UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Leonardo Soares Paulino

DESENVOLVIMENTO DE UM DRIVER PARA COMUNICAÇÃO SÍNCRONA SOBRE IEEE1394

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. José Mazzucco Jr.
Orientador

Florianópolis, Fevereiro de 2003.

DESENVOLVIMENTO DE UM DRIVER PARA COMUNICAÇÃO SÍNCRONA SOBRE IEEE1394

Leonardo Soares Paulino

Esta dissertação foi julgada adequada para a obtenção do título de **Mestre em Ciência da Computação** especialidade **Sistemas de Computação** e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Fernando Álvaro Ostuni Gauthier

Banca Examinadora

_____

Prof. Dr. José Mazzucco Jr.

_____

Prof. Dr. Thadeu Botteri Corso

_____

Prof. Dr. Luis Fernando Friedrich

“Se assim foi, assim pôde ser;

Se assim fosse, assim poderia ser;

Porém como não é, não é. Isso é lógica”.

(Lewis Carroll).

“Se os fatos não combinam com a teoria,

mude os fatos”.

(Albert Einstein).

Dedicatória

A meus pais, Luiz e Vera.

A meus avós paternos, Manuel e Maria.

Em memória de meus avós maternos, Arthur e Odília.

À minha família.

Agradecimentos

Ao Prof. Dr. José Mazzucco Jr., pela paciência e confiança depositada neste trabalho.

Aos Professores Membros da Banca Examinadora, pelas contribuições feitas no intuito de aperfeiçoar este trabalho.

Aos amigos de mestrado, que estiveram presentes dentro e fora da Universidade, apoiando e incentivando a conclusão deste trabalho.

Aos colegas Karlos e Fernando, pelo auxílio durante a fase de implementação, o qual foi imprescindível.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta um estudo desenvolvido sobre o padrão IEEE1394 (Firewire, i.Link), com o objetivo de servir como base de conhecimento para o desenvolvimento de um *driver* de comunicação síncrona, utilizando-se esta mesma tecnologia.

O projeto de pesquisa teve como motivação maior a necessidade de diminuição do tempo de comunicação entre máquinas que executem processos paralelos e distribuídos, com confiabilidade e baixo custo. O Projeto *Crux* é, no momento, a fonte de requisitos para implementação, em nível físico, de primitivas bloqueantes de comunicação.

Desenvolvido para o sistema operacional Linux, o módulo SYNC1394 oferece funções de envio, recepção e gerência de mensagens. A partir destas funções, foi possível propor atualizações ao padrão IEEE1394, com intuito de se atingir as metas propostas. Como pré-requisitos para a compreensão deste trabalho estão conceitos de programação paralela, sistemas distribuídos, redes de computadores e do sistema operacional Linux.

Palavras-chave: Comunicação síncrona, IEEE1394, Firewire, Linux.

Abstract

This work presents a study about IEEE1394 (Firewire, i.Link) standard, in order to serve as knowledge base for the development of a synchronous communication driver, using this same technology.

The research project had as biggest motivation the necessity of reduction of communication time between machines that execute parallel and distributed processes, with trustworthiness and low cost. The Crux Project is, at the moment, the requirements source for implementation, in physical level, of blocking ommunication primitives.

Developed to the Linux operational system, the SYNC1394 module offers functions of sending, reception and message management. With these functions, it was possible to consider updates to IEEE1394 standard, with intention of reach the proposals goals. As prerequisite for the understanding of this work, these are concepts about parallel programming, distributed systems, computer networks and Linux operational system.

Keywords: Syncronous communication, IEEE1394, Firewire, Linux.

1 Introdução

A comunicação de dados é, atualmente, um dos principais problemas entre muitos que a ciência da computação tem como objetivo resolver, principalmente quando esse processo deve ser realizado de forma síncrona, rápida e confiável. Buscando aperfeiçoar modelos de comunicação síncrona, vem-se desenvolvendo novos componentes (hardware e software), os quais nem sempre atendem completamente às necessidades dos diversos segmentos de usuários.

Propor então, um modelo alternativo àqueles conhecidos e estudados, tornou-se motivação para o desenvolvimento desta pesquisa, tendo como ponto de partida o trabalho de Corso [COR99], onde percebemos a pertinência desta pesquisa, já que um modelo de multicomputador será a principal aplicação a utilizar esta implementação, na forma do núcleo de sistema do Acrux de Budag [BUD02]. Para o desenvolvimento dos trabalhos, utilizou-se componentes físicos e lógicos disponibilizados pelo Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Santa Catarina.

Desta forma, como meio físico, escolheu-se a arquitetura *Firewire*. Este nome - *Firewire*, foi cunhado pela Apple Computer Inc[1]. (sua criadora), e ainda permanece como sinônimo de IEEE1394, referência adotada oficialmente pelo órgão IEEE[2] (*Institute of Electrical and Electronics Engineers*) [IEE96]. Outros nomes foram atribuídos, por exemplo, i.Link (adotado pela Sony Semiconductors[3]).

O padrão IEEE1394 está fortemente relacionado com termos extremamente importantes, onde temos baixo-custo, tempo real (devido a largura de banda relacionada às transações isócronas), interfaceamento entre computadores, periféricos e produtos eletrônicos consumíveis, como: filmadoras, vídeo cassetes, impressoras, TVs, e câmeras digitais.

[1] http://www.apple.com
[2] http://www.ieee.org
[3] http://products.sel.sony.com/semi/

Quanto ao sistema operacional, escolheu-se o Linux, distribuição Red Hat 7.3, disponível na Internet, com código aberto, bem documentado e, principalmente, pelo fato de tratar-se de um sistema operacional em constante em evolução.

Como base de referência para a implementação deste módulo de comunicação síncrona sobre Linux, foram adotados os trabalhos de Anderson [AND99], Boszorm'enyi [BOSZ], Bouwhuis [BOU02], Collins [COL02] e Teener [TEE02], sobre a arquitetura IEEE1394 e a implementação de clusters sobre essa tecnologia e, de Rubini [RUB01] que se refere ao desenvolvimento de módulos no sistema operacional Linux.

Outras fontes de referência foram utilizadas e merecem serem citadas, como é o caso de: Russell [RUS00], Bovet [BOV02], Becker [BEC02], Maxwell [MAX00] e a API do Kernel Linux [LKAPI], que tratam de técnicas, ferramentas e interfaces de programação C para Linux; Hoffman [HOF02] que também estudou o padrão e a especificação do uso de códigos de resposta [1394S] do padrão IEEE1394 e afins e, ainda, Canosa [CAN99] por constituir-se numa referência acessível para quem pretende ter um primeiro contato com o padrão IEEE1394.

Pretendeu-se, ao final dessa pesquisa, apresentar um sistema de transferências síncronas de dados sobre um hardware que não possui tal característica. Por esse motivo, ao longo deste trabalho serão apresentados**:** no Capítulo 2, a topologia, os tipos de transferências, as camadas do protocolo, o funcionamento e o futuro do padrão IEEE1394**;** no Capítulo 3, a integração com o sistema operacional Linux e alguns projetos desenvolvidos sobre IEEE1394**;** no Capítulo 4**,** a descrição da proposta e implementação do módulo síncrono, além de testes de desempenho comparativos do padrão IEEE1394, do módulo e do padrão Ethernet.

O Capítulo 5 traz um estudo sobre a performance do sistema implementado, apontando correções e atualizações. Finalmente, o Capítulo 6 conclui o trabalho e descreve um conjunto de possíveis ajustes capazes de torná-lo, num futuro próximo, parte integrante do padrão IEEE1394.

2 O Padrão IEEE1394 (Firewire)

Este capítulo conduz à compreensão do embasamento teórico do trabalho, pois define o que vem a ser a tecnologia IEEE1394 e, também, o rumo que será seguido nos demais capítulos. Desta forma, uma boa compreensão deste texto é fundamental para o entendimento do sistema IEEE1394 e da proposta e desenvolvimento de um módulo para comunicação síncrona.

As informações aqui contidas são uma compilação dos trabalhos de Anderson [AND99], Canosa [CAN99] e Jain [JAI02] que descrevem a arquitetura IEEE1394 desde o nível físico até a lógica das transações. Também constam referências técnicas [IEE96, 1394O] e conhecimentos obtidos através do estudo dos códigos referentes aos sistemas aqui descritos.

2.1 A História do IEEE1394

Elevado ao status oficial de padrão IEEE em 1995, o padrão 1394-1995[4] surgiu na segunda metade da década de 80, através de pesquisas promovidas pela Apple Computers, responsável direta pela alcunha "*Firewire*", adotada mundialmente. Em 1994 essa mesma empresa liderou a criação da *1394 Trade Association* (da qual vieram participar: Microsoft, Intel, Philips e Compaq), com o objetivo de oferecer suporte ao desenvolvimento de sistemas computacionais e eletrônicos de consumo, que permitissem interconexão direta entre dispositivos multimídia.

O IEEE1394 consiste de um padrão para conexões de alta velocidade, enumeradas na Tabela 2.1, e de baixo custo para aplicação em periféricos, como: câmeras de vídeo e fotográficas, gravadoras, PCs, discos rígidos, entre outros, destacando-se algumas de suas aplicações para: armazenamento em massa, videoconferência, produção de vídeo, conexão de pequenas redes, impressoras de alta velocidade e equipamentos de entretenimento.

[4] Este trabalho irá se referir ao padrão IEEE1394-1995, que constitui a especificação básica, apenas como IEEE1394.

Tabela 2.1 – Velocidade de transmissão associada ao padrão IEEE.

Padrão IEEE	Velocidade de transmissão
1394	98 304 (100) Mbps
1394	196 608 (200) Mbps
1394	393 216 (400) Mbps
1394b	800 Mbps
1394b	1.6 Gbps*
1394b	3.2 Gbps*

Em relação às principais características desse padrão, temos:

- performance escalonável;
- *plug & play*;
- espaço de endereçamento de 356TB (terabytes) por nó;
- suporte para até 63 nós (em um único barramento);
- suporte para 1024 barramentos;
- modelo de software e hardware em camadas;
- suporte para dois tipos de transações;
- suporte para transações *peer-to-peer* e broadcast;
- detecção e tratamento de erros.

O padrão IEEE1394 descreve um espaço de endereçamento de 64 bits, registradores de controle, operações de leitura e escrita, cabos, sinalização, gerenciamento e distribuição de energia, além de outros requisitos necessários para o sistema.

O suplemento 1394a foi apresentado com intuito de complementar e corrigir alguns detalhes na versão anterior, além de definir melhoramentos capazes de aprimorar o

* Em desenvolvimento.

desempenho e adicionar novos recursos a esse padrão. Ainda, com o objetivo de obter-se valores mais expressivos, encontra-se em desenvolvimento a versão 1394b.

Dentro da aplicação acadêmica, com os protocolos atualmente implementados, o padrão encontra-se no patamar dos 400Mbps, sendo que uma nova versão capaz de alcançar 800Mbps foi recentemente introduzida no mercado.

2.2 A Estrutura da Arquitetura IEEE1394

A arquitetura IEEE1394 é formada por um conjunto de especificações e padrões, como pode ser observado de forma resumida na Figura 2.1, já que existem diversas outras especificações. Tal arquitetura[5] compreende uma variedade de dispositivos que podem ser implementados como sendo *nós 1394*. Alguns destes dispositivos podem ser reconhecidos como "unidades arquiteturais", pois definem detalhes de implementação como protocolos, registradores e outros.

Partindo-se da base desse conjunto temos o padrão IEEE1212 (também conhecido como ISO 13212) referente ao padrão de definição de registradores de controle e status, e que serve de suporte ao padrão IEEE1394.

Logo acima, na pilha, temos o protocolo IEEE1394 acompanhado de:

- P1394a – melhoramentos básicos, incluindo tanto correções quanto aprimoramentos:
    - define a distribuição de energia através do barramento, incluindo níveis de tensão, tipo de fonte, etc;
    - define os estados de energia, registradores CSR (*Control and Status Registers*) e a memória (ROM) de configuração;
    - define mecanismos de conservação de energia, como a implementação de características de pausa (*suspend*) e reinício (*resume*);
    - aceleração do processo de arbitragem do barramento.

[5] O "P" significa que está em fase de projeto, ou seja, não é um padrão finalizado.

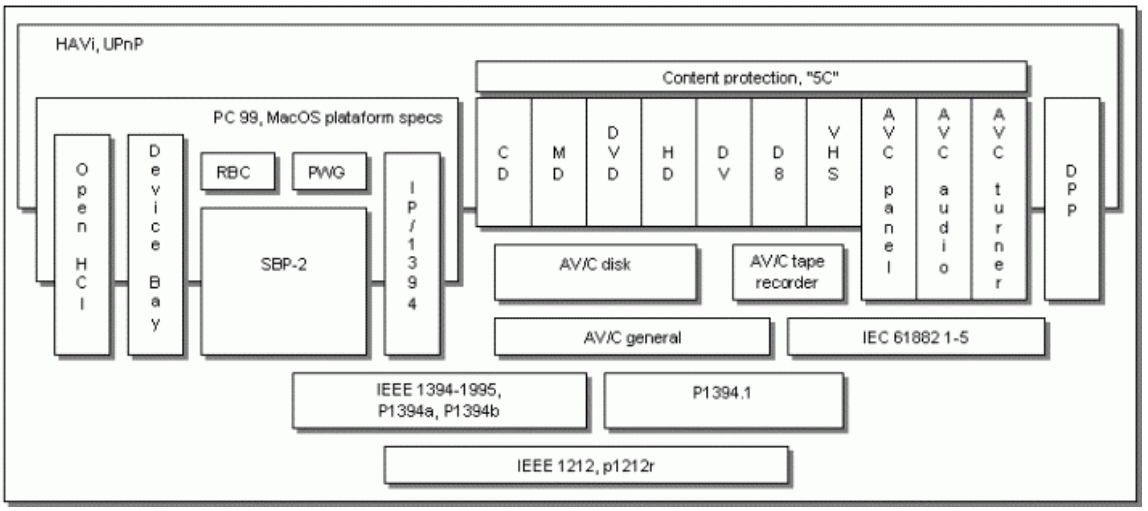

Figura 2.1 – Estrutura (resumida) da especificação IEEE1394.

- P1394b – projeto que busca elevar a arquitetura ao nível dos gigabits, permitir distâncias superiores a 100mts entre os nós, e reduzir o custo de produção através de técnicas alternativas de construção dos componentes;
- P1394.1 – especifica a conexão entre barramentos múltiplos, permitindo assim que mais de 63 nós se comuniquem, buscando manter a qualidade das sub-redes sem afetar o conjunto. Outro atributo deste projeto é prover condições para que reinicializações e tráfego de gerenciamento de um barramento não afetem os demais.

No patamar superior estão definições referentes às especificações baseadas em aplicações de consumo e computacionais, como conexão de periféricos (CD, HD, DVD e muitos outros). Além destas, temos especificações baseadas em PC para propósito geral, dividida em: comunicação com outros dispositivos (SBP-2 e IP1394) e padrões de interface (OHCI e DeviceBay) – e, de plataformas (referentes ao tipo de sistema operacional).

2.3 O Componente Hardware

Como foi citado anteriormente, IEEE1394 é um barramento serial, onde os dados trafegam “em fila”, diferentemente dos barramentos paralelos, SCSI, por exemplo. Este tópico tratará da parte física dessa arquitetura, descrevendo os nós, cabos, placas, topologia e endereçamento.

2.3.1 Módulo, Nó e Unidade

Inicialmente, define-se uma terminologia básica: um barramento IEEE1394 é constituído de **módulos**. Módulos representam um dispositivo físico ligado ao barramento, podendo cada um daqueles conter um ou mais **nós**. Nó representa a entidade lógica, sendo a ele atribuído um endereço e um espaço de endereçamento. Um nó pode ser compartilhado entre uma ou mais **unidades**, que são os componentes funcionais do nó e estão associadas aos protocolos de alto nível (unidade de vídeo, áudio, disco rígido, etc). A Figura 2.2 é um exemplo desta definição.

O nó IEEE1394 como entidade endereçável desta arquitetura de barramento serial, pode apresentar até 64 entidades (valores de 0 a 62, onde o valor 63 é reservado para *broadcast*) por barramento. Esta mesma arquitetura é capaz de conectar 1024 barramentos diferentes, tornando assim possível interligar, mesmo que indiretamente, até 65536 nós.

Durante o seu funcionamento normal, o módulo não é visto pelo software do usuário, estando este em contato com outros nós através dos protocolos associados às unidades funcionais. Tal situação pode ser contestada quando há falhas na interface do barramento ou um software específico, de diagnóstico ou mapeamento, por exemplo, está sendo executado.

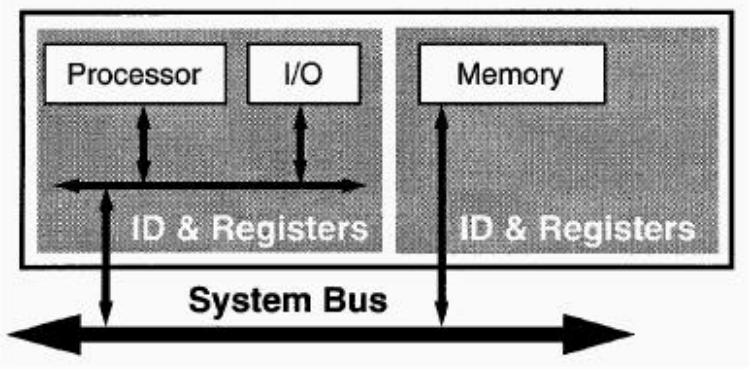

Figura 2.2 – Módulo, nós e unidades funcionais.

2.3.2 Topologia

O barramento IEEE1394 é baseado em comunicação *peer-to-peer* (nó a nó), ou seja, cada nó pode se comunicar diretamente com outro nó sem necessidade de intermédio de um dispositivo servidor, como é o caso do USB (*Universal Serial Bus*). Este sistema é um dos responsáveis pela alta velocidade obtida nas transações.

Ainda, tem-se que o barramento é auto-configurável, isto é, a adição e/ou remoção de um nó provoca um novo processo de configuração dos nós, fornecendo a cada um deles um endereço no respectivo barramento, podendo este ser o mesmo utilizado anteriormente ou não. Neste ponto é importante citar que o barramento não permite configurações onde esteja presente qualquer *loop*, ou seja, não pode haver um caminho fechado na rede que tenha como ponto de partida e chegada o mesmo nó. Esta restrição será comentada, novamente, no tópico sobre o processo de configuração.

A Figura 2.3 ilustra um barramento hipotético. Neste barramento temos seis nós conectados, onde o nó **E** por apresentar o maior número de conexões é referenciado com nó "raiz" (com exceção, já que o nó raiz pode ser "forçado", via o *driver* do sistema operacional). Já os nós **A**, **D** e **F** são nós "folhas", por onde se dá início o processo de configuração do barramento (tratado no decorrer do capítulo), e os nós **B**, **C** e **G** são ditos "ramos".

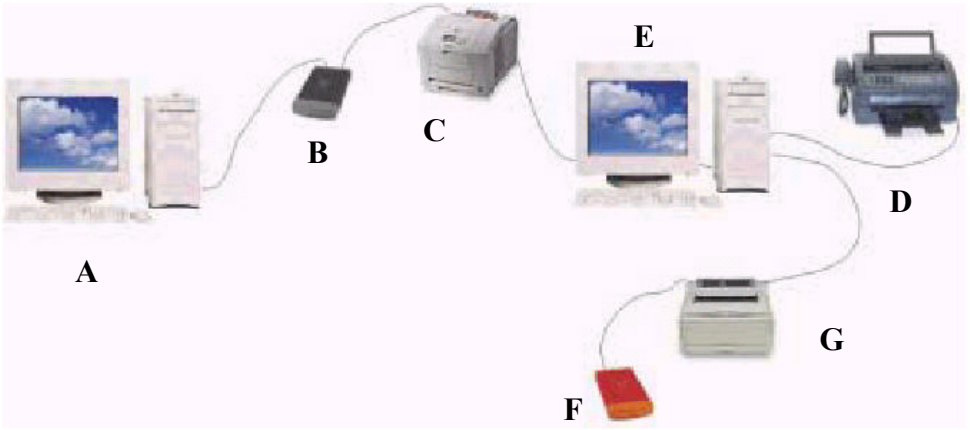

Figura 2.3 – Exemplo de barramento IEEE1394

O nó "raiz" é responsável por duas tarefas especiais: como controlador de ciclo ele provê os ciclos de relógio utilizados nas transferências isócronas e, também, é de sua responsabilidade a arbitragem pelo barramento, determinando qual será o dispositivo a usar o barramento.

Tendo em vista o lado físico, a topologia do barramento pode ser dividida em dois ambientes, um referente aos cabos e suas especificações e, o outro, referente à placa. Ambos serão abordados a seguir.

2.3.3 O Ambiente dos Cabos

A interconexão de dispositivos no padrão IEEE1394 segue através do ambiente de cabos uma rede não-cíclica, significando, como já abordado, que não são permitidos *loops*.

Em relação à parte física, o meio consiste de dois pares de condutores e um par para alimentação, conectando portas (conjunto de terminal, *transceiver* e uma lógica simplificada) de diferentes nós. Atualmente, estes cabos estão restritos a um comprimento máximo de 4,5m. A Figura 2.4 ilustra a seção transversal de um cabo IEEE1394.

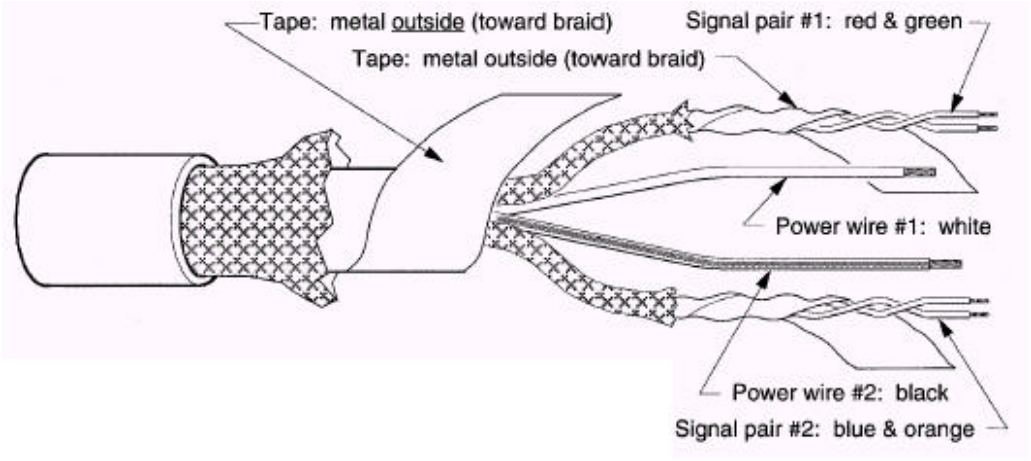

Figura 2.4 – Seção transversal do cabo IEEE1394.

Existem dois modelos de cabos IEEE1394, com 4 ou 6 pinos. O primeiro utiliza todos os fios para transferência de dados, enquanto o segundo possui dois fios extras para suprimento de força (8 – 40V, com até 1,5A) a dispositivos sem fonte própria de energia. Um terceiro tipo de cabo pode ser utilizado para funcionar como adaptador, sendo que cada uma de suas extremidades possui 4 e 6 pinos respectivamente.

A transferência de dados é *half-duplex*, isto é, o fluxo de dados existe em apenas um sentido por vez. Embora esta tecnologia utilize dois pares trançados, sua função se restringe a transferência diferencial dos dados, onde um par carrega os dados (cada fio carrega uma voltagem oposta, por exemplo, enquanto no fio A é +1V, no outro será –1V) e o outro controla o ciclo de relógio.

2.3.4 O Ambiente da Placa

Trabalhando desde o início, junto à Apple Computers, a Texas Instruments desenvolveu o primeiro controlador IEEE1394, exemplificado através da Figura 2.5, sendo que, atualmente, várias outras empresas disputam este mercado. Essas placas controladoras apresentam uma separação em interface física (PHY) e uma camada de enlace (o

verdadeiro controlador), ambas independentes, e que se comunicam através de uma interface de enlace.

A interface é formada por: oito linhas de sinalização de dados para transferência de dados entre unidades – todas as linhas são utilizadas somente por nós que trabalhem a 400mbps; duas linhas de sinalização de controle, para especificar o modo de transferência; uma linha de requisição para determinar o modo atual de transferência e, uma para o relógio do sistema. Além dessas, há linhas de isolam as duas unidades.

