção) de dados. Também é possível que o *driver* receba interrupções de *hardware* para tratamento (mensagens enviadas pela tarefa HARDWARE).

Durante a configuração inicial, o INET espera que a interface de rede tenha um endereço físico, que pode ser definido a partir de uma variável de ambiente, ou lida do *hardware*, caso este ofereça suporte. A requisição de configuração também inclui o modo de recepção de pacotes solicitado:

Tabela 4 – Modos de recepção de pacotes da rede

Modo	**Descrição**
DL_PROMISC_REQ	Receber quaisquer pacotes (modo promíscuo).
DL_MULTI_REQ	Receber pacotes *multicast*.
DL_BROAD_REQ	Receber pacotes *broadcast*.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

Por padrão, a interface de rede deve ser capaz de receber pacotes endereçados a si mesmo. É possível configurar a interface para recepção de qualquer pacote no meio de transmissão, mesmo que não seja destinada a ela, usando o modo promíscuo. De acordo com a necessidade e configuração da rede, também é possível receber pacotes de difusão ampla, usando o modo *broadcast*, ou então pacotes de difusão seletiva, usando o modo *multicast*.

Quando a transmissão ou recepção dos pacotes de rede ocorre, a mensagem de solicitação contém um vetor de requisições de E/S, contendo as concessões de memória e endereços para transferência dos dados com as funções para cópia segura de memória:

Tabela 5 – Funções para cópia segura de memória

Função	**Descrição**	**Parâmetros**
`sys_safecopyfrom`	Copia dados de um outro processo, a partir do local indicado.	Processo de origem, Concessão de memória, Desvio (relativo ao endereço inicial), Endereço inicial dos dados, Tamanho dos dados.
`sys_safecopyto`	Copia dados para um outro processo, no local indicado.	Processo de destino, Concessão de memória, Desvio (relativo ao endereço inicial), Endereço inicial dos dados, Tamanho dos dados.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2016.

A função `sys_safecopyfrom` é usada para copiar os dados a serem transmitidos do INET para o *driver*, que prepara os dados para transmissão conforme especificações da interface de rede, enquanto que a função `sys_safecopyto` é usada para copiar os dados que foram recebidos e tratados pelo *driver* para o INET.

3 ESTUDO DE CASO: O DRIVER SGE

3.1 Interface de rede SiS190/SiS191

A SiS191 é uma interface de rede integrada ao *southbridge* SiS968 (controlador de E/S na arquitetura Intel), fabricada pela Silicon Integrated Systems Corp. De acordo com as especificações da fabricante (SILICON INTEGRATED SYSTEMS CORP, 2007), é compatível com o padrão IEEE 802.3 (Ethernet), suporta conectividade *half-duplex* e *full-duplex* em par trançado nas velocidades de 10/100/1000Mbps. Outra versão de baixo custo também foi lançada como SiS190, com o mesmo funcionamento, porém suportando apenas as velocidades 10/100 Mbps. A funcionalidade para a camada física é fornecida por meio de um transceptor[1] externo, este utilizando a interface MII (*Media-Independent Interface*) ou RGMII (*Reduced Gigabit Media-Independent Interface*) para comunicação com o controlador da interface. A comunicação com o sistema é realizada por meio do barramento PCI, e a transferência de dados é realizada por acesso direto à memória (DMA – *Direct Memory Access*).

Antes de iniciar o desenvolvimento do *driver* para este dispositivo, é necessário verificar a viabilidade do projeto e obter informações sobre o *hardware*. A seguir, as perguntas propostas por Cort (2016) são respondidas para avaliar a possibilidade de prosseguir com o desenvolvimento:

- **O dispositivo já é suportado?** Não. Atualmente não há suporte para o dispositivo em nenhuma versão do MINIX.
- **Um dispositivo similar já é suportado?** Sim. O MINIX oferece suporte para dispositivos de rede.
- **Alguém já está trabalhando neste dispositivo?** Não. Até o momento da realização deste trabalho, não foram encontrados trabalhos em desenvolvimento para o mesmo dispositivo.
- **Existe documentação disponível?** Parcialmente. A fabricante disponibilizou um *driver* em código aberto para o núcleo Linux 2.6.9, mas não disponibiliza o *datasheet* publicamente.
- **Existe hardware disponível?** Sim. Esta interface de rede é utilizada como solução integrada na placa-mãe em computadores de mesa e portáteis de diversos fabricantes.

[1] Unidade que realiza transmissão e recepção de quadros no meio físico como sinais elétricos.

- **O dispositivo é suportado por outros sistemas operacionais de código aberto?** Sim. Há suporte para os sistemas operacionais Linux, Haiku, OpenBSD e FreeBSD por meio de *drivers* desenvolvidos pela comunidade.
- **É possível utilizar o dispositivo em um sistema suportado pelo MINIX?** Sim. Este dispositivo é utilizado somente na arquitetura x86, que já é suportada pelo MINIX.

Considerando que não há documentação oficial disponível além do *driver* fornecido pela fabricante e que este não foi atualizado para suportar versões mais recentes do núcleo Linux, utilizou-se adicionalmente como referência os códigos-fonte para *drivers* disponíveis em outros sistemas operacionais de código aberto (Haiku, OpenBSD, NetBSD), *datasheet* para a interface de rede SiS900 (modelo mais antigo, que compartilha algumas similaridades com a SiS191) e o código-fonte para o *driver* e1000 (Intel Pro/1000) no MINIX (para entendimento da estrutura e funcionamento).

O nome escolhido para o *driver* (SGE, acrônimo para *SiS Gigabit Ethernet*) é igual ao nome do *driver* para o sistema operacional FreeBSD, e segue a convenção de nomes curtos usada também no MINIX.

Para realizar o desenvolvimento do *driver*, optou-se por utilizar *hardware* real, pois não existe emulador deste dispositivo disponível, e então desta forma a depuração do código pôde ser realizada por meio de saída em vídeo ou em *log* do sistema.

3.2 Especificação de funcionalidades

O novo *driver* de dispositivo de rede deve prover as seguintes funcionalidades ao sistema operacional: configuração (inicialização) do *hardware*, transmissão/recepção de pacotes e relatório das estatísticas de erros. A maior parte do trabalho ocorre durante a configuração, pois cada dispositivo tem sua própria rotina de inicialização. Quanto à recepção de pacotes, é necessário ler os pacotes do *hardware* e copiar para o serviço de rede, enquanto na transmissão os pacotes são recebidos do serviço de rede e copiados para o *hardware* executar a transmissão. O relatório das estatísticas de erros depende de a interface de rede ser capaz de fornecer o status de cada operação realizada, ou de todas as operações.

O manual de referência do produto deve descrever como programar o *hardware* para realizar cada funcionalidade, mas caso não seja possível o acesso ao manual (alguns fabricantes disponibilizam este manual apenas para empresas parceiras, somente mediante acordo de confidencialidade), também é possível entender o funcionamento das operações observando o

código de *drivers* disponíveis em outros sistemas operacionais de código aberto, que podem ter sido desenvolvidos pela própria fabricante ou então pela comunidade.

3.2.1 Configuração

O *driver* deve utilizar as funções do sistema para tratamento de dispositivos PCI para encontrar um dispositivo compatível e configurá-lo, assim como deve alocar a memória que será utilizada durante a operação.

A busca do dispositivo é realizada utilizando o ID de fabricante e ID de dispositivo. O *driver* espera encontrar um dispositivo que se identifique com o ID de fabricante 0x1039 (atribuído à *Silicon Integrated Systems*) e ID de dispositivo 0x0190 (SiS190) ou 0x0191 (SiS191). Os IDs de fabricante são atribuídos pelo *PCI Special Interest Group*, associação de empresas que mantem as especificações do barramento PCI padrão, enquanto que os IDs de dispositivos são atribuídos a critério do fabricante. Uma lista com IDs de fabricante e IDs de dispositivo pode ser encontrada em (PCI DATABASE, s.d.).

Durante a configuração do dispositivo no BIOS, os registradores de controle do dispositivo são mapeados em memória e tem seu endereço armazenado em um dos registradores de configuração PCI. Este endereço é um endereço físico e deve ser mapeado para o espaço de memória do processo do *driver* para utilizar na realização dos comandos. Então, os registradores de controle devem ser inicializados para um estado conhecido, evitando que qualquer valor estranho afete a configuração da interface de rede.

O dispositivo faz uso de *buffers* para armazenar temporariamente os pacotes de rede e seus descritores. O uso recurso *bus mastering* deve ser habilitado, permitindo que a interface de rede realize transferências de dados com acesso direto à memória (DMA) para estes espaços. A memória que será utilizada para estes armazenamentos temporários deve ser alocada em um espaço contíguo na memória física, e este endereço deve ser informado ao dispositivo. Por fim, é possível habilitar a transmissão e recepção dos pacotes, configurando os registradores de controle de acordo.

3.2.2 Transmissão e recepção de pacotes

A principal funcionalidade da interface de rede é a de transmitir e receber dados. Quando recebidos, os pacotes são armazenados em memória no local informado previamente durante a configuração, o dispositivo gera uma interrupção para avisar o sistema que novos dados estão

disponíveis para processamento, o *driver* então trata a interrupção, iniciando o processamento dos pacotes, para então copiá-los para o INET e liberar o espaço para reutilização.

Para a transmissão de pacotes, o *driver* recebe do INET uma solicitação para transmissão de pacote, os dados são copiados do INET para o espaço de memória alocado para a interface de rede, o *driver* marca o pacote para avisar que este está pronto para envio, e então a interface de rede, ao detectar que há um novo pacote pronto que ainda não foi enviado, realiza a transmissão deste e libera o espaço para reutilização.

3.2.3 Estatísticas de erros

No decorrer dos processos de transmissão e recepção, erros podem acontecer por motivos diversos (cabeamento ruim, congestionamento, entre outros). As informações sobre os erros ocorridos durante a operação devem ser armazenadas para reportar ao sistema, quando solicitado. Dependendo da interface de rede, esses dados podem ser obtidos verificando o status de cada pacote enviado ou recebido, como no caso da SiS900, ou então, cumulativamente em um registrador específico para obter a contagem de erros, como acontece na Intel Pro/1000.

3.3 Implementação

O *driver* desenvolvido é composto por quatro arquivos (listados no Apêndice A), que devem estar no diretório "`/minix/drivers/net/sge/`", os quais serão descritos a seguir:

- **`Makefile`**: Arquivo responsável por configurar o compilador com as opções necessárias para a correta compilação do *driver* e ligação com as bibliotecas do sistema utilizadas.
- **`sge.conf`**: Arquivo de configuração que descreve o tipo de dispositivo, indica as funções do sistema que requer permissão para uso, e os serviços com que pode se comunicar. Para *drivers* de dispositivos PCI também indica quais dispositivos é capaz de controlar, identificados por meio dos IDs de fabricante e de dispositivo.
- **`sge.h`**: Arquivo de cabeçalho, contém definições de constantes e declaração de estruturas utilizadas durante a implementação.
- **`sge.c`**: Principal arquivo, contendo o código para execução de todas as funções do *driver*.

3.3.1 Arquivo “Makefile”

A compilação do *driver* requer a inclusão das dependências utilizadas, como as bibliotecas do sistema que se encontram em diversos diretórios do código-fonte do sistema. Para evitar que o usuário precise digitar comandos extensos a cada parte do sistema que deve ser compilada e ligada com as bibliotecas, este processo pode ser automatizado utilizando os arquivos “`Makefile`”, lidos pelo programa “`make`”. Esta ferramenta é amplamente utilizada em outros sistemas operacionais (por exemplo: GNU Make no Linux, nmake no Microsoft Visual Studio), porém a sintaxe dos comandos pode ser diferente entre cada distribuição. O MINIX usa uma versão do `make` baseada na distribuição Berkeley Unix, conhecida como BSD Make.

Quando o comando `make` é executado, o programa lê o arquivo `Makefile` e configura o ambiente de acordo com as instruções especificadas. A Figura 9 mostra o código do arquivo `Makefile` utilizado para compilação do *driver* que foi desenvolvido.

Figura 9 – Código fonte do arquivo Makefile

```
1   # Makefile for the SiS 190/191 Ethernet Controller driver.
2   PROG= sge
3   SRCS= sge.c
4
5   FILES=$(PROG).conf
6   FILESNAME=$(PROG)
7   FILESDIR= /etc/system.conf.d
8
9   DPADD+=     ${LIBNETDRIVER} ${LIBSYS}
10  LDADD+=     -lnetdriver -lsys
11
12  .include <minix.service.mk>
```

Fonte: Próprio autor.

Comentários podem ser adicionados para auxiliar a leitura do arquivo pelo programador, e suas linhas são precedidas pelo símbolo “`#`” (cerquilha) e estas linhas são ignoradas pelo interpretador de comandos (linha 1). Em seguida, diversas variáveis são definidas: a variável `PROG` define o nome do programa que será compilado e a variável `SRCS` define os arquivos de entrada para o compilador (linhas 2 e 3), a variável `FILES` é definida para indicar os arquivos adicionais que serão instalados (quando for solicitada a instalação do programa compilado), a variável `FILESNAME` é usada opcionalmente para indicar o nome de cada arquivo, caso seja necessário renomear os arquivos de saída, e a variável `FILESDIR` indica o diretório onde estes arquivos serão instalados (linhas 5 a 7). A variável `DPADD` é usada para especificar as

dependências utilizadas enquanto que a variável `LDADD` adiciona as opções ao compilador para utilizar as dependências especificadas (linhas 9 e 10). O comando "`.include`" (linha 12) adiciona um arquivo externo contendo instruções adicionais para processamento (o arquivo "`/minix/share/mk/minix.service.mk`" contém configurações para compilação de serviços ou *drivers*). Com esta configuração, o arquivo "`sge.c`" é compilado para o binário "`sge`", ligado às bibliotecas utilizadas, a saída é copiada para "`/service/sge`", e o arquivo "`sge.conf`" é copiado para "`/etc/system.conf.d/sge`".

Detalhes sobre as demais opções disponíveis para uso podem ser consultadas no arquivo "`/minix/share/mk/bsd.README`", no código-fonte do MINIX, e no manual de uso do comando, que pode ser consultado por meio do comando "`man make`", dentro do MINIX.

3.3.2 Arquivo "sge.conf"

O arquivo de configuração, presente para cada *driver* e serviço no diretório "`/etc/system.conf.d`", contém as configurações para execução do programa por meio do `minix-service` (gerenciador de serviços do sistema), incluindo privilégios, com quem está autorizado a se comunicar, quais chamadas de sistema pode usar e quais dispositivos PCI podem ser controlados (no caso dos *drivers*). A Figura 10 mostra o código do arquivo de configuração para o *driver* de rede desenvolvido.

Figura 10 – Código fonte do arquivo sge.conf

```
1   service sge
2   {
3      type net;
4      descr "SiS 190/191 Ethernet Controller";
5      system
6            UMAP     # 14
7            IRQCTL   # 19
8            DEVIO    # 21
9      ;
10     pci device     1039:0191;
11     pci device     1039:0190;
12     ipc
13           SYSTEM pm rs tty ds vm
14           pci inet lwip
15     ;
16  };
```

Fonte: Próprio autor.

Na primeira linha é indicado o executável que inicia o serviço/*driver*: o programa `sge` que foi compilado. Em seguida é definido o tipo de serviço (atributo "`type`") e uma descrição textual (atributo "`descr`") é adicionada (linhas 3 e 4). A seção "`system`" é onde as chamadas de sistema utilizadas são especificadas (linhas 5 a 9). A Tabela 6 mostra algumas das configurações possíveis para este atributo. É possível consultar todas as chamadas de sistema disponíveis no arquivo "`/minix/include/minix/com.h`".

Tabela 6 – Configurações do atributo `system` para chamadas de sistema

`ALL`	Permite o uso de todas as chamadas de sistema.
`BASIC`	Permite o uso somente das chamadas de sistema definidas pelo macro `SYS_BASIC_CALLS` no arquivo "`minix/com.h`"
`NONE`	Não permite o uso de chamadas de sistema.
`UMAP`	Mapeamento de endereços virtuais para endereços físicos.
`VIRCOPY`	Cópia de dados utilizando endereçamento virtual.
`PHYSCOPY`	Cópia de dados utilizando endereçamento físico.
`DEVIO`	Ler ou gravar dados em um dispositivo de E/S.
`IRQCTL`	Definir ou redefinir uma política de interrupções

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2016.

O atributo "`pci device`" é usado para especificar quais dispositivos o serviço tem permissão para controlar, e para cada dispositivo identificado pela dupla dos IDs de fabricante e IDs de dispositivo, uma linha é adicionada com estas informações (linhas 10 e 11). Por fim, o atributo "`ipc`" é usado para definir os processos com o qual o serviço pode se comunicar (linhas 12 a 15). A Tabela 7 mostra algumas das possíveis opções para esta configuração. Também é possível identificar cada processo explicitamente por seu nome: "`pci`" para o serviço PCI e "`inet`" para o serviço de rede, por exemplo.

Tabela 7 – Configurações do atributo `ipc` para comunicação interprocessos

`ALL`	Comunicação com todos os processos possíveis.
`ALL_SYS`	Comunicação com todos os processos de sistema.
`NONE`	Comunicação não permitida.
`SYSTEM`	Comunicação com os processos responsáveis pelas chamadas de sistema.
`USER`	Comunicação com processos de usuário.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

Detalhes sobre o formato do arquivo e opções disponíveis podem ser consultadas no manual de uso, que pode ser acessado por meio do comando "`man system.conf`", dentro do MINIX.

3.3.3 Arquivo “sge.h”

O código deste arquivo está listado no Apêndice A (a partir da página 58). Este arquivo de cabeçalho contém as constantes e declaração de estruturas utilizadas no decorrer do programa. O código inteiro deste arquivo está envolvido na construção “`#ifndef`” (linha 13) e “`#endif`” (linha 258) para que o arquivo seja incluído apenas uma única vez durante a compilação, evitando problemas como tentativas de redefinição de constantes e tipos já definidos. O arquivo de cabeçalho “`net/gen/ether.h`” é incluído (linha 16) para podermos utilizar as constantes relacionadas ao tamanho de um pacote Ethernet (Tabela 8):

Tabela 8 – Constantes definidas no arquivo net/gen/ether.h

Constante	**Descrição**
`ETH_MIN_PACK_SIZE`	Tamanho mínimo de um pacote Ethernet.
`ETH_MAX_PACK_SIZE`	Tamanho máximo de um pacote Ethernet, sem o cabeçalho.
`ETH_MAX_PACK_SIZE_TAGGED`	Tamanho máximo de um pacote Ethernet, com cabeçalho.
`ETH_HDR_SIZE`	Tamanho do cabeçalho.
`ETH_CRC_SIZE`	Tamanho da soma de verificação.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

O arquivo “`net/gen/eth_io.h`” também é incluído (linha 17) para utilizar a estruturas do tipo “`eth_stat_t`”, usadas para armazenar as estatísticas de transmissão e de erros.