Normalmente, a interface física é o "gargalo" nos dispositivos IEEE1394, dado ao fato de existir o isolamento da camada de enlace, e o que os impede de alcançar velocidades ainda maiores.

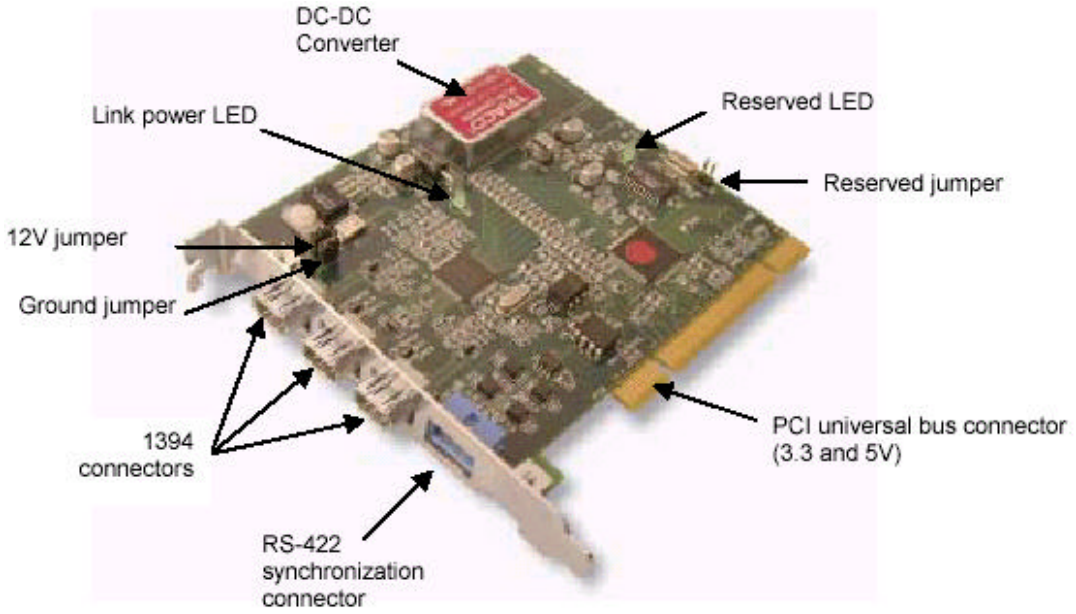

Figura 2.5 – Exemplo de placa controladora IEEE1394.

Dentro das funcionalidades associadas a este nível, camada de enlace, há:

- disponibilidade de transações: cada nó possui esta característica, caso contrário não se comunicaria com outros nós (não inclui transferências isócronas);

- disponibilidade isócrona: garante a possibilidade de executar transferências isócronas;
- disponibilidade para controle de ciclo: fornece o relógio de controle utilizado pelos canais isócronos;
- disponibilidade para gerenciamento de barramento: executa a configuração da topologia do barramento, ao receber os pacotes identificadores dos nós, ele determina a melhor estrutura possível.

2.3.5 Endereçamento

O espaço de memória definido no padrão IEEE1394 é baseado na arquitetura CSR, com endereçamento de 64 bits, sendo a cada nó atribuído um espaço de endereçamento de 48 bits. No total, temos um espaço de memória na casa dos 16 exabytes (18446744073709551616 bytes), o que representa 256 terabytes (281474976710656 bytes) por nó.

Nesta região reservada aos nós, os 16 bits superiores são utilizados o identificador do nó (*node ID*), tamanho suficiente para prover cerca de 64 mil nós. Já, este identificador, é subdividido em 10 bits específicos para o identificador de barramento (*bus ID*) e 6 bits para o identificador físico (*physical ID*).

O espaço de endereçamento é dividido, como mostra a Figura 2.6, em: espaço de registro inicial (*initial register space*), reservado para recursos da arquitetura CSR, registros específicos ao barramento serial e à área de identificador de ROM; espaço de unidade inicial (*initial units space*), reservado aos recursos específicos do nó; espaço privado (*private space*), para utilização local do nó; e, espaço de memória inicial (*initial memory space*).

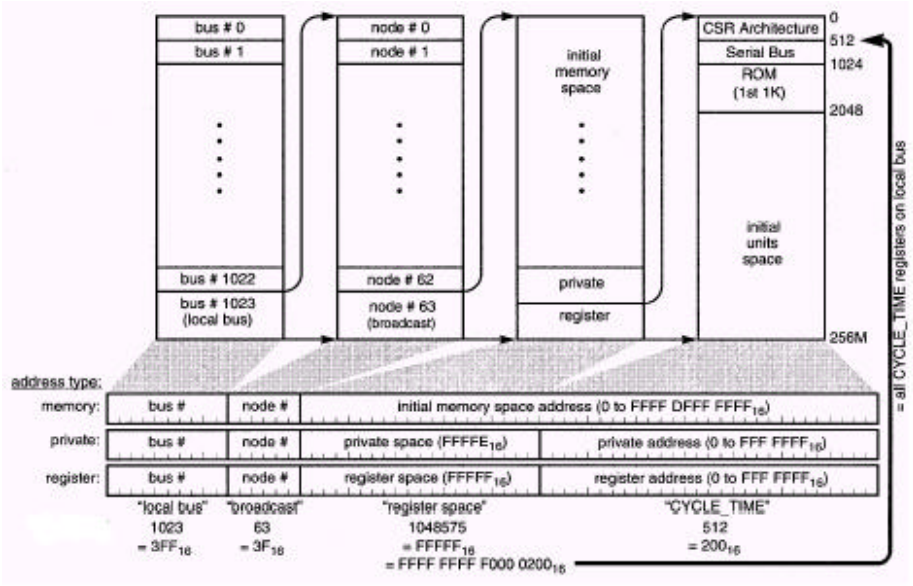

Figura 2.6 – Representação do espaço de endereçamento.

2.4 As Camadas do Protocolo

O protocolo IEEE1394 é dividido em camadas, sendo a cada uma delas definidas funções. As três principais camadas são:

- camada de transação: define o protocolo de requisição/resposta para executar as transações do barramento. Esta camada não inclui serviços isócronos;
- camada de enlace: fornece à camada superior um serviço de datagramas de reconhecimento, endereçamento, checagem de dados e empacotamento de dados. Além disso, fornece diretamente à aplicação, serviços de transferências de dados isócronos;
- camada física: traduz os dados em sinais elétricos e vice-versa, garante a exclusividade do barramento durante uma transação e define o mecanismo de interfaces para o barramento serial.

Ainda aparece o gerenciador do barramento serial, que fornece funções para controle dos nós e gerência dos recursos do barramento, e o gerente de recursos isócronos, que organiza a alocação de largura de banda e outros recursos. A Figura 2.7 mostra um esquema de organização das camadas.

2.4.1 Camada de Transação – *Transaction Layer*

A camada superior, de transação, é usada para efetuar transações assíncronas (mais de uma transação pode ser iniciada por vez), utilizando um mecanismo de requisição/resposta para tal. Dentre os tipos de transações temos: leitura (*read*), onde dados são lidos a partir de um endereço em um determinado nó, que os transfere de volta; escrita (*write*), onde dados são transferidos para um ou mais nós; e, trava (*lock*), onde dados são transmitidos a um outro nó, comparado com os dados de um determinado endereço e então, transferidos de volta.

Em relação aos tipos de transações, ainda podemos defini-las como sendo:

- separadas (*split*): ocorre quando um dispositivo não responde, no mesmo instante, a uma requisição, enviando um pacote de reconhecimento (*acknowledge*). Ao final de todo o processamento, o nó pede pela posse do barramento e envia para o requisitante as devidas informações. Neste meio tempo, o barramento pode ser utilizado por outros dispositivos;
- concatenadas (*concatenated*): quando um dispositivo consegue responder rapidamente a uma requisição, através da concatenação dos dados processados ao pacote de reconhecimento, eliminando a necessidade de uma nova arbitragem pelo barramento.
- unificadas (*unified*): no caso de transações de escrita, o reconhecimento também pode vir a ser a resposta para a requisição, caso o dispositivo consiga aceitar os dados em tempo suficiente. Isto elimina a necessidade de uma transação de resposta.

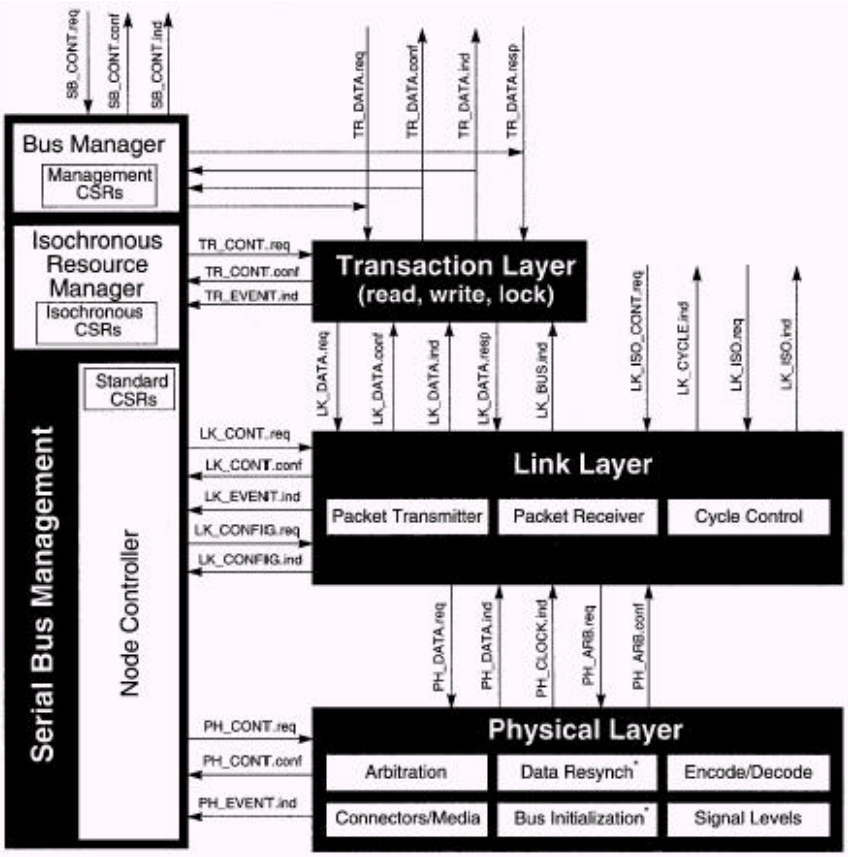

Figura 2.7 – Esquema das camadas do protocolo IEEE1394.

2.4.1.1 Serviços da Camada de Transação

As quatro primitivas de serviço oferecidas por esta camada são:

- requisição (*request*): inicia uma transação;
- indicação (*indication*): notifica da chegada de uma requisição;
- resposta (*response*): retorna o estado ou, até mesmo, dados para o nó que fez a requisição;
- confirmação (*confirmation*): notifica a chegada de uma resposta.

2.4.2 Camada de Enlace - Link Layer

Esta camada está entre as de transação e física, sendo a ela atribuída a função do serviço de entrega dos pacotes de dados *half-duplex*. Algumas de suas principais atribuições são: checar o campo CRC (*Cyclic Redundancy Check*) para verificar a integridade dos pacotes, além de calculá-lo (o CRC) e adicioná-lo aos pacotes; enviar e receber dados isócronos e, examinar o cabeçalho dos pacotes para determinar o tipo de transação que está sendo executada.

Sobre este serviço de entrega de pacotes, podemos destacar duas sub-ações:

- sub-ação assíncrona: uma certa quantidade de dados e vários bytes de informação, da camada de transação, são transferidos para um determinado endereço. Há espera por confirmação;
- sub-ação isócrona: certa quantidade de dados são transferidos durante intervalos regulares. Não há espera por confirmação.

Em uma camada de enlace espera-se encontrar a interface PHY, um mecanismo de geração e checagem de CRC, filas FIFO (*First-In First-Out*) de transmissão e recepção, registradores de interrupção, uma interface para o dispositivo e, no mínimo, um canal DMA (*Direct Memory Access*).

2.4.2.1 Serviços da Camada de Enlace

Os serviços oferecidos por esta camada são:

- requisição (*request*): inicia uma transação para outro dispositivo;
- indicação (*indication*): notificação da chegada de pacote originado em outro dispositivo, em resposta;
- resposta (*response*): transmissão de um reconhecimento pelo dispositivo de resposta;

- confirmação (*confirmation*): notifica a chegada de um pacote de reconhecimento.

2.4.3 Camada Física - Physical Layer

A camada física (PHY) é responsável pela transmissão e recepção de bits de dados, arbitragem pelo barramento e, também, prover uma interface mecânica e elétrica, ou seja, esta camada fornece a conexão com os demais nós do barramento, os quais podem possuir mais de um conector, o que possibilita uma grande variedade de topologias.

Cada uma dessas conexões (portas) retransmite os bits que chegaram pelas outras (re-sincronização de dados), trabalhando como um repetidor. Outra função da camada física é a inicialização do barramento, reconfigurando a topologia do barramento a cada mudança detectada.

2.4.4 Camada de Gerenciamento

Responsável pelo gerenciamento da arquitetura IEEE1394, esta camada que pode estar distribuída por vários nós ou coexistir em um único, implementa o controlador de ciclo, gerenciador de barramento e de recursos isócronos.

2.4.4.1 Controlador de Ciclo – *Cycle Master*

O controlador de ciclo é responsável por iniciar os ciclos de transmissão a cada 125ìs, sendo esta tarefa atribuída ao nó raiz. Caso um nó que não possua a capacidade de executar tal tarefa tornar-se o nó raiz, o barramento é reinicializado, e um novo nó é escolhido até que esta condição seja atendida.

2.4.4.2 Gerenciador de Barramento – *Bus Manager*

O gerenciador de barramento apresenta várias funções:

- manutenção do mapa de velocidade, indicando qual velocidade de transmissão deve ser utilizada entre dois ou mais nós;
- manutenção da topologia, definindo o melhor percurso para envio de dados;
- otimização do tráfego no barramento, permitindo o melhor aproveitamento de banda;
- gerenciamento de energia.

Também podemos considerar como função desse dispositivo, a busca por um nó capaz de executar a tarefa de controlador de ciclo.

2.4.4.3 Gerenciador de Recursos Isócronos – Isochronous Resourses Manager

O gerenciador de recursos isócronos através da implementação de registradores (*Bus Manager ID Register*, *Bus Bandwidth Allocation Register*, e *Channel Allocation Register*), como o nome já indica, é responsável por:

- alocação de banda isócrona, garantindo assim a velocidade da transmissão;
- alocação dos números dos canais, controlando as transmissões;
- seleção do controlador de ciclo, cooperando com o gerente de barramento.

2.5 Tipos de Transferências

O protocolo IEEE1394 aceita dois tipos de transferências de dados, assíncrona e isócrona. O primeiro tipo, o assíncrono, consiste na transmissão de pacotes de dados com espera por confirmação. Já o segundo tipo, o isócrono, baseia-se no envio de um fluxo de dados através de um canal e sem espera por confirmação. A Figura 2.8 ilustra um ciclo de

transmissão, onde cerca de 80% dos 125ìs deste ciclo são reservados para transferências isócronas, e os demais 20% para as transferências assíncronas.

Antecipando o processo de cada transação, há necessidade de se alocar o barramento. Esse passo chama-se **arbitragem**, o qual é executada pelo nó raiz, escolhido durante a configuração do barramento. Descrevem-se, a seguir, cada um destes tipos de transferências de dados e o processo de arbitragem.

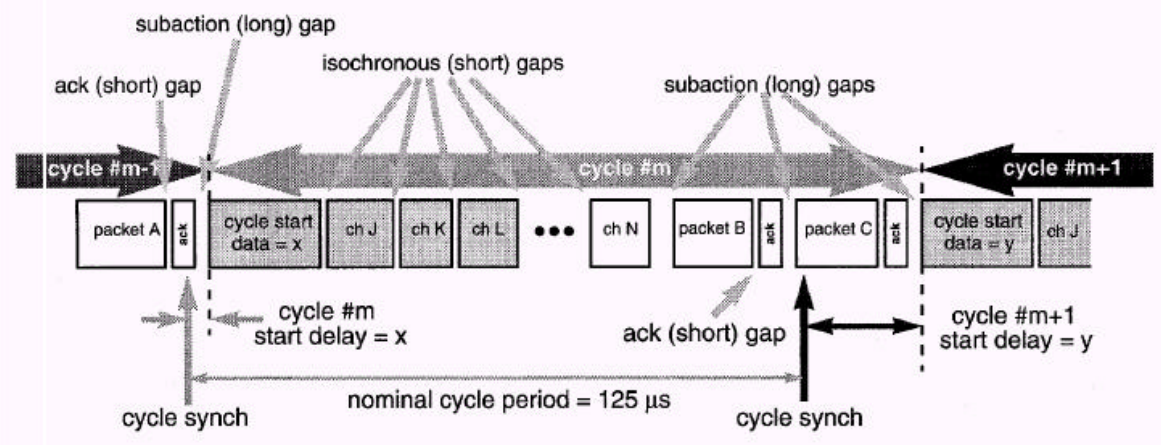

Figura 2.8 – Ciclo de transmissão do protocolo IEEE1394.

2.5.1 Arbitragem

O processo de arbitragem ocorre quando a configuração do barramento está completa e as operações do mesmo estão prontas para ter início. Primeiramente, há necessidade de se definir o que vem ser um ciclo. **Ciclo** é uma fatia de tempo (período) equivalente a 125ìs, determinado pelo nó raiz do barramento.

Cada ciclo é iniciado por um pacote, enviado por broadcast, a fim de sincronizar todos os dispositivos no barramento. Este é seguido por um pacote de início de ciclo, indicando que os nós que desejem efetuar uma transmissão estão liberados para indicar sua intenção. A **arbitragem** consiste na indicação ao nó pai desta intenção de transmitir, que este por sua vez transmitirá ao seu nó pai, sucessivamente, até o nó raiz. Assim como os

demais protocolos que utilizam algum tipo de disputa pelo barramento, a requisição que primeiro chegar ao nó raiz será a privilegiada.

Dentro do tempo de cada ciclo, as transferências isócronas são as primeiras a terem acesso ao processo de arbitragem e, também, são as que ocupam a maior parte do processo - 80%. Assim, é interessante ressaltar que, todos os nós que desejarem transmitir um pacote isócrono, terão acesso a um pedaço reservado desta fatia de tempo. Ao final de todas as transferências isócronas, o processo de arbitragem se volta para as transferências assíncronas, seguindo o mesmo esquema anteriormente descrito, exceto por esta divisão da fatia de tempo.

Para evitar que os nós mais próximos ao nó raiz, ou ele próprio monopolize o barramento, faz-se uso de um artifício, o *fairness interval*. Neste período os nós mais próximos ao nó raiz que já efetuaram sua transmissão habilitam um *bit* de controle, no intuito de não voltarem, por um determinado tempo, a requisitar o barramento. Dessa forma, os nós mais distantes têm chance de, finalmente, executar suas transferências.

O projeto P1394a introduz três novos tipos de arbitragem:

- *acknowledged accelerated arbitration*: quando um nó termina de efetuar resposta a uma requisição, e este também deseja transmitir, imediatamente transmite seu pacote, sem passar pelo processo de arbitragem;
- *fly-by arbitration*: quando um nó está retransmitindo um pacote a outro nó, este pode anexar o seu pacote ao que está processando. Este pacote concatenado deve ser transmitido na mesma velocidade do pacote original, e funciona somente com pacotes isócronos e de confirmação;
- *token-style arbitration*: a partir de um grupo de nós cooperantes, o nó que vencer a arbitragem passa a um nó mais distante da raiz o direito pelo barramento. Na medida que este nó transmite, todos os nós que estão no meio deste percurso utilizam-se do processo de arbitragem anterior para que também possam transmitir.

2.5.2 Transferências Assíncronas

O primeiro modo de transferência apresentado é o assíncrono, utilizado em aplicações que necessitem da garantia de entrega dos dados transmitidos, como por exemplo, um sistema de armazenamento de dados. A integridade dos dados é verificada pelo barramento, através do campo CRC e pacotes de confirmação. Esta característica impõe ao padrão um determinado limite de tamanho aos pacotes, sendo eles enumerados na Tabela 2.2.

Tabela 2.2 – Velocidade de transmissão para pacotes assíncronos.

Velocidade de transmissão	**Tamanho máximo por pacote (bytes)**
100 Mbps	512
200 Mbps	1024
400 Mbps	2048
800 Mbps	4096
1.6 Gbps	8192
3.2 Gbps	16384

Todas as transações assíncronas são executadas conforme o protocolo requisita/responde (*request/response*) que divide cada uma delas em duas sub-ações:

- requisita (*request*): transfere o endereço, o comando e dados (quando for o caso) do nó requisitante ao nó que irá responder;
- responde (*response*): retorna o status da transação e, eventualmente, pode retornar dados.

A Figura 2.9 ilustra um exemplo genérico de transferência assíncrona, onde esta pode ser definida em três tipos de transações básicas, são elas:

- *reads:* O nó requisitante deseja um dado armazenado em outro nó. O primeiro envia um pacote primário ao nó que contém a informação desejada

e, este último, retorna um pacote contendo o status da transação e o dado requisitado;

- *writes:* O nó requisitante pretende escrever um determinado valor em outro nó. O requisitante envia um pacote primário com os dados necessários, sendo do encargo do segundo nó retornar um pacote de confirmação da transação*;*
- *locks*: Um determinado nó deseja executar uma operação atômica sobre outro nó. O nó que pretende executar a operação envia um pacote contendo informações necessárias para bloquear um determinado nó alvo, enquanto este garante o acesso exclusivo ao requisitante;

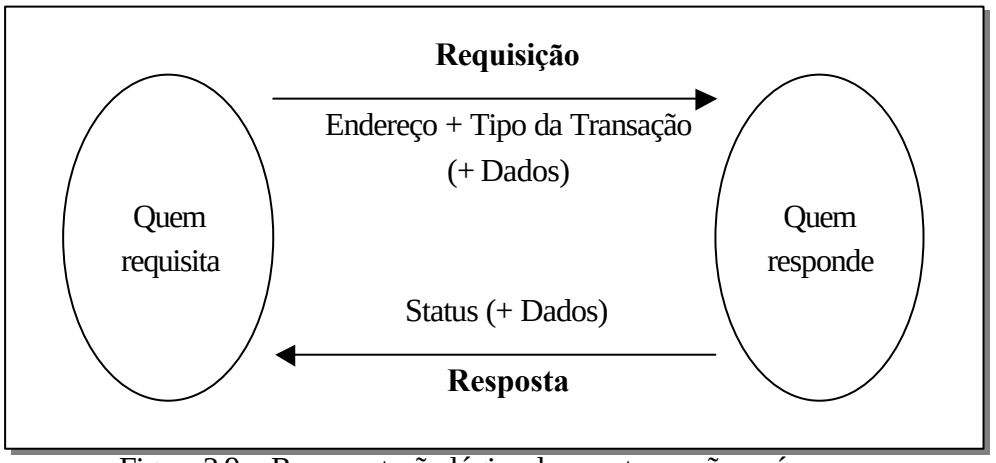

Figura 2.9 – Representação lógica de uma transação assíncrona.

As Figuras 2.10 e 2.11 são exemplos típicos de pacotes primários e de confirmação, respectivamente, para transações assíncronas. Ainda podemos reafirmar como suas principais características: confirmação na entrega de pacotes e tratamento por pacotes (cada pacote assíncrono recebido é diretamente enviado ao seu tratador).

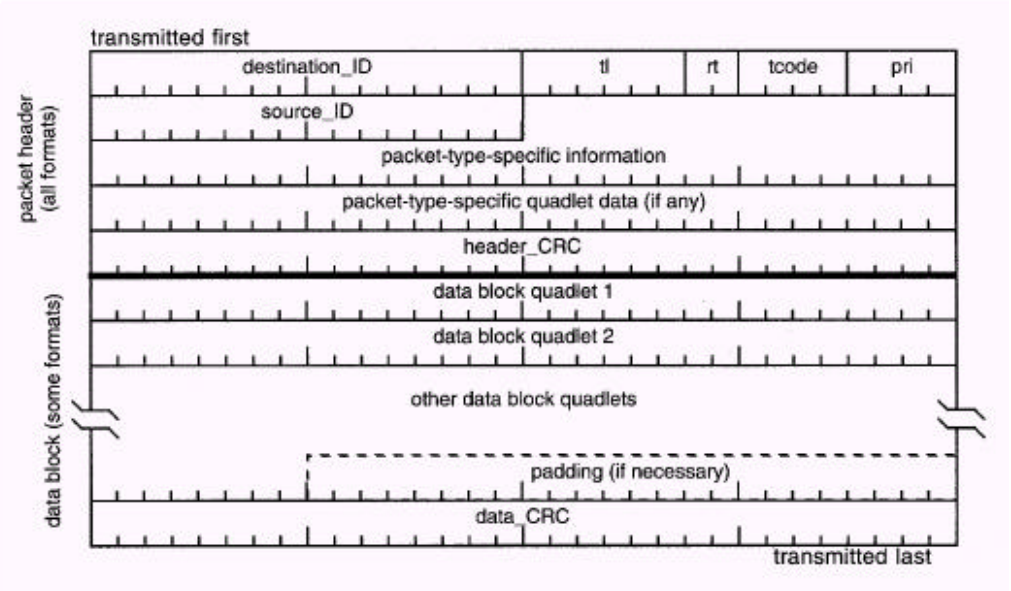

Figura 2.10 – Exemplo de pacote assíncrono em bloco.

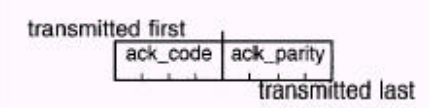

Figura 2.11 – Exemplo de pacote de confirmação.

2.5.3 Transferências Isócronas

As transferências isócronas trabalham com o preceito de entrega constante de dados, ou seja, a garantia de entrega e a integridade dos dados são substituídas pela taxa de entrega (velocidade de transmissão). A Tabela 2.3 aponta o tamanho máximo que cada pacote pode carregar em uma determinada velocidade de transferência. Como exemplo de aplicações isócronas, temos: transmissões de vídeo e som, aplicações multimídia em geral, etc.

A Figura 2.12 representa uma transação isócrona, onde o nó requisitante faz uso de um canal de transmissão pré-definido, o tipo da transação e os dados associados. Cada nó do barramento tem direito a uma fração dos 125µs fornecidos a cada ciclo.

Tabela 2.3 – Velocidade de transmissão para pacotes isócronos.

Velocidade de transmissão	Tamanho máximo por pacote (bytes)
100 Mbps	1024
200 Mbps	2048
400 Mbps	4096
800 Mbps	8192
1.6 Gbps	16384
3.2 Gbps	32768

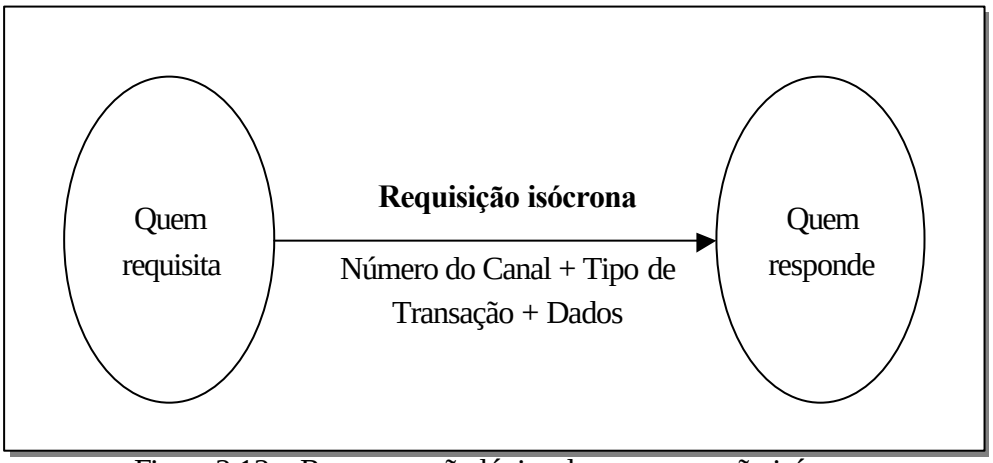

Figura 2.12 – Representação lógica de uma transação isócrona.

Uma transação isócrona pode ter um ou mais dispositivos alvo (permite *multicast*), sendo cada transferência associada a um determinado canal (existem 63 canais disponíveis). Antes de iniciar o processo, a aplicação deve alocar uma largura de banda para sua operação, através do nó que opere como gerente de recursos isócronos.

Como principais características das transferências isócronas, temos: velocidade de transmissão garantida e tratamento por buffer (os dados são tratados em lote, apenas quando o buffer de recepção está completamente cheio). A Figura 2.13 ilustra os campos de um pacote isócrono.

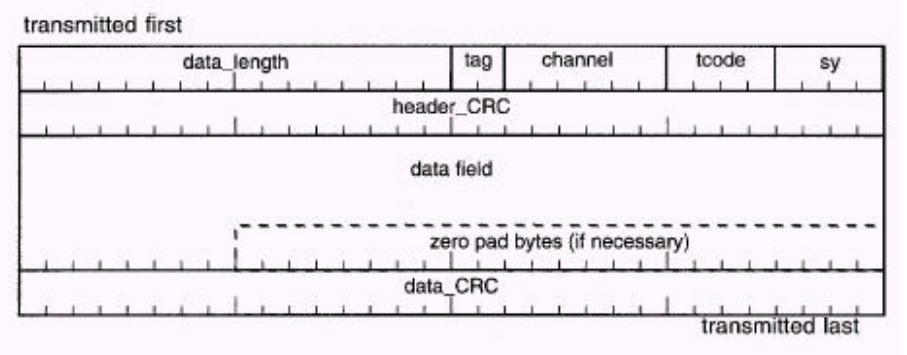

Figura 2.13 – Exemplo de pacote isócrono.

2.6 Processo de Configuração

Executado a partir da camada física, o processo de configuração consiste na transformação de uma topologia física em uma estrutura de árvore lógica, com o nó raiz em seu topo. Este processo é iniciado a cada alteração no barramento (adição ou remoção de dispositivos), sendo dividido em três fases: inicialização do barramento, identificação de árvore e auto-identificação.

2.6.1 Inicialização do Barramento – *Bus Reset*

Esta etapa consiste em limpar qualquer informação de topologia nos nós, sem afetar as definições de recursos isócronos. Tal tarefa é posta em execução, por um nó, atribuindo às linhas de sinalização (TPA e TPB) o valor lógico "1". Ao detectar a presença desta condição em uma de suas portas, o nó retransmite o sinal para suas demais portas, assumindo um estado de espera (*idle*) durante um período de tempo, a fim de permitir que o sinal se propague para os demais nós.

2.6.2 Identificação de Árvore – *Tree Identification*

Neste ponto, a topologia do barramento é definida. Durante este processo, a topologia física é mapeada, diretamente, em uma estrutura lógica, sendo um dos seus resultados finais a eleição do nó raiz.

Realizado em microssegundos, o processo de identificação de árvore inicia-se, genericamente, com uma sinalização dos nós folhas aos seus respectivos pais (*Parent Notify*). No momento que um nó ramo recebe este sinal, ele marca a porta pela qual a recebeu como contendo um dispositivo filho, retornando através delas um sinal em resposta (*Child Notify*). Os nós folhas ao receberem este sinal marcam suas portas como portas pai, removendo assim a sua sinalização inicial e confirmando sua condição de nó filho.

Assim que os nós folha terminam o processo de identificação, o passo se repete com os nós que estão na camada superior, sinalizando aos seus nós pai (*Parent Notify*) e aguardando uma confirmação (*Child Notify*). Seguindo esta lógica até a última camada, o nó que apresentar todas as suas portas marcadas com nós filho, tornar-se-á o nó raiz. Com o transcorrer do processo, números são aplicados às portas, os quais serão utilizados na fase subseqüente. O exemplo de um resultado desta etapa é mostrado na Figura 2.14.

As disputas pela condição de nó raiz são resolvidas através do uso de *timers* randômicos. O nó que requer tornar-se nó raiz via software, por exemplo, o faz utilizando-se de um artifício, ou seja, prorrogando sua participação no processo de sinalização.

A Figura 2.15 ilustra um estado topológico inconsistente, devido a presença de um *loop.* Tal inconsistência decorre do fato de que o processo de identificação de árvore não consegue tratar este modelo, pois entra em ciclo. Após um determinado tempo (*time-out*), o protocolo reconhece este estado não-válido e retorna à camada de tratamento (*driver*) um sinal indicando erro.

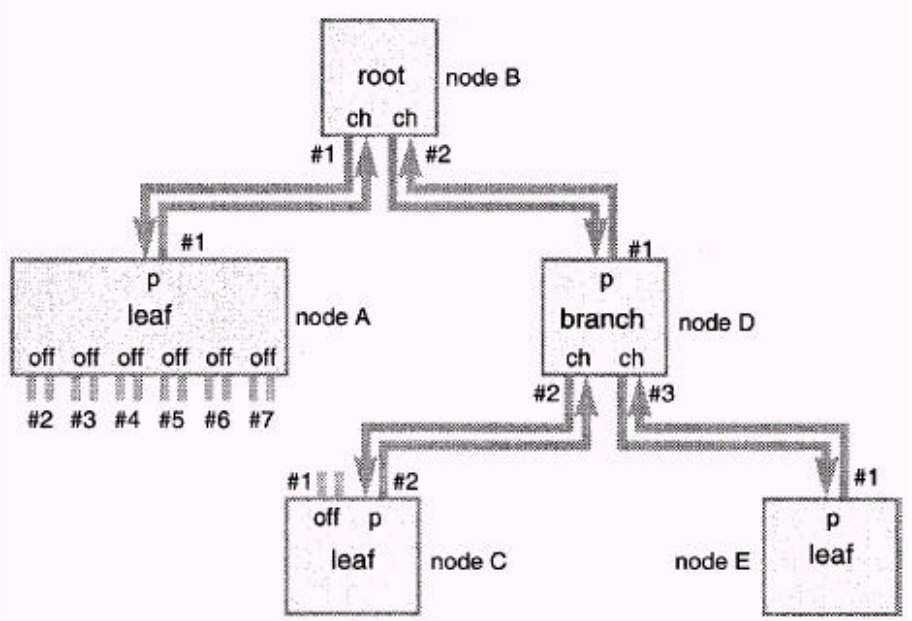

Figura 2.14 – Exemplo de processo de identificação de árvore.

2.6.3 Auto-Identificação – *Self Identification*

Uma vez terminada a etapa de identificação de árvore, tem início a fase de auto-identificação, que consiste na atribuição de identificadores físicos (*physical ID*) para cada nó do barramento. Outro resultado é a troca, entre nós vizinhos, de informações sobre suas velocidades de transmissão.

A fase inicia-se com o nó raiz enviando um sinal de confirmação de arbitragem (*arbitration grant*) pela sua porta de menor numeração, quando este sinal alcançar um nó ramo, este irá repassá-lo por sua porta de menor numeração, até que o sinal chegue em um nó folha. O nó folha, por sua vez, atribui a si mesmo o identificador físico **0** e retorna um pacote de auto-identificação (*self ID*). Então, o pacote contendo a identificação é enviado para todos os nós do barramento, fazendo com que estes incrementem seus contadores de auto-identificação.

Repetindo-se este mesmo processo para todos os demais nós, os que ainda não foram identificados, é possível atribuir os valores de seus identificadores físicos. Como pode ser observado, o nó raiz será, sempre, o nó com o maior identificador físico.

Durante as transferências que ocorrem no processo, os dispositivos expõem a máxima velocidade com que são capazes de transmitir, o que leva ao fato de que, qualquer transação entre dois nós deverá ser realizada com a velocidade do dispositivo mais lento em seu percurso.

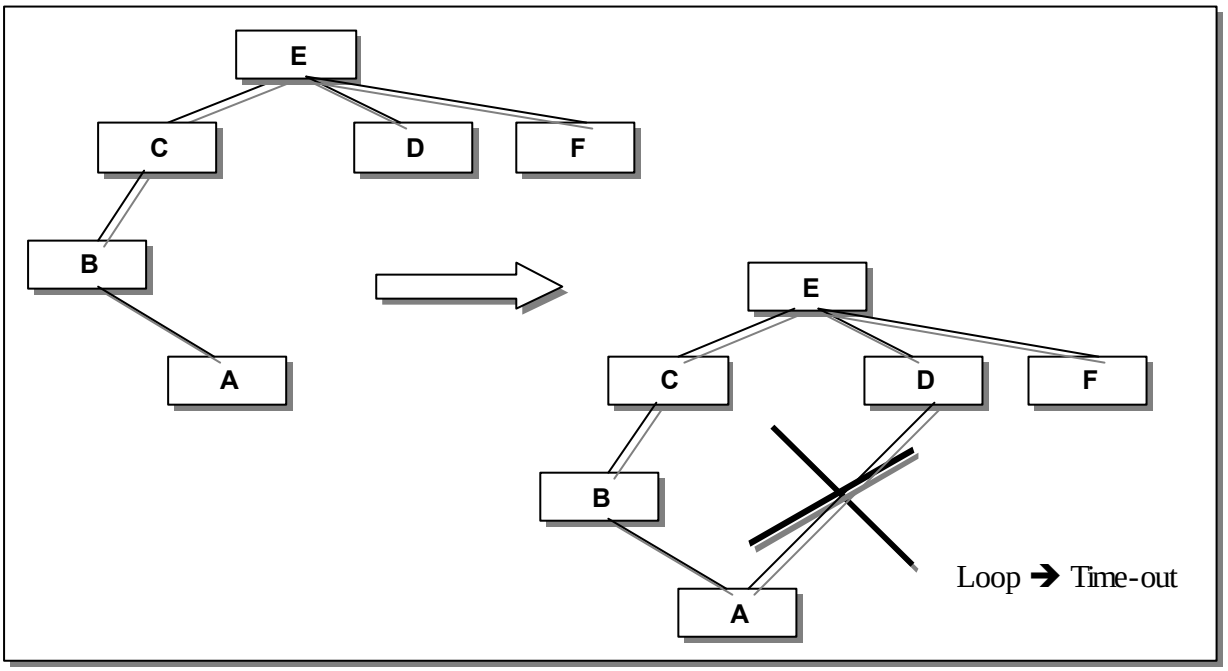

Figura 2.15 – Topologia válida *versus* topologia inconsistente.

3 O Sistema IEEE1394

Estando a arquitetura IEEE1394 definida podemos, então, descrever a camada de software envolvida nesse trabalho. Inicialmente, explicaremos como configurar o sistema operacional, seguido de uma breve descrição dos seus principais *drivers*. Finalizando, mostraremos alguns projetos em andamento sobre o padrão IEEE1394.

Assim sendo, o sistema operacional escolhido para este trabalho foi o Linux, distribuição Red Hat 7.3, por esse apresentar as características adiante descritas:

- sistema multitarefa preemptivo;
- proteção de memória;
- gerenciamento de memória virtual;
- velocidade e estabilidade;
- e, principalmente, pelo suporte ao padrão IEEE1394.

Mais detalhes sobre a estrutura, configuração e programação do sistema operacional Linux podem ser obtidos através dos trabalhos de Goldt [GOL95], Becker [BEC02], Rubini [RUB01], Bovet [BOV02], Maxwell [MAX00] e Russell [RUS00] e nas interfaces de programação Linux [GKAPI, LKAPI].

3.1 Linux - Módulo Experimental

Inicialmente, **módulo** é uma porção de código encarregada de exercer um conjunto pré-definido de funções, a fim de dar suporte ao kernel do sistema operacional. O sistema operacional Linux é composto de vários *drivers* que podem vir a ser tratados como módulos do sistema, ou seja, podem ser carregados, utilizados e retirados da memória principal após terminar a execução de sua tarefa, sem que venham, necessariamente, a interferir no funcionamento de outro módulo. A partir daí, podemos caracterizar o que vem a ser um módulo experimental.

Módulo experimental (de rede, arquivo de sistemas, protocolos, etc...) é aquele que ainda está em fase de desenvolvimento, onde a sua funcionalidade e estabilidade não são suficientes para uso geral ou ainda não foi submetido a uma bateria de testes significativa. Ele também é conhecido, comumente, como versão "alfa". Todo suporte à arquitetura IEEE1394 é feito por módulos experimentais.

3.1.1 Como Configurar o Kernel Linux

Para iniciar o processo de utilização das placas IEEE1394 e efetuar a configuração das mesmas, tem-se que seguir uma rotina de procedimentos concatenados. Este tópico se encarrega de ilustrar tal processo, sendo de responsabilidade dos usuários obter e instalar o sistema operacional Linux, preferencialmente, com versão de kernel posterior a 2.4.18.

Versões de kernel anteriores[6] àquela, também podem ser utilizadas mas, tem-se que levar em consideração as devidas diferenças de configuração. Outras versões, posteriores a 2.5, apresentam alterações na implementação dos módulos IEEE1394, os quais não são abordados e, também, não foram utilizados no desenvolvimento deste trabalho.

Estando, então, o Linux devidamente instalado, assim como a placa IEEE1394, o processo de configuração[7] do suporte às placas segue a seguinte rotina:

1. Acessar o diretório */usr/src/linux*;
2. Digitar *make xconfig* ou *make menuconfig*, para configurar o kernel;
3. Selecionar a opção *Code maturity level options*, para permitir ao Linux trabalhar com módulos de nível experimental. A Figura 3.1 ilustra este passo;

[6] Mais detalhes em http://www.linux1394.org/start_req.html.
[7] Em modo superusuário.

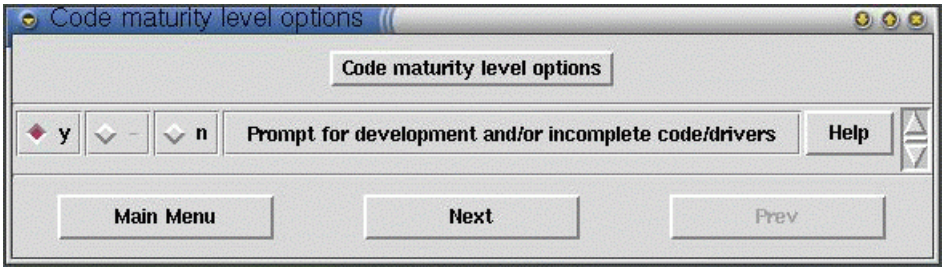

Figura 3.1 – Ativando o *Code maturity level options*.

4. Selecionar *Main Menu*;
5. Selecionar a opção *IEEE1394 (Firewire) Support*, para configurar o módulo de baixo nível. A Figura 3.2 e 3.3 mostram quais opções devem ser selecionadas para configurar o *driver* como módulo ou parte do kernel, respectivamente;

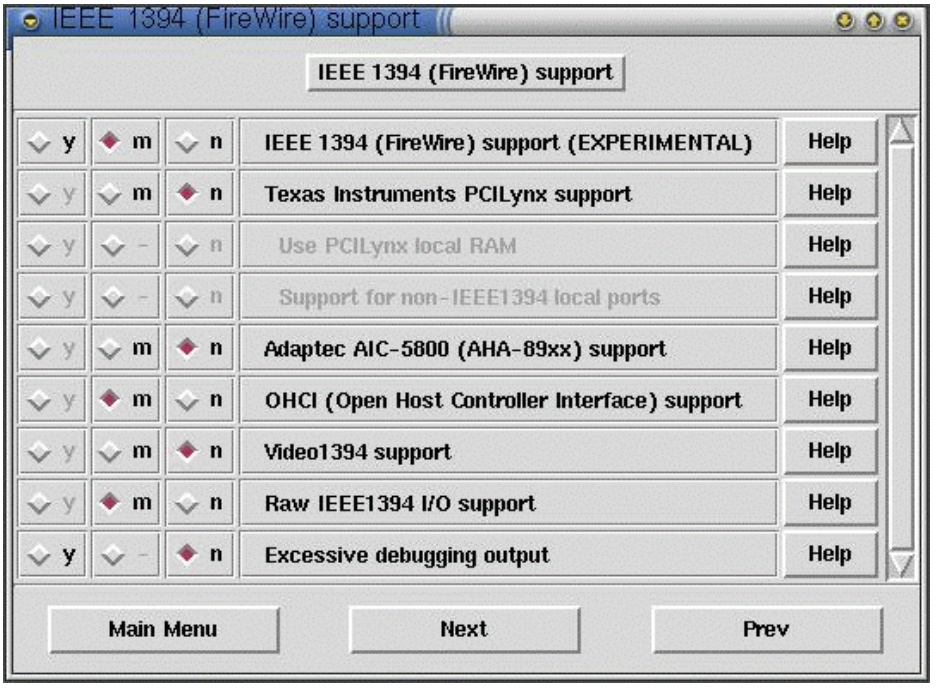

Figura 3.2 – Configurando o *IEEE1394 (Firewire) Support como módulo*.

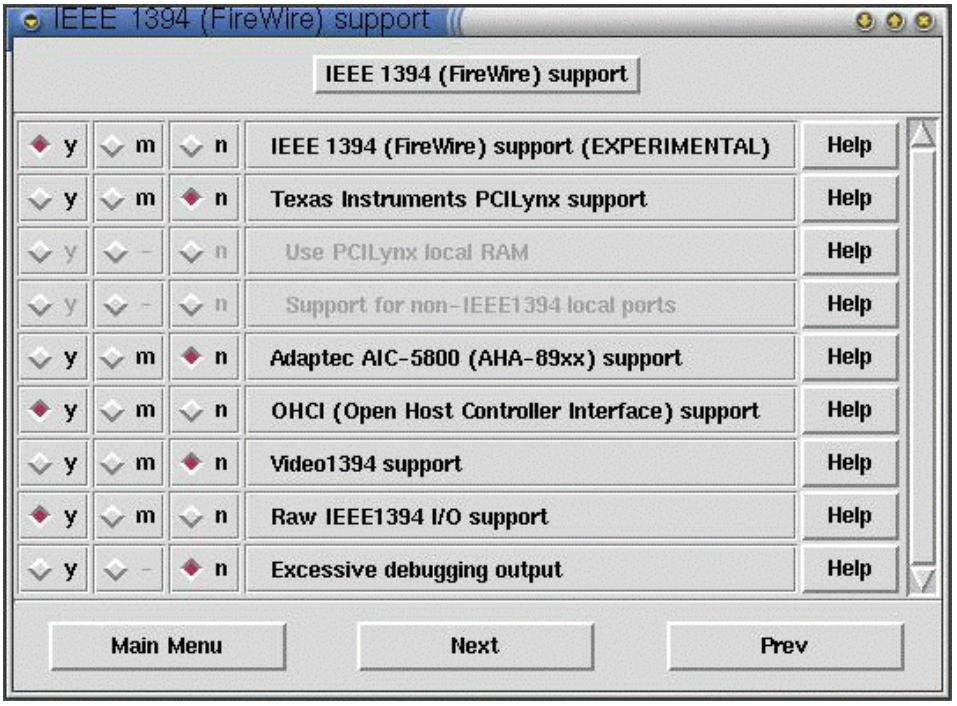

Figura 3.3 – Configurando o *IEEE1394 (Firewire) Support* como parte do kernel.

6. Selecionar *Main Menu*;
7. Selecionar *Save and Exit*.

Neste momento, as configurações do kernel já estão prontas, mas é necessário recompilá-lo. Para isto, basta:

1. Reconstruir o kernel, módulos e suas dependências com os comandos *make dep, make bzImage* e *make modules;*
2. Reinstalar o kernel e os módulos com make *modules_install,* seguido de *cp arch/i386/boot/bzImage /boot/vmlinuz-2.4_ieee1394.* E, finalmente, reconfigurar o LILO (gerenciador de *boot*), para o novo sistema de arranque com suporte a IEEE1394.

3.1.2 Módulos IEEE1394, OHCI1394 e RAW1394

Estando o sistema devidamente configurado, pode-se navegar pelos arquivos que compõem o *driver* de baixo nível. Eles estão divididos em três módulos principais, sendo estes compostos de um conjunto de arquivos escritos em C ANSI (padrão utilizado no desenvolvimento do kernel e módulos Linux). Estes módulos são:

- IEEE1394, que tem interação direta com o hardware, fornecendo primitivas básicas de configuração;
- OHCI1394, módulo que fornece as principais interfaces de conexão e transmissão, utilizando chamadas de sistema e podendo ser acessado, apenas, por outros módulos;
- RAW1394, que é o módulo mais próximo ao usuário e que fornece a interface de gerenciamento e transações às aplicações em nível de usuário.

Entre as funções tidas como desenvolvidas, estão: transmissão e recepção de requisições assíncronas, transmissão e recepção de respostas assíncronas, recepção isócrona, recepção isócrona para função *mmap* DMA (acesso direto a memória compartilhada), configuração da ROM (placa) e transmissão isócrona[8] (em fase de testes).

Ainda, entre as funções na espera por implementações, estão: transações (envio e recepção) de pacotes de fluxo assíncrono e recuperação de erros de DMA.