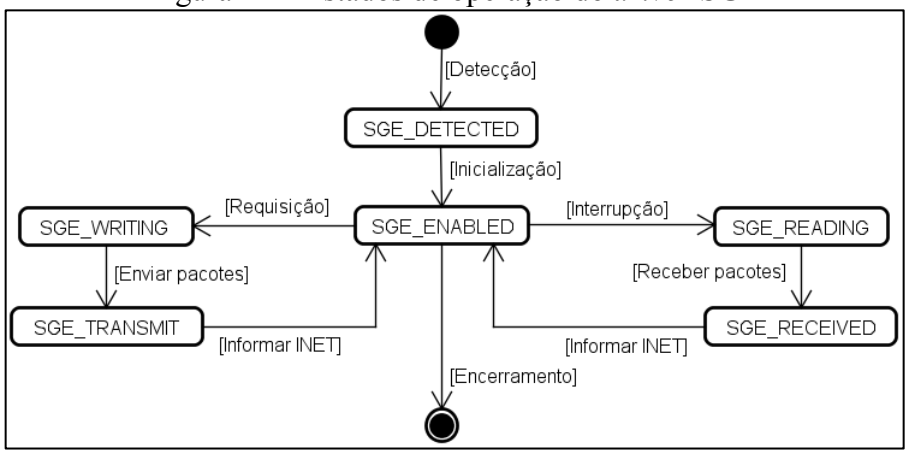
Em seguida são declaradas algumas constantes que serão utilizadas pelo *driver*. Por convenção, as constantes e funções recebem um prefixo padronizado em seus nomes para diferenciá-las das constantes e funções do sistema.

A constante “`SGE_ENVVAR`” (linha 20) é usada para definir qual variável de ambiente o *driver* deve ler para obter os parâmetros definidos pelo usuário na inicialização, como endereço MAC (quando não se deseja utilizar o endereço próprio da interface de rede). As constantes “`SGE_DEV_0190`” e “`SGE_DEV_0191`” (linhas 23 e 24) contém os IDs de dispositivo para os modelos SiS190 e SiS191, respectivamente, usados na função de busca do *hardware* para diferenciar cada modelo.

Foram definidos os estados de operação possíveis da interface de rede (linhas 27 a 32). Usar estados de operação definidos impedem que o sistema tente executar uma operação sem ter antes concluído a anterior, ou que tente transmitir/receber dados sem ter antes inicializado

o *hardware* de maneira correta ou informado ao serviço de rede sobre a conclusão da última operação realizada. A Figura 11 mostra o diagrama dos estados de operação do *driver*.

Figura 11 – Estados de operação do *driver* SGE



Fonte: Próprio autor.

O primeiro estado (SGE_DETECTED) é definido quando o *driver* detecta a presença do dispositivo. Caso o dispositivo não esteja conectado, o programa encerra retornando erro.

- **`SGE_DETECTED`**: O *hardware* foi detectado no barramento PCI e aguarda configuração.
- **`SGE_ENABLED`**: O *hardware* foi inicializado, configurado e está pronto para transmissão ou recepção de pacotes.
- **`SGE_READING`**: Uma operação de leitura (recepção de pacote) está em execução.
- **`SGE_WRITING`**: Uma operação de escrita (transmissão de pacote) está em execução.
- **`SGE_RECEIVED`**: A operação de leitura foi concluída e deve ser reportada ao serviço de rede.
- **`SGE_TRANSMIT`**: A operação de escrita foi concluída e deve ser reportada ao serviço de rede.

Do mesmo modo, também são definidas constantes para identificar o modo de recepção de pacotes na configuração (linhas 35 a 37), representados como:

- **`SGE_PROMISC`**: Habilitar recepção de qualquer pacote, mesmo que não seja endereçado para a interface (modo promíscuo).
- **`SGE_MULTICAST`**: Habilitar recepção de pacotes *multicast* (pacotes que são endereçados a grupos específicos de máquinas na rede).
- **`SGE_BROADCAST`**: Habilitar recepção de pacotes *broadcast* (pacotes que são endereçados a todas as máquinas na rede).

Nas linhas 40 a 44 são definidas as constantes que representam a velocidade de conexão e o modo de comunicação no meio físico negociados pela interface (Tabela 9):

Tabela 9 – Constantes relacionadas a velocidade de conexão e modo de comunicação

Constante	**Descrição**
`SGE_SPEED_10`	Velocidade de conexão: 10Mbps
`SGE_SPEED_100`	Velocidade de conexão: 100Mbps (*Fast Ethernet*)
`SGE_SPEED_1000`	Velocidade de conexão: 1000Mbps (*Gigabit Ethernet*)
`SGE_DUPLEX_ON`	Modo *full-duplex*
`SGE_DUPLEX_OFF`	Modo *half-duplex*

Fonte: Próprio autor.

As velocidades de conexão possíveis são as velocidades padrões do Ethernet, e permitem comunicação em *full-duplex*, com transmissão e recepção simultânea de dados ou *half-duplex* que realiza transmissão ou recepção de forma alternada.

As constantes relacionadas aos *buffers* de recepção e transmissão são definidas nas linhas 47 a 55, descritas na Tabela 10. Com estes valores calculados é possível alocar a quantidade necessária de memória para a operação do dispositivo.

Tabela 10 – Constantes relacionadas aos buffers de recepção/transmissão

Constante	**Descrição**
`SGE_IOVEC_NR`	Quantidade máxima de vetores de requisição de E/S utilizados em cada operação.
`SGE_BUF_SIZE`	Tamanho do *buffer* (em bytes) para cada pacote.
`SGE_RXDESC_NR`	Quantidade de descritores de pacotes para recebimento.
`SGE_TXDESC_NR`	Quantidade de descritores de pacotes para envio.
`SGE_RXB_TOTALSIZE`	Tamanho total (em bytes) do buffer de pacotes para recepção.
`SGE_TXB_TOTALSIZE`	Tamanho total (em bytes) do buffer de pacotes para transmissão.
`SGE_RXD_TOTALSIZE`	Tamanho total (em bytes) do buffer de descritores de pacotes para recebimento.
`SGE_TXD_TOTALSIZE`	Tamanho total (em bytes) do buffer de descritores de pacote para transmissão.
`SGE_DESC_FINAL`	Marcador do último descritor de pacote na memória.

Fonte: Próprio autor.

A quantidade de vetores de requisição de E/S utilizados durante as operações de leitura e escrita será limitado ao definido na constante `SGE_IOVEC_NR`. As outras constantes definem a quantidade de descritores de pacote que podem ser armazenados, o tamanho reservado para

cada pacote individual e para todos os pacotes, incluindo seus descritores. O descritor de pacote é uma estrutura usada pela interface de rede que contém informações sobre cada pacote armazenado no *buffer*. O *buffer* de descritores de pacote se comporta como um vetor, então para identificar o último índice disponível, o último descritor recebe um marcador que o identifique como o fim da lista. Por fim, é declarada a constante contendo o valor que serve para marcar o último descritor disponível (`SGE_DESC_FINAL`). Desta forma, quando a interface de rede alcançar este descritor na memória, reconhece que está no fim do espaço de memória e retoma a busca por descritores livres para uso a partir do primeiro descritor.

O acesso aos registradores de controle da interface de rede é realizado tendo como ponto de origem o endereço base obtido nos registradores de configuração PCI. Para cada registrador de controle foi definido uma constante contendo seu desvio relativo ao endereço base (linhas 58 a 92), então para obter a localização de um registrador específico, basta somar o endereço base dos registradores de controle com o desvio relativo de tal registrador e com este endereço resultante poderá ser realizada a operação que for necessária. A Figura 12 demonstra o acesso ao valor de um registrador com desvio dado por "`offset`" a partir do endereço base definido em "`baseaddr`".

Figura 12 – Acesso a registrador de controle

```
u32_t valor;
valor = *(volatile u32_t *)(baseaddr + offset);
```

Fonte: Próprio autor.

Como o *driver* oficial disponibilizado pelo fabricante para o sistema operacional Linux não é claro quanto a função de cada registrador, as informações foram obtidas no código-fonte do *driver* para o sistema operacional Haiku (HAIKU INC, 2017). O Anexo A contém uma lista com todos os registradores de operação conhecidos da interface de rede SiS191 e uma descrição breve de sua função. As demais constantes (linhas 94 a 193) tratam-se de valores usados na manipulação destes registradores, e serão descritas conforme são utilizadas no próximo tópico.

Outra importante declaração feita neste arquivo é a da estrutura do tipo "`sge_desc_t`" (linhas 196 a 203) que representa um descritor de pacote, contendo os seguintes campos:

- **`pkt_size`**: Tamanho dos dados contidos no *buffer*.
- **`status`**: Informações de controle sobre o pacote.
- **`buf_ptr`**: Ponteiro para o local no *buffer* onde os dados se encontram. Este deve ser um endereço físico, pois é usado para transferências com acesso direto à memória pela interface de rede.

- `flags`: Tamanho total disponível para o pacote no *buffer*. Este atributo é definido durante a inicialização do espaço de memória para utilização como *buffer*.

A estrutura do tipo "`sge_t`" (linhas 205 a 246) é usada para armazenar informações sobre o estado do driver, contendo informações sobre o *hardware*, sobre os *buffers* e sobre as requisições do serviço de rede. A estrutura "`mii_phy`" (linhas 248 a 256) é usada para armazenar informações sobre o transceptor: endereço, identificação de fabricante e modelo, estado e tipo.

3.3.4 Arquivo "sge.c"

Este é o principal arquivo do *driver*, que contém todas as funções realizadas. O código deste arquivo está listado no Apêndice A (a partir da página 64).

O funcionamento do código será descrito a seguir na ordem em que aparecem no arquivo. Antes de declarar os protótipos das funções, são incluídos alguns arquivos de cabeçalho necessários para uso de mais algumas funções do sistema ("`minix/drivers.h`", que importa todos os outros cabeçalhos necessários para implementação de *drivers*, "`minix/netdriver.h`" para utilizar as funções do serviço de rede e "`machine/pci.h`" para utilizar as funções do barramento PCI), e o arquivo de cabeçalho "`sge.h`" visto anteriormente (linhas 13 a 16).

As variáveis globais "`sge_instance`" e "`sge_state`" (linhas 18 e 19) representam uma instância do *driver* para o sistema: cada interface de rede corresponde a uma instância de *driver*. Atualmente, ainda não é suportada a utilização de mais de uma interface de rede com o mesmo *driver*, mas esta função está planejada para ser implementada na próxima atualização do servidor de rede.

Em seguida, são declarados os protótipos das funções que serão implementadas (linhas 21 a 52). A Tabela 11 lista todas as funções implementadas no *driver* e descreve brevemente a função de cada uma.

Tabela 11 – Funções no *driver* SGE

(continua)

Função	**Descrição**
`main`	Função principal do *driver*.
`sge_init`	Trata a solicitação de configuração do dispositivo.
`sge_init_pci`	Inicializa o subsistema PCI.
`sge_probe`	Detecta o dispositivo e o mapeia em memória.
`sge_init_hw`	Realiza a configuração inicial do *hardware*.

(conclusão)

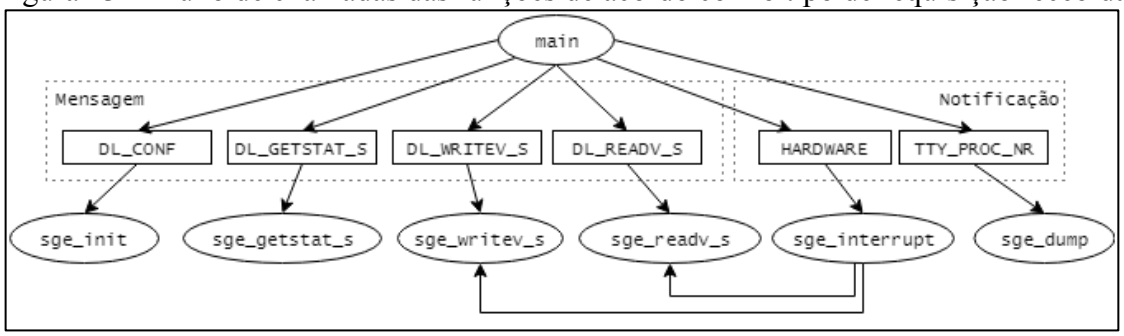
Função	Descrição
sge_init_addr	Define o endereço MAC da interface.
sge_init_buf	Aloca memória para os *buffers*.
sge_reset_hw	Redefine os registradores da interface com valores padrão.
sge_interrupt	Realiza o tratamento das interrupções.
sge_stop	Encerra a execução do *driver*.
sge_reg_read	Leitura de registrador.
sge_reg_write	Escrita em registrador.
sge_reg_set	Reescreve o valor lido do registrador.
sge_reg_unset	Limpa o valor do registrador.
read_eeprom	Leitura da EEPROM.
sge_mii_probe	Detecta o transceptor associado à interface.
sge_mii_read	Leitura de registrador do transceptor.
sge_mii_write	Escrita em registrador do transceptor.
sge_writev_s	Realiza a transmissão de um pacote.
sge_readv_s	Realiza a recepção de um pacote.
sge_getstat_s	Trata a solicitação do relatório de erros.
sge_default_phy	Configura o transceptor.
sge_reset_phy	Reinicia o transceptor.
sge_phymode	Obtém a velocidade e o modo de transmissão negociados pelo transceptor.
sge_macmode	Define a velocidade e o modo de transmissão na interface de rede.
reply	Prepara e envia respostas para o serviço de rede.
mess_reply	Função para resposta da solicitação de configuração.
sge_dump	Imprime em tela informações para depuração.
sef_local_startup	Registro no *System Event Framework*.
sef_cb_init_fresh	Registra o *driver* para uso no serviço de rede.
sef_cb_signal_handler	Tratamento do sinal de encerramento (SIGTERM)

Fonte: Próprio autor.

A função principal "`main`" (linhas 57 a 103) chama a função "`sef_local_startup`" para registrar o *driver* com o *System Event Framework*, e implementa o mecanismo de tratamento de mensagens utilizado no MINIX, executando a função para recebimento de mensagens do serviço de rede dentro de um laço de repetição. A função "`netdriver_receive`" é bloqueada quando não existem novas mensagens e retoma somente quando uma nova mensagem está pronta para recebimento. Desta forma, não há desperdício de recursos com espera ocupada. A função também retorna a classificação da mensagem (síncrona, no caso de mensagens, ou assíncrona, no caso de notificações). A macro "`is_ipc_notify`" verifica se a mensagem recebida é uma notificação de evento assíncrono, como interrupções de *hardware*. A origem da mensagem é determinada, e caso seja uma interrupção o tratador de

interrupções é executado. Então, o tipo de mensagem é identificado para decidir qual função será executada (Figura 13):

Figura 13 – Fluxo de chamadas das funções de acordo com o tipo de requisição recebida



Fonte: Próprio autor.

A função "`sef_local_startup`" (linhas 105 a 124) é durante a inicialização dentro da função "`main`" para associar ponteiros das funções implementadas aos pontos de chamada usados nos eventos do servidor de reencarnação, descritos na Tabela 12:

Tabela 12 – Funções de registro do *System Event Framework*

Função	**Descrição**
`sef_setcb_init_fresh`	Função chamada durante a inicialização limpa.
`sef_setcb_init_lu`	Função chamada na solicitação do *live update* sem interrupção do serviço.
`sef_setcb_init_restart`	Função chamada quando é solicitado o reinício do serviço.
`sef_setcb_lu_prepare`	Função chamada quando é necessário preparar para o *live update*.
`sef_setcb_lu_state_isvalid`	Função chamada para verificar se o estado é válido para o *live update*.
`sef_setcb_signal_handler`	Função chamada para tratar sinais POSIX.

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

Como o *driver* atualmente não suporta a funcionalidade *live update* do SEF, a função associada é a mesma para a inicialização limpa. Após o registro das funções, "`sef_startup()`" é chamado para confirmar as associações.

Em "`sef_cb_init_fresh`" (linhas 129 a 152) é utilizada a função "`fkey_map`" para registrar um conjunto de teclas para chamar a função para escrever informações de depuração em tela (`sge_dump`). O primeiro atributo (`fkeys`) é referente às teclas F1 a F12, e o segundo atributo (`sfkeys`) é referente às teclas Shift+F1 a Shift+F12. É registrado o conjunto Shift+F7, por ser um dos espaços não registrados pelo sistema: somente o atributo "`sfkeys`" é fornecido.

O número da instância (fornecido pelo sistema operacional em uma variável de ambiente) é obtido, a estrutura que armazenará as informações da interface de rede (`sge_state`) é inicializada totalmente com zeros para eliminar os valores desconhecidos que possam existir, e então a função "`netdriver_announce()`" é chamada para informar ao serviço de rede que o *driver* foi carregado.

A função "`sef_cb_signal_handler`" para tratamento de sinais (linhas 157 a 166) recebe como argumento um identificador de sinal (todos os sinais possíveis estão listados no arquivo "`/sys/sys/signal.h`"), e trata somente os sinais do tipo `SIGTERM` (sinal para término do processo), executando a função "`sge_stop`" para encerrar o *driver* corretamente.

A função "`sge_init`" (linhas 171 a 207) é chamada para tratar a mensagem de configuração (`DL_CONF`), recebendo-a por meio de um argumento do tipo "`message`". Esta função executa a rotina de inicialização do subsistema PCI, obtém as configurações de modo de recepção de pacotes (do único campo da mensagem: "`m_net_netdrv_dl_conf.mode`"), armazenando esta configuração na estrutura de controle do *driver*, e executa a inicialização do *hardware* com a função "`sge_init_hw`". Por fim, uma mensagem de resposta do tipo "`DL_CONF_REPLY`" (Figura 14) é preparada, e reporta ao serviço de rede se a inicialização do *hardware* ocorreu com sucesso ou não (campo `m_netdrv_net_dl_conf.stat`), e caso positivo também inclui o endereço físico da interface (campo `m_netdrv_net_dl_conf.hw_addr`).