Novas funções e interfaces estão sendo desenvolvidas neste instante, fazendo do IEEE1394 sobre Linux, alvo de interesse nas mais diversas áreas de atuação.

3.1.3 Principais estruturas lógicas IEEE1394

Algumas variáveis, estruturas lógicas e funções presentes neste sistema merecem ser apresentadas formalmente, pois estas são as principais responsáveis pela caracterização da

[8] No momento que este trabalho estava sendo escrito, uma nova interface de transmissão isócrona foi apresentada, estando ela disponível em http://linux1394.org.

arquitetura, além de que, são de suma importância para o desenvolvimento do módulo síncrono e das modificações no código a serem propostas.

As primeiras variáveis a serem descritas são CSR_FCP_COMMAND e CSR_FCP_RESPONSE, responsáveis por mapear o endereço lógico dos registradores físicos de transmissão de pacotes assíncronos. São utilizadas como parâmetros na função de envio, definindo assim o contexto da mensagem enviada.

A estrutura HPSB_HIGHLEVEL_OPS associa as interrupções de hardware aos seus devidos procedimentos de tratamento, referentes a: adição e remoção de nós, inicialização do barramento, e recepção de pacotes assíncronos e isócronos. Utilizada no gerenciamento dos nós do barramento.

Continuando, temos a estrutura que é, provavelmente, a mais importante, HPSB_PACKET a qual descreve o pacote IEEE1394 que será transmitido pelo barramento. Dentre os campos por ela definidos, estão: ACK_CODE, código de confirmação de recebimento de pacotes; TLABEL, identificador único de transação; SPEED_CODE, que indica a velocidade utilizada para transmissão do pacote; e DATA, o conteúdo do pacote propriamente dito, entre outros.

HPSB_WRITE é a função de transmissão de pacotes, sendo esta formada por um conjunto de outras funções (HPSB_MAKE_WRITEQPACKET, HPSB_MAKE_WRITEBPACKET, GET_TLABEL, FILL_ASYNC_WRITEBLOCK, FILL_ASYNC_WRITEQUAD, HPSB_PACKET_SUCCESS, por exemplo) de extrema relevância para este trabalho, por ser ela ponto chave no desempenho de todo o sistema.

A variável PAGE_SIZE define o tamanho da página de memória acessível pelo gerente de memória, sendo que o driver a utiliza para referenciar o tamanho do maior pacote transmissível pelo barramento.

E finalmente, RAW1394_REQUEST, estrutura usada pelo módulo RAW1394 para se comunicar com o espaço de memória do usuário. Esta estrutura é utilizada pela biblioteca Libraw1394, que será abordada a seguir.

3.2 Libraw1394 – Interface em Nível de Usuário

Com intuito de expandir as referencias ao modelo de software IEEE1394, torna-se importante citar, também, outros trabalhos desenvolvidos sobre esta tecnologia. Um dos principais projetos está descrito no trabalho de Bombe [BOM01], a biblioteca Libraw1394.

Essa biblioteca foi criada para intermediar a comunicação entre o espaço de usuário e o módulo RAW1394, escondendo das aplicações a existência do protocolo IEEE1394. Dentre suas funcionalidades temos transações READ, WRITE e LOCK, recepção de fluxos isócronos e transferências assíncronas, apresentando como vantagens:

- o protocolo só precisa ser implementado uma única vez;
- todos os projetos podem ser feitos sobre um conjunto de funções, ao invés de utilizar uma estrutura de comando complicada;
- alterações no módulo RAW1394 refletem em alterações somente na biblioteca Libraw1394, mantendo as aplicações intactas.

A Figura 3.4 mostra os comandos necessários para a instalação da biblioteca Libraw1394.

Tamanha é a importância desta implementação, que pode-se citar algumas aplicações, que dela fazem uso. A Tabela 3.1 mostra alguns utilitários, acompanhados de uma breve descrição, enquanto a Tabela 3.2 apresenta duas outras bibliotecas desenvolvidas a partir desta.

Tabela 3.1 – Aplicações sobre a biblioteca Libraw1394.

gscanbus	Mostra os dispositivos conectados no barramento.
Coriander	Utilitário gráfico que permite controlar câmeras digitais.
dvgrab	Transforma vídeos de uma filmadora em arquivos AVI.
dvbackup	Através de linhas de comando, armazena qualquer tipo de dado em uma filmadora.

1. Compile a biblioteca libraw1394:

   - `tar xvfz libraw1394_0.9.0.tar.gz`
   - `cd libraw1394-0.9.0`
   - `./configure`
   - `make`
   - `make install`

2. Criar o novo dispositivo (/dev/raw1394):

   - `make dev`

3. Reiniciar:

   - `shutdown -r now`

4. Carregar os módulos:

   - `modprobe ohci1394`
   - `modprobe raw1394`

Figura 3.4 – Configurando e carregando a biblioteca Libraw1394.

Tabela 3.2 – Bibliotecas desenvolvidas a partir da Libraw1394.

libdc1394	Biblioteca que provê uma interface de alto nível para programadores que desejem controlar câmeras digitais IEEE1394.
libavc1394	Biblioteca para controle de dispositivos de áudio/vídeo.

3.3 Outros Projetos sobre IEEE1394

Dando continuidade a introdução de alguns projetos desenvolvidos sobre IEEE1394, podemos citar: eth1394 e ip1394 [FIA02, IPS01, STI01], IICP488 [1394TA], HotNet

[BOU02], PPDT (P1394.3) [PPD00, IEE01], SBP 2 [SBP01, STK01], entre outros. Esta relação nos mostra tanto a diversidade dos sistemas em implementação, quanto a expectativa que está sendo depositada na usabilidade deste barramento.

Nessa mesma linha, estão em pauta, também, aplicações multimídia, redes e em tempo real, que necessitam ou não de garantias de entrega, velocidade e praticidade, em geral, qualidade de serviço. Além de estar presente em soluções de redes domésticas, empresas e universidades, tem-se, como exemplo de sua alta aplicabilidade, a ampla utilização nas indústrias automotiva, de telecomunicações e aeroespacial, sendo a Boeing umas de suas principais usuárias.

Com vistas a demonstrar a representatividade desta tecnologia, e o impacto que ela vem provocar no mercado de hardware/software, descreve-se, a seguir, alguns protocolos, desenvolvidos sobre o sistema operacional Linux, Windows e/ou MacOS.

3.3.1 IP sobre 1394

O primeiro protocolo apresentado é o IP1394 (do qual foi derivado o ETH1394). Esse protocolo é parte da família dos protocolos de alto nível, consistindo de uma camada que provê interface entre o protocolo TCP/IP do sistema operacional e o driver IEEE1394, oferecendo compatibilidade e versatilidade às aplicações.

Algumas aplicações como Telnet, Ping, FTP, RPC e RLOGIN já passaram por testes de desempenho, como mostra o trabalho de Fiala [FIA02], mas com a restrição de funcionar atualmente com especificações de IPv4.

Sua tarefa consiste, apenas, de receber um datagrama IP, empacotá-lo e retransmiti-lo através do barramento IEEE1394 para outro nó IP, para isso, utiliza uma tabela de referência que associa um endereço IP a um endereço IEEE1394, pela execução de um procedimento de ARP (*Address Resolution Protocol*).

A Figura 3.5 ilustra a estrutura e posição deste protocolo junto ao sistema. Além disto, mais referências podem ser obtidas nos manuais do protocolo IP1394 [IPS01, STI01] e no trabalho de TEENER [TEE02].

O protocolo ETH1394 é um projeto que corre em paralelo ao IP1394, sendo ele, atualmente, uma parte complementar (de uso experimental) ao *driver* IEEE1394 que acompanha as distribuições Linux com kernel superior a versão 2.5.

3.3.2 SBP 2 – Serial Bus Protocol versão 2

O protocolo SBP 2 (*Serial Bus Protocol*), descrito em manuais técnicos [SBP01, STK01], é tanto um *driver* de alto nível quando se refere ao padrão IEEE1394, quanto um *driver* de baixo nível ao se referir ao padrão SCSI (*Small Computer System Interface*).

Diante da necessidade de estabelecer comunicação entre periféricos, empresas como Microsoft e Apple, empregam este protocolo devido sua baixa carga sobre a CPU e por se fazer um modelo de acesso direto à memória escalonável. A Figura 3.6 ilustra a arquitetura do protocolo SBP 2. Já a Figura 3.7 mostra a organização do protocolo SBP 2 em relação a outros módulos do sistema e das aplicações de usuário.

Em relação a sua arquitetura, ela provê um mecanismo de transporte de comandos e dados, que torna o periférico um dispositivo com acesso direto à memória. Isto pode ser feito, mapeando-se a memória da CPU diretamente no espaço de memória do sistema 1394, auxiliado por um conjunto de ponteiros e *buffers* de dados.

O protocolo SBP 2 acompanha as distribuições do IEEE1394 para Linux, estando presente desde suas primeiras versões.

A Tabela 3.3 lista alguns aparelhos compatíveis com a tecnologia IEEE1394 e que trabalham com o protocolo SBP 2.

Tabela 3.3 – Dispositivos IEEE1394 compatíveis com SBP 2.

Western Digital IEEE-1394 hard drives
Maxtor IEEE-1394 hard drives
VST (SmartDisk) IEEE-1394 hard drives and Zip drives
QPS IEEE-1394 CD-RW/DVD drives and hard drives

Tabela 3.3 – Dispositivos IEEE1394 compatíveis com SBP 2 (cont.).

BusLink IEEE-1394 hard drives
Epson IEEE-1394 scanner
Sony IEEE-1394 CD-RW drives
Iomega IEEE-1394 Zip/Jazz drives
EzQuest IEEE-1394 hard drives and CD-RW drives
Bellstor IEEE-1394 hard drives and CD-RW drives
FirePower IEEE-1394 hard drives
Castlewood/ADS IEEE-1394 ORB drives

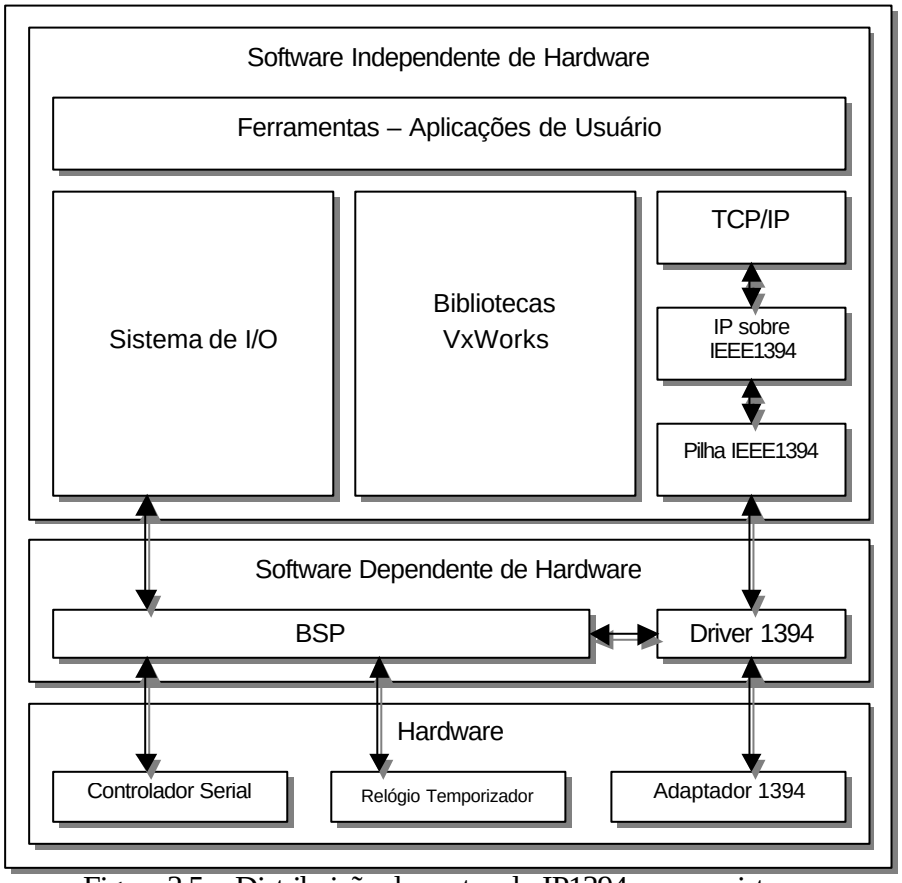

Figura 3.5 – Distribuição do protocolo IP1394 em um sistema.

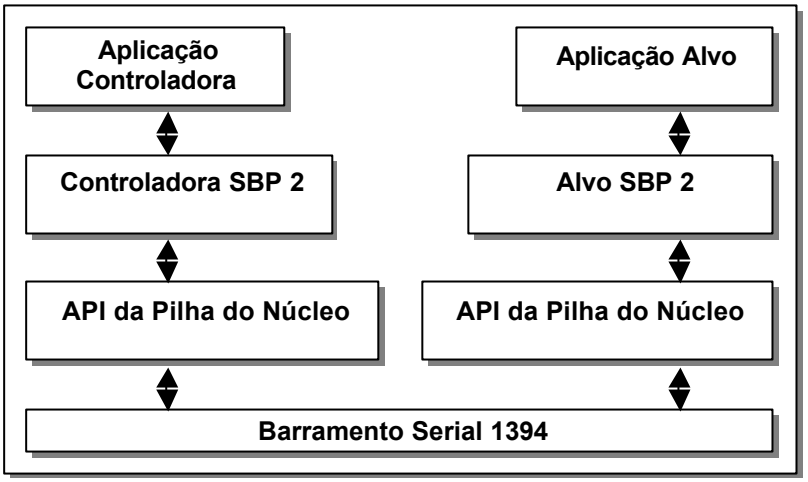

Figura 3.6 – Arquitetura do Protocolo SBP 2.

3.3.3 PPDT - Peer-to-Peer Data Transmission

O PPDT (*Peer-to-Peer Data Transmission*) é um protocolo que estabelece um ”túnel” de comunicação entre dois pontos (dispositivos). De um lado está a aplicação cliente, e do outro está a aplicação servidora. Para efetivar o transporte de dados, este protocolo utiliza-se do protocolo SBP 2 em ambos os lados da conexão.
A Figura 3.8 ilustra este relacionamento. Já os trabalhos técnicos [PPD00, IEE01] servem de referência à compreensão deste protocolo.

Algumas utilidades para este protocolo são:

- busca por dispositivos e serviços – pode ser utilizado em conjunto com o padrão P1394.1 (especifica a conexão de múltiplos barramentos), afim de localizar e identificar dispositivos e serviços oferecidos as aplicações clientes;
- autoconfiguração (*plug and play*) – oferece a possibilidade de conectar *drivers* e dispositivos de forma dinâmica, simplificando a operação dos usuários, e;
- gerenciamento de conexões – um dispositivo PPDT pode estabelecer e gerenciar conexões para transferências de dados com outro dispositivo PPDT, podendo ser elas bloqueantes ou não.

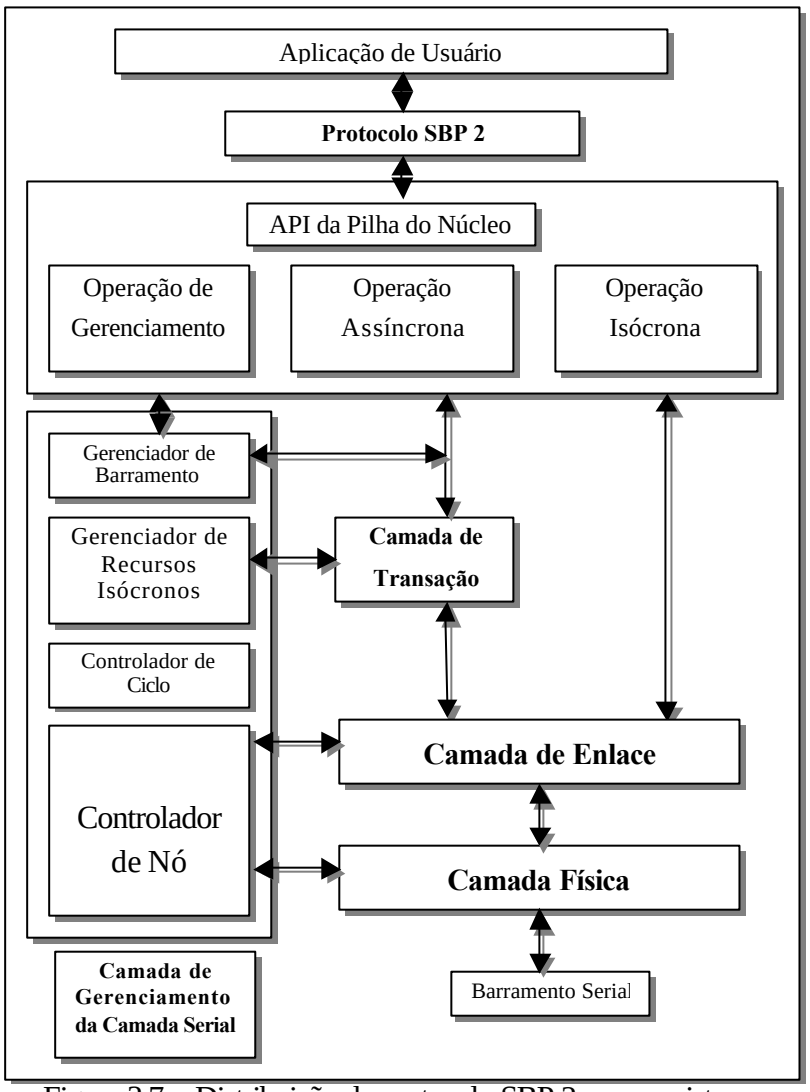

Figura 3.7 – Distribuição do protocolo SBP 2 em um sistema.

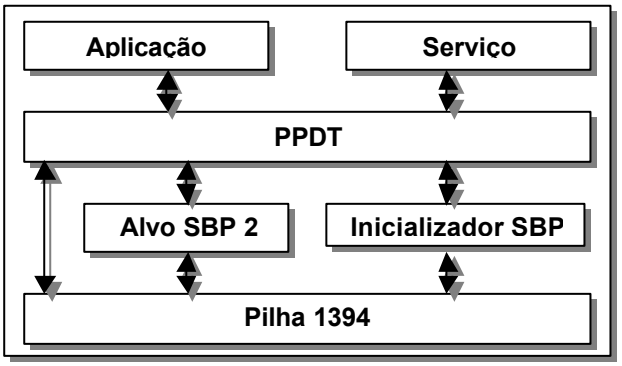

Figura 3.8 – Disposição do protocolo PPDT.

4 O Módulo SYNC1394

O módulo SYNC1394 (*Synchronous IEEE1394*) foi desenvolvido para atuar sobre o sistema operacional Linux[9] como um *driver* intermediário entre aplicações em nível de usuário/kernel e o *driver* de baixo nível, responsável pela comunicação direta com o hardware. Sua função é fornecer um conjunto de primitivas de comunicação síncrona:

- SEND bloqueante;
- RECEIVE bloqueante;
- RECEIVE ANY bloqueante.

A partir destas funções principais e de um conjunto auxiliar somos capazes de introduzir o padrão IEEE1394, que não possui tal característica, para uma nova gama de aplicações paralelas e distribuídas.

4.1 Ajustes Preliminares

Devido ao fato de que o *driver* IEEE1394 (de baixo nível), que acompanha as distribuições do Linux, estar em fase de desenvolvimento, alguns ajustes foram necessários para garantir um melhor aproveitamento do hardware. Tais ajustes visam possibilitar o envio de pacotes de até 2048 bytes através do barramento, já que os registradores responsáveis por tal operação são definidos, originalmente, com uma capacidade de apenas 512 bytes.

Tais alterações consistem no redimensionamento de variáveis e na remoção de duas linhas de código, as quais podem ser vistas nas Figuras 4.1 e 4.2 a seguir:

[9] A partir das versões de kernel 2.2 (as versões 2.2 e 2.4 necessitam do patch de atualização http://www.linux1394.org).

csr.h

```
/* register offsets relative to CSR_REGISTER_BASE */
#define CSR_STATE_CLEAR           0x0
#define CSR_STATE_SET             0x4
#define CSR_NODE_IDS              0x8
#define CSR_RESET_START           0xc
#define CSR_SPLIT_TIMEOUT_HI      0x18
#define CSR_SPLIT_TIMEOUT_LO      0x1c
#define CSR_CYCLE_TIME            0x200
#define CSR_BUS_TIME              0x204
#define CSR_BUSY_TIMEOUT          0x210
#define CSR_BUS_MANAGER_ID        0x21c
#define CSR_BANDWIDTH_AVAILABLE   0x220
#define CSR_CHANNELS_AVAILABLE_HI 0x224
#define CSR_CHANNELS_AVAILABLE_LO 0x228
#define CSR_CONFIG_ROM            0x400
#define CSR_CONFIG_ROM_END        0x800
#define CSR_FCP_COMMAND           0xB00
#define CSR_FCP_RESPONSE          0x1300 //D00
#define CSR_FCP_END               0x1B00 //F00
#define CSR_TOPOLOGY_MAP          0x2000 //1000
#define CSR_TOPOLOGY_MAP_END      0x2400 //1400
#define CSR_SPEED_MAP             0x3000 //2000
#define CSR_SPEED_MAP_END         0x4000 //3000
```

Figura 4.1 – Arquivo csr.h (definição de registradores).

csr.c

```
static int write_fcp(struct hpsb_host *host, int nodeid, int
dest, quadlet_t *data, u64 addr, unsigned int length)
{
        int csraddr = addr - CSR_REGISTER_BASE;

      //    if (length > 512)
      //          return RCODE_TYPE_ERROR;

     switch (csraddr) {
        case CSR_FCP_COMMAND:
                highlevel_fcp_request(host, nodeid, 0, (u8
                *)data, length);
...
}
```

Figura 4.2 – Arquivo csr.c (Envio de pacotes assíncronos a baixo nível).

4.2 Proposta de um Protocolo de Sincronização

Inicialmente pretendia-se, neste trabalho, a implementação de uma interface de comunicação para o multicomputador CRUX [COR99] como solução alternativa às atuais propostas (RS232, USB e Ethernet). Partindo-se de um hardware sem características para comunicações síncronas, foram implementadas três primitivas de comunicação bloqueantes. Com o decorrer do processo de familiarização da tecnologia e do desenvolvimento do módulo, observou-se que mesmo possuindo restrições quanto à velocidade de transmissão, as transações assíncronas eram a opção a ser utilizada, devido a forma de tratamento de seus pacotes.

Quanto às restrições impostas pelas transações assíncronas temos:

- O tamanho das mensagens enviadas de no máximo 512 bytes, e;
- A baixa velocidade de transmissão.

Em relação à primeira restrição, temos a solução através das alterações sugeridas no item anterior. Tais alterações devem ser efetuadas em todas as máquinas que tenham intenção de executar como nós aptos à execução deste módulo e aproveitar ao máximo dos 2048 bytes disponíveis. Uma forma alternativa de evitar alterações no *driver* do Linux, o que obrigaria a recompilação do mesmo, seria a carga do módulo SYNC1394 com o parâmetro MTU com o valor 512 (MTU=512), desta forma estaria informando que o maior pacote transitável pelo barramento é de 512 bytes. Em contrapartida, haveria uma perda significativa de performance, como será explicado em um momento posterior.

Quanto à baixa velocidade de transmissão, não se pode fazer maiores considerações neste primeiro momento, sendo que este assunto será tratado em um capítulo à parte.

A compilação e a instalação do módulo SYNC1394 se faz a partir da seguinte seqüência de comandos, como ilustrado através da Figura 4.3:

- Para compilar o módulo:
gcc -c sync1394.c -I/lib/modules/2.4.18-3custom/build/include -D__KERNEL__ -DMODULE -DLINUX -O2 -march=i686 -DEXPORT_SYMTAB -DSYNC1394_DEBUG* -DSYNC1394_BENCHMARK*

- Para carregar o módulo:
$modprobe ohci1394 attempt_root=1* -k
$insmod sync1394.o control_node=1* MTU=max_packet_size* -k

*opcional

Figura 4.3 – Compilação e instalação do módulo SYNC1394.

As linhas de comando desta figura têm os seguintes significados:

- **DSYNC1394_DEBUG**: ativa as mensagens de log;
- **DSYNC1394_BENCHMARK**: executa um benchmark simplificado sobre as operações de *send*, *receive* e *receive any*;
- **attempt_root=1**: deve ser utilizado por um único nó (geralmente o nó de controle, em estruturas cliente – servidor) e este requisitar maior prioridade de acesso sobre o barramento;
- **control_node=1**: deve ser utilizado por um único nó (o nó de controle, em estruturas cliente – servidor), sendo que no processo de configuração este nó informará seu status ao demais, vide item 4.3.4.1;
- **MTU=max_packet_size**: define o tamanho do maior pacote transitável pelo barramento IEEE1394, podendo variar de 4 bytes (mínimo) à 512 bytes (2048 bytes, no caso dos devidos ajustes tenham sido feitos).

Através da Figura 4.4 demonstra-se um esquema resumido da troca de mensagens entre dois nós que rodam o módulo SYNC1394. De um lado está o nó transmissor, o qual executa a chamada da primitiva *send*, e do outro lado, o nó receptor, executando ou a primitiva *receive* ou *receive any*. Em ambas as situações, existem a opção da primitiva em nível de usuário (*send*, *receive, receive_any*) e em nível de kernel (*send1394*, *receive1394, receive_any1394*). Após os nós informarem suas intenções, de transmitir e receber um

pacote, o processo de sincronização é efetuado, a transferência realizada e o processo concluído.

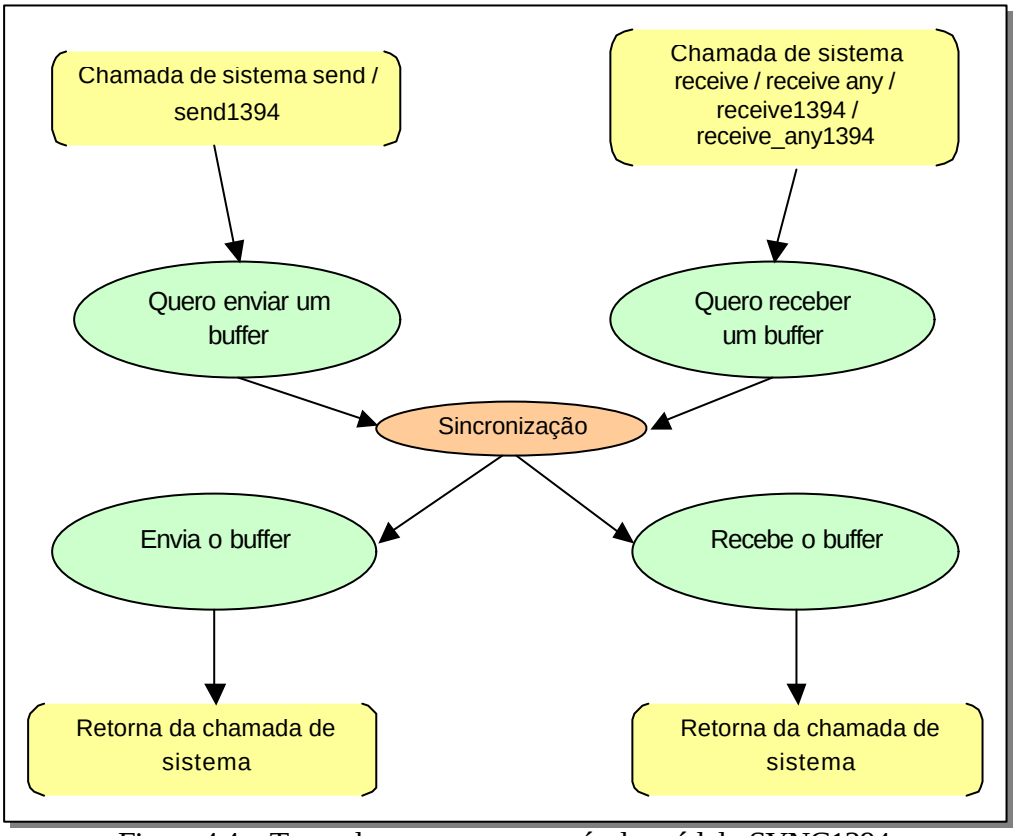

Figura 4.4 – Troca de mensagens através do módulo SYNC1394.

4.3 Primitivas de Sincronização

São três as primitivas de sincronização do módulo SYNC1394: *send*, *receive* e *receive any* – sendo as mesmas implementadas com semáforos e/ou espera ocupada (*busy wait*). Além destas, foram implementadas algumas funções complementares, para que sirvam de apoio, são elas: *sync_1394, get_1394id, get_control_1394id, reset_1394* e *get_node_count*.

A implementação dessas funções gerou algumas implicações importantes e, assim, alguns cuidados tiveram que ser tomados, pois a execução de qualquer uma delas em

contexto de interrupção, sem um devido tratamento, pode causar a parada do sistema, tanto por *deadlock* quanto por pane no kernel.

Tais primitivas são implementadas como chamadas de sistema para aplicações em nível de usuário e exportadas na tabela de símbolos do sistema operacional no caso de aplicações em nível de kernel.

Elas estão descritas a seguir.

4.3.1 Send Bloqueante

A primeira primitiva a ser abordada é o *send* bloqueante, implementada sobre o sistema de transferência assíncrona ela se encarrega de sincronizar os sistemas, dividir o buffer em pacotes, quando necessário, e executar a interface IEEE1394, através da função *hpsb_write*.

Na Figura 4.5 estão as interfaces fornecidas, consistindo de três argumentos:

```
/*   send for user-aplications      */

int send(void *buffer, const int size, const int
dest)

/*   send for kernel-aplications    */

int sync1394_send(void *buffer, const int size,
const int dest)
```

Figura 4.5 – Interface da primitiva *send*.

- Buffer: ponteiro para a primeira posição de memória do buffer a ser transmitido;
- Size: tamanho do buffer a ser transmitido, devendo ser maior do que zero e menor do que o maior espaço de memória alocável pelo nó que executa o receive;

- Dest: endereço físico do nó que receberá o pacote podendo variar de 0 a 62.

A Figura 4.6 mostra o fluxo principal de execução da primitiva *send*. Este fluxo consiste na execução de uma fase de inicialização de variáveis, seguido por um teste que verifica se o nó parceiro já informou sua intenção de receber um dado. Caso o tenha feito, o dado é enviado e o processo finalizado. Em contrapartida, se o nó receptor ainda não se pronunciou, o nó transmissor envia sua intenção de transmitir e aguarda a liberação em um ponto de sincronização. Assim que este for sinalizado, o processo retorna, executando a transferência e finalizando a primitiva.

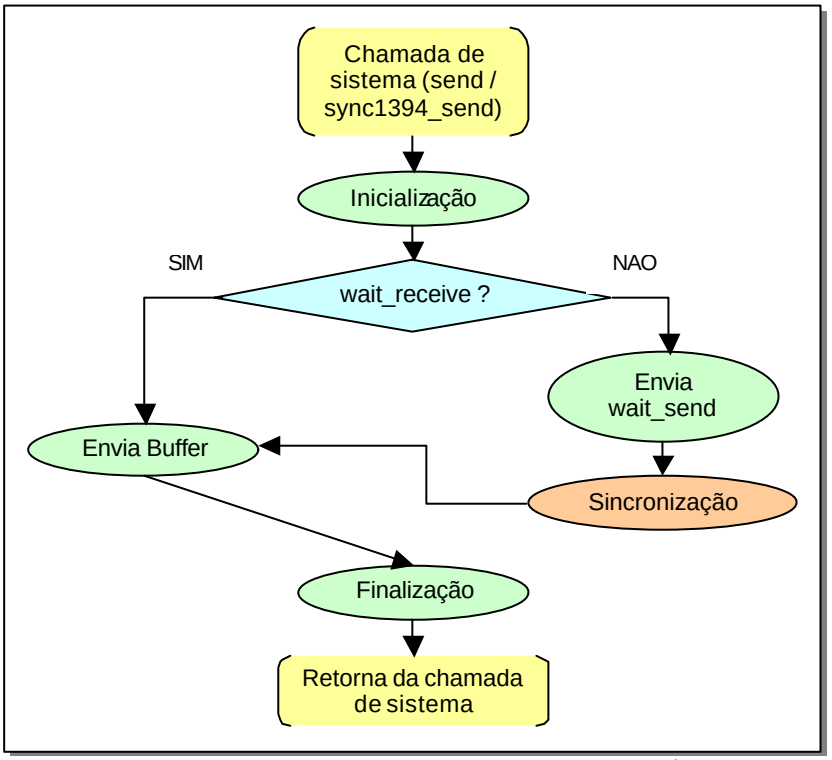

Figura 4.6 – Fluxo de execução da primitiva *send*.

4.3.2 Receive Bloqueante

A segunda primitiva a ser abordada é o *receive* bloqueante, implementada sobre o sistema de transferência assíncrona ela se encarrega de sincronizar os sistemas, receber o

buffer montando-o a partir de mais de um pacote, quando necessário, e retorná-lo à aplicação que fez a requisição.

Na Figura 4.7 estão ilustradas as interfaces fornecidas:

- Buffer_pckt: ponteiro para uma região de memória previamente alocada pela aplicação, devendo esta ser grande o suficiente para receber o buffer;

```
/*   receive for user-aplications   */

int receive(void *buffer_pckt, int size, int source)

/*   receive for kernel-aplications  */

static int sync1394_receive(void *buffer_pckt, int
size, int source)
```

Figura 4.7 – Interface da primitiva *receive*.

- Size: tamanho do buffer a ser recebido, utilizado internamente pela chamada para fim de *buffering*;
- Source: endereço físico do nó de que se origina o buffer.

A Figura 4.8 representa o fluxo principal de execução da primitiva *receive*. Ao se iniciar o processo de recepção, a primitiva inicializa o ambiente de variáveis das quais fará uso e efetua um teste que indica se o nó transmissor já informou sua intenção de transmissão. Caso o nó parceiro já tenha se pronunciado, o nó receptor envia um pacote de controle liberando o início do processo de transferência e entra em uma etapa de sincronização aguardando pelo dado. Por outro lado, se o nó transmissor ainda não foi executado, o nó receptor envia um pacote informando sua intenção e passa a esperar pelo dado em um estágio de sincronização. Ao final da transação o dado é repassado ao processo requisitante e a primitiva finalizada.

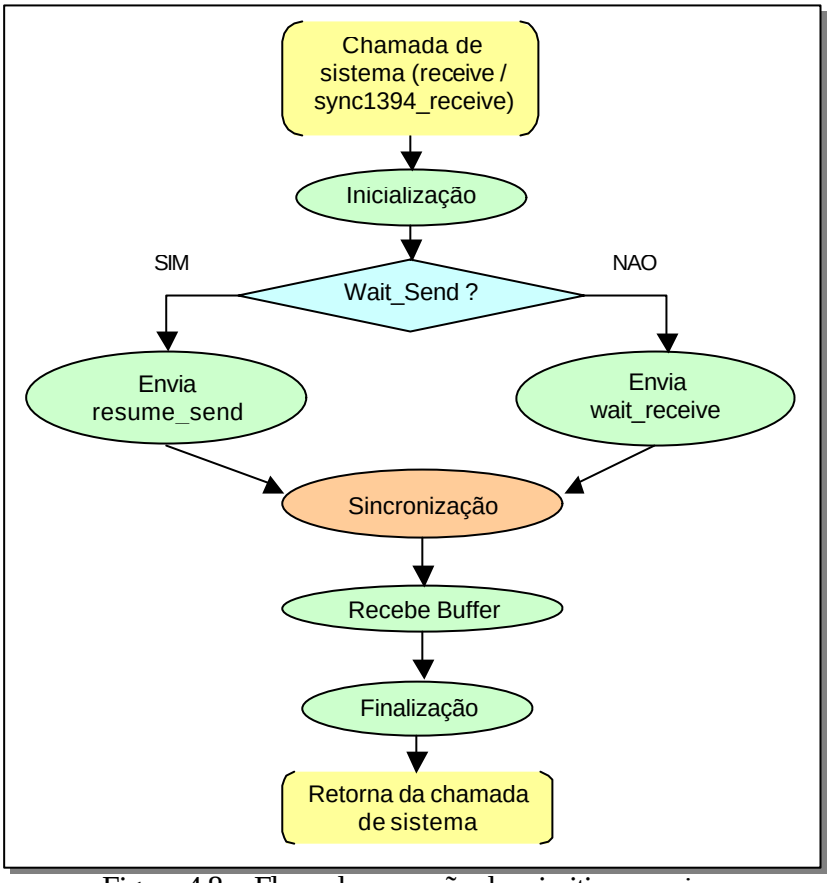

Figura 4.8 – Fluxo de execução da primitiva *receive*.

4.3.3 Receive Any Bloqueante

A terceira primitiva a ser abordada é o *receive any* bloqueante, implementada sobre o sistema de transferência assíncrona ela se encarrega de sincronizar os sistemas, receber o buffer de qualquer nó que deseje lhe transmitir, montando-o a partir de mais de um pacote, quando necessário, e retorná-lo à aplicação que fez a requisição junto com o endereço do nó transmissor.

Na Figura 4.9 estão representados os protótipos fornecidos, consistindo de três argumentos:

```
/*   receive_any for user-aplications     */

int receive_any(void *buffer_pckt, int size, int
**source)

/*   receive_any for kernel-aplications   */

static int sync1394_receive_any(void *buffer_pckt,
int size, int **source)
```

Figura 4.9 – Interface da primitiva *receive any*.

- Buffer_pckt: ponteiro para uma região de memória previamente alocada pela aplicação, devendo esta ser grande o suficiente para receber o buffer;
- Size: tamanho do buffer a ser recebido, utilizado internamente pela chamada para fim de *buffering*;
- Source: ponteiro para um valor inteiro que receberá o endereço físico do nó de que se origina o buffer.

A Figura 4.10 mostra o fluxo principal de execução da primitiva *receive any*. Após a etapa de inicialização da primitiva, o processo entra em um *loop – while* onde testa a existência de indicações de intenção de transmissão. Ao encontrar tal indicação, ou seja, um nó transmissor pronunciou-se, o nó receptor envia um pacote liberando o nó parceiro para iniciar a transação, aguardando em um estado de sincronização. Finalizada a comunicação, a primitiva retorna os dados ao processo requisitante e encerra sua operação.

4.3.4 Funções Complementares

Com a finalidade de auxiliar o desenvolvimento de aplicações que utilizam este módulo, fez-se necessária a implementação de um conjunto de funções de apoio, sendo estas capazes de sincronizar endereços, retorná-los às devidas aplicações e gerenciar o estado do nó.

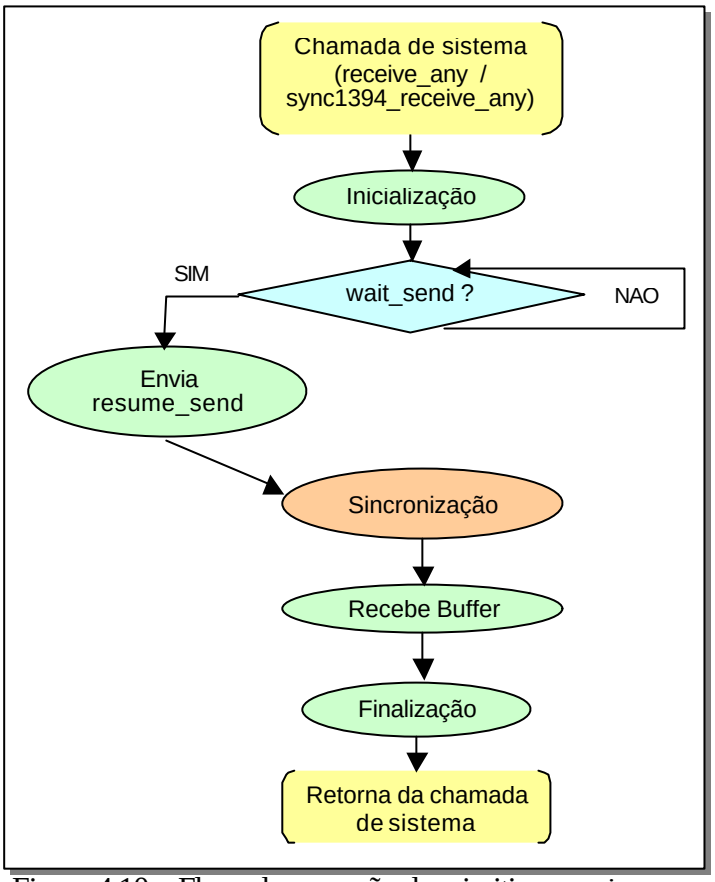

Figura 4.10 – Fluxo de execução da primitiva *receive any*.

4.3.4.1 SYNC_1394

A função *sync_1394*, apresentada na Figura 4.11, é responsável por transmitir o endereço físico do nó de controle, quando este parâmetro for carregado pelo módulo, ou transmitir a requisição de um nó de trabalho ao nó de controle para obter seu endereço.

Este processo é útil para aplicação em sistemas que utilizem uma política cliente – servidor e não há necessidade de "prender" o nó de controle a um determinado endereço físico.

Sendo esta primitiva executada, o fluxo inicia-se com um teste que garanta a não sincronização do nó com os demais. O nó não estando sincronizado passa para um novo teste onde identifica sua característica, nó servidor ou de trabalho. Caso este nó seja de

trabalho, ele envia um pacote (*broadcast*) requisitando o endereço do nó servidor e entra em um estado de sincronização. Caso este seja um servidor, ele enviará um pacote de controle contendo seu endereço (*broadcast*), liberando os demais nós para que continuem seus processos de sincronização. Na Figura 4.12 está representado o fluxo principal de execução da função *sync_1394*.

```
/*   sync_1394 for user-aplications  */
int sync_1394(void)
/*   sync_1394 for kernel-aplications     */
static int sync1394_sync_1394(void)
```

Figura 4.11 – Interface da função *sync_1394*.

4.3.4.2 GET_1394ID

A função *get_1394id*, Figura 4.13, retorna para a aplicação o endereço físico do nó, seja ele de controle ou de trabalho. Este endereço será um valor entre 0 e 62.

4.3.4.3 GET_CONTROL_ID_1394

A função *get_control_1394id*, Figura 4.14, retorna para a aplicação o endereço físico do nó de controle. Este endereço será um valor entre 0 e 62 para um nó de controle válido e –1 quando o seu endereço não estiver sincronizado. Esta função pode e deve ser utilizada em conjunto com a função *sync_1394*, para garantir a lógica correta das aplicações.

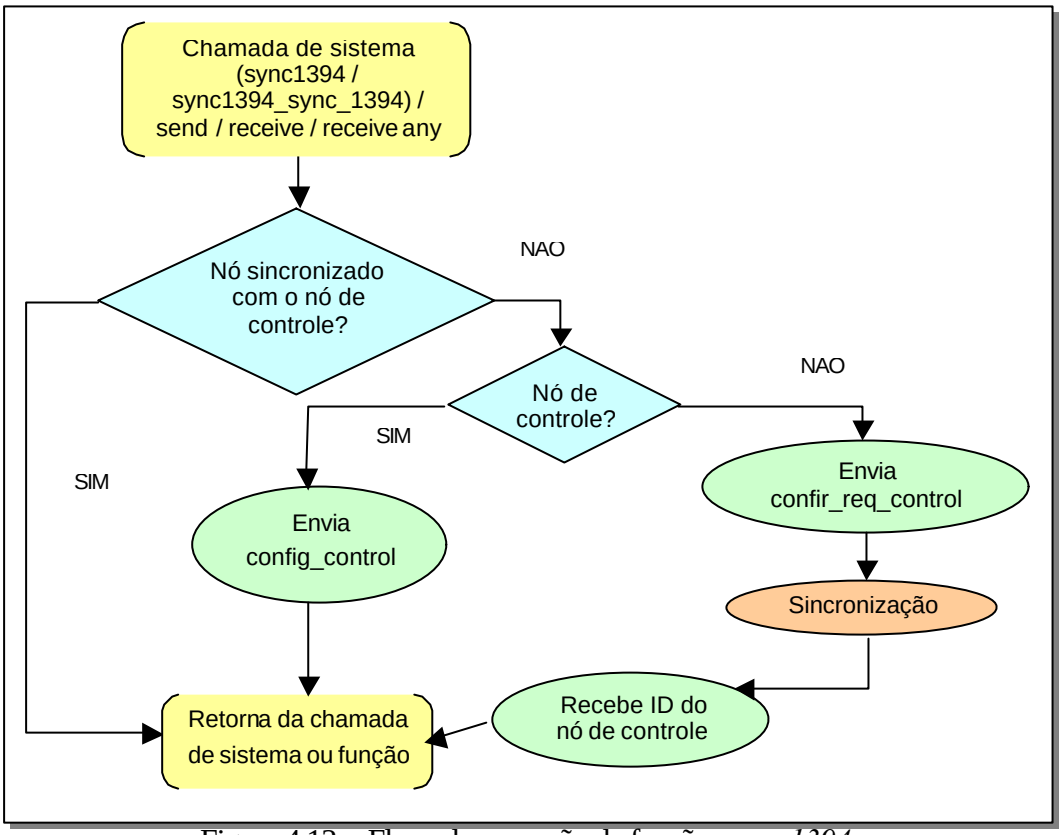

Figura 4.12 – Fluxo de execução da função *sync_1394*.

```
/*   get_1394ID for user-aplications      */

int get_1394id(void)

/*   get_1394ID for kernel-aplications    */

int sync1394_get_1394id(void)
```

Figura 4.13 – Interface da função *get_1394id*.

```
/* get_control_1394id for user-aplications    */

int get_control_1394id(void)

/* get_control_1394id for kernel-aplications  */

static int sync1394_get_control_1394id(void)
```

Figura 4.14 – Interface da função *get_control_id_1394*.

4.3.4.4 GET_NODE_COUNT

A função *get_node_count*, Figura 4.15, retorna para a aplicação o número de nós ativos no barramento, sendo este um valor entre 1 - para o caso de quando apenas o próprio nó está ativo - e 63 para um barramento completo.

```
/* get_node_count for user-aplications    */
int get_node_count(void)
/* get_node_count for kernel-aplications  */
static int sync1394_get_node_count(void)
```

Figura 4.15 – Interface da função *get_node_count*.

4.3.4.5 RESET_1394

A função *reset_1394*, ilustrada na Figura 4.16, reinicializa as principais variáveis do módulo, na tentativa de evitar estados inconsistentes após erros de transmissão ou paradas não previstas. Faz-se uma boa política executar esta função ao iniciar uma nova aplicação.

```
/* reset_1394 for user-aplications    */
int reset_1394(void)
/* reset_1394 for kernel-aplications  */
static int sync1394_reset_1394(void)
```

Figura 4.16 – Interface da função *reset_1394*.

4.4 Análise de Performance

Com o intuito de validar o módulo SYNC1394 e propor otimizações e novas implementações descrevemos neste tópico os testes de performance do módulo SYNC1394, do padrão IEEE1394 e uma comparação com o padrão Ethernet.

O ambiente de execução tanto para o módulo SYNC1394 quanto para o padrão IEEE1394 consiste de uma aplicação produtora/consumidora, onde uma máquina executa a transmissão de um pacote de tamanho T conhecido N vezes. No caso do padrão Ethernet foi utilizado a ferramenta de benchmark NetPIPE [TUR02], que simula um algoritmo ping-pong para obtenção dos tempos de transmissão.

Vale salientar que em todos os casos foram utilizados dois computadores com os seguintes recursos:

- Tamanho máximo de pacote transmitido de 2048 bytes;
- Biblioteca Libraw1394 versão 0.9;
- Sistema operacional Linux Red Hat 7.3, Kernel 2.4.18 com suporte à IEEE1394;
- Processadores AMD K7 1.1MHz;
- 128 MB de memória RAM;
- Placa IEEE1394 (Firewire) Pyro BasicDV da ADS Technologies 400Mbps;
- Cabos IEEE1394 de conexão 6x6 1.8m da Lábramo Centronis.

4.4.1 SYNC1394

A coleta de tempos do módulo SYNC1394 fez-se em duas etapas separadas, a primeira com a transmissão contínua tendo o receptor executado a primitiva *receive* com endereço do nó origem conhecido. Num segundo momento com o receptor executando a primitiva *receive any*.

Em ambos os casos, as transações foram realizadas em um laço com 200 (duzentas) repetições, e coletando-se o tempo total. A partir desses valores foram calculadas as médias

de tempo para cada iteração, o número de pacotes transmitidos em cada iteração, a média de tempo de transmissão de cada pacote e a largura de banda (bandwidth).

4.4.1.1 Usando a Primitiva Receive

Através da Tabela 4.1 enumera-se os valores coletados e, na Figura 4.17 apresenta-se o gráfico equivalente à largura de banda obtida.