Figura 14 – Mensagem de resposta de configuração (DL_CONF_REPLY)

```
typedef struct {
    int stat;
    uint8_t hwaddr[6];
    uint8_t padding[46];
} mess_netdrv_net_dl_conf;
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A função "`sge_init_pci`" (linhas 212 a 224) executa a função "`pci_init()`" para inicializar o subsistema PCI, permitindo o uso das funções relacionadas ao controle do barramento PCI no decorrer do programa, define um nome para a identificação da instância nos *logs* do sistema, que é armazenado na estrutura de controle do driver, e inicia a função para busca de dispositivos compatíveis.

A função "`sge_probe`" (linhas 229 a 308) recebe como parâmetro um ponteiro para a estrutura de controle do *driver* (a maioria das funções a seguir recebe este ponteiro) e um índice da instância que será controlada, e retorna se houve sucesso na busca.

A busca de dispositivos limita-se aos dispositivos com identificação descrita no arquivo de configuração "`sge.conf`". Se a busca encontra um dispositivo, seu índice é obtido para usar nos passos seguintes, e o estado do *driver* é alterado para "`SGE_DETECTED`". Caso nenhum dispositivo seja encontrado, a função é encerrada, indicando no retorno que não houve sucesso na busca. O índice é usado para reservar o dispositivo para uso, obter a configuração da linha de interrupção, obter o endereço base dos registradores de controle e habilitar o *bus mastering*. Dados sobre o dispositivo são armazenados na estrutura de controle para uso posterior. O endereço base dos registradores de controle é mapeado em memória usando chamada de sistema "`vm_map_phys`" (Figura 15), isso permite o acesso aos registradores como variáveis comuns a partir de seu ponteiro.

Figura 15 – Protótipo da função vm_map_phys

```
void *vm_map_phys(endpoint_t who, void *physaddr, size_t len);
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

O primeiro argumento identifica o processo para o qual o endereço deve ser mapeado ("`SELF`" representa o processo que chamou a função). O segundo argumento deve conter o endereço físico que será mapeado, e o terceiro argumento indica o tamanho do espaço que será mapeado.

A função "`sge_init_hw`" (linhas 313 a 395) é executada para associar a linha de interrupção da interface ao *driver* e ativar as interrupções utilizando as funções da biblioteca do sistema relacionadas ao tratamento de interrupções (Figura 16):

Figura 16 – Protótipo das funções relacionadas a interrupções

```
int sys_irqsetpolicy(int irq_vec, int policy, int *hook_id);
int sys_irqenable(int *hook_id);
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2016.

A função "`sys_irqsetpolicy`" indica ao sistema operacional que o *driver* está interessado em receber notificações sobre interrupções para a linha de interrupção especificada no primeiro argumento (`irq_vec`). O segundo argumento (`policy`) é usado para indicar se as interrupções serão habilitadas automaticamente após a notificação. O terceiro argumento (`hook_id`) é um ponteiro para o número da linha de interrupção atribuído ao dispositivo. Esse valor será alterado para um identificador que será usado na função para reativar as interrupções. A função "`sys_irqenable`" recebe como argumento somente o identificador recebido na

função anterior (`hook_id`) e reativa as interrupções, quando as interrupções não estão configuradas para habilitar automaticamente.

A função também coordena a rotina de inicialização chamando funções auxiliares: reiniciar o *hardware* para um estado conhecido, obter o endereço MAC, inicializar os *buffers*, encontrar e configurar o transceptor para aceitar pacotes e habilitar a transmissão/recepção de pacotes pela interface de rede. O modo de recepção de pacotes é configurado por meio de um filtro de pacotes, que permite a recepção apenas dos tipos de pacotes configurados (um ou mais de um tipo combinado), de acordo com o que foi solicitado pelo serviço de rede. A Tabela 13 mostra as possíveis configurações (constantes definidas em "`sge.h`").

Tabela 13 – Tipos de pacotes para o filtro

Tipo	**Descrição**
`SGE_RXCTRL_BCAST`	Pacotes broadcast.
`SGE_RXCTRL_ALLPHYS`	Qualquer outro pacote.
`SGE_RXCTRL_MCAST`	Pacotes multicast.
`SGE_RXCTRL_MYPHYS`	Pacotes endereçados à interface.

Fonte: Próprio autor.

A função "`sge_init_addr`" (linhas 400 a 451) é usada para configurar o endereço MAC, podendo este ser lido da variável de ambiente "`SGEETH#_EA`" (onde # é o número da instância atual). Caso a variável de ambiente não esteja definida, o endereço MAC é lido da EEPROM do dispositivo. O endereço é salvo em um campo do tipo "`ether_addr_t`" na estrutura de controle.

A função "`sge_init_buf`" (linhas 456 a 555) é usada para inicializar os *buffers*, utilizados nas transferências de dados. Todas as alocações de memória nesta seção são realizadas utilizando a função "`alloc_contig`" (Figura 17), que garante que a memória está fisicamente contígua, e retorna tanto o endereço virtual, que pode ser usado pelo *driver*, quanto o endereço físico, que pode ser usado pelo *hardware*.

Figura 17 – Protótipo da função alloc_contig

```
void *alloc_contig(size_t len, int flags, phys_bytes *phys);
```

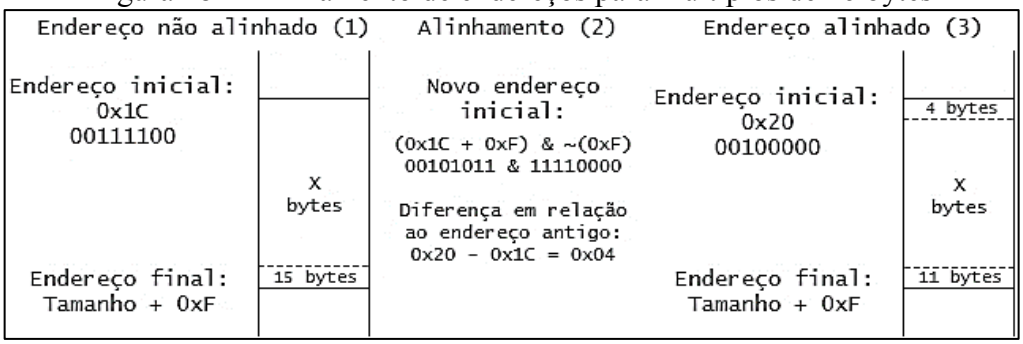
Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

O primeiro argumento desta função é o tamanho do espaço que será alocado (`len`), o segundo argumento (`flags`) define requisitos para a memória alocada, e o terceiro argumento

(`phys`) é um ponteiro para a variável que armazenará o endereço físico. O retorno da função é um ponteiro para o endereço virtual.

Os endereços que serão utilizados devem ser múltiplos de 16 *bytes*, como requerido pela interface de rede. Para garantir isto, é realizado o alinhamento dos endereços como no exemplo mostrado na Figura 18. Os espaços são alocados com 15 *bytes* além do necessário (1) e o alinhamento é realizado (2) somando 0xF (15, em decimal) ao endereço obtido e arredondado para baixo redefinindo os últimos 4 bits do endereço para 0. O novo endereço calculado (3) está alinhado e mantém o tamanho necessário utilizando parte do espaço adicional, caso necessário.

Figura 18 – Alinhamento de endereços para múltiplos de 16 bytes



Fonte: Próprio autor.

Depois de alocar todos os espaços de memória, o conteúdo destes é preenchido com zeros para que valores desconhecidos não causem problemas, e o espaço que armazenará os descritores é inicializado como um vetor da estrutura "`sge_desc_t`". Todos os endereços obtidos (virtuais e físicos) são gravados na estrutura de controle do *driver* para uso posterior.

O espaço dos descritores é inicializado. Como esse espaço é de uso compartilhado do *driver* e do dispositivo, é necessário evitar que ambos tentem escrever no mesmo descritor simultaneamente, ou ler dados que ainda estão sendo escritos. Para isso, um marcador no campo "`status`" é definido para identificar que o descritor está em uso, e ao final da operação o marcador é removido. Durante a primeira inicialização, todos os descritores para recepção de pacotes são marcados como propriedade do *hardware*, e todos os descritores para transmissão de pacotes não recebem marcação, sendo propriedade do *driver*. O último descritor contido no vetor é marcado no campo "`flags`" como final, para que o *hardware* retorne ao endereço inicial quando o alcançar e não tente escrever fora dos limites.

A função "`sge_reset_hw`" (linhas 560 a 611) redefine os valores dos registradores de controle para valores padrão, deixando a interface em um estado conhecido. Por falta de

documentação, não foi possível encontrar o motivo de cada valor definido, e esta função somente replica o comportamento observado no *driver* oficial.

Para realizar a transmissão de dados, a função "`sge_writev_s`" (linhas 616 a 699) é usada, recebendo como parâmetro uma indicação se a chamada foi causada por uma interrupção ou não. Se a função foi chamada devido a uma interrupção, esta é para sinalizar que a transmissão de um pacote finalizou e deve ser notificado ao serviço de rede, e o estado do *driver* é alterado para "`SGE_TRANSMIT`". Se a origem da chamada não for de uma interrupção, a função foi requisitada pelo serviço de rede para transmitir um pacote. A mensagem de requisição recebida pelo *driver* neste caso é uma estrutura do tipo "`mess_net_netdrv_dl_writev_s`" (Figura 19).

Figura 19 – Mensagem de requisição para transmissão (DL_WRITEV_S)

```
typedef struct {
    cp_grant_id_t grant;
    int count;
    uint8_t padding[48];
} mess_net_netdrv_dl_writev_s;
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A mensagem de requisição contém uma concessão para acessar o vetor de requisições de E/S fornecido pelo INET (campo `grant`) e a quantidade de concessões no vetor (campo `count`). A função para cópia segura de memória é usada para armazenar o vetor de requisições de E/S em uma variável local. Cada requisição de E/S no vetor é uma estrutura do tipo "`iovec_s_t`" (Figura 20), que armazena uma concessão de memória (campo `iov_grant`) e o tamanho do espaço concedido (campo `iov_size`).

Figura 20 – Estrutura de uma requisição de E/S

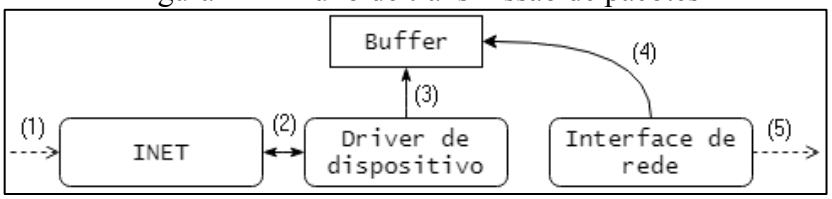
```
typedef struct {
    cp_grant_id_t iov_grant;
    vir_bytes iov_size;
} iovec_s_t;
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

Para escrever o conteúdo do pacote a ser transmitido no dispositivo, também é usada a função para cópia segura de memória, desta vez usando a concessão contida na requisição de E/S, transferindo dados do INET para o *buffer* de transmissão apontado por um descritor livre para uso. O descritor é marcado como utilizado para que o dispositivo envie o pacote. Neste

ponto, o estado é alterado para "SGE_WRITING". De forma simplificada, a Figura 21 mostra o fluxo dos dados para a transmissão de pacotes.

Figura 21 – Fluxo de transmissão de pacotes

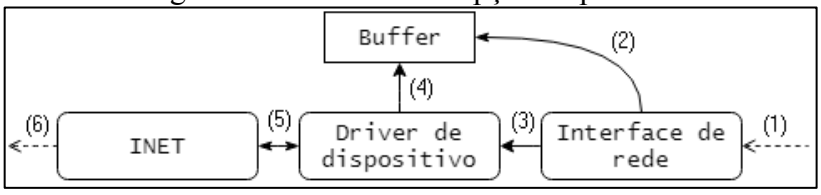


Fonte: Próprio autor.

O serviço de rede INET recebe do sistema uma solicitação para envio de dados na rede (1) e envia uma mensagem ao *driver* de dispositivo (2) para realizar a transmissão contendo a concessão para acesso à memória com o conteúdo a ser transmitido. O *driver* copia os dados para o *buffer* de pacotes (3) e configura um descritor de pacote que estiver livre para uso. A interface de rede detecta que existem pacotes pendentes de transmissão enquanto varre os descritores no *buffer* de transmissão e realiza a transmissão no meio físico (5), por fim liberando o *buffer* para uso posterior.

A função "sge_readv_s" (linhas 704 a 782) realiza a recepção de dados. Também recebe como parâmetro uma indicação se a chamada foi causada por uma interrupção ou não. A interrupção neste caso indica que existem novos pacotes recebidos para processamento pelo *driver*, e quando a função não é chamada pela interrupção, é o serviço de rede aguardando novos dados para receber, alterando o estado para "SGE_READING". Da mesma forma como ocorre na transmissão, o *driver* obtém as concessões de memória do serviço de rede e copia os dados para esta memória, libera o uso dos descritores lidos, e o estado é alterado para "SGE_RECEIVED". A Figura 22 mostra o fluxo para recepção de pacotes.

Figura 22 – Fluxo de recepção de pacotes



Fonte: Próprio autor.

A interface de rede recebe um novo pacote (1) e copia este para o *buffer* (2). Uma interrupção de *hardware* é gerada para avisar o driver que existem novos pacotes (3). O *driver* varre os descritores para encontrar os pacotes ainda não processados (4) e envia uma mensagem

para o INET contendo a concessão para acessar os dados (5). O INET recebe e dá destino aos dados (6).

A função "`sge_getstat_s`" (linhas 787 a 814) é chamada para retornar as estatísticas de uso e erros para o serviço de rede. Novamente, por falta de documentação não foi possível implementar esta funcionalidade, mas a presença desta função é obrigatória para o correto funcionamento do *driver*, pois o sistema aguarda resposta para a mensagem que solicita estas estatísticas. Uma estrutura do tipo "`eth_stat_t`" deve ser preenchida com as informações das estatísticas de operação e retornada ao serviço de rede. A Tabela 14 mostra os campos desta estrutura.

Tabela 14 – Campos da estrutura eth_stat_t

Campo	**Descrição**
`ets_recvErr`	Erros de recepção.
`ets_sendErr`	Erros de transmissão.
`ets_OVW`	*Buffer* sobrescrito (pacotes chegam mais rápido do que podem ser processados)
`ets_CRCerr`	Pacote corrompido, falhando a verificação da soma de verificação.
`ets_frameAll`	Quadro não alinhado.
`ets_missedP`	Pacote perdido.
`ets_packetR`	Quantidade de pacotes recebidos.
`ets_packetT`	Quantidade de pacotes transmitidos.
`ets_transDef`	Transmissão adiada (uma transmissão já está em andamento por outra estação).
`ets_collision`	Ocorrência de colisão durante a transmissão.
`ets_transAb`	Transmissão abortada devido a excesso de colisões.
`ets_carrSense`	Perda de conectividade com a mídia.
`ets_fifoUnder`	Condição de *buffer underrun*: o serviço de rede processa pacotes mais rápido que a interface.
`ets_fifoOver`	Condição de *buffer overrun*: a interface processa pacotes mais rápido que o serviço de rede.
`ets_CDheartbeat`	Impossibilidade de transmitir sinal de colisão.
`ets_OWC`	Fora da janela de colisão (não conseguiu adquirir a mídia para transmissão).

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

A função "`sge_interrupt`" (linhas 819 a 860) é a função que trata as interrupções de *hardware*. Ao receber uma interrupção, identifica o motivo e desativa as interrupções enquanto realiza o tratamento desta interrupção. Na implementação são tratados quatro tipos de interrupção (Tabela 15):

Tabela 15 – Interrupções tratadas pelo *driver*

Interrupção	**Descrição**
SGE_INTR_TX_DONE	Transmissão concluída
SGE_INTR_TX_IDLE	Transmissor ocioso
SGE_INTR_RX_DONE	Recepção concluída
SGE_INTR_RX_IDLE	Receptor ocioso
SGE_INTR_LINK	Alteração do enlace

Fonte: Próprio autor.

As interrupções "SGE_INTR_TX_DONE" e "SGE_INTR_TX_IDLE" chamam a função "sge_writev_s" para relatar ao serviço de rede sobre o fim da operação ou iniciar uma nova transmissão. As interrupções "SGE_INTR_RX_DONE" e "SGE_INTR_RX_IDLE" chamam a função "sge_readv_s" para processar os pacotes recebidos e enviá-los para o serviço de rede. A interrupção "SGE_INTR_LINK" indica que houve uma alteração no enlace (desconexão ou conexão). Não foi implementada função para tratar esta interrupção, pois o serviço de rede atual (INET) não trata o status do enlace, mas esta funcionalidade estará presente no novo serviço de rede. Depois de executar as funções correspondentes às interrupções que tratou, o *driver* reativa as interrupções.

A função "sge_stop" (linhas 862 a 882) é executada quando é solicitado o encerramento do *driver*. A interface é reiniciada e as interrupções são desativadas, e então a execução do processo é encerrada "graciosamente" com a função "exit()".

Um conjunto de funções para leitura e gravação de registradores de controle da interface de rede, da memória EEPROM e dos registradores de controle do transceptor encontra-se nas linhas 887 a 1025. A sintaxe de cada função é similar, mudando apenas o local onde ocorre a operação com uma pequena alteração nas funções para leitura/escrita dos registradores no transceptor, o qual deve ser identificado pelo seu índice como um parâmetro adicional. Os protótipos destas funções são apresentados na Figura 23:

Figura 23 – Protótipo das funções de leitura e gravação em registradores

```
uint32_t sge_reg_read(sge_t *e, uint32_t reg);
void sge_reg_write(sge_t *e, uint32_t reg, uint32_t value);
void sge_reg_set(sge_t *e, uint32_t reg, uint32_t value);
void sge_reg_unset(sge_t *e, uint32_t reg, uint32_t value);
uint16_t read_eeprom(sge_t *e, int reg);
uint16_t sge_mii_read(sge_t *e, uint32_t phy, uint32_t reg);
void sge_mii_write(sge_t *e, uint32_t phy, uint32_t reg,
    uint32_t data);
```

Fonte: Próprio autor.

Todas as funções recebem como primeiro parâmetro o ponteiro para a estrutura de controle do *driver*. As funções "`sge_reg_read`" e "`sge_reg_write`" são usadas para leitura e gravação nos registradores de controle da interface de rede, recebendo em seu segundo argumento (`reg`) o deslocamento do registrador (armazenado nas constantes definidas no arquivo "sge.h") em relação ao endereço base dos registradores, e no caso de gravação o valor a ser gravado no terceiro argumento (`value`). A função "`sge_reg_set`" lê o valor do registrador e grava novamente este mesmo valor, e a função "`sge_reg_unset`" remove o conteúdo do registrador usando uma regra de álgebra booleana, onde qualquer valor e seu inverso é sempre falso (zero). As funções "`sge_mii_read`" e "`sge_mii_write`" realizam leitura e gravação dos registradores do transceptor, recebe no seu segundo parâmetro (`phy`) o endereço do transceptor que foi detectado durante a configuração, o deslocamento do registrador no terceiro parâmetro (e o valor a ser gravado no quarto parâmetro, no caso de gravação). A função "`read_eeprom`" é usada para leitura dos dados na memória EEPROM embutida, e recebe no seu segundo parâmetro o deslocamento em relação ao início do armazenamento. A Figura 24 exemplifica o uso da função para leitura do registrador de controle para obter o conteúdo do registrador de controle "`SGE_REG_RX_CTL`":

Figura 24 – Operação de leitura de registrador

```
uint32_t valor;
valor = sge_reg_read(e, SGE_REG_RX_CTL);
```

Fonte: Próprio autor.

A função "`sge_mii_probe`" (linhas 1030 a 1123) é executada durante a configuração da interface para detectar o transceptor e seu endereço, e chamar a função para configuração do transceptor. A função "`sge_default_phy`" (linhas 1129 a 1172) define o transceptor padrão, caso mais de um seja detectado. A função "`sge_reset_phy`" (linhas 1177 a 1191) é usada para reiniciar o transceptor.

A função "`sge_phymode`" (linhas 1196 a 1246), chamada durante a detecção do transceptor, é usada para obter informações sobre a conexão negociada pelo transceptor para o enlace atual, como velocidade e modo de comunicação. As informações são guardadas na estrutura de controle. A função "`sge_macmode`" (linhas 1251 a 1285) é usada para definir as informações sobre o enlace, obtidas na função anterior, nos registradores de controle da interface de rede.

Todas as funções que precisam responder mensagens do serviço de rede usam a função "`reply`" (linhas 1290 a 1331) para compor a resposta como uma mensagem do tipo

"DL_TASK_REPLY", representada pela estrutura do tipo "mess_netdrv_net_dl_task" (Figura 25):

Figura 25 – Estrutura da mensagem do tipo DL_TASK_REPLY

```
typedef struct {
    int count;
    uint32_t flags;
    uint8_t padding[48];
} mess_netdrv_net_dl_task;
```

Fonte: STICHTING MINIX RESEARCH FOUNDATION, 2014.

No caso de leitura de pacote, é informado que se trata de um recebimento (a mensagem recebe o marcador "DL_PACK_RECV" no campo "flags"), e o tamanho do pacote também é informado neste momento (no campo "count"). No caso de escrita de pacote, é só reconhecido que o pedido de transmissão foi tratado (a mensagem recebe o marcador "DL_PACK_SEND"), e o estado do *driver* retorna para "SGE_ENABLED". A função "mess_reply" (linhas 1336 a 1344) é usada para enviar as mensagens de resposta no passo de inicialização do *driver*.

Por fim, a função "sge_dump" (linhas 1349 a 1443) é chamada quando o usuário pressiona a combinação de teclas que foi associada ao driver (no caso, Shift+F7). Esta função imprime em tela informações úteis para depuração, como o modelo da interface detectada, endereço MAC, velocidade e modo de comunicação no enlace, marca e modelo do transceptor, conteúdo dos registradores de controle, conteúdo da EEPROM, conteúdo dos registradores do transceptor, e conteúdo dos últimos descritores de transmissão e recepção utilizados.

3.4 Instalação

Para realizar a instalação do *driver* em um computador que possui a interface de rede suportada é necessário primeiro obter o código-fonte do MINIX a partir do repositório oficial, contendo todos os arquivos que foram referenciados no código desenvolvido. Então, os arquivos para o novo *driver* devem ser copiados para o diretório que contém os *drivers* de rede ("/minix/drivers/net/"). Até o momento, o MINIX não oferece suporte a dispositivos USB na arquitetura x86, então para copiar os arquivos de código-fonte para um computador sem acesso à rede é possível gravar discos no formato ISO-9660 em outro computador, e usar o programa "isoread" para ler estes discos no MINIX. Instruções para uso deste programa podem ser consultadas com o comando "man isoread". Devido a limitações no formato do disco de dados (quantidade máxima de diretórios e nomeação dos arquivos) pode ser

conveniente utilizar ferramentas para compactar os vários arquivos em um único arquivo, tais como TAR e GZIP, e descompactá-los na máquina de destino. Quando todos os arquivos e programas necessários estiverem presentes, é possível iniciar a compilação com o comando "`make`" dentro do diretório contendo o código do *driver*, que processará o arquivo "`Makefile`" para realizar a compilação. Caso o processo termine com êxito, o comando "`make install`" deve ser usado para instalar o *driver* compilado no sistema. Após a instalação, o programa "`netconf`" deve ser executado para reconfigurar o serviço de rede, agora permitindo a utilização do novo *driver* instalado. A Figura 26 mostra a realização da configuração da interface de rede que será utilizada no sistema, por meio do programa "`netconf`".

Após a configuração, em cada inicialização do sistema o *driver* é carregado em memória automaticamente. É possível verificar a conectividade da rede executando ferramentas do sistema como "`ping`" ou "`traceroute`", a exemplo do teste demonstrado na Figura 27 utilizando o programa "`ping`" para verificar se um servidor (minix3.org) pode ser alcançado na rede. Outros programas podem ser instalados usando o gerenciador de pacotes "`pkgin`", como o navegador web Links, mostrado na Figura 28.

Figura 26 – Configurando a interface de rede para uso com o netconf

```
# netconf
MINIX 3 currently supports the following Ethernet cards. PCI cards detected
by MINIX are marked with *. Please choose:

 0.   No Ethernet card (no networking)
 1.   3Com 501 or 3Com 509 based card
 2.   Realtek 8029 based card (also emulated by Qemu)
 3.   NE2000, 3com 503 or WD based card (also emulated by Bochs)
 4.   lan8710a (on BeagleBone, BeagleBone Black)
 5.   Attansic/Atheros L2 FastEthernet
 6.   DEC Tulip 21140A in VirtualPC
 7.   Intel PRO/1000 Gigabit
 8.   Intel PRO/100
 9.   AMD LANCE (also emulated by VMWare and VirtualBox)
10.   Realtek 8139 based card
11.   Realtek 8169 based card
12. * SiS 190/191 Fast Ethernet Controller
13.   Virtio network device
14.   Different Ethernet card (no networking)

Ethernet card? [12] _
```

Fonte: Próprio autor.

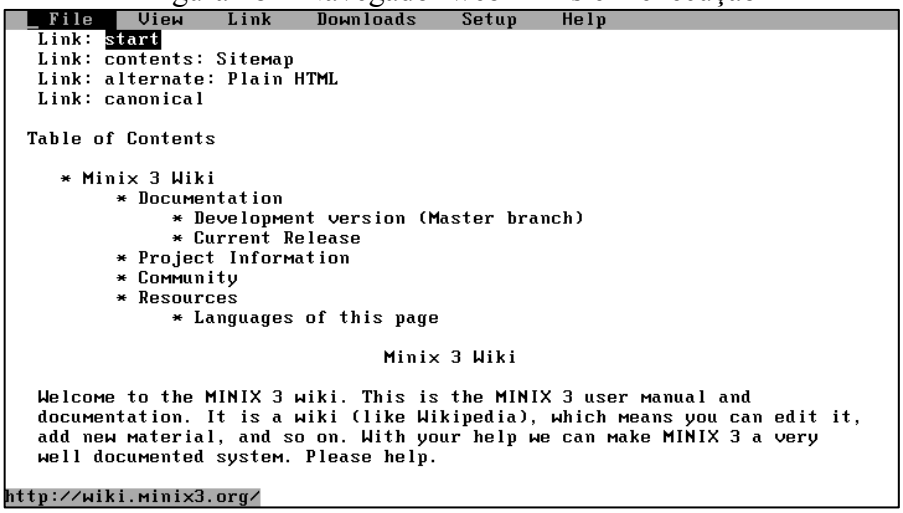
Figura 27 – Teste de conectividade com o programa ping

```
# ping -c 10 minix3.org
PING minix3.org (66.147.238.215): 56 data bytes
64 bytes from 66.147.238.215: icmp_seq=0 ttl=52 time=183.333333 ms
64 bytes from 66.147.238.215: icmp_seq=1 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=2 ttl=52 time=200.000000 ms
64 bytes from 66.147.238.215: icmp_seq=3 ttl=52 time=200.000000 ms
64 bytes from 66.147.238.215: icmp_seq=4 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=5 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=6 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=7 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=8 ttl=52 time=166.666667 ms
64 bytes from 66.147.238.215: icmp_seq=9 ttl=52 time=166.666667 ms