Tabela 4.1 – Tabela com a primitiva *receive.*

Tamanho do Buffer (bytes)	Tempo de Transmissão (s)	Média p/ iteração (s)	N° de Pacotes	Média p/ pacote (s)	Bandwidth (Mbps)
1	1,17418	0,00587	1	0,00587	0,0012995
4	0,94279	0,00471	1	0,00471	0,0064739
16	0,96851	0,00484	1	0,00484	0,0252079
32	0,85953	0,00430	1	0,00430	0,0568079
80	0,98625	0,00493	1	0,00493	0,1237722
160	0,89420	0,00447	1	0,00447	0,2730269
240	0,96470	0,00482	1	0,00482	0,3796112
512	0,98157	0,00491	1	0,00491	0,7959188
800	0,93277	0,00466	1	0,00466	1,3086861
1024	0,96471	0,00482	1	0,00482	1,6196577
1600	0,91541	0,00458	1	0,00458	2,6670085
1800	1,17223	0,00586	1	0,00586	2,3430402
2000	1,04930	0,00525	1	0,00525	2,9083749
2048	1,21010	0,00605	1	0,00605	2,5824312
4000	1,78030	0,00890	2	0,00445	3,4283635
6000	2,61127	0,01306	3	0,00435	3,5060616
10000	4,43763	0,02219	5	0,00444	3,4384996
20000	8,16144	0,04081	10	0,00408	3,7392394
40000	16,25300	0,08127	20	0,00406	3,7553163
100000	39,74000	0,19870	49	0,00406	3,8396550
500000	197,99200	0,98996	245	0,00404	3,8533853

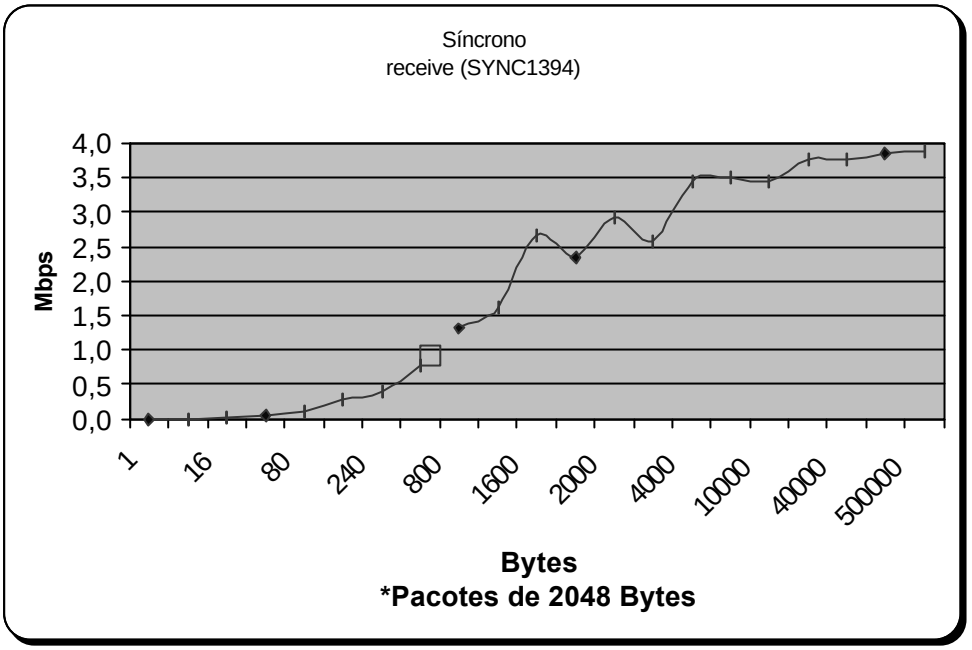

Figura 4.17 – Largura de banda da primitiva *receive* (SYNC1394).

Nota-se aqui dois pontos importantes, o fato da largura de banda tender a 4 (quatro) Mbps, assim como o tempo de transmissão por pacotes tender a 0,004s, fazendo com que o tempo de envio de pacotes maiores que 2048 tornem-se múltiplos deste número. A baixa velocidade está diretamente relacionada ao tipo de transação – assíncrona – utilizada, como foi explicado em capítulo anterior.

As Figuras 4.18 e 4.19 mostram, respectivamente, o gráfico de desempenho com os tempos de transmissão por pacotes variando de 1 a 500000 bytes, e seu detalhamento em um conjunto reduzido de valores que variam de 1 a 2048 bytes. As variações expostas neste último gráfico não refletem, necessariamente, uma não-linearidade, apenas ilustra que a corrida pela tomada do barramento pode levar a pequenas variações no tempo de envio dos pacotes menores que o maior pacote transitável, levando em conta que com o aumento das mensagens enviadas o tempo de transmissão de cada pacote tende a um valor constante.

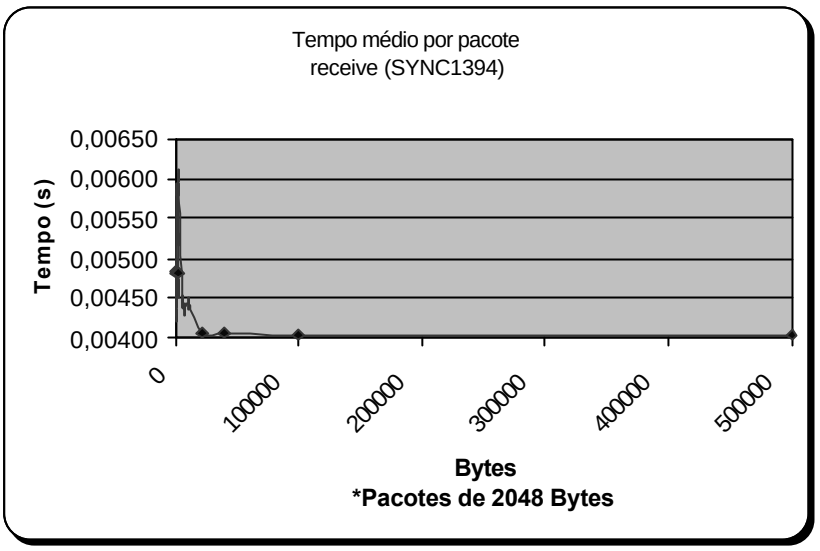

Figura 4.18 – Tempos de Transmissão da primitiva receive (SYNC1394).

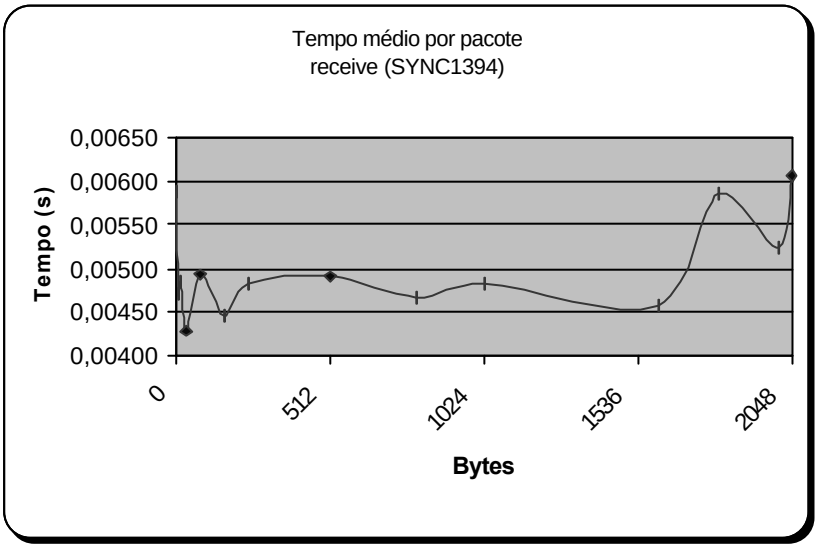

Figura 4.19 - Tempos de Transmissão da primitiva receive - detalhamento (SYNC1394).

4.4.1.2 Usando a Primitiva Receive Any

Repetimos o processo, agora com a primitiva *receive any*, temos na Tabela 4.2 os valores obtidos nos testes de transmissão.

Tabela 4.2 - Tabela com a primitiva receive any.

Tamanho do Buffer (bytes)	Tempo de Transmissão (s)	Média p/ iteração (s)	N° de Pacotes	Média p/ pacote (s)	Bandwidth (Mbps)
1	1,90460	0,00952	1	0,00952	0,0008012
4	1,99564	0,00998	1	0,00998	0,0030584
16	1,89432	0,00947	1	0,00947	0,0128880
32	1,97096	0,00985	1	0,00985	0,0247738
80	1,75725	0,00879	1	0,00879	0,0694667
160	1,95315	0,00977	1	0,00977	0,1249984
240	1,92761	0,00964	1	0,00964	0,1899819
512	1,84766	0,00924	1	0,00924	0,4228321
800	1,83102	0,00916	1	0,00916	0,6666793
1024	1,85681	0,00928	1	0,00928	0,8414970
1600	1,89071	0,00945	1	0,00945	1,2912643
1800	1,94277	0,00971	1	0,00971	1,4137453
2000	1,92532	0,00963	1	0,00963	1,5850652
2048	1,95343	0,00977	1	0,00977	1,5997502
4000	2,70609	0,01353	2	0,00677	2,2554740
6000	3,29011	0,01645	3	0,00548	2,7826648
10000	5,05509	0,02528	5	0,00506	3,0185000
20000	8,87891	0,04439	10	0,00444	3,4370861
40000	16,92740	0,08464	20	0,00423	3,6057018
100000	40,30230	0,20151	49	0,00411	3,7860839
500000	198,03600	0,99018	245	0,00404	3,8525291

Podemos notar um acréscimo nos tempos, pontualmente, enquanto que a tendência dos valores do tempo de transmissão e de largura de banda são mantidos os mesmos do caso anterior.

O aumento aqui citado é devido à lógica utilizada para recepção de mensagens de origens não predefinidas. Para recepção através da primitiva *receive*, o nó que a executou envia uma mensagem de controle ao nó transmissor funcionando como uma requisição. Este passo diminui o número de mensagens de controle transitando no barramento e, conseqüentemente, melhora o seu desempenho. No caso da primitiva *receive any*, o nó aguarda um primeiro contado do nó transmissor para então enviar uma mensagem liberando o início da transação. Aqui temos, obrigatoriamente, um tráfego maior de dados no barramento.

A Figura 4.20 mostra a largura de banda obtida nos testes da primitiva, com uma tendência aos 4 (quatro) Mbps. Enquanto as Figuras 4.21 e 4.22 mostram, respectivamente, os tempos de transmissão de pacotes variando de 1 a 500000 bytes e um detalhamento, variando de 1 a 2048 bytes. Nesta última encontramos os mesmos tipos de distorções encontradas nos testes da primitiva *receive*, o que nos leva as mesmas conclusões.

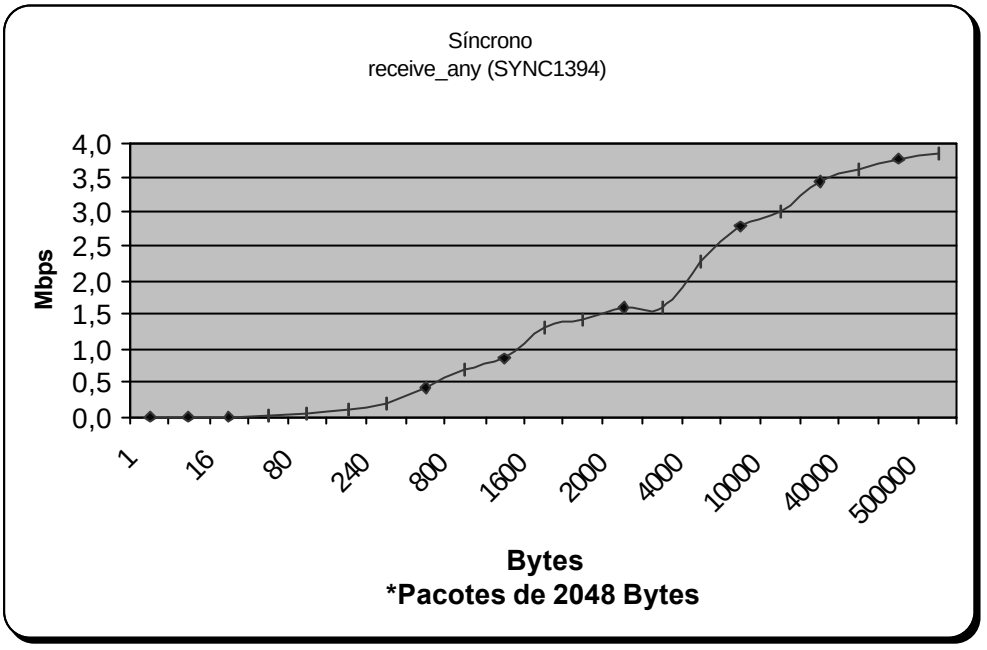

Figura 4.20 - Largura de banda da primitiva receive any (SYNC1394).

4.4.2 IEEE1394

Os testes com o padrão IEEE1394 foram efetuados com o auxílio da biblioteca *libraw* de alto nível, sendo esta descrita em capítulo anterior.

Observando-se a Tabela 4.3, podemos fazer uma comparação com os tempos de transmissão do módulo aqui proposto, chegando à conclusão de que o módulo SYNC1394 não sofre uma interferência significativa pela utilização das mensagens de controle, uma vez que transmissor e receptor atingem um estado de comunicação harmônico Em ambos os casos, a performance do barramento mostrou-se similar, principalmente na transmissão

de grandes mensagens onde o elevado número de pacotes de dados compensam o tempo com a sincronização dos nós.

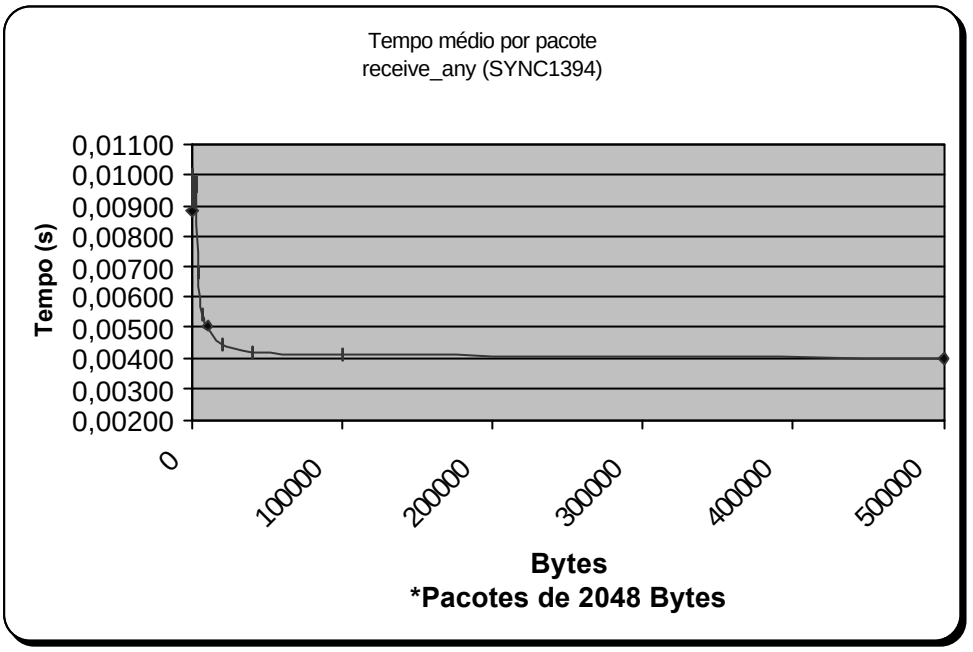

Figura 4.21 – Tempos de Transmissão da primitiva receive any (SYNC1394).

Tabela 4.3 - Tabela com o padrão IEEE1394.

Tamanho do Buffer (bytes)	Tempo de Transmissão (s)	Bandwidth (Mbps)
1	0,0064	0,0012
4	0,0046	0,0067
16	0,0061	0,02
32	0,0060	0,041
80	0,0060	0,1024
160	0,0046	0,2658
240	0,0060	0,3053
512	0,0056	0,6983
800	0,0045	1,3618
1024	0,0045	1,7479
1600	0,0048	2,5181
1800	0,0046	2,9777
2000	0,0051	3,0137
2048	0,0056	2,7797
4000	0,0086	3,5344

Tabela 4.3 - Tabela com o padrão IEEE1394 (cont.).

6000	0,0135	3,3821
10000	0,0207	3,6877
20000	0,0409	3,7308
40000	0,0811	3,7614
100000	0,1992	3,8297
500000	0,9905	3,8512

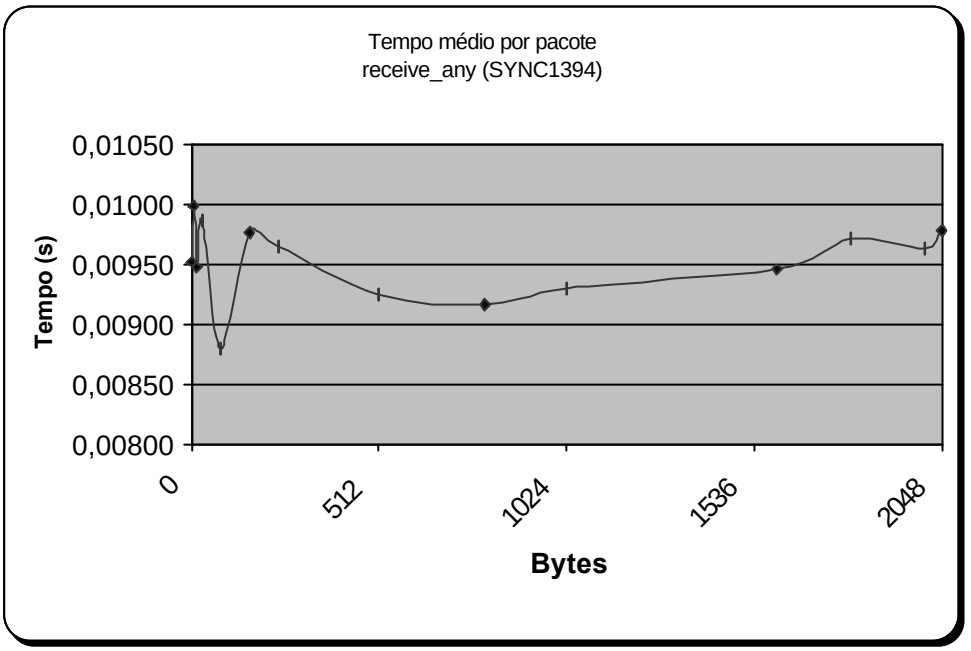

Figura 4.22 - Tempos de Transmissão da primitiva receive any – detalhamento (SYNC1394).

O gráfico na Figura 4.23 ilustra o conteúdo da tabela anterior, assemelhando-se ao gráfico da primitiva *receive*, onde mais de 90% da largura de banda é alcançada com pacotes de aproximadamente 10000 bytes.

4.4.3 Ethernet (TCP/IP)

As tomadas de tempo com o padrão Ethernet, para efeito de comparação, foram feitas a partir da ferramenta NetPIPE [TUR02] que utiliza sockets TCP para comunicação.

Ele implementa um algoritmo ping-pong (a mensagem é enviada em um sentido e então reenviada ao nó de origem), a fim de permitir uma maior precisão nos valores, já que a primitiva *send* encontrada na implementação dos sockets não é bloqueante.

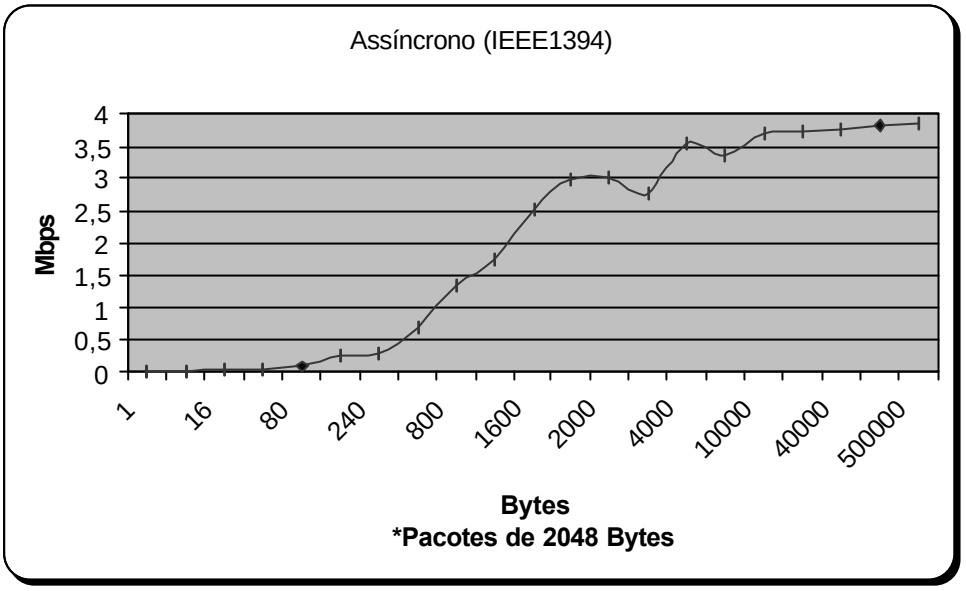

Figura 4.23 - Largura de banda da transação assíncrona (IEEE1394).

A Tabela 4.4 mostra um subconjunto dos valores obtidos com a ferramenta, ilustrada no gráfico da Figura 4.24, sendo ela executada com o seguinte suporte de hardware:

- Placas de rede Ethernet 100Mbps;
- Switch 3COM 100Mbps.

Tabela 4.4 - Tabela com o padrão Ethernet.

Tamanho do Buffer (bytes)	Tempo de Transmissão (s)	Bandwidth (Mbps)
1	0,000045	0,1695
8	0,000048	1,2716
12	0,00005	1,8311
13	0,00005	1,9836
16	0,000049	2,4912
19	0,000049	2,9583
21	0,000049	3,2697

Tabela 4.4 - Tabela com o padrão Ethernet.

93	0,000066	10,751
96	0,000075	9,7656
99	0,00007	10,79
125	0,000073	13,064
128	0,000083	11,766
515	0,000149	26,37
765	0,000194	30,085
768	0,000196	29,895
771	0,000197	29,859
1021	0,000246	31,665
1024	0,000242	32,283
1027	0,000244	32,112
1533	0,000333	35,123
2048	0,000361	43,283

A diferença de performance aqui encontrada, tanto para o módulo SYNC1394 quanto para o padrão IEEE1394 (assíncrono) é muito alta, fazendo com que a otimização do módulo desenvolvido seja crucial. Proposta de otimização e atualização do módulo SYNC1394 serão descritas no próximo capítulo.

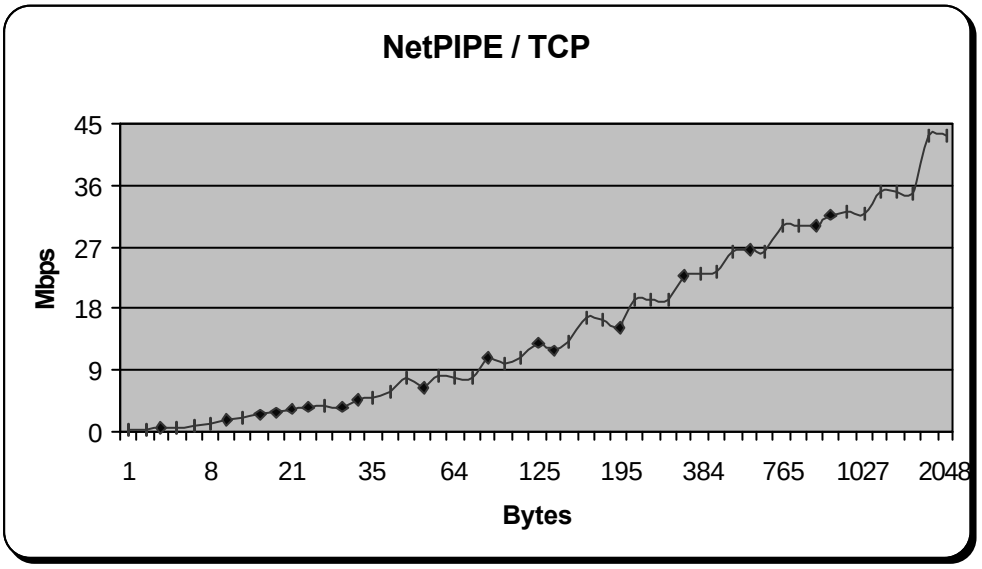

Figura 4.24 - Largura de banda do padrão Ethernet.