----minix3.org PING Statistics----
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 166.666667/175.000000/200.000000/14.163943 ms
```

Fonte: Próprio autor.

Figura 28 – Navegador web Links em execução



Fonte: Próprio autor.

3.5 Testes iniciais

Os testes iniciais de desempenho foram realizados com metodologia adaptada de Linnenbank (2009). Para realização destes, foram utilizados dois computadores (com suas especificações descritas na Tabela 16) diretamente conectados por meio de um cabo crossover em um enlace de 100 Mbps. O desempenho obtido nos testes mostra que o *driver* é satisfatório para uso, exceto no caso de uploads UDP de alta velocidade.

Tabela 16 – Especificação de computadores utilizados nos testes

Computador	Computador 1	Computador 2
Processador	Intel Pentium T4300 (2.1 GHz)	AMD A10-7860K (3.6 GHz)
Memória	4GB	8GB
Interface de rede	SiS191 (100Mbps)	Realtek 8111G (1000Mbps)

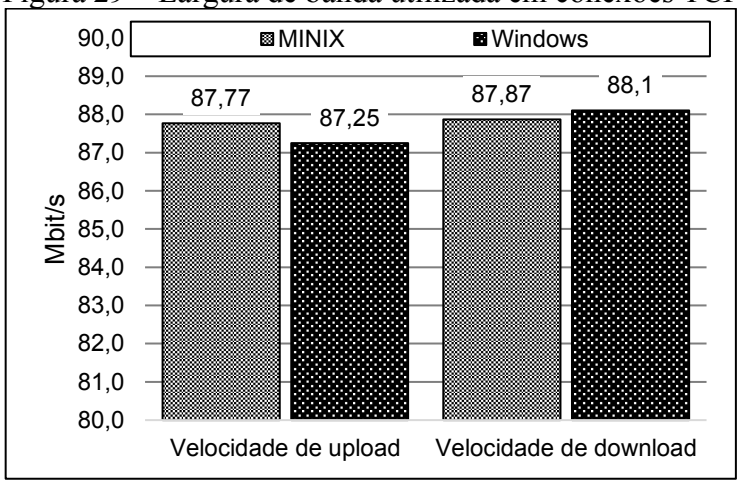
Fonte: Próprio autor.

O primeiro teste consiste em verificar a largura de banda máxima efetivamente utilizada durante as transferências de dados utilizando o programa "`iperf`". Este programa não é incluído no sistema por padrão, porém está disponível para instalação utilizando o gerenciador de pacotes "`pkgin`" ou a partir do site dos autores do programa[2]. As medições foram realizadas utilizando conexões TCP e UDP. Para efeitos de comparação, os mesmos testes são executados no sistema operacional Microsoft Windows 7, utilizando o *driver* fornecido pelo fabricante.

[2] Consultar "iPerf - The ultimate speed test tool for TCP, UDP and SCTP" em https://iperf.fr/

A Figura 29 mostra que o *driver* desenvolvido para o MINIX consegue alcançar velocidades equivalentes ao *driver* do fabricante no Microsoft Windows, ocorrendo somente diferenças pequenas entre os sistemas operacionais testados (menos de 1 Mbps de diferença).

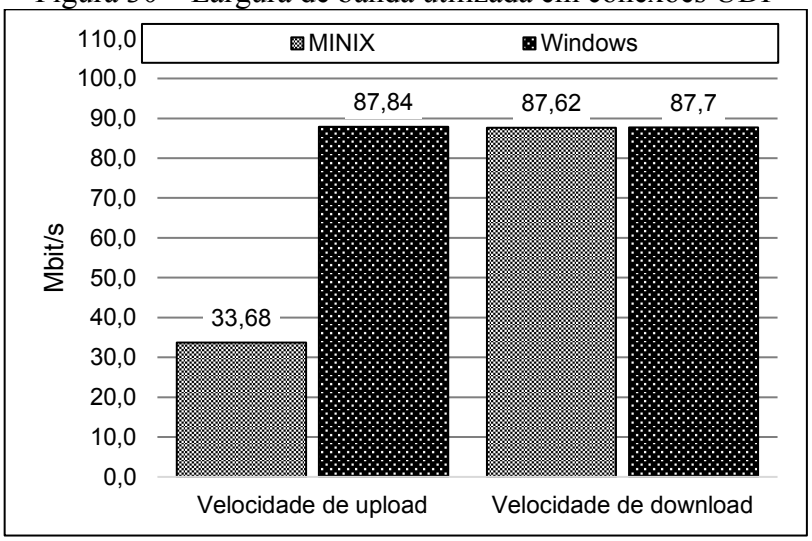
Figura 29 – Largura de banda utilizada em conexões TCP



Fonte: Próprio autor.

A Figura 30 mostra que a velocidade de *download* em conexões UDP se mantém equivalente ao *driver* para Microsoft Windows. No entanto, uma diferença considerável na velocidade de *upload* foi verificada no MINIX, que alcançou o uso de somente 33 Mbit/s. O mesmo teste foi realizado com o *driver* "lance" (para a interface de rede AMD PCNET emulada no VMware), e este comportamento se repetiu. A razão para esta diferença é desconhecida, mas pode estar relacionada ao funcionamento do INET ou de suas dependências (Linnenbank, 2009).

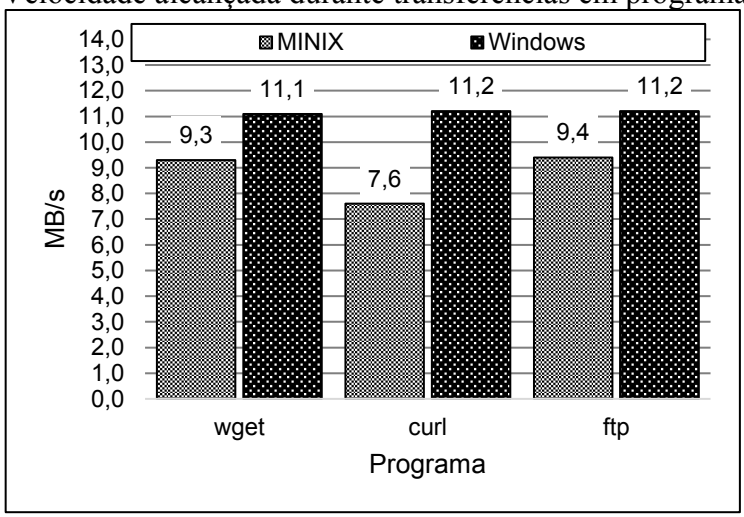
Figura 30 – Largura de banda utilizada em conexões UDP



Fonte: Próprio autor.

O segundo teste consiste em verificar a velocidade alcançada durante as transferências de dados utilizando os programas de usuário "`curl`", "`wget`" e "`ftp`", e os resultados são mostrados na Figura 31.

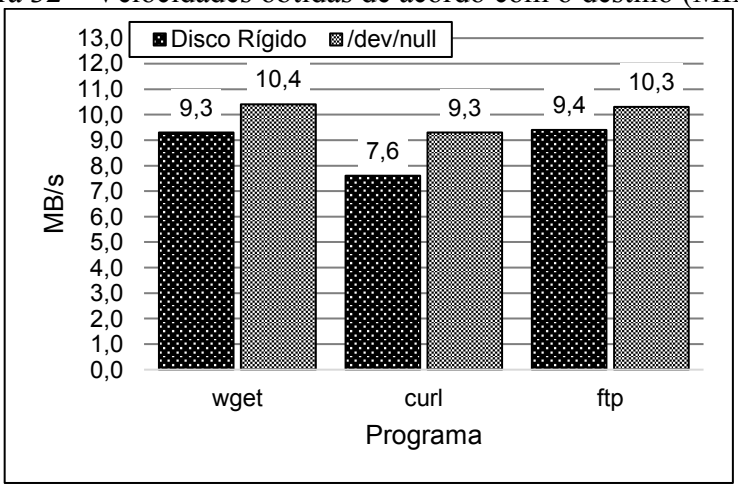
Figura 31 – Velocidade alcançada durante transferências em programas de usuário



Fonte: Próprio autor.

Neste teste, o Computador 2 foi configurado como um servidor FTP (*File Transfer Protocol*) usando o programa "FileZilla Server", compartilhando um arquivo de 500MB com conteúdo aleatório. A Figura 32 mostra as velocidades obtidas de acordo com o local escolhido para armazenar os dados recebidos: um arquivo no disco rígido ou redirecionar para "/dev/null" (portanto, descartando os dados). Percebe-se que há uma diferença de velocidade entre cada programa testado no MINIX, e observou-se também durante os testes que gravar o arquivo em disco influencia na velocidade alcançada pelos programas.

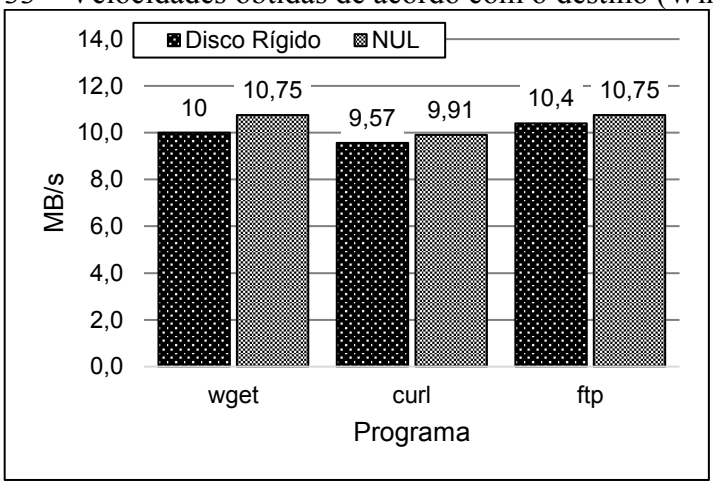
Figura 32 – Velocidades obtidas de acordo com o destino (MINIX)



Fonte: Próprio autor.

O mesmo teste foi realizado no Windows com o *driver* do fabricante, e o desempenho obtido é mostrado na Figura 33. Para descartar os dados no Microsoft Windows, a saída foi redirecionada para “NUL”, que é equivalente ao “/dev/null”. Diferente do observado no MINIX, houve pouca diferença entre os programas testados.

Figura 33 – Velocidades obtidas de acordo com o destino (Windows)



Fonte: Próprio autor.

4 CONCLUSÃO

A realização deste trabalho permitiu estudar na prática o processo de desenvolvimento de um *driver* de dispositivo para um sistema operacional real. Dentre as dificuldades encontradas durante o desenvolvimento, destacam-se:

- A quantidade reduzida de informações disponíveis sobre o *hardware* que foi utilizado, não permitindo que algumas funções pudessem ser implementadas, como o relatório de estatísticas da interface (que depende de mais detalhes sobre o status dos pacotes).
- A inconsistência da documentação disponível na *wiki* do MINIX 3 fez necessário o estudo do código-fonte para algumas funções ou então a consulta à lista de discussão por explicações adicionais.
- A máquina utilizada para desenvolvimento não permitiu o uso de recursos de depuração como o GDB (GNU *Project Debugger*), comunicação por porta serial ou virtualização de *hardware*, diminuindo a eficiência na detecção e correção de erros.

Apesar destas dificuldades, acredita-se que o objetivo proposto foi alcançado, resultando no desenvolvimento de um *driver* funcional com documentação de apoio para entendimento das funcionalidades e do processo de desenvolvimento. Espera-se, portanto, que este trabalho possa contribuir com a comunidade do MINIX.

4.1 Trabalhos futuros

Durante o desenvolvimento deste trabalho, foi anunciado a substituição do serviço de rede atual (INET) pelo novo serviço de rede (LWIP) que será usado na próxima versão do MINIX. Como ainda não ocorreu o lançamento da versão estável do MINIX 3.4, manteve-se o desenvolvimento para a última versão estável até o momento (versão 3.3), que ainda utiliza o serviço de rede INET. Quando ocorrer o lançamento da próxima versão estável, pode-se atualizar o estudo para usar o novo serviço de rede.

Para obter informações mais precisas sobre a causa do baixo desempenho no tráfego UDP será necessário implementar a funcionalidade de relatório de estatísticas. Os testes de desempenho então poderiam ser relacionados aos resultados reportados pelo próprio *driver*.

Como esse trabalho teve seu escopo limitado ao desenvolvimento de *drivers* para interfaces de rede, pode-se futuramente estender o estudo para abranger outras classes de dispositivos.

REFERÊNCIAS

BEACH, A. **3Com 3c905C Network Interface Card Driver for the Minix-vmd Operating System**. Fourth-Year Project (Bachelor of Computer Science) – Department of Mathematics & Computer Science, Laurentian University. Ontario, Canadá, 2003. Disponível em: <http://web.archive.org/web/20060820133621/http://users.eastlink.ca:80/~alvinbeach/portfolio/3c920/3c905C_report-1.1.pdf>. Acesso em: 31 maio 2017.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers**. 3ª Ed. Califórnia, EUA: O'Reilly, 2005. ISBN 978-0-596-00590-0.

CORT, T. Device Driver Development Demystified. **MINIXCon 2016**. Amsterdã, Holanda, fev. 2016. Disponível em: <http://www.minix3.org/conference/2016/slides/Cort.pdf>. Acesso em: 04 jun. 2017.

HAIKU INC. **Haiku Project – Haiku's main repository**. New York, Estados Unidos, 2017. Disponível em: <http://cgit.haiku-os.org/haiku/>. Acesso em: 27 maio 2017.

HERDER, J. N.; BOS, H.; GRAS, B.; HOMBURG, P. TANENBAUM, A. S. Fault Isolation for Device Drivers. **2009 IEEE/IFIP International Conference on Dependable Systems and Networks**. Lisboa, Portugal, p. 33-42, jun. 2009. Disponível em: <http://dx.doi.org/10.1109/DSN.2009.5270357>. Acesso em: 02 jul. 2017.

LINNENBANK, N. K. **Implementing the Intel Pro/1000 on MINIX 3**. Amsterdã, Holanda, 2009. Disponível em: <http://www.minix3.org/theses/linnenbank-ipa.pdf>. Acesso em: 03 maio 2017.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª Ed. Rio de Janeiro: LTC, 2007. ISBN 978-85-216-1548-4.

PCI DATABASE. **PCIDatabase.com – PCI Vendor and Devices Lists**. [S.l.], s.d. Disponível em: <http://pcidatabase.com/>. Acesso em: 15 ago. 2017.

SILICON INTEGRATED SYSTEMS CORP. **SiS968 Overview**. Taiwan, 2007. Disponível em: <http://web.archive.org/web/20071021022734/http://www.sis.com/products/sis968.htm>. Acesso em: 28 maio 2017.

STICHTING MINIX RESEARCH FOUNDATION. **MINIX Source Code**. R3.3.0. Amsterdã, Holanda, 2014. Disponível em: <http://git.minix3.org>. Acesso em: 04 jun. 2017.

STICHTING MINIX RESEARCH FOUNDATION. **MINIX 3 Kernel API**. Amsterdã, Holanda, 2016. Disponível em: <http://wiki.minix3.org/doku.php?id=developersguide:kernelapi>. Acesso em: 18 jun. 2017.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 3ª Ed. São Paulo: Pearson Prentice Hall, 2009. ISBN 978-85-7605-237-1.

TANENBAUM, A. S. Lessons Learned from 30 Years of MINIX. **Communications of the ACM**, New York, EUA, v. 59, n. 3, p. 70-78, mar. 2016. Disponível em: <http://doi.acm.org/10.1145/2795228>. Acesso em: 31 maio 2017.

TANENBAUM, A. S.; WOODHULL, A. S. **Sistemas Operacionais**: Projeto e Implementação. 3ª Ed. Porto Alegre: Bookman, 2008. ISBN 978-85-7780-057-5.

APÊNDICE A – CÓDIGO FONTE DO DRIVER SGE

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                                Makefile                                  X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
01   # Makefile for the SiS 190/191 Fast Ethernet Controller driver.
02   PROG= sge
03   SRCS= sge.c
04
05   FILES=$(PROG).conf
06   FILESNAME=$(PROG)
07   FILESDIR= /etc/system.conf.d
08
09   DPADD+=    ${LIBNETDRIVER} ${LIBSYS}
10   LDADD+=    -lnetdriver –lsys
11
12   .include <minix.service.mk>

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                                sge.conf                                  X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
01   service sge
02   {
03        type net;
04        descr "SiS 190/191 Ethernet Controller";
05        system
06             UMAP       # 14
07             IRQCTL     # 19
08             DEVIO      # 21
09        ;
10        pci device      1039:0191;
11        pci device      1039:0190;
12        ipc
13             SYSTEM pm rs tty ds vm
14             pci inet lwip
15        ;
16   };

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                                 sge.h                                    X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
001  /* sge.h
002   *
003   * SiS 190/191 Ethernet Controller driver
004   *
005   * Parts of this code are based on the FreeBSD implementation
006   * (https://svnweb.freebsd.org/base/head/sys/dev/sge/), the
007   * e1000 driver by Niek Linnenbank, and the official
```

```
008   * SiS 190/191 GNU/Linux driver by K.M. Liu.
009   *
010   * Created: May 2017 by Marcelo Alencar <marceloalves@ufpa.br>
011   */
012
013  #ifndef _SGE_H_
014  #define _SGE_H_
015
016  #include <net/gen/ether.h>
017  #include <net/gen/eth_io.h>
018
019  /* MAC Override */
020  #define SGE_ENVVAR         "SGEETH"
021
022  /* Device IDs */
023  #define SGE_DEV_0190  0x0190 /* SiS190 */
024  #define SGE_DEV_0191  0x0191 /* SiS191 */
025
026  /* Ethernet driver statuses */
027  #define SGE_DETECTED       (1 << 0)
028  #define SGE_ENABLED        (1 << 1)
029  #define SGE_READING        (1 << 2)
030  #define SGE_WRITING        (1 << 3)
031  #define SGE_RECEIVED       (1 << 4)
032  #define SGE_TRANSMIT       (1 << 5)
033
034  /* Ethernet driver modes */
035  #define SGE_PROMISC        (1 << 0)
036  #define SGE_MULTICAST      (1 << 1)
037  #define SGE_BROADCAST      (1 << 2)
038
039  /* Speed/Duplex */
040  #define SGE_SPEED_10       10
041  #define SGE_SPEED_100      100
042  #define SGE_SPEED_1000     1000
043  #define SGE_DUPLEX_ON      1
044  #define SGE_DUPLEX_OFF     0
045
046  /* Buffer info */
047  #define SGE_IOVEC_NR       16
048  #define SGE_BUF_SIZE       2048
049  #define SGE_RXDESC_NR      32
050  #define SGE_TXDESC_NR      32
051  #define SGE_RXB_TOTALSIZE  SGE_RXDESC_NR*SGE_BUF_SIZE
052  #define SGE_TXB_TOTALSIZE  SGE_TXDESC_NR*SGE_BUF_SIZE
053  #define SGE_RXD_TOTALSIZE  SGE_RXDESC_NR*sizeof(sge_desc_t)
054  #define SGE_TXD_TOTALSIZE  SGE_TXDESC_NR*sizeof(sge_desc_t)
055  #define SGE_DESC_FINAL     0x80000000
056
```

```
057  /* Register Addresses */
058  #define    SGE_REG_TX_CTL            0x00
059  #define    SGE_REG_TX_DESC           0x04
060  #define    SGE_REG_RESERVED0         0x08
061  #define    SGE_REG_TX_NEXT           0x0c
062
063  #define    SGE_REG_RX_CTL            0x10
064  #define    SGE_REG_RX_DESC           0x14
065  #define    SGE_REG_RESERVED1         0x18
066  #define    SGE_REG_RX_NEXT           0x1c
067
068  #define    SGE_REG_INTRSTATUS        0x20
069  #define    SGE_REG_INTRMASK          0x24
070  #define    SGE_REG_INTRCONTROL       0x28
071  #define    SGE_REG_INTRTIMER         0x2c
072
073  #define    SGE_REG_PMCONTROL         0x30
074  #define    SGE_REG_RESERVED2         0x34
075  #define    SGE_REG_EEPROMCONTROL     0x38
076  #define    SGE_REG_EEPROMINTERFACE   0x3c
077  #define    SGE_REG_STATIONCONTROL    0x40
078  #define    SGE_REG_GMIICONTROL       0x44
079  #define    SGE_REG_GMACIOCR          0x48
080  #define    SGE_REG_GMACIOCTL         0x4c
081  #define    SGE_REG_TXMACCONTROL      0x50
082  #define    SGE_REG_TXMACTIMELIMIT    0x54
083  #define    SGE_REG_RGMIIDELAY        0x58
084  #define    SGE_REG_RESERVED3         0x5c
085  #define    SGE_REG_RXMACCONTROL      0x60
086  #define    SGE_REG_RXMACADDR         0x62
087  #define    SGE_REG_RXHASHTABLE       0x68
088  #define    SGE_REG_RXHASHTABLE2      0x6c
089  #define    SGE_REG_RXWAKEONLAN       0x70
090  #define    SGE_REG_RXWAKEONLANDATA   0x74
091  #define    SGE_REG_RXMPSCONTROL      0x78
092  #define    SGE_REG_RESERVED4         0x7c
093
094  /* Registers Interface */
095  #define    SGE_REGSC_FDX             0x00001000
096  #define    SGE_REGSC_SPEED_MASK      0x00000c00
097  #define    SGE_REGSC_SPEED_10        0x00000400
098  #define    SGE_REGSC_SPEED_100       0x00000800
099  #define    SGE_REGSC_SPEED_1000      0x00000c00
100
101  /* RX mode */
102  #define    SGE_RXCTRL_BCAST          0x0800
103  #define    SGE_RXCTRL_MCAST          0x0400
104  #define    SGE_RXCTRL_MYPHYS         0x0200
105  #define    SGE_RXCTRL_ALLPHYS        0x0100
```

```
106
107  /* TX descriptor command/status */
108  #define SGE_TXSTATUS_TXOWN      0x80000000
109  #define SGE_TXSTATUS_TXINT      0x40000000
110  #define SGE_TXSTATUS_THOL3      0x30000000
111  #define SGE_TXSTATUS_BSTEN      0x00800000
112  #define SGE_TXSTATUS_EXTEN      0x00400000
113  #define SGE_TXSTATUS_DEFEN      0x00200000
114  #define SGE_TXSTATUS_BKFEN      0x00100000
115  #define SGE_TXSTATUS_CRSEN      0x00080000
116  #define SGE_TXSTATUS_COLSEN     0x00040000
117  #define SGE_TXSTATUS_CRCEN      0x00020000
118  #define SGE_TXSTATUS_PADEN      0x00010000
119
120  /* RX descriptor status */
121  #define SGE_RXSTATUS_CRCOK      0x00010000
122  #define SGE_RXSTATUS_COLON      0x00020000
123  #define SGE_RXSTATUS_NIBON      0x00040000
124  #define SGE_RXSTATUS_OVRUN      0x00080000
125  #define SGE_RXSTATUS_MIIER      0x00100000
126  #define SGE_RXSTATUS_LIMIT      0x00200000
127  #define SGE_RXSTATUS_SHORT      0x00400000
128  #define SGE_RXSTATUS_ABORT      0x00800000
129  #define SGE_RXSTATUS_RXINT      0x40000000
130  #define SGE_RXSTATUS_RXOWN      0x80000000
131
132  /* Interrupts */
133  #define    SGE_INTR_SOFT        0x40000000
134  #define    SGE_INTR_TIMER       0x20000000
135  #define    SGE_INTR_LINK        0x00010000
136  #define    SGE_INTR_RX_IDLE     0x00000080
137  #define    SGE_INTR_RX_DONE     0x00000040
138  #define    SGE_INTR_TXQ1_IDLE   0x00000020
139  #define    SGE_INTR_TXQ1_DONE   0x00000010
140  #define    SGE_INTR_TX_IDLE     0x00000008
141  #define    SGE_INTR_TX_DONE     0x00000004
142  #define    SGE_INTR_RX_HALT     0x00000002
143  #define    SGE_INTR_TX_HALT     0x00000001
144  #define    SGE_INTRS \
145   (SGE_INTR_RX_IDLE | SGE_INTR_RX_DONE | SGE_INTR_TXQ1_IDLE | \
146    SGE_INTR_TXQ1_DONE |SGE_INTR_TX_IDLE | SGE_INTR_TX_DONE | \
147    SGE_INTR_TX_HALT | SGE_INTR_RX_HALT)
148
149  /* EEPROM Addresses */
150  #define    SGE_EEPADDR_SIG      0x00 /* Signature */
151  #define    SGE_EEPADDR_CLK      0x01 /* Clock */
152  #define    SGE_EEPADDR_INFO     0x02 /* Info */
153  #define    SGE_EEPADDR_MAC      0x03 /* MAC Address */
154
```

```
155  /* EEPROM Interface */
156  #define SGE_EEPROM_DATA              0xffff0000
157  #define SGE_EEPROM_DATA_SHIFT        16
158  #define SGE_EEPROM_OFFSET_SHIFT      10
159  #define SGE_EEPROM_READ              0x00000200
160  #define SGE_EEPROM_REQ               0x00000080
161
162  /* MII Addresses */
163  #define SGE_MIIADDR_CONTROL          0x00
164  #define SGE_MIIADDR_STATUS           0x01
165  #define SGE_MIIADDR_PHY_ID0          0x02
166  #define SGE_MIIADDR_PHY_ID1          0x03
167  #define SGE_MIIADDR_AUTO_ADV         0x04
168  #define SGE_MIIADDR_AUTO_LPAR        0x05
169  #define SGE_MIIADDR_AUTO_EXT         0x06
170  #define SGE_MIIADDR_AUTO_GADV        0x09
171  #define SGE_MIIADDR_AUTO_GLPAR       0x0a
172
173  /* MII Interface */
174  #define SGE_MIISTATUS_LINK           0x0004
175  #define SGE_MIISTATUS_AUTO_DONE      0x0020
176  #define SGE_MIISTATUS_CAN_TX         0x2000
177  #define SGE_MIISTATUS_CAN_TX_FDX     0x4000
178
179  #define SGE_MIICTRL_RST_AUTO         0x0200
180  #define SGE_MIICTRL_ISOLATE          0x0400
181  #define SGE_MIICTRL_AUTO             0x1000
182  #define SGE_MIICTRL_RESET            0x8000
183
184  #define SGE_MII_DATA                 0xffff0000
185  #define SGE_MII_DATA_SHIFT           16
186  #define SGE_MII_REQ                  0x00000010
187  #define SGE_MII_READ                 0x00000000
188  #define SGE_MII_WRITE                0x00000020
189
190  #define SGE_MIIAUTON_NP              0x8000
191  #define SGE_MIIAUTON_TX              0x0080
192  #define SGE_MIIAUTON_TX_FULL         0x0100
193  #define SGE_MIIAUTON_T_FULL          0x0040
194
195  /* TX/RX Descriptor */
196  typedef struct sge_desc
197  {
198       uint32_t pkt_size;
199       uint32_t status;
200       uint32_t buf_ptr;
201       uint32_t flags;
202  }
203  sge_desc_t;
```

```
204
205  typedef struct sge
206  {
207        char name[8];
208        int model;
209        int status;
210        int flags;
211        int irq;
212        int irq_hook;
213        int revision;
214        u8_t *regs;
215        ether_addr_t address;
216
217        struct mii_phy *mii;
218        struct mii_phy *first_mii;
219        uint32_t cur_phy;
220
221        int link_speed;
222        int duplex_mode;
223        int autoneg_done;
224
225        uint32_t cur_rx;
226        uint32_t cur_tx;
227
228        sge_desc_t *rx_desc;
229        phys_bytes rx_desc_p;
230        char *rx_buffer;
231        phys_bytes rx_buffer_p;
232
233        sge_desc_t *tx_desc;
234        phys_bytes tx_desc_p;
235        char *tx_buffer;
236        phys_bytes tx_buffer_p;
237
238        int client;
239        message rx_message;
240        message tx_message;
241        size_t rx_size;
242
243        int RGMII;
244        int MAC_APC;
245  }
246  sge_t;
247
248  struct mii_phy
249  {
250        struct mii_phy *next;
251        int addr;
252        uint16_t id0;
```

```
253         uint16_t id1;
254         uint16_t status;
255         uint16_t types;
256     };
257
258     #endif /* !_SGE_H_ */
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X sge.c X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
1     /* sge.h
2      *
3      * SiS 190/191 Ethernet Controller driver
4      *
5      * Parts of this code are based on the FreeBSD implementation
6      * (https://svnweb.freebsd.org/base/head/sys/dev/sge/), the
7      * e1000 driver by Niek Linnenbank, and the official
8      * SiS 190/191 GNU/Linux driver by K.M. Liu.
9      *
10     * Created: May 2017 by Marcelo Alencar <marceloalves@ufpa.br>
11     */
12
13    #include <minix/drivers.h>
14    #include <minix/netdriver.h>
15    #include <machine/pci.h>
16    #include "sge.h"
17
18    static int sge_instance;
19    static sge_t sge_state;
20
21    static void sge_init(message *mp);
22    static void sge_init_pci(void);
23    static int sge_probe(sge_t *e, int skip);
24    static int sge_init_hw(sge_t *e);
25    static void sge_init_addr(sge_t *e);
26    static void sge_init_buf(sge_t *e);
27    static void sge_reset_hw(sge_t *e);
28    static void sge_interrupt(message *mp);
29    static void sge_stop(sge_t *e);
30    static uint32_t sge_reg_read(sge_t *e, uint32_t reg);
31    static void sge_reg_write(sge_t *e, uint32_t reg,
          uint32_t value);
32    static void sge_reg_set(sge_t *e, uint32_t reg,
          uint32_t value);
33    static void sge_reg_unset(sge_t *e, uint32_t reg,
          uint32_t value);
34    static uint16_t read_eeprom(sge_t *e, int reg);
35    static int sge_mii_probe(sge_t *e);
36    static uint16_t sge_mii_read(sge_t *e, uint32_t phy,
```

```
        uint32_t reg);
37  static void sge_mii_write(sge_t *e, uint32_t phy, uint32_t reg,
        uint32_t data);
38  static void sge_writev_s(message *mp, int from_int);
39  static void sge_readv_s(message *mp, int from_int);
40  static void sge_getstat_s(message *mp);
41  static uint16_t sge_default_phy(sge_t *e);
42  static uint16_t sge_reset_phy(sge_t *e, uint32_t addr);
43  static void sge_phymode(sge_t *e);
44  static void sge_macmode(sge_t *e);
45  static void reply(sge_t *e);
46  static void mess_reply(message *req, message *reply);
47  static void sge_dump(message *m);
48
49  /* SEF functions and variables. */
50  static void sef_local_startup(void);
51  static int sef_cb_init_fresh(int type, sef_init_info_t *info);
52  static void sef_cb_signal_handler(int signo);
53
54  /*===========================================================*
55   *                          main                             *
56   *===========================================================*/
57  int main(int argc, char *argv[])
58  {
59      /* This is the main driver task. */
60      message m;
61      int ipc_status;
62      int r;
63
64      /* SEF local startup. */
65      env_setargs(argc, argv);
66      sef_local_startup();
67
68      /*
69       * Enter the main driver loop.
70       */
71      while (TRUE)
72      {
73       if ((r= netdriver_receive(ANY, &m, &ipc_status)) != OK)
74       {
75            panic("netdriver_receive failed: %d", r);
76       }
77
78       if (is_ipc_notify(ipc_status))
79       {
80            switch (_ENDPOINT_P(m.m_source))
81            {
82            case HARDWARE:
83                 sge_interrupt(&m);
```

```
84                  break;
85              case CLOCK:
86                  break;
87              case TTY_PROC_NR:
88                  sge_dump(&m);
89                  break;
90              }
91              continue;
92          }
93          switch (m.m_type)
94          {
95          case DL_CONF:       sge_init(&m);               break;
96          case DL_GETSTAT_S:  sge_getstat_s(&m);          break;
97          case DL_WRITEV_S:   sge_writev_s(&m, FALSE);    break;
98          case DL_READV_S:    sge_readv_s(&m, FALSE);     break;
99          default:
100             panic("illegal message: %d", m.m_type);
101         }
102     }
103 }
104
105 /*===========================================================*
106  *                  sef_local_startup                        *
107  *===========================================================*/
108 static void sef_local_startup()
109 {
110   /* Register init callbacks. */
111   sef_setcb_init_fresh(sef_cb_init_fresh);
112   sef_setcb_init_lu(sef_cb_init_fresh);
113   sef_setcb_init_restart(sef_cb_init_fresh);
114
115   /* Register live update callbacks. */
116   sef_setcb_lu_prepare(sef_cb_lu_prepare_always_ready);
117   sef_setcb_lu_state_isvalid(sef_cb_lu_state_isvalid_workfree);
118
119   /* Register signal callbacks. */
120   sef_setcb_signal_handler(sef_cb_signal_handler);
121
122   /* Let SEF perform startup. */
123   sef_startup();
124 }
125
126 /*===========================================================*