4.5 Dificuldades na Implementação

Durante o processo de criação do módulo SYNC1394, tivemos vários problemas, tais como: a falta de documentação sobre o barramento, ficando restrito ao trabalho de Anderson [AND99]; falta de documentação sobre o *driver* IEEE1394 para Linux, fato normal em implementações de código aberto, onde vários desenvolvedores opinam e contribuem para com o todo.

Passado o processo de compreensão do sistema a ser criado, encontramos mais algumas dificuldades, agora no processo de sincronização de um barramento que é naturalmente assíncrono. Para tanto se recorreu a utilização de espera ocupada (*busy-wait*) e semáforos no processo de sincronização do nó. O tratamento de *deadlocks* causados por falha de transmissão e/ou mau uso das primitivas fornecidas, não faz parte do escopo deste trabalho.

5 Aperfeiçoando o Módulo SYNC1394

Como pôde ser observado no capítulo anterior, ao fim do desenvolvimento do módulo SYNC1394 não foram obtidas velocidades de comunicação que pudessem ser aproveitadas em sistemas de alta performance. Esperando alcançar esta meta, iniciou-se um estudo que possibilitasse apontar os pontos onde, devidamente adaptados à nossa necessidade, pudéssemos superar os tempos de transmissão do padrão Ethernet. Assim não só aprimorar o sistema, como também, torná-lo uma nova opção para as redes de alto desempenho. Para tal meta podemos citar como importantes pontos de referência as especificações de utilização de códigos de resposta IEEE1394 [SPT95], da camada de transações [SCR96] e da interface do controlador [1394O], além do trabalho de Bartholdy [BAR01].

Adiante serão sugeridas algumas idéias de como acelerar as transações sobre o barramento, assim como novas implementações que possibilitarão a utilização do módulo SYNC1394 em um maior número de aplicações, de forma estável e confiável.

5.1 Otimizando o Protocolo

A primeira meta a ser alcançada é ultrapassar a barreira dos 100 Mbps nas transferências de mensagens, a fim de superar as transações com sockets TCP/IP. Para este fim, devemos nos concentrar no tipo de transação que está sendo utilizada neste momento, transações assíncronas com confirmação de pacotes.

O elevado tempo de transmissão associado à espera pela mensagem de confirmação por qualquer ação assíncrona é o responsável direto pelo baixo desempenho e mau aproveitamento do barramento. Tentando encontrar a melhor maneira de tratar este problema, propomos aqui três possíveis soluções, tanto em nível de código do módulo SYNC1394 quanto em nível do *driver* IEEE1394.

Sendo este um padrão protegido pelas regras de Licença GPL (GNU General Public License) – vide Anexo D, temos total liberdade de desenvolvermos um "novo" *driver* de baixo nível capaz de satisfazer nossas necessidades.

5.1.1 Por Que Não Utilizar Outro Tipo de Transação?

Relembrando o Capítulo 2, dentro do escopo das transações sobre o barramento IEEE1394, temos os seguintes modos de transmissão:

- Transações Isócronas, sem confirmação de pacotes e tratamento por buffer;
- Transações Assíncronas, com confirmação de pacotes e tratamento por pacote;
- Fluxo de Transmissão Assíncrona, ainda não implementado.

Analisando esses modos, podemos formular a pergunta: "Por que não utilizar outro tipo de transação no módulo SYNC1394?". A resposta é simples, os outros tipos de transação não atendem as nossas necessidades de maneira a torná-las uma real opção.

No caso das transações de fluxo assíncrono, assim que implementado, o tempo de transmissão será baixo, conforme afirmação de Anderson [AND99], já que ele seguirá uma política com pacotes isócronos e arbitragem assíncrona. A Figura 5.1 ilustra a estrutura de um pacote de fluxo assíncrono, sendo este o mesmo utilizado pelas transações isócronas.

Em contrapartida, a utilização de transações isócronas permitiria tempos de transmissão muito menores, podendo alcançar valores semelhantes aos obtidos por Boszorm'enyi [BOSZ], Fung [FUN02] e Yamagiwa [YAMW]. Para tal seria necessário efetuar um grande número de alterações e adaptações tanto no módulo SYNC1394 quanto no *driver* de baixo nível.

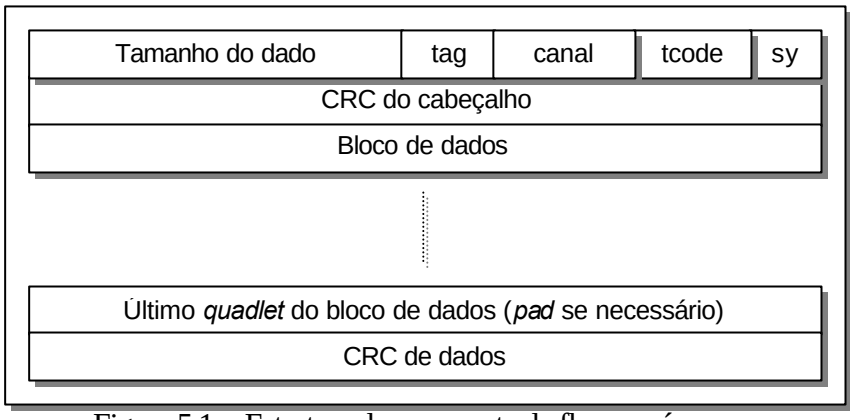

Figura 5.1 – Estrutura de um pacote de fluxo assíncrono.

Tais alterações consistiriam no gerenciamento do buffer de recepção de cada nó do barramento, onde os pacotes isócronos seriam tratados no momento de sua chegada, através da execução de uma operação de *flush* sobre o buffer, em tempos regulares (o que ocuparia o processador em uma tarefa desnecessária). Ou senão, adicionar ao final das mensagens um bloco de *pad,* de forma que toda mensagem transmitida, por menor que fosse, provocasse o seu tratamento imediato pelo nó receptor. Ambas as soluções resultariam em um *overhead*, além de que nenhuma das duas garantiria a integridade do pacote e seu correto tratamento.

5.1.2 Uma Nova Política para Pacotes de Confirmação

Outra possibilidade estaria em trabalhar sobre o modelo assíncrono, em busca de uma solução melhor. Esta solução passaria pela reescrita do código de envio de pacotes e remoção do trecho responsável pela espera de confirmação, e pelo gerenciamento da variável de controle de confirmações (*tlabel*), destacada na Figura 5.2. Para mais detalhes, leia as considerações de Anderson [AND99].

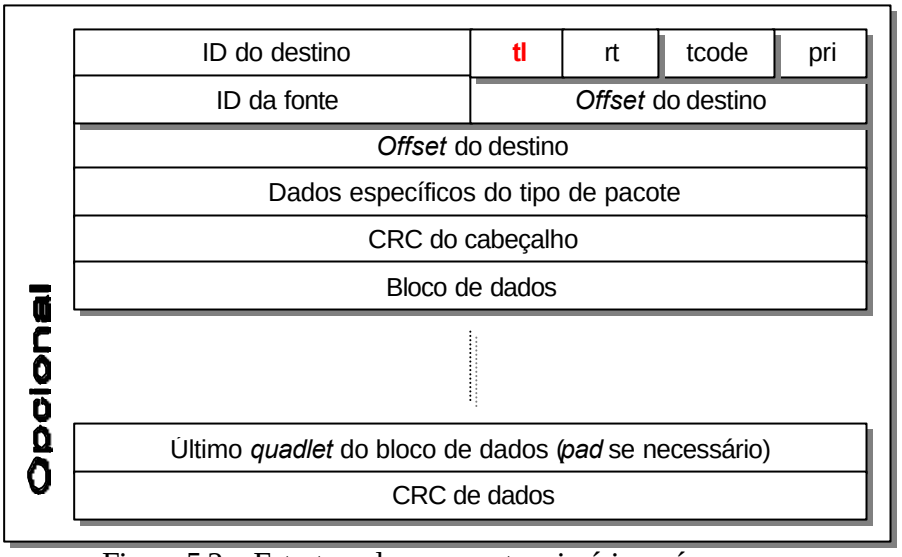

Figura 5.2 – Estrutura de um pacote primário assíncrono.

Testes preliminares mostraram que uma nova política sobre esta variável permitiria a transferência de pacotes em modo assíncrono com velocidade equiparável à isócrona. Por outro lado, abandonaríamos o padrão, pois não mais teríamos controle sobre os pacotes de confirmação, o que geraria a instabilidade do barramento e, ocasionalmente, o travamento do sistema.

A partir daí, temos dois caminhos distintos para a remoção da espera pelos pacotes de confirmação: liberar de vez a variável *tlabel*; ou, gerenciar separadamente sua criação e destruição.

5.1.2.1 Liberando a variável *tlabel*

A primeira tentativa está em reescrever o código e liberar as variáveis *tlabel*. O trecho de código na Figura 5.3, a seguir, mostra como deveria ficar a nova função de transmissão de pacotes.

```
struct hpsb_packet *p;

p = alloc_hpsb_packet(length);
if (!p) return -ENOMEM;

p->host = local_host;
p->tlabel = get_tlabel(local_host, 0xffc0 | dest, 1);
p->generation = get_hpsb_generation(local_host);
p->node_id = 0xffc0 | dest;

if (length > 4 ) {
     if (length % 4)
     p->data[length / 4] = 0;

     if (pt == msg)
          fill_async_writeblock(p, MSG_ADDR, length);
     else
          fill_async_writeblock(p, OTHER_ADDR, length);

     memcpy(p->data, buffer, length);
     }
else {
     if (pt == msg)
          fill_async_writequad(p, MSG_ADDR, *buffer);
     else
          fill_async_writequad(p, OTHER_ADDR, *buffer);
     }

err = hpsb_send_packet(p);

free_tlabel(local_host, 0xffc0 | dest, p->tlabel);
free_hpsb_packet(p);

return err;
```

Figura 5.3 – Envio de pacotes sem espera por confirmação.

Após o envio do pacote, tanto a variável *tlabel* quanto o próprio pacote devem ser liberados, deixando assim espaço para que um novo pacote possa ser transmitido. O problema desta abordagem está no funcionamento do sistema de confirmação. A Figura 5.4 ilustra este funcionamento.

Ao se remover *tlabel*, a mensagem de confirmação que carrega anexada a si uma cópia desta variável não é mais capaz de encontrar na fila de espera a mensagem com a

qual forma par, dessa forma ela passa a ser reconhecida como um pacote não solicitado. O recebimento consecutivo de pacotes não solicitados torna o sistema instável e provoca a sua pane.

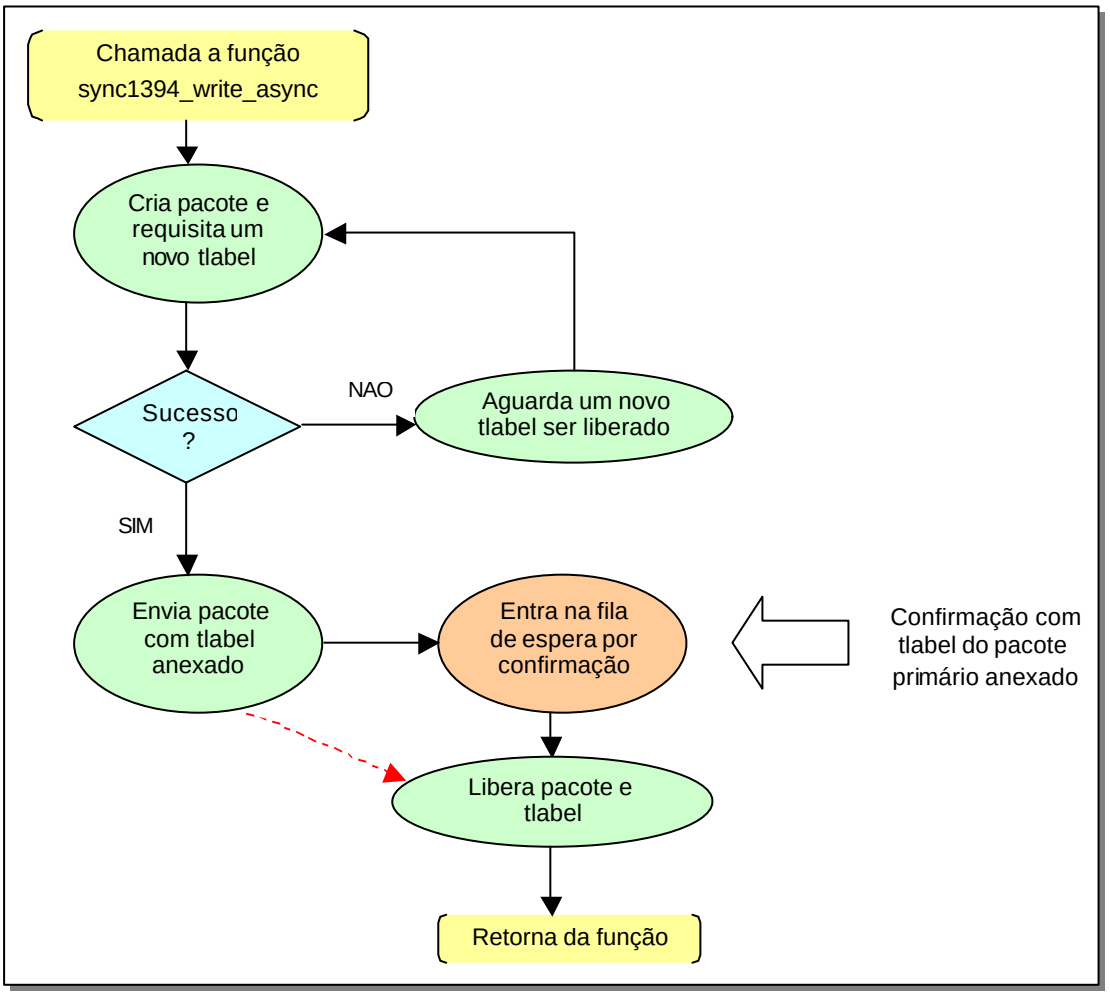

Figura 5.4 – Fluxo de execução do envio de pacotes.

5.1.2.2 Gerenciando o campo *tlabel*

Outra tentativa válida seria gerenciar as variáveis *tlabel*, criando-as, armazenando-as em uma estrutura e liberando-as quando os respectivos pacotes de confirmação

chegassem, mas com uma pequena diferença, a liberação ocorreria em blocos de 63 (número de *tlabels* disponíveis). O seguinte trecho de código, Figura 5.5, exemplifica esta idéia.

Tal política permitiria que blocos de até no máximo 63 pacotes (129024 bytes) fossem enviados antes que fosse necessário liberar o grupo de *tlabels*, esperando que o tempo necessário para alcançar a 64ª transmissão seja suficiente para a recepção dos pacotes de confirmação, referente às 63 transações anteriores. E assim, estes não seriam reconhecidos como tráfego indesejável. Uma forma mais simples de visualização seria imaginando uma rajada de pacotes, onde cada uma delas conteria até 63 pacotes, sendo que para enviar o próximo pacote seria necessário recarregar o nó transmissor com *tlabels* utilizadas anteriormente.

5.1.3 Declarando um Novo Tipo

Como terceira opção, tanto pelo fator abordagem do problema quanto pelo lado estético e lógico de implementação, a criação de um tipo de transação misto – que contenha características assíncronas e isócronas – seria a melhor opção a ser tomada. Este novo tipo deveria conter as seguintes qualidades:

- Tratamento por pacotes, como nas transações assíncronas;
- Sem confirmação de pacotes, como nas transações isócronas.

Para tal, observemos a Tabela 5.1 que ilustra os códigos utilizados atualmente pelas transações assíncronas. Nesta tabela há espaços não utilizados, que podem vir a ser aproveitados na nova implementação.

```
struct hpsb_packet *p;
static int n_tlabel = 0;

p = alloc_hpsb_packet(length);
if (!p) return -ENOMEM;

p->host = local_host;
if (n_tlabel == 63) {
     free_manager_fifo();
     ntlabel = 0;
}
p->tlabel = get_tlabel(local_host, 0xffc0 | dest, 1);
n_tlabel++;
p->generation = get_hpsb_generation(local_host);
p->node_id = 0xffc0 | dest;

add_packet_to_manager_fifo(p);

if (length > 4 ) {
     if (length % 4)
     p->data[length / 4] = 0;

     if (pt == msg)
          fill_async_writeblock(p, MSG_ADDR, length);
     else
          fill_async_writeblock(p, OTHER_ADDR, length);

     memcpy(p->data, buffer, length);
     }
else {
     if (pt == msg)
          fill_async_writequad(p, MSG_ADDR, *buffer);
     else
          fill_async_writequad(p, OTHER_ADDR, *buffer);
     }

return hpsb_send_packet(p);
```

Figura 5.5 – Gerenciando as variáveis *tlabel*.

Desta forma obteríamos um terceiro tipo, capaz de satisfazer as necessidades de uma aplicação síncrona e de alta performance.

Apenas como forma de exemplificação tomemos os seguintes tipos de transação:

- WRITE_DATA_REQUEST_NO_CONFIRM, para a nova transação de pacotes com no máximo 4 bytes, sem confirmação;
- WRITE_BLOCK_REQUEST_NO_CONFIRM, para a nova transação de pacotes com mais de 4 bytes, sem confirmação.

Tabela 5.1 – Tabela de códigos de transação assíncrona.

Nome da Transação	**Código (Hex)**
Write Request for data quadlet	0
Write Request for data block	1
Write Response	2
Reserved	3
Read Request for data quadlet	4
Read Request for data block	5
Read Response for data quadlet	6
Read Response for data block	7
Cycle start	8
Lock request	9
Asynchronous Streaming Packet	A
Lock response	B
Reserved	**C**
Reserved	**D**
Utilizado internamente	E
Reserved	F

Estes tipos podem ser atribuídos, respectivamente, aos valores C e D. Acompanhados de um tratamento lógico adequado, através da alteração das funções de envio e tratamento de pacotes, podemos obter um modelo de transação assíncrono sem confirmação, o qual teoricamente resolveria os problemas de performance do barramento.

Tais implementações devem ser feitas no *driver* de baixo nível, mais especificamente, nos arquivos *ieee1394_transaction* (para o algoritmo de envio) e *ieee1394_core* (para o algoritmo de tratamento). Certamente outros ajustes serão necessários.

5.2 Previsões de Desempenho

A partir de testes preliminares, executados através da remoção das linhas de código referentes à espera dos pacotes de confirmação (como apresentado no item 5.1.2), foi possível obter valores de performance superiores aos atuais e muito próximos aos por nós esperados em uma implementação final.

Neste teste foi utilizado o mesmo algoritmo (Capítulo 4) para captura dos tempos de transmissão. Tendo um único porém, o teste ficou limitado a transações de no máximo 2048 bytes, pois o envio de mais de um pacote, consecutivamente, causava pane no kernel do sistema. Pelo mesmo motivo, não foi possível utilizar o laço de iterações, sendo o processo de envio feito em etapa única. Como antes citado, há uma grande necessidade de buscar uma solução que não só garanta a entrega rápida e correta do pacote, mas que também garanta a estabilidade do sistema como um todo.

Na Tabela 5.2 temos os valores obtidos com este teste.

Tabela 5.2 - Teste com o modelo assíncrono sem confirmação.

Tamanho do Buffer (bytes)	Tempo de Transmissão (s)	Bandwidth (Mbps)
1	0,00004623875	0,165
4	0,00004805918	0,635
16	0,00004296738	2,841
32	0,00004601218	5,306
80	0,00004606079	13,251
160	0,00004394180	27,78
240	0,00004497359	40,714
512	0,00004897198	79,765
800	0,00004895502	124,676
1024	0,00004997537	156,327
1600	0,00008398197	145,353
1800	0,00008998578	152,612
2000	0,00008199154	186,102
2048	0,00010696487	146,076

Observamos aqui, uma melhora significativa nos tempos de transmissão, sendo que em determinados momentos podemos obter taxas próximas a 270 Mbps.

As Figuras 5.6 e 5.7 ilustram os valores obtidos.

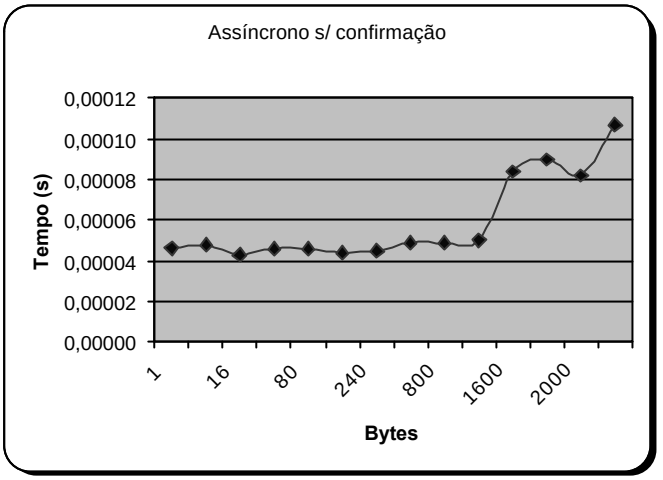

Figura 5.6 – Tempos de Transmissão da transação assíncrona sem confirmação.

Analisando estes gráficos, podemos observar que a partir de 800 bytes, o sistema passa a transmitir com velocidades acima de 100 Mbps, tornando-se uma boa opção frente ao padrão Ethernet, e até mesmo sobre o trabalho de [FIA02], que obteve tempos de no máximo 58 Mbps com seu *IP over IEEE1394*.

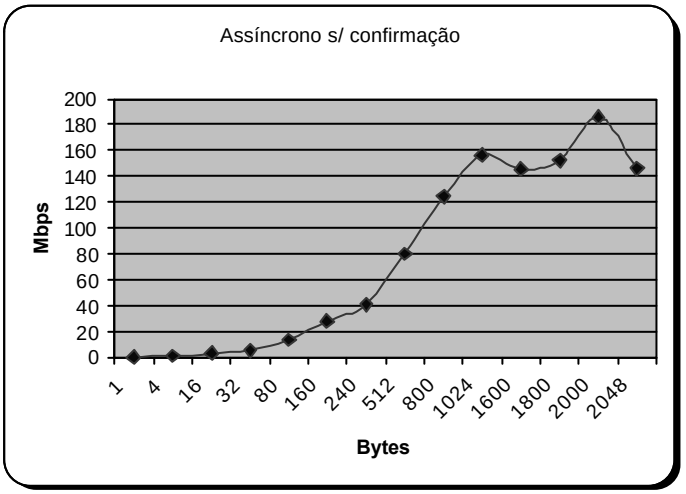

Figura 5.7 – Largura de banda da transação assíncrona sem confirmação.

6 Conclusão

Pretendeu-se com este trabalho realizar uma primeira abordagem sobre utilização da tecnologia IEEE1394 (Firewire) como suporte às transmissões síncronas de dados. O projeto teve como ponto de partida a necessidade de atualização da camada física do cluster CRUX, o qual possui módulos implementados no padrão Ethernet.

Como parte do esforço de desenvolvimento pode-se citar: o reconhecimento dos requisitos de comunicação do ambiente CRUX, o estudo de uma nova tecnologia de comunicação, a compreensão do funcionamento dos módulos Linux (suporte ao IEEE1394 e funções associadas), a programação de um *driver* síncrono e, finalmente, a detecção dos pontos que deverão ser corrigidos, para que se tenha um aproveitamento ótimo do sistema.

Observando-se os resultados obtidos, pode-se chegar a conclusão que a utilização do padrão IEEE1394, para tal fim, ainda apresenta algumas restrições, pois este não possui um modo de transferência de dados 100% compatível com as necessidades delineadas.

Em princípio, transferências isócronas fornecem uma boa resposta quanto a velocidade de transmissão, mas pecam no sistema de tratamento de pacotes (por *buffer*). Já, as transferências assíncronas apresentam a situação inversa, permitem um tratamento de pacotes ideal (por pacotes), mas a latência associada a espera pelos pacotes de confirmação torna esta opção inviável.

A partir dos conhecimentos adquiridos com o estudo desta tecnologia (hardware e software), pôde-se propor uma solução alternativa, a futura implementação de um novo tipo de transferência (síncrona), cujas principais características seriam a velocidade de transmissão, decorrente da não utilização de mensagens de confirmação e, o tratamento de mensagens por pacotes.

O resultado deste trabalho, no entanto, não se limitou apenas na proposta de um novo modo de transferência, como também serviu de ponto de partida para renovação das tecnologias utilizadas no meio acadêmico, mais especificamente, no Laboratório de Computação Paralela e Distribuída (LACPAD) da UFSC. Além do que, alcançou-se a implementação de um módulo de comunicação síncrona sobre IEEE1394, o qual atende

tanto a desenvolvedores com aplicações em nível de kernel, como também ao nível de usuário.