127  *                   sef_cb_init_fresh                       *
128  *===========================================================*/
129 static int sef_cb_init_fresh(int UNUSED(type),
        sef_init_info_t *UNUSED(info))
130 {
131      /* Initialize the SiS FE Driver. */
```

```
132       int r, fkeys, sfkeys;
133       long v;
134
135       /* Request function key for debug dumps */
136       fkeys = sfkeys = 0;
137       bit_set(sfkeys, 7);
138       if ((r = fkey_map(&fkeys, &sfkeys)) != OK)
139            printf("sge: couldn't bind Shift+F7 key (%d)\n", r);
140
141       v = 0;
142       (void)env_parse("instance", "d", 0, &v, 0, 255);
143       sge_instance = (int) v;
144
145       /* Clear state. */
146       memset(&sge_state, 0, sizeof(sge_state));
147
148       /* Announce we are up! */
149       netdriver_announce();
150
151       return(OK);
152  }
153
154  /*===========================================================*
155   *                    sef_cb_signal_handler                  *
156   *===========================================================*/
157  static void sef_cb_signal_handler(int signo)
158  {
159   sge_t *e;
160   e = &sge_state;
161
162   /* Only check for termination signal, ignore anything else. */
163   if (signo != SIGTERM) return;
164
165   sge_stop(e);
166  }
167
168  /*===========================================================*
169   *                         sge_init                          *
170   *===========================================================*/
171  static void sge_init(message *mp)
172  {
173       static int first_time = 1;
174       message reply_mess;
175       sge_t *e;
176
177       /* Configure PCI devices, if needed. */
178       if (first_time)
179       {
180            first_time = 0;
```

```
181             sge_init_pci();
182       }
183       e = &sge_state;
184
185       e->flags &= ~(SGE_PROMISC|SGE_MULTICAST|SGE_BROADCAST);
186       if (mp->m_net_netdrv_dl_conf.mode & DL_PROMISC_REQ)
187             e->flags |= SGE_PROMISC|SGE_MULTICAST|SGE_BROADCAST;
188       if (mp->m_net_netdrv_dl_conf.mode & DL_MULTI_REQ)
189             e->flags |= SGE_MULTICAST;
190       if (mp->m_net_netdrv_dl_conf.mode & DL_BROAD_REQ)
191             e->flags |= SGE_BROADCAST;
192
193       /* Initialize hardware, if needed. */
194       if (!(e->status & SGE_ENABLED) && !(sge_init_hw(e)))
195       {
196             reply_mess.m_type  = DL_CONF_REPLY;
197             reply_mess.m_netdrv_net_dl_conf.stat = ENXIO;
198             mess_reply(mp, &reply_mess);
199             return;
200       }
201       /* Reply back to INET. */
202       reply_mess.m_type  = DL_CONF_REPLY;
203       reply_mess.m_netdrv_net_dl_conf.stat = OK;
204       memcpy(reply_mess.m_netdrv_net_dl_conf.hw_addr,
              e->address.ea_addr,
205             sizeof(reply_mess.m_netdrv_net_dl_conf.hw_addr));
206       mess_reply(mp, &reply_mess);
207 }
208
209 /*=========================================================*
210  *                        sge_init_pci                     *
211  *=========================================================*/
212 static void sge_init_pci()
213 {
214       sge_t *e;
215
216       /* Initialize the PCI bus. */
217       pci_init();
218
219       /* Try to detect sge's. */
220       e = &sge_state;
221       strlcpy(e->name, "sge#0", sizeof(e->name));
222       e->name[4] += sge_instance;
223       sge_probe(e, sge_instance);
224 }
225
226 /*=========================================================*
227  *                        sge_probe                        *
228  *=========================================================*/
```

```
229  static int sge_probe(sge_t *e, int skip)
230  {
231       int r, devind, ioflag;
232       u16_t vid, did, cr;
233       u32_t status[2];
234       u32_t base, size;
235       u32_t gfpreg, sector_base_addr;
236
237       /*
238        * Attempt to iterate the PCI bus. Start at the
beginning.
239        */
240       if ((r = pci_first_dev(&devind, &vid, &did)) == 0)
241       {
242            return FALSE;
243       }
244       /* Loop devices on the PCI bus. */
245       while (skip--)
246       {
247            if (!(r = pci_next_dev(&devind, &vid, &did)))
248            {
249                 return FALSE;
250            }
251       }
252
253       /*
254        * Successfully detected an SiS Ethernet Controller on
the
             PCI bus.
255        */
256       e->status = SGE_DETECTED;
257
258       /*
259        * Set card specific properties.
260        */
261       switch (did)
262       {
263       case SGE_DEV_0190:
264            /* Inform the user about the new card. */
265            e->model = SGE_DEV_0190;
266            break;
267       case SGE_DEV_0191:
268            /* Inform the user about the new card. */
269            e->model = SGE_DEV_0191;
270            break;
271       default:
272            break;
273       }
274
```

```
275       /* Reserve PCI resources found. */
276       if ((r = pci_reserve_ok(devind)) != OK)
277       {
278            panic("failed to reserve PCI device: %d", r);
279       }
280       /* Read PCI configuration. */
281       e->irq   = pci_attr_r8(devind, PCI_ILR);
282
283       if ((r = pci_get_bar(devind, PCI_BAR, &base, &size,
              &ioflag)) != OK)
284            panic("failed to get PCI BAR (%d)", r);
285       if (ioflag) panic("PCI BAR is not for memory");
286
287       e->regs  = vm_map_phys(SELF, (void *) base, size);
288       if (e->regs == (u8_t *) -1) {
289            panic("failed to map hardware registers from PCI");
290       }
291
292       cr = pci_attr_r16(devind, PCI_CR);
293       if (!(cr & PCI_CR_MAST_EN))
294            pci_attr_w16(devind, PCI_CR, cr | PCI_CR_MAST_EN);
295
296       /* Where to read MAC from? */
297       int isAPC = pci_attr_r8(devind, 0x73);
298       if ((isAPC & 0x1) == 0)
299       {
300            e->MAC_APC = 0;
301       }
302       else
303       {
304            e->MAC_APC = 1;
305       }
306
307       return TRUE;
308  }
309
310  /*============================================================*
311   *                        sge_init_hw                         *
312   *============================================================*/
313  static int sge_init_hw(e)
314  sge_t *e;
315  {
316       int r, i;
317       uint32_t control;
318       uint16_t filter;
319
320       e->status = SGE_ENABLED;
321       e->irq_hook = e->irq;
322
```

```
323         /*
324          * Set the interrupt handler and policy. Do not
                automatically
325          * re-enable interrupts. Return the IRQ line number on
                interrupts.
326          */
327         if ((r = sys_irqsetpolicy(e->irq, 0, &e->irq_hook)) !=
OK)
328         {
329              printf("sys_irqsetpolicy failed: %d", r);
330              return -EFAULT;
331         }
332         if ((r = sys_irqenable(&e->irq_hook)) != OK)
333         {
334              printf("sys_irqenable failed: %d", r);
335              return -EFAULT;
336         }
337         /* Reset hardware. */
338         sge_reset_hw(e);
339
340         /* Initialization routine */
341         sge_init_addr(e);
342         sge_init_buf(e);
343
344         if (sge_mii_probe(e) == 0)
345         {
346              return -ENODEV;
347         }
348
349         sge_reg_write(e, SGE_REG_RXMACADDR, 0);
350
351         /* Set filter */
352         filter = sge_reg_read(e, SGE_REG_RXMACCONTROL);
353         /* disable packet filtering before address is set */
354         filter &= ~(SGE_RXCTRL_BCAST | SGE_RXCTRL_ALLPHYS |
                SGE_RXCTRL_MCAST |
355               SGE_RXCTRL_MYPHYS);
356         sge_reg_write(e, SGE_REG_RXMACCONTROL, filter);
357         /* Write MAC to registers */
358         for (i = 0; i < 6 ; i++)
359         {
360              uint8_t w;
361
362              w = (uint8_t) e->address.ea_addr[i];
363              sge_reg_write(e, SGE_REG_RXMACADDR + i, w);
364         }
365
366         /* Enable filter */
367         filter |= SGE_RXCTRL_MYPHYS;
```

```
368       if (e->flags & SGE_PROMISC)
369       {
370            filter |= (SGE_RXCTRL_BCAST | SGE_RXCTRL_MCAST |
              SGE_RXCTRL_ALLPHYS);
371       }
372       else if (e->flags & SGE_MULTICAST)
373       {
374            filter |= (SGE_RXCTRL_BCAST | SGE_RXCTRL_MCAST);
375       }
376       else
377       {
378            filter |= SGE_RXCTRL_BCAST;
379       }
380
381       sge_reg_write(e, SGE_REG_RXMACCONTROL, filter);
382       sge_reg_write(e, SGE_REG_RXHASHTABLE, 0xffffffff);
383       sge_reg_write(e, SGE_REG_RXHASHTABLE2, 0xffffffff);
384
385       /* Enable interrupts */
386       sge_reg_write(e, SGE_REG_INTRMASK, SGE_INTRS);
387
388       /* Enable TX/RX */
389       control = sge_reg_read(e, SGE_REG_TX_CTL);
390       sge_reg_write(e, SGE_REG_TX_CTL, control | 0x1);
391       control = sge_reg_read(e, SGE_REG_RX_CTL);
392       sge_reg_write(e, SGE_REG_RX_CTL, control | 0x1 | 0x10);
393
394       return TRUE;
395  }
396
397  /*=============================================================*
398   *                    sge_init_addr                            *
399   *=============================================================*/
400  static void sge_init_addr(e)
401  sge_t *e;
402  {
403       static char eakey[] = SGE_ENVVAR "#_EA";
404       static char eafmt[] = "x:x:x:x:x:x";
405       u16_t val;
406       int i;
407       long v;
408
409       /*
410        * Do we have a user defined ethernet address?
411        */
412       eakey[sizeof(SGE_ENVVAR)-1] = '0' + sge_instance;
413
414       for (i = 0; i < 6; i++)
415       {
```

```
416             if (env_parse(eakey, eafmt, i, &v, 0x00L, 0xFFL) !=
                    EP_SET)
417                   break;
418             e->address.ea_addr[i] = v;
419       }
420
421       /* Nothing was read or not everything was read? */
422       if (i != 6)
423       {
424             /* Embedded SiS190 has MAC in southbridge. */
425             if (e->MAC_APC)
426             {
427                   /*
428                   // ISA Bridge read not implemented.
429                   */
430                   panic("Read MAC from APC not
                          implemented...\n");
431             }
432             /* Standalone SiS190 has MAC in EEPROM. */
433             else
434             {
435                   for (i = 0; i < 3; i++)
436                   {
437                         val = read_eeprom(e, SGE_EEPADDR_MAC+i);
438                         e->address.ea_addr[(i*2)] = (val & 0xff);
439                         e->address.ea_addr[(i*2)+1] = (val &
                                0xff00) >> 8;
440                   }
441                   if ((read_eeprom(e, SGE_EEPADDR_INFO) &
                          0x80) != 0)
442                   {
443                         e->RGMII = 1;
444                   }
445                   else
446                   {
447                         e->RGMII = 0;
448                   }
449             }
450       }
451 }
452
453 /*=============================================================*
454  *                        sge_init_buf                         *
455  *=============================================================*/
456 static void sge_init_buf(e)
457 sge_t *e;
458 {
459       /* This function initializes the TX/RX rings, used for
DMA
```

```
                transfers */
460         int i, rx_align, tx_align;
461         phys_bytes rx_buff_p;
462         phys_bytes tx_buff_p;
463         phys_bytes rx_desc_p;
464         phys_bytes tx_desc_p;
465
466         if (!e->rx_desc)
467         {
468               /* Allocate RX descriptors.    */
469               /* rx_desc: Virtual address    */
470               /* rx_desc_p: Physical address */
471               if ((e->rx_desc = alloc_contig(SGE_RXD_TOTALSIZE+15,
                      AC_ALIGN4K,
472                     &rx_desc_p)) == NULL)
473               {
474                     panic("%s: Failed to allocate RX descriptors.
                          \n", e->name);
475               }
476               memset(e->rx_desc, 0, SGE_RXD_TOTALSIZE + 15);
477
478               /* Allocate RX buffers */
479               if ((e->rx_buffer = alloc_contig(SGE_RXB_TOTALSIZE +
                      15, AC_ALIGN4K,
480                     &rx_buff_p)) == NULL)
481               {
482                     panic("%s: Failed to allocate RX buffers.\n",
                          e->name);
483               }
484               memset(e->rx_buffer, 0, SGE_RXB_TOTALSIZE + 15);
485
486               /* Align addresses to multiple of 16 bit */
487               rx_align = ((rx_buff_p + 0xf) & ~0xf) - rx_buff_p;
488               rx_buff_p = ((rx_buff_p + 0xf) & ~0xf);
489
490               e->cur_rx = 0;
491
492               /* Setup receive descriptors. */
493               for (i = 0; i < SGE_RXDESC_NR; i++)
494               {
495                     /* RX descriptors are initially held by
                          hardware */
496                     e->rx_desc[i].pkt_size = 0;
497                     e->rx_desc[i].status = SGE_RXSTATUS_RXOWN |
                          SGE_RXSTATUS_RXINT;
498                     e->rx_desc[i].buf_ptr = rx_buff_p + (i *
                          SGE_BUF_SIZE);
499                     e->rx_desc[i].flags = (SGE_BUF_SIZE & 0xfff8);
500               }
```

```
501             /* Last descriptor is marked as final */
502             e->rx_desc[SGE_RXDESC_NR - 1].flags |=
                    SGE_DESC_FINAL;
503         }
504
505         if (!e->tx_desc)
506         {
507             /* Allocate TX descriptors.    */
508             /* tx_desc: Virtual address    */
509             /* tx_desc_p: Physical address */
510             if ((e->tx_desc = alloc_contig(SGE_TXD_TOTALSIZE +
                    15, AC_ALIGN4K,
511                  &tx_desc_p)) == NULL)
512             {
513                 panic("%s: Failed to allocate TX
                        descriptors.\n", e->name);
514             }
515             memset(e->tx_desc, 0, SGE_TXD_TOTALSIZE + 15);
516
517             /* Allocate TX buffers */
518             if ((e->tx_buffer = alloc_contig(SGE_TXB_TOTALSIZE +
                    15, AC_ALIGN4K,
519                  &tx_buff_p)) == NULL)
520             {
521                 panic("%s: Failed to allocate TX buffers.\n",
                        e->name);
522             }
523             memset(e->tx_buffer, 0, SGE_TXB_TOTALSIZE + 15);
524
525             /* Align addresses to multiple of 16 bit */
526             tx_align = ((tx_buff_p + 0xf) & ~0xf) - tx_buff_p;
527             tx_buff_p = ((tx_buff_p + 0xf) & ~0xf);
528
529             e->cur_tx = 0;
530
531             /* Setup receive descriptors. */
532             for (i = 0; i < SGE_TXDESC_NR; i++)
533             {
534                 /* TX descriptors will be filled by software */
535                 e->tx_desc[i].pkt_size = 0;
536                 e->tx_desc[i].status = 0;
537                 e->tx_desc[i].buf_ptr = 0;
538                 e->tx_desc[i].flags = 0;
539             }
540             /* Last descriptor is marked as final */
541             e->tx_desc[SGE_TXDESC_NR-1].flags = SGE_DESC_FINAL;
542         }
543
544         /* Apply alignment to virtual addresses, and store at
```

```
             status */
545         e->tx_buffer += tx_align;
546         e->rx_buffer += rx_align;
547         e->tx_buffer_p = tx_buff_p;
548         e->rx_buffer_p = rx_buff_p;
549         e->tx_desc_p = tx_desc_p;
550         e->rx_desc_p = rx_desc_p;
551
552         /* Inform card where the buffer is */
553         sge_reg_write(e, SGE_REG_TX_DESC, tx_desc_p);
554         sge_reg_write(e, SGE_REG_RX_DESC, rx_desc_p);
555   }
556
557   /*=============================================================*
558    *                          sge_reset_hw                       *
559    *=============================================================*/
560   static void sge_reset_hw(e)
561   sge_t *e;
562   {
563        sge_reg_write(e, SGE_REG_INTRMASK, 0);
564        sge_reg_write(e, SGE_REG_INTRSTATUS, 0xffffffff);
565
566        sge_reg_write(e, SGE_REG_TX_CTL, 0x00001c00);
567        sge_reg_write(e, SGE_REG_RX_CTL, 0x001e1c00);
568
569        sge_reg_write(e, SGE_REG_INTRCONTROL, 0x8000);
570        sge_reg_read(e, SGE_REG_INTRCONTROL);
571        micro_delay(100);
572        sge_reg_write(e, SGE_REG_INTRCONTROL, 0x0);
573
574        sge_reg_write(e, SGE_REG_INTRMASK, 0);
575        sge_reg_write(e, SGE_REG_INTRSTATUS, 0xffffffff);
576
577        sge_reg_write(e, SGE_REG_TX_DESC, 0x0);
578        sge_reg_write(e, SGE_REG_RESERVED0, 0x0);
579        sge_reg_write(e, SGE_REG_RX_DESC, 0x0);
580        sge_reg_write(e, SGE_REG_RESERVED1, 0x0);
581
582        sge_reg_write(e, SGE_REG_PMCONTROL, 0xffc00000);
583        sge_reg_write(e, SGE_REG_RESERVED2, 0x0);
584
585        if (e->RGMII)
586        {
587             sge_reg_write(e, SGE_REG_STATIONCONTROL,0x04008001);
588        }
589        else
590        {
591             sge_reg_write(e, SGE_REG_STATIONCONTROL,0x04000001);
592        }
```

```
593
594         sge_reg_write(e, SGE_REG_GMACIOCR, 0x0);
595         sge_reg_write(e, SGE_REG_GMACIOCTL, 0x0);
596
597         sge_reg_write(e, SGE_REG_TXMACCONTROL, 0x00002364);
598         sge_reg_write(e, SGE_REG_TXMACTIMELIMIT, 0x0000000f);
599
600         sge_reg_write(e, SGE_REG_RGMIIDELAY, 0x0);
601         sge_reg_write(e, SGE_REG_RESERVED3, 0x0);
602         sge_reg_write(e, SGE_REG_RXMACCONTROL, 0x12);
603
604         sge_reg_write(e, SGE_REG_RXHASHTABLE, 0x0);
605         sge_reg_write(e, SGE_REG_RXHASHTABLE2, 0x0);
606
607         sge_reg_write(e, SGE_REG_RXWAKEONLAN, 0x80ff0000);
608         sge_reg_write(e, SGE_REG_RXWAKEONLANDATA, 0x80ff0000);
609         sge_reg_write(e, SGE_REG_RXMPSCONTROL, 0x0);
610         sge_reg_write(e, SGE_REG_RESERVED4, 0x0);
611   }
612
613   /*============================================================*
614    *                          sge_writev_s                      *
615    *============================================================*/
616   static void sge_writev_s(mp, from_int)
617   message *mp;
618   int from_int;
619   {
620         sge_t *e = &sge_state;
621         sge_desc_t *desc;
622         iovec_s_t iovec[SGE_IOVEC_NR];
623         int r, i, bytes = 0, size;
624         uint32_t command;
625         uint32_t current;
626         uint32_t status;
627
628         /* Are we called from the interrupt handler? */
629         if (!from_int)
630         {
631               if (!(e->autoneg_done))
632               {
633                     return;
634               }
635
636               /* Copy write message. */
637               e->tx_message = *mp;
638               e->client = mp->m_source;
639               e->status |= SGE_WRITING;
640
641               /*
```

```
642              * Copy the I/O vector table.
643              */
644             if ((r = sys_safecopyfrom(e->tx_message.m_source,
645                e->tx_message.m_net_netdrv_dl_writev_s.grant, 0,
646                (vir_bytes) iovec,
647                e->tx_message.m_net_netdrv_dl_writev_s.count *
648                sizeof(iovec_s_t))) != OK)
649             {
650                  panic("sys_safecopyfrom() failed: %d", r);
651             }
652
653             current = e->cur_tx % SGE_TXDESC_NR;
654             desc = &e->tx_desc[current];
655
656             /* Loop vector elements. */
657             for (i = 0; i <
                   e->tx_message.m_net_netdrv_dl_writev_s.count; i++)
658             {
659                  size=iovec[i].iov_size < (SGE_BUF_SIZE-bytes) ?
660                       iovec[i].iov_size : (SGE_BUF_SIZE-bytes);
661
662                  /* Copy bytes to TX queue buffers. */
663                  if ((r=sys_safecopyfrom(e->tx_message.m_source,
664                       iovec[i].iov_grant, 0,
665                       (vir_bytes) e->tx_buffer + bytes +
666                       (current * SGE_BUF_SIZE), size)) != OK)
667                  {
668                       panic("sys_safecopyfrom() failed: %d",
r);
669                  }
670
671                  bytes += size;
672             }
673
674             /* Mark this descriptor ready. */
675             desc->pkt_size = size & 0xffff;
676             desc->status = (SGE_TXSTATUS_PADEN |
                   SGE_TXSTATUS_CRCEN |
677                   SGE_TXSTATUS_DEFEN | SGE_TXSTATUS_THOL3 |
                         SGE_TXSTATUS_TXINT);
678             desc->buf_ptr = e->tx_buffer_p + (current *
                   SGE_BUF_SIZE);
679             desc->flags |= size & 0xffff;
680             if (e->duplex_mode == 0)
681             {
682                  desc->status |= (SGE_TXSTATUS_COLSEN |
                         SGE_TXSTATUS_CRSEN |
683                           SGE_TXSTATUS_BKFEN);
684                  if (e->link_speed == SGE_SPEED_1000)
```

```
685                         desc->status |= (SGE_TXSTATUS_EXTEN |
                               SGE_TXSTATUS_BSTEN);
686             }
687             desc->status |= SGE_TXSTATUS_TXOWN;
688
689             /* Increment tail. Start transmission. */
690             e->cur_tx = (current + 1) % SGE_TXDESC_NR;
691             command = sge_reg_read(e, SGE_REG_TX_CTL);
692             sge_reg_write(e, SGE_REG_TX_CTL, 0x10 | command);
693        }
694        else
695        {
696             e->status |= SGE_TRANSMIT;
697        }
698        reply(e);
699   }
700
701   /*==========================================================*
702    *                        sge_readv_s                       *
703    *==========================================================*/
704   static void sge_readv_s(mp, from_int)
705   message *mp;
706   int from_int;
707   {
708        sge_t *e = &sge_state;
709        sge_desc_t *desc;
710        iovec_s_t iovec[SGE_IOVEC_NR];
711        int r, i, bytes = 0, size;
712        uint32_t command;
713        uint32_t current;
714        uint32_t status;
715        uint32_t pkt_size;
716
717        /* Are we called from the interrupt handler? */
718        if (!from_int)
719        {
720             /* Copy read message. */
721             e->rx_message = *mp;
722             e->client = mp->m_source;
723             e->status |= SGE_READING;
724             e->rx_size = 0;
725        }
726
727        if (e->status & SGE_READING)
728        {
729             /*
730              * Copy the I/O vector table.
731              */
732             if ((r = sys_safecopyfrom(e->rx_message.m_source,
```

```
733                 e->rx_message.m_net_netdrv_dl_readv_s.grant, 0,
734                 (vir_bytes) iovec,
735                 e->rx_message.m_net_netdrv_dl_readv_s.count *
736                 sizeof(iovec_s_t))) != OK)
737             {
738                 panic("sys_safecopyfrom() failed: %d", r);
739             }
740
741             /* Select packet not owned by the card. */
742             current = e->cur_rx % SGE_RXDESC_NR;
743             desc = &e->rx_desc[current];
744
745             /* Give up if none found. */
746             if (desc->status & SGE_RXSTATUS_RXOWN)
747             {
748                 return;
749             }
750
751             pkt_size = (desc->pkt_size & 0xffff);
752
753             /* Copy to vector elements. */
754             for (i = 0;
                   i < e->rx_message.m_net_netdrv_dl_readv_s.count &&
755                 bytes < pkt_size; i++)
756             {
757                 size = iovec[i].iov_size < (pkt_size - bytes) ?
758                     iovec[i].iov_size : (pkt_size - bytes);
759
760                 if ((r = sys_safecopyto(e->rx_message.m_source,
                      iovec[i].iov_grant,
761                     0, (vir_bytes) e->rx_buffer + bytes +
762                     (current * SGE_BUF_SIZE), size)) != OK)
763                 {
764                     panic("sys_safecopyto() failed: %d", r);
765                 }
766                 bytes += size;
767             }
768
769             /* Flip ownership back to the card */
770             desc->pkt_size = 0;
771             desc->status = SGE_RXSTATUS_RXOWN |
                SGE_RXSTATUS_RXINT;
772
773             /* Update current and reenable. */
774             e->cur_rx = (current + 1) % SGE_RXDESC_NR;
775             command = sge_reg_read(e, SGE_REG_RX_CTL);
776             sge_reg_write(e, SGE_REG_RX_CTL, 0x10 | command);
777
778             e->rx_size = bytes;
```

```
779             e->status |= SGE_RECEIVED;
780       }
781       reply(e);
782   }
783
784   /*==========================================================*
785    *                        sge_getstat_s                     *
786    *==========================================================*/
787   static void sge_getstat_s(mp)
788   message *mp;
789   {
790       int r;
791       eth_stat_t stats;
792
793       stats.ets_recvErr   = 0;
794       stats.ets_sendErr   = 0;
795       stats.ets_OVW       = 0;
796       stats.ets_CRCerr    = 0;
797       stats.ets_frameAll  = 0;
798       stats.ets_missedP   = 0;
799       stats.ets_packetR   = 0;
800       stats.ets_packetT   = 0;
801       stats.ets_collision = 0;
802       stats.ets_transAb   = 0;
803       stats.ets_carrSense = 0;
804       stats.ets_fifoUnder = 0;
805       stats.ets_fifoOver  = 0;
806       stats.ets_CDheartbeat = 0;
807       stats.ets_OWC = 0;
808
809       sys_safecopyto(mp->m_source,
             mp->m_net_netdrv_dl_getstat_s.grant, 0,
810            (vir_bytes)&stats, sizeof(stats));
811       mp->m_type  = DL_STAT_REPLY;
812       if((r=ipc_send(mp->m_source, mp)) != OK)
813            panic("sge_getstat: ipc_send() failed: %d", r);
814   }
815
816   /*==========================================================*
817    *                        sge_interrupt                     *
818    *==========================================================*/
819   static void sge_interrupt(mp)
820   message *mp;
821   {
822       sge_t *e;
823       u32_t status;
824
825       /*
826        * Check the card for interrupt reason(s).
```

```
827         */
828        e = &sge_state;
829
830        status = sge_reg_read(e, SGE_REG_INTRSTATUS);;
831        if (!(status == 0xffffffff || (status & SGE_INTRS) == 0))
832        {
833             //Acknowledge and disable interrupts
834             sge_reg_write(e, SGE_REG_INTRSTATUS, status);
835             sge_reg_write(e, SGE_REG_INTRMASK, 0);
836
837             /* Read the Interrupt Cause Read register. */
838             if ((status & SGE_INTRS) == 0)
839             {
840                  /* Nothing */
841                  return;
842             }
843             sge_reg_write(e, SGE_REG_INTRSTATUS, status);
844             if (status & (SGE_INTR_TX_DONE | SGE_INTR_TX_IDLE))
845                  /* Tx interrupt */
846                  sge_writev_s(&e->tx_message, TRUE);
847             if (status & (SGE_INTR_RX_DONE | SGE_INTR_RX_IDLE))
848                  /* Rx interrupt */
849                  sge_readv_s(&e->rx_message, TRUE);
850             if (status & SGE_INTR_LINK)
851                  printf("%s: Link changed.\n", e->name);
852        }
853
854        /* Re-enable interrupts. */
855        sge_reg_write(e, SGE_REG_INTRMASK, SGE_INTRS);
856        if (sys_irqenable(&e->irq_hook) != OK)
857        {
858             panic("failed to re-enable IRQ");
859        }
860   }
861
862   /*=========================================================*
863    *                         sge_stop                        *
864    *=========================================================*/
865   static void sge_stop(e)
866   sge_t *e;
867   {
868        uint32_t val;
869        printf("%s: stopping...\n", e->name);
870
871        sge_reset_hw(e);
872
873        sge_reg_write(e, SGE_REG_INTRMASK, 0x0);
874        micro_delay(2000);
875
```

```
876         val = sge_reg_read(e, SGE_REG_INTRCONTROL) | 0x8000;
877         sge_reg_write(e, SGE_REG_INTRCONTROL, val);
878         micro_delay(50);
879         sge_reg_write(e, SGE_REG_INTRCONTROL, val & ~0x8000);
880
881         exit(EXIT_SUCCESS);
882   }
883
884   /*============================================================*
885    *                          sge_reg_read                      *
886    *============================================================*/
887   static uint32_t sge_reg_read(e, reg)
888   sge_t *e;
889   uint32_t reg;
890   {
891         uint32_t value;
892
893         /* Read from memory mapped register. */
894         value = *(volatile u32_t *)(e->regs + reg);
895
896         /* Return the result. */
897         return value;
898   }
899
900   /*============================================================*
901    *                         sge_reg_write                      *
902    *============================================================*/
903   static void sge_reg_write(e, reg, value)
904   sge_t *e;
905   uint32_t reg;
906   uint32_t value;
907   {
908         /* Write to memory mapped register. */
909         *(volatile u32_t *)(e->regs + reg) = value;
910   }
911
912   /*============================================================*
913    *                          sge_reg_set                       *
914    *============================================================*/
915   static void sge_reg_set(e, reg, value)
916   sge_t *e;
917   uint32_t reg;
918   uint32_t value;
919   {
920         uint32_t data;
921
922         /* First read the current value. */
923         data = sge_reg_read(e, reg);
924
```

```
925       /* Set value, and write back. */
926       sge_reg_write(e, reg, data | value);
927  }
928
929  /*=============================================================*
930   *                        sge_reg_unset                        *
931   *=============================================================*/
932  static void sge_reg_unset(e, reg, value)
933  sge_t *e;
934  uint32_t reg;
935  uint32_t value;
936  {
937       uint32_t data;
938
939       /* First read the current value. */
940       data = sge_reg_read(e, reg);
941
942       /* Unset value, and write back. */
943       sge_reg_write(e, reg, data & ~value);
944  }
945
946  /*=============================================================*
947   *                          sge_mii_read                       *
948   *=============================================================*/
949  static uint16_t sge_mii_read(e, phy, reg)
950  sge_t *e;
951  uint32_t phy;
952  uint32_t reg;
953  {
954       u32_t read_cmd;
955       u32_t data;
956
957       phy = (phy & 0x1f) << 6;
958       reg = (reg & 0x1f) << 11;
959       read_cmd = SGE_MII_REQ | SGE_MII_READ | phy | reg;
960
961       sge_reg_write(e, SGE_REG_GMIICONTROL, read_cmd);
962       micro_delay(50);
963
964       do
965       {
966            data = sge_reg_read(e, SGE_REG_GMIICONTROL);
967            micro_delay(50);
968       } while ((data & SGE_MII_REQ) != 0);
969
970       return (u16_t)((data & SGE_MII_DATA) >>
             SGE_MII_DATA_SHIFT);
971  }
972
```

```
973  /*============================================================*
974   *                        sge_mii_write                       *
975   *============================================================*/
976  static void sge_mii_write(e, phy, reg, data)
977  sge_t *e;
978  uint32_t phy;
979  uint32_t reg;
980  uint32_t data;
981  {
982       u32_t write_cmd;
983
984       phy = (phy & 0x1f) << 6;
985       reg = (reg & 0x1f) << 11;
986       data = (data & 0xffff) << SGE_MII_DATA_SHIFT;
987       write_cmd = SGE_MII_REQ | SGE_MII_WRITE | phy | reg |
               data;
988
989       sge_reg_write(e, SGE_REG_GMIICONTROL, write_cmd);
990       micro_delay(500);
991
992       do
993       {
994            data = sge_reg_read(e, SGE_REG_GMIICONTROL);
995            micro_delay(50);
996       } while ((data & SGE_MII_REQ) != 0);
997  }
998
999  /*============================================================*
1000  *                        read_eeprom                         *
1001  *============================================================*/
1002 static uint16_t read_eeprom(e, reg)
1003 sge_t *e;
1004 int reg;
1005 {
1006      u32_t data;
1007      u32_t read_cmd;
1008
1009      /* Request EEPROM read. */
1010      read_cmd = SGE_EEPROM_REQ | SGE_EEPROM_READ |
1011           (reg << SGE_EEPROM_OFFSET_SHIFT);
1012      sge_reg_write(e, SGE_REG_EEPROMINTERFACE, read_cmd);
1013
1014      /* Wait 500ms */
1015      micro_delay(500);
1016
1017      /* Wait until ready. */
1018      do
1019      {
1020           data = sge_reg_read(e, SGE_REG_EEPROMINTERFACE);
```

```
1021            micro_delay(100);
1022      } while ((data & SGE_EEPROM_REQ) != 0);
1023
1024      return (u16_t)((data & SGE_EEPROM_DATA) >>
SGE_EEPROM_DATA_SHIFT);
1025 }
1026
1027 /*============================================================*
1028  *                      sge_mii_probe                         *
1029  *============================================================*/
1030 static int sge_mii_probe(e)
1031 sge_t *e;
1032 {
1033      int timeout = 10000;
1034      int autoneg_done = 0;
1035      struct mii_phy *phy;
1036      u32_t addr;
1037      u16_t status;
1038      u16_t link_status = SGE_MIISTATUS_LINK;
1039
1040      /* Search for PHY */
1041      for (addr = 0; addr < 32; addr++)
1042      {
1043           status = sge_mii_read(e, addr, SGE_MIIADDR_STATUS);
1044           status = sge_mii_read(e, addr, SGE_MIIADDR_STATUS);
1045
1046           if (status == 0xffff || status == 0)
1047                continue;
1048
1049           phy = alloc_contig(sizeof(struct mii_phy), 0, NULL);
1050           phy->id0 = sge_mii_read(e, addr,
                SGE_MIIADDR_PHY_ID0);
1051           phy->id1 = sge_mii_read(e, addr,
                SGE_MIIADDR_PHY_ID1);
1052           phy->addr = addr;
1053           phy->status = status;
1054           phy->types = 0x2;
1055           phy->next = e->mii;
1056           e->mii = phy;
1057           e->first_mii = phy;
1058      }
1059
1060      if (e->mii == NULL)
1061      {
1062           printf("%s: No transceiver found!\n", e->name);
1063           return 0;
1064      }
1065
1066      e->mii = NULL;
```

```
1067
1068        sge_default_phy(e);
1069
1070        status = sge_reset_phy(e, e->cur_phy);
1071
1072        if(status & SGE_MIISTATUS_LINK)
1073        {
1074             int i;
1075             for (i = 0; i < timeout; i++)
1076             {
1077                  if (!link_status)
1078                       break;
1079
1080                  micro_delay(1000);
1081
1082                  link_status = link_status ^ (sge_mii_read(e,
                        e->cur_phy,
1083                       SGE_MIIADDR_STATUS) & link_status);
1084             }
1085
1086             if (i == timeout)
1087             {
1088                  printf("%s: reset phy and link down now\n",
                        e->name);
1089                  return -ETIME;
1090             }
1091        }
1092
1093        status = sge_mii_read(e, e->cur_phy, SGE_MIIADDR_STATUS);
1094        if (status & SGE_MIISTATUS_LINK)
1095        {
1096             for(int i = 0; i < 1000; i++)
1097             {
1098                  status = sge_mii_read(e, e->cur_phy,
                        SGE_MIIADDR_STATUS);
1099                  if(status & SGE_MIISTATUS_AUTO_DONE)
1100                  {
1101                       autoneg_done = 1;
1102                       break;
1103                  }
1104                  micro_delay(100);
1105             }
1106        }
1107
1108        if(autoneg_done)
1109        {
1110             sge_phymode(e);
1111             sge_macmode(e);
1112        }
```

```
1113
1114      if (e->mii->status & SGE_MIISTATUS_LINK)
1115      {
1116           if(e->RGMII)
1117           {
1118                sge_reg_write(e, SGE_REG_RGMIIDELAY, 0x0441);
1119                sge_reg_write(e, SGE_REG_RGMIIDELAY, 0x0440);
1120           }
1121      }
1122
1123      return 1;
1124 }
1125
1126 /*============================================================*
1127  *                      sge_default_phy                       *
1128  *============================================================*/
1129 static uint16_t sge_default_phy(e)
1130 sge_t *e;
1131 {
1132      struct mii_phy *phy = NULL;
1133      struct mii_phy *default_phy = NULL;
1134      u16_t status;
1135
1136      for(phy = e->first_mii; phy; phy = phy->next)
1137      {
1138           status = sge_mii_read(e, phy->addr,
                        SGE_MIIADDR_STATUS);
1139           status = sge_mii_read(e, phy->addr,
                        SGE_MIIADDR_STATUS);
1140
1141           if ((status & SGE_MIISTATUS_LINK) && !default_phy &&
                        (phy->types != 0))
1142           {
1143                default_phy = phy;
1144           }
1145           else
1146           {
1147                status = sge_mii_read(e, phy->addr,
                             SGE_MIIADDR_CONTROL);
1148                sge_mii_write(e, phy->addr,
                             SGE_MIIADDR_CONTROL,
1149                               status | SGE_MIICTRL_AUTO |
                                     SGE_MIICTRL_ISOLATE);
1150                if (phy->types == 0x02)
1151                     default_phy = phy;
1152            }
1153      }
1154
1155      if (!default_phy)
```

```
1156            default_phy = e->first_mii;
1157
1158       if(e->mii != default_phy )
1159       {
1160            e->mii = default_phy;
1161            e->cur_phy = default_phy->addr;
1162       }
1163
1164       status = sge_mii_read(e, e->cur_phy,
SGE_MIIADDR_CONTROL);
1165       status = status & ~SGE_MIICTRL_ISOLATE;
1166
1167       sge_mii_write(e, e->cur_phy, SGE_MIIADDR_CONTROL,
status);
1168       status = sge_mii_read(e, e->cur_phy, SGE_MIIADDR_STATUS);
1169       status = sge_mii_read(e, e->cur_phy, SGE_MIIADDR_STATUS);
1170
1171       return status;
1172  }
1173
1174  /*============================================================*
1175   *                         sge_reset_phy                      *
1176   *============================================================*/
1177  static uint16_t sge_reset_phy(e, addr)
1178  sge_t *e;
1179  uint32_t addr;
1180  {
1181       int i = 0;
1182       u16_t status;
1183
1184       status = sge_mii_read(e, addr, SGE_MIIADDR_STATUS);
1185       status = sge_mii_read(e, addr, SGE_MIIADDR_STATUS);
1186
1187       sge_mii_write(e, addr, SGE_MIIADDR_CONTROL,
1188            (SGE_MIICTRL_RESET | SGE_MIICTRL_AUTO |
                 SGE_MIICTRL_RST_AUTO));
1189
1190       return status;
1191  }
1192
1193  /*============================================================*
1194   *                          sge_phymode                       *
1195   *============================================================*/
1196  static void sge_phymode(e)
1197  sge_t *e;
1198  {
1199       u32_t status;
1200       u16_t anadv;
1201       u16_t anrec;
```

```
1202       u16_t anexp;
1203       u16_t gadv;
1204       u16_t grec;
1205       status = sge_mii_read(e, e->cur_phy, SGE_MIIADDR_STATUS);
1206       status = sge_mii_read(e, e->cur_phy, SGE_MIIADDR_STATUS);
1207
1208       if (!(status & SGE_MIISTATUS_LINK))
1209            return;
1210
1211       anadv = sge_mii_read(e, e->cur_phy,
SGE_MIIADDR_AUTO_ADV);
1212       anrec = sge_mii_read(e, e->cur_phy,
              SGE_MIIADDR_AUTO_LPAR);
1213       anexp = sge_mii_read(e, e->cur_phy,
SGE_MIIADDR_AUTO_EXT);
1214
1215       e->link_speed = SGE_SPEED_10;
1216       e->duplex_mode = SGE_DUPLEX_OFF;
1217
1218       if((e->model == SGE_DEV_0191) && (anrec &
SGE_MIIAUTON_NP)
1219            && (anexp & 0x2))
1220       {
1221            gadv = sge_mii_read(e, e->cur_phy,
                  SGE_MIIADDR_AUTO_GADV);
1222            grec = sge_mii_read(e, e->cur_phy,
                  SGE_MIIADDR_AUTO_GLPAR);
1223            status = (gadv & (grec >> 2));
1224            if(status & 0x200)
1225            {
1226                 e->link_speed = SGE_SPEED_1000;
1227                 e->duplex_mode = SGE_DUPLEX_ON;
1228            }
1229            else if (status & 0x100)
1230            {
1231                 e->link_speed = SGE_SPEED_1000;
1232                 e->duplex_mode = SGE_DUPLEX_OFF;
1233            }
1234       }
1235       else
1236       {
1237            status = anadv & anrec;
1238
1239            if (status & (SGE_MIIAUTON_TX |
                  SGE_MIIAUTON_TX_FULL))
1240                 e->link_speed = SGE_SPEED_100;
1241            if (status & (SGE_MIIAUTON_TX_FULL |
                  SGE_MIIAUTON_T_FULL))
1242                 e->duplex_mode = SGE_DUPLEX_ON;
```

```
1243      }
1244
1245      e->autoneg_done = 1;
1246 }
1247
1248 /*=========================================================*
1249  *                         sge_macmode                     *
1250  *=========================================================*/
1251 static void sge_macmode(e)
1252 sge_t *e;
1253 {
1254      u32_t status;
1255
1256      status = sge_reg_read(e, SGE_REG_STATIONCONTROL);
1257      status = status & ~(0x0f000000 | SGE_REGSC_FDX |
            SGE_REGSC_SPEED_MASK);
1258
1259      switch (e->link_speed)
1260      {
1261           case SGE_SPEED_1000:
1262                status |= (SGE_REGSC_SPEED_1000 | (0x3 << 24) |
                     (0x1 << 26));
1263                break;
1264           case SGE_SPEED_100:
1265                status |= (SGE_REGSC_SPEED_100 | (0x1 << 26));
1266                break;
1267           case SGE_SPEED_10:
1268                status |= (SGE_REGSC_SPEED_10 | (0x1 << 26));
1269                break;
1270           default:
1271                printf("%s: Unsupported link speed.\n",
                     e->name);
1272      }
1273
1274      if (e->duplex_mode)
1275      {
1276           status = status | SGE_REGSC_FDX;
1277      }
1278
1279      if(e->RGMII)
1280      {
1281           status = status | (0x3 << 24);
1282      }
1283
1284      sge_reg_write(e, SGE_REG_STATIONCONTROL, status);
1285 }
1286
1287 /*=========================================================*
1288  *                          reply                          *
```

```
1289  *===========================================================*/
1290 static void reply(e)
1291 sge_t *e;
1292 {
1293       message msg;
1294       int r;
1295 
1296       /* Only reply to client for read/write request. */
1297       if (!(e->status & SGE_READING || e->status &
SGE_WRITING))
1298       {
1299            return;
1300       }
1301       /* Construct reply message. */
1302       msg.m_type   = DL_TASK_REPLY;
1303       msg.m_netdrv_net_dl_task.flags = DL_NOFLAGS;
1304       msg.m_netdrv_net_dl_task.count = 0;
1305 
1306       /* Did we successfully receive packet(s)? */
1307       if (e->status & SGE_READING && e->status & SGE_RECEIVED)
1308       {
1309            msg.m_netdrv_net_dl_task.flags |= DL_PACK_RECV;
1310            msg.m_netdrv_net_dl_task.count =
1311                 e->rx_size >= ETH_MIN_PACK_SIZE ?
1312                      e->rx_size  : ETH_MIN_PACK_SIZE;
1313 
1314            /* Clear flags. */
1315            e->status &= ~(SGE_READING | SGE_RECEIVED);
1316       }
1317       /* Did we successfully transmit packet(s)? */
1318       if (e->status & SGE_TRANSMIT && e->status & SGE_WRITING)
1319       {
1320            msg.m_netdrv_net_dl_task.flags |= DL_PACK_SEND;
1321 
1322            /* Clear flags. */
1323            e->status &= ~(SGE_WRITING | SGE_TRANSMIT);
1324       }
1325 
1326       /* Acknowledge to INET. */
1327       if ((r = ipc_send(e->client, &msg)) != OK)
1328       {
1329            panic("ipc_send() failed: %d", r);
1330       }
1331 }
1332 
1333 /*===========================================================*
1334  *                          mess_reply                       *
1335  *===========================================================*/
1336 static void mess_reply(req, reply_mess)
```

```
1337 message *req;
1338 message *reply_mess;
1339 {
1340      if (ipc_send(req->m_source, reply_mess) != OK)
1341      {
1342           panic("unable to send reply message");
1343      }
1344 }
1345
1346 /*============================================================*
1347  *                          sge_dump                          *
1348  *============================================================*/
1349 static void sge_dump(m)
1350 message *m;
1351 {
1352      sge_t *e;
1353      e = &sge_state;
1354      long i;
1355      char *dname;
1356
1357      switch (e->model)
1358      {
1359           case SGE_DEV_0190:
1360                dname = "SiS 190 PCI Fast Ethernet Adapter";
1361                break;
1362           case SGE_DEV_0191:
1363                dname = "SiS 191 PCI Gigabit Ethernet Adapter";
1364                break;
1365      }
1366
1367      printf("%s is a %s\n", e->name, dname);
1368
1369      /* MAC Address */
1370      printf("Ethernet Address %x:%x:%x:%x:%x:%x\n",
1371           e->address.ea_addr[0], e->address.ea_addr[1],
1372           e->address.ea_addr[2], e->address.ea_addr[3],
1373           e->address.ea_addr[4], e->address.ea_addr[5]);
1374
1375      /* Link speed */
1376      printf("Media Link On %d Mbps %s-duplex \n",
1377           e->link_speed,
1378           e->duplex_mode ? "full" : "half");
1379
1380      /* PHY Transceiver */
1381      printf("PHY Transceiver (%0x/%0x) found at address %d\n",
1382           e->mii->id0, (e->mii->id1 & 0xFFF0), e->mii->addr);
1383
1384      /* Mac Registers (Memory Mapped)*/
1385      printf("MAC Registers:\n");
```

```
1386       for(i = 0; i < 0x80; i+=4)
1387       {
1388            if((i%16) == 0)
1389                 printf("%2.2xh: ", (char)i);
1390
1391            printf("%8.8x ", sge_reg_read(e, i));
1392
1393            if((i%16) == 12)
1394                 printf("\n");
1395       }
1396
1397       /* EEPROM */
1398       printf("EEPROM Dump:\n");
1399       for(i = 0; i < 0x10; i+=1)
1400       {
1401            if(i%0x8 == 0)
1402                 printf("%2.2xh: ", (char)i);
1403
1404            printf("%4.4x ", read_eeprom(e, i));
1405
1406            if(i == 0x7)
1407                 printf("\n");
1408       }
1409       printf("\n");
1410
1411       /* EEPROM */
1412       printf("PHY Registers:\n");
1413       for(i = 0; i < 0x20; i+=1)
1414       {
1415            if((i%8) == 0 )
1416                 printf("%2.2xh: ", (char)i);
1417
1418            printf("%4.4x ", sge_mii_read(e, e->cur_phy, i));
1419
1420            if((i%8)==7)
1421                 printf("\n");
1422       }
1423       printf("\n");
1424       printf("Current descriptors: TX: %d, RX: %d\n", e-
>cur_tx,
              e->cur_rx);
1425       printf("Current descriptor data: TX: %8.8x %8.8x %8.8x
              %8.8x\n",
1426            e->tx_desc[e->cur_tx].pkt_size,
                 e->tx_desc[e->cur_tx].status,
1427            e->tx_desc[e->cur_tx].buf_ptr,
                 e->tx_desc[e->cur_tx].flags);
1428       printf("Current descriptor data: RX: %8.8x %8.8x %8.8x
              %8.8x\n",
```

```
1429            e->rx_desc[e->cur_rx].pkt_size,
                   e->rx_desc[e->cur_rx].status,
1430            e->rx_desc[e->cur_rx].buf_ptr,
                   e->rx_desc[e->cur_rx].flags);
1431       if (e->cur_tx != 0)
1432       {
1433            printf("Last descriptor data: TX: %8.8x %8.8x %8.8x
                   %8.8x\n",
1434                 e->tx_desc[(e->cur_tx) - 1].pkt_size,
                        e->tx_desc[(e->cur_tx) - 1].status,
1435                 e->tx_desc[(e->cur_tx) - 1].buf_ptr,
                        e->tx_desc[(e->cur_tx) - 1].flags);
1436       }
1437       if (e->cur_rx)
1438       {
1439            printf("Last descriptor data: RX: %8.8x %8.8x %8.8x
                   %8.8x\n",
1440                 e->rx_desc[(e->cur_rx) - 1].pkt_size,
                        e->rx_desc[(e->cur_rx) - 1].status,
1441                 e->rx_desc[(e->cur_rx) - 1].buf_ptr,
                        e->rx_desc[(e->cur_rx) - 1].flags);
1442       }
1443  }
```

ANEXO A – REGISTRADORES DE OPERAÇÃO DA INTERFACE

(continua)

Nome	Desvio	Descrição
SGE_REG_TX_CTL	0x00	Registrador de controle/estado do transmissor.
SGE_REG_TX_DESC	0x04	Ponteiro para o primeiro descritor de pacote para transmissão na memória.
SGE_REG_RESERVED0	0x08	Uso restrito.
SGE_REG_TX_NEXT	0x0c	Ponteiro para o próximo descritor para transmissão que será usado.
SGE_REG_RX_CTL	0x10	Registrador de controle/estado do receptor.
SGE_REG_RX_DESC	0x14	Ponteiro para o primeiro descritor de pacote para recepção na memória.
SGE_REG_RESERVED1	0x18	Uso restrito.
SGE_REG_RX_NEXT	0x1c	Ponteiro para o próximo descritor para recepção que será usado.
SGE_REG_INTRSTATUS	0x20	Identifica a causa de uma interrupção.
SGE_REG_INTRMASK	0x24	Usado para mascarar (ignorar) interrupções específicas.
SGE_REG_INTRCONTROL	0x28	Registrador de controle para interrupções.
SGE_REG_INTRTIMER	0x2c	Temporizador de interrupção.
SGE_REG_PMCONTROL	0x30	Registrador de controle/estado da gerência de energia.
SGE_REG_RESERVED2	0x34	Uso restrito.
SGE_REG_EEPROMCONTROL	0x38	Registrador de controle/estado da EEPROM (memória externa).
SGE_REG_EEPROMINTERFACE	0x3c	Registrador de interface com a EEPROM.
SGE_REG_STATIONCONTROL	0x40	Registrador de controle da portadora (velocidade e modo de comunicação).
SGE_REG_GMIICONTROL	0x44	Registrador de controle do transceptor.
SGE_REG_GMACIOCR	0x48	Registrador de comando para interface Gigabit.
SGE_REG_GMACIOCTL	0x4c	Registrador de controle/status para interface Gigabit.
SGE_REG_TXMACCONTROL	0x50	Registrador de controle/status para o transmissor da interface com o transceptor.
SGE_REG_TXMACTIMELIMIT	0x54	Temporizador de transmissões.

(continuação)

Nome	Desvio	Descrição
SGE_REG_RGMIIDELAY	0x58	Atraso para realização de comandos no transceptor.
SGE_REG_RESERVED3	0x5c	Uso restrito.
SGE_REG_RXMACCONTROL	0x60	Registrador de controle/status para o receptor da interface com o transceptor.
SGE_REG_RXMACADDR	0x62	Endereço MAC da interface para identificação na rede.
SGE_REG_RXHASHTABLE	0x68	1º Endereço MAC do grupo *multicast*.
SGE_REG_RXHASHTABLE2	0x6c	2º Endereço MAC do grupo *multicast*.
SGE_REG_RXWAKEONLAN	0x70	Configuração da função "*Wake on LAN*". (não implementado)
SGE_REG_RXWAKEONLANDATA	0x74	Dados da função "*Wake on LAN*". (não implementado)
SGE_REG_RXMPSCONTROL	0x78	Tamanho máximo de pacotes recebidos.
SGE_REG_RESERVED4	0x7c	Uso restrito.

Fonte: HAIKU INC, 2017.