Ainda em tempo, pode-se citar que testes preliminares junto ao ambiente CRUX foram realizados, e estes validaram, até o momento, o funcionamento do módulo SYNC1394 e, também, confirmaram a necessidade de otimização em relação a velocidade de comunicação.

Buscando o aprimoramento deste trabalho, alguns trabalhos futuros fazem-se necessários. Primeiramente, a conclusão das pesquisas quanto a implementação do novo modo de transferência síncrona, levando-se em conta a manutenção do padrão IEEE, ou optando-se pelo desenvolvimento paralelo de um padrão específico para uso no LACPAD.

Um segundo ponto importante seria o estudo e, se possível, a implementação de um sistema de *zero-copying*, onde o nó receptor não tenha necessidade de armazenar os pacotes recebidos em um buffer, para então transferi-la ao espaço de usuário. A implementação de um sistema de memória compartilhada, para tal fim, simplificaria as trocas de contexto, diminuiria o tempo associado e aumentaria a capacidade das transferências.

Outro trabalho posposto é o de implementação de uma tabela de associação de endereços físicos e lógicos, criando uma camada de abstração. Ou seja, a atribuição de endereços lógicos aos nós permitiria maior flexibilidade aos usuários, pois não teriam que se preocupar com eventuais reconfigurações do barramento, e conseqüentemente, a mudança nos valores dos endereços físicos.

A inclusão de parâmetro referente a limites de espera (*time-out*) junto às primitivas de comunicação, pois desta forma poder-se-ia prever, tratar e evitar a ocorrência de *deadlocks* diretos (por má programação) ou indiretos (pela falha de algum nó e/ou má programação).

E por fim, mas não menos importante, o estudo de viabilidade de utilização do padrão IEEE1394 com aplicações em tempo real, fazendo-se uso dos próprios relógios de sincronização fornecidos pela arquitetura e utilizados nas transações.

7 Glossário

Acknowledge packet (Pacote de confirmação): pacote de 8 bits que pode ser transmitido em resposta à recepção de um pacote primário.

Arbitration (Arbitragem): processo pelo qual os nós competem pelo controle do barramento. Ao fim da decisão, o nó vencedor é capaz de transmitir um pacote.

Arbitration reset gap (Intervalo de reinício de arbitragem): o período mínimo de inatividade do barramento que separa intervalos "*fairness*".

Asynchronous packet (Pacote assíncrono): um pacote primário transmitido de acordo com as regras de arbitragem assíncrona (fora do período isócrono).

Asynchronous transaction (Transação assíncrona): uma transação executada durante um intervalo "*fairness*" que é usada por aplicações que não necessitam de uma largura de banda garantida e que pode tolerar longas latências entre transferências de dados.

Concatenated transaction (Transação concatenada): uma transação dividida, composta de sub-ações concatenadas.

Cycle start packet (Pacote de início de ciclo): um pacote primário enviado pelo controlador de ciclos que indica o início de um período isócrono.

Cycle master (Controlador de ciclo): um nó raiz responsável por iniciar cada intervalo isócrono (~125 µs) pela transmissão de um pacote de início de ciclo.

Fairness interval (Intervalo de igualdade): um período de tempo delimitado pelos intervalos de reinício de arbitragem. Dentro deste intervalo, o número total de pacotes

assíncronos que podem ser transmitidos por um nó é limitado. Cada limite de nó pode ser estabelecido explicitamente pelo gerenciador de barramento ou ele pode ser implícito.

Gap (Intervalo): um período de inatividade do barramento.

Isochronous (Isócrono): uniforme no tempo (isto é, tem duração igual) e ocorre em intervalos regulares.

Isochronous gap (Intervalo Isócrono): para uma sub-ação isócrona, o período de inatividade do barramento que precede a arbitragem.

Isochronous resource manager (Gerenciador de recursos isócronos): nó que implementa os registradores BUS_MANAGER_ID, BANDWIDTH_AVAILABLE, CHANNELS_AVAILABLE, e BROADCAST_CHANNEL (alguns quais permitem a alocação cooperativa de recursos isócronos). Subseqüente a cada inicialização do barramento, um gerenciador de recursos isócronos é selecionado de todos os nós capazes desta função.

Node (Nó)**:** um dispositivo do barramento serial que pode se endereçado independentemente de outros nós. Um nó mínimo consiste de somente uma camada física (PHY) sem um enlace habilitado. Se o enlace e outras camadas estão presentes e habilitadas elas serão consideradas parte do nó.

PHY: abreviação de Physical Layer (Camada Física).

Quadlet: um conjunto de quatro bytes de dados (tipo *u32* no Linux).

Repeater (Repetidor): função do PHY necessária para repassar o tráfego do barramento serial entre portas de nós multi-portas.

Request (Requisição): uma sub-ação usada para iniciar uma transação.

Response (Resposta): uma sub-ação usada para reportar o estado de finalização para o nó requisitante.

Transaction (Transação): uma transferência completa de informação de uma aplicação pata outra através do barramento serial. Transações consistem de um único pacote para transações isócronas e dois ou quatro pacotes para assíncronas.

8 Referências Bibliográficas

[1394H] **1394 High Performance Serial Bus**: Technical Overview. Disponível na Internet http://www.ti.com/sc/docs/products/msp/intrface/1394/tech.htm.

[1394O] **1394 Open Host Controller Interface Specification**. Disponível na Internet http://inanna.ecs.soton.ac.uk/~swh/mLAN/ohcir100.pdf. 2000. 210p.

[1394S] **1394 Specification for Response Code Usage**. Disponível na Internet http://www.1394ta.org/Technology/Specifications/Descriptions/RespCode.htm. 1996. 10p.

[1394T] **1394 Trade Association**. Disponível na Internet http://www.1394ta.org.

[1394TA] **1394TA IICP488 Specification for IEEE-488 Communications Using the Instrument & Industrial Control Protocol over IEEE1394**. Disponível na Internet http://ftp.agilent.com/pub/mpusup/ieee1394/ii-wg/iicp488_1_00rc2.pdf. 1394 Trade Association. 1999. 24p.

[ALE00] ALEXANDER, Bill. **IEEE1394 References and Programming page**. Disponível na Internet http://www.geocities.com/SiliconValley/Haven/4824/ieee1394.html.

[ALI00] ALICKE, F.; BARTHOLDY, F.; BLOZIS, S.; et al. **Comparing Bus Solution**. Texas Instruments. Disponível na Internet http://www-s.ti.com/sc/psheets/slla067/slla067.pdf. 2000. 67p.

[AND99] ANDERSON, Don; MINDSHARE, Inc. **Firewire System Architecture**: IEEE1394a. Reading, MA: Addison Wesley, 1999. 2ª Ed. 509 p.

[ANG02] ANGLANO, Cosimo. **Cluster Benchmarks Web Page**. Disponível na Internet http://www.mfn.unipmn.it/~mino/cluster/benchmarks/. 2002.

[BEC02] BECKER, Donald. **Using Linux Device Drivers as Modules**. Disponível na Internet http://www.embedded.com/1999/9906/9906feat2.htm. 2002.

[BOM01] BOMBE, Andreas. **Libraw1394: version 0.9.** Disponível na Internet http://dcine. dyndns.org/stuff/libraw1394-doc/. 1991.

[BOSZ] BOSZORM'ENYI, L'aszl'o; HOLZL, Gunther; PIRKER, Emanuel. **Parallel Cluster Computing with IEEE1394-1995**. Disponível na Internet http://citeseer. nj.nec.com/35537.html.

[BOU02] BOUWHUIS, Jarno G. **Real-Time Networking with IEEE1394 and HOTnet**. University of Twente. Disponível na Internet http://wwwes.cs .utwente.nl/dies/archive/master/PDFs/jarno.bouwhuis.pdf. 2002. 89p.

[BOV02] BOVET, Daniel P.; CESATI, Marco. **Understanding the Linux Kernel**: Process Scheduling. Disponível na Internet http://www.oreilly.com/ catalog/linuxkernel/chapter/ch10.html. 2002.

[BUD02] BUDAG, Karlos H. **Implementação do Núcleo do Sistema Operacional Distribuído Acrux**. Dissertação de Mestrado, CPGCC / UFSC. 2002.

[CAN99] CANOSA, John. **Fundamentals of Firewire**. Disponível na Internet http://www. embedded.com/1999/9906/9906feat2.htm. 1999.

[COL02] COLLINS, Ben; DENNEDY, Dan; MAAS, Dan; et al. **IEEE1394 for Linux**. Disponível na Internet http://www.linux1394.org/. 2002.

[COR99] CORSO, Thadeu B. **CRUX**: Ambiente Multicomputador Configurável por Demanda. Tese de Doutorado, CPGCC / UFSC. 1999.

[FIA02] FIALA, Guido. **IP over IEEE1394 under Linux**. Disponível na Internet http://www.s.netic.de/gfiala/IP_over_1394.html. 2002.

[FUN02] FUNG Y.F.; CHEUNG W.L.; SIRISENA, H.R. **A study of IEEE1394 for network computing**. Disponível na Internet http://www.cris.vt.edu/files /abstracts/crisabsIV5.pdf. 2002.

[GKAPI] GATLIFF, William. **The Linux Kernel's Interrupt Controller API**. Disponível na Internet http://billgatliff.com/articles/emb-linux/interrupts .pdf. 22p.

[GOL95] GOLDT, Sven; MEER, Sven van der; BURKETT, Scott; et al. **The Linux Programmer's Guide**. Disponível na Internet http://www.tldp.org/ LDP/lpg/lpg.html. 1995.

[GUS02] GUSTAVSON, Dave. **ATM, FibreChannel, HIPPI, Serialbus, SerialPlus SCI/LAMP, etc**. Disponível na Internet http://www.scizzl.com/ SCIvsEtc.html. 2002.

[HAR02] HARIRI, Salim. **IEEE CS Task Force on Cluster Computing**: Network Technologies. Disponível na Internet http://www.ece.arizona.edu/ ~hpdc/tfcc-network/. 2002.

[HOF02] HOFFMAN, Gary; MOORE, Daniel. **IEEE1394**: **A Ubiquitous Bus**. Disponível na Internet http://www.skipstone.com/compcon.html. 2002.

[IEE01] **IEEE1394 PPDT Protocol: Reference Manual.** Mindready Solutions Inc. Disponível na Internet http://www.emjembedded.com/programs/mindready manuals/PPDTReference Manual Ed.1Rev.pdf. 2001. 26p.

[IEE02] **IEEE1394 Technology**. Microsoft Corporation. Disponível na Internet http://www.microsoft.com/hwdev/bus/1394/default.asp. 2002.

[IEE96] **IEEE Standard for a High Performance Serial Bus**. Nova York, NY: The Institute of Electrical And Electronics Engineers, Inc. Disponível na Internet http://www.aquezada.com/staff/julian/school/ieee1394/. 1996. 392p.

[IPS01] **IP over IEEE1394 Stack: User Manual.** Mindready Solutions Inc. Disponível na Internet http://www.emjembedded.com/programs/mindreadymanuals/ 1394StackReferenceManualEd.pdf. 2001.

[JAI02] JAIN, Aakash; KOHLBECHER, Philipp; LIEBALD, Benjamin. **On IEEE1394**. Disponível na Internet http://www.faculty.iu-bremen.de/birk/lectures/ comparch/reports/ieee1394paper .pdf. 2002. 28p.

[KEVT] KEVILLE, Kurt L.; TOMPKIN, Robert. **IEEE1394 and RFC 2734; a viable HSI for hypercubes.** Disponível na Internet http://web.mit.edu/ kkeville/www/firewire/.

[LAW99] LAWSON, Kelvin. **Computer Networking over the IEEE1394 Serial Bus**. Beng Electronic & Computer Engineering. Disponível na Internet http://www.dcs.napier.ac.uk/ ~bill/pdf/1294.pdf.1999. 111p.

[LKAPI] **The Linux Kernel API**. Disponível na Internet http://kernelnewbies. org/documents/kdoc/ kernel-api.pdf. 391p.

[LPKA] LIM, Hyo-Sang; PARK, Dong-Hwan; KANG, Soon-Ju; et al. **Priority Queue-Based IEEE1394 Device Driver Supporting Real-Time Characteristics**. Disponível na Internet http://rtlab.knu.ac.kr/paper/priorityQueue-Consumer .pdf.

[MAX00] MAXWELL, Scott; **Kernel do Linux**. São Paulo, SP: Makron Books 2000. 308p.

[PPD00] **P1394.3 Draft Standard for a High Performance Serial Bus Peer-to-Peer Data Transport Protocol (PPDT)**. Disponível na Internet http://www.pwg.org/p1394/PPDT_r11.pdf. 2000. 80p.

[RUB01] RUBINI, Alessandro. **Linux Device Drivers**. Disponível na Internet http://www.xml.com/ ldd/chapter/book/. 2001. 586p.

[RUS00] RUSSELL, Paul Rusty. **Unreliable Guide To Locking**. Disponível na Internet http://kernelbook.sourceforge.net/kernel-locking.pdf. 2000. 24p.

[SBP01] **SBP-2 Initiator IEEE1394 SBP-2 Protocol.** Mindready. Disponível na Internet http://www.emjembedded.com/programs/mindreadymanuals/SBP-2Initiator ReferenceManu .pdf. 2001. 51p.

[SED01] **SedNet2 1394 Serial Bus Real-Time API**. Mindready. Disponível na Internet http://www.emjembedded.com/programs/mindreadymanuals/SedNet2Refere nceManualEd4.pdf. 2001. 114p.

[SPT95] **Specification for TRAN Layer Services**. 1995. 10p.

[STI01] **STK-IP-UM IP Over IEEE1394 Stack**. Mindready. Disponível na Internet

http://www.emjembedded.com/programs/mindreadymanuals/IPover1394UserManualEd2.pdf. 2001. 25p.

[STK01] **STKSBP-2-SRC-VXW IEEE1394 SBP-2 Target**. Mindready. Disponível na Internet http://www.emjembedded.com/programs/mindreadymanuals/SBP-2TargetReference Manual.pdf. 2001. 39p.

[TEE02] TEENER, Michael D. Johas. **Understandig Firewire**: The IEEE1394 Standards and Specifications. Disponível na Internet http://www.chipcenter .com/networking/ieee1394 /main.html. 2002.

[TUR02] TURNER, Dave. **NetPIPE:** A Network Protocol Independent Performance Evaluator. Disponível na Internet http://www.scl.ameslab.gov/netpipe/. 2002.

[YAMW] YAMAGIWA, Shinichi; WADA, Koichi. **Design and Implementation of Message Passing Library on Maestro Network**. Disponível na Internet http://citeseer.nj. nec.com/484348.html.

Anexo A – Código Fonte do SYNC1394

Interface em Nível de Usuário – sync1394.h

```
/*
        Interface SYNC1394 – Aplicação em nível de usuário

              Copyright 2003 Leonardo Soares Paulino.

   This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation. You should have
received a copy of the GNU General Public License along with this
program; if not, write to the

 Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA
02139, USA.

*/

#include <asm/unistd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>

/*****************************************************************/
/**     Syscall send_pckt(...)         - 230                    **/
/**     Syscall receive_size()         - 231                    **/
/**     Syscall receive_any_size()     - 233                    **/
/**     Syscall get_1394id()           - 234                    **/
/**     Syscall get_control_1394id()   - 235                    **/
/**     Syscall sync_1394()            - 236                    **/
/**     Syscall get_node_count()       - 237                    **/
/**     Syscall reset_1394()           - 238                    **/
/*****************************************************************/

#define __NR_send_pckt 230

_syscall3(int,send_pckt, void*, x, int, s, int, y)

#define __NR_receive_size 231

_syscall4(int ,receive_size, void*, b, int, s, int, o, int, d)

#define __NR_receive_any_size 233
```

```
_syscall4(int,receive_any_size, void*, b, int, s, int*, o, int, d)

#define __NR_get_1394id 234

_syscall0(int,get_1394id)

#define __NR_get_control_1394id 235

_syscall0(int,get_control_1394id)

#define __NR_sync_1394 236

_syscall0(int,sync_1394)

#define __NR_get_node_count 237

_syscall0(int,get_node_count)

#define __NR_reset_1394 238

_syscall0(int,reset_1394)


int send(void *buffer, const int size, const int dest) {

   return send_pckt(buffer, size, dest);
}

/*****************************************************************/
/** Receive : returns the buffer size, negative value if error  **/
/** *buffer_pckt : pointer to buffer packet who will return     **/
/** size : buffer size                                          **/
/** source : source node address                                **/
/*****************************************************************/

int receive(void *buffer_pckt, int size, int source) {

   int error;

   if (size <= 0) {
      printf("Error: Size value out of range = %d\n", size);

      return -EINVAL;
   }

   error = receive_size(buffer_pckt, size, source, 0);

   if (error < 0) {
```

```
      printf("Error: Transmition error receiving packet : %d\n",
error);

      return error;
   }

   return size;
}

/*************************************************************/
/** Receive_any : returns the buffer size, < 0 if  error       **/
/** *buffer_pckt : pointer to buffer packet who will return    **/
/** size : buffer size                                        **/
/** **source : pointer to value of source node address        **/
/*************************************************************/

int receive_any(void *buffer_pckt, int size, int **source) {

   int error;

   if (size <= 0) {
      printf("Error: Size value out of range = %d\n", size);

      return -EINVAL;
   }

   *source = malloc(sizeof(int));

   if (*source == NULL) {
      printf("Error: Not enough memory\n");

      return -ENOMEM;
   }

   error = receive_any_size(buffer_pckt, size, *source, 0);

    if (error < 0) {
      printf("Error: Transmition error, receiving packet : %d\n",
error);

      free(*source);

      return error;
   }

     return size;
}
```

Interface em Nível de Kernel – sync1394_mod.h

```
/*
        Interface SYNC1394 – Aplicação em nível de kernel

             Copyright 2003 Leonardo Soares Paulino.

   This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the  Free  Software  Foundation.  You  should  have
received a copy of the GNU General Public License along with this
program; if not, write to the

 Free  Software  Foundation,  Inc.,  675  Mass  Ave,  Cambridge,  MA
02139, USA.

*/

extern int sync1394_send(void*, int, int);
extern int sync1394_recv_size(void*, int, int);
extern int sync1394_recv_any_size(void*, int, int*);
extern int sync1394_get_1394id(void);
extern int sync1394_get_control_1394id(void);
extern int sync1394_sync_1394(void);
extern int sync1394_get_node_count(void);
extern int sync1394_reset_1394(void);

/*************************************************************/
/** Receive : returns the buffer size, negative value if error **/
/** *buffer_pckt : pointer to buffer packet who will return    **/
/** size : buffer size                                        **/
/** source : source node address                              **/
/*************************************************************/

static int sync1394_receive(void *buffer_pckt, int size, int
source) {

   int error;

   if (size <= 0) {
      printk("Error: Size value out of range = %d\n", size);

      return -EINVAL;
   }

   error = sync1394_recv_size(buffer_pckt, size, source);

   if (error < 0) {
```

```
      printk("Error: Transmition error, receiving packet : %d\n",
error);

      return error;
   }

   return size;
}

/*************************************************************/
/** Receive_any : returns the buffer size, < 0 if  error    **/
/** *buffer_pckt : pointer to buffer packet who will return  **/
/** size : buffer size                                       **/
/** **source : pointer to value of source node address       **/
/*************************************************************/

static int sync1394_receive_any(void *buffer_pckt, int size, int
**source) {

   int error;

   if (size <= 0) {
      printk("Error: Size value out of range = %d\n", size);

      return -EINVAL;
   }

   error = sync1394_recv_any_size(buffer_pckt, size, *source);

   if (error < 0) {
      printk("Error: Transmition error, receiving packet : %d\n",
error);

      return error;
   }

   return size;
}
```

Arquivo de Definições do Módulo – sync1394_core.h

```
/*
                    Definições SYNC1394 – Módulo

             Copyright 2003 Leonardo Soares Paulino.

   This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation. You should have
received a copy of the GNU General Public License along with this
program; if not, write to the

 Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA
02139, USA.

*/

#define __NO_VERSION__
#define MODULE

#ifndef __SYNC1394__
#define __SYNC1394__
#endif

#define EXPORT_SYMTAB

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/unistd.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/spinlock.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>
#include <asm/smplock.h>
#include <linux/vmalloc.h>

#include "ieee1394_types.h"
#include "ieee1394.h"
#include "hosts.h"
#include "ieee1394_core.h"
#include "highlevel.h"
#include "ieee1394_transactions.h"
```

```
#include "csr.h"
#include "nodemgr.h"
#include "raw1394.h"
#include "ieee1394_hotplug.h"
#include "ohci1394.h"

#define SYNC1394_DRIVER_NAME "sync1394"
char sync1394_dev_name[9]= "sync1394\0" ;

#ifdef DBGMSG
#undef DBGMSG
#endif

#ifdef SYNC1394_DEBUG
#define DBGMSG(card, fmt, args...) \
printk(KERN_ALERT "%s: " fmt "\n" , SYNC1394_DRIVER_NAME, ## args)
#else
#define DBGMSG(card, fmt, args...)
#endif


#define NODE_COUNT 3           // number of nodes
#define ISO_BUFFER 2048        // iso buffer size

/*
 * SYNC1394 types
 */

typedef enum { s_free, s_send, s_recv, s_off}
__attribute__((packed)) status_t;

struct sync1394_cluster_node_t {

     status_t status;
     // s_free = avaiable to transmit ou receive a packet
     // s_send = blocked waiting to complete a connection
     // s_recv = blocked waiting a connection
     // s_off  = node is disconnected

     int peer_addr;  // node what's connected or waiting
connection, 65 indicates disconnected
};

typedef struct sync1394_cluster_node_t
cluster_table_t[NODE_COUNT];

typedef enum { config, config_control, config_req_control,
unconfig, resume_send, wait_send, wait_receive, msg}
__attribute__((packed)) packet_type_t;
```

```
struct cluster_packet_t {
     int node_source;     // source node
     int node_target;     // target node
     packet_type_t type;
     /*
     config = configuration packet (connect, disconnect, cluster
configuration)
     mesg = mensage packet
     */
     __u8 signal;         // reserved to UNIX signals
     int protocol;        // reserved to new protocols (0 - 99)
     char data[8];        // config information (in cfg pckt)
};


typedef struct cluster_packet_t sync1394_packet_t;


/*
 * SYNC1394 encode/decode functions
 */

#define CR_NODE_MASK      34
#define CR_PROTOCOL_MASK  100
#define CR_SIGNAL_MASK    255


inline void sync1394_encode_msg(sync1394_packet_t *cp, char
*buffer) {

     int res;

     // packet structure (##@@!&&&%%%)
     // ## - node source
     // @@ - node target
     // !  - type
     // &&& - protocol
     // %%% - signal


     res = sprintf(buffer, "%d%d%d%d%d%d", cp->node_source +
CR_NODE_MASK, cp->node_target + CR_NODE_MASK, cp->type, cp-
>protocol + CR_PROTOCOL_MASK, cp->signal + CR_SIGNAL_MASK);

}

inline void sync1394_decode_msg(char * buffer, sync1394_packet_t*
cp) {
```

```
     int x;
     char temp[3];
     char temp2[50];

     temp[0] = buffer[0]; temp[1] = buffer[1]; temp[2] = '\0';
     cp->node_source = (int) simple_strtol ((char*)temp, (char **)
NULL, 10) - CR_NODE_MASK;

     temp[0] = buffer[2]; temp[1] = buffer[3]; temp[2] = '\0';
     cp->node_target = (int) simple_strtol ((char*)temp, (char **)
NULL, 10) - CR_NODE_MASK;

     temp[0] = buffer[4]; temp[1] = '\0';
     cp->type = (int) simple_strtol ((char*)temp, (char **) NULL,
10);

     temp[0] = buffer[5]; temp[1] = buffer[6]; temp[2] =
buffer[7];
     cp->protocol = (int) simple_strtol ((char*)temp, (char **)
NULL, 10) - CR_PROTOCOL_MASK;

     temp[0] = buffer[8]; temp[1] = buffer[9]; temp[2] =
buffer[10];
     cp->signal = (int) simple_strtol ((char*)temp, (char **)
NULL, 10) - CR_SIGNAL_MASK;

     for (x=11; x<strlen(buffer); x++)
          buffer[x-11] = buffer[x];
     buffer[strlen(buffer)-11] = '\0';
     strcpy(cp->data, buffer);
}

inline char* sync1394_encode_cfg(struct sync1394_cluster_node_t*
cn) {

     int res;
     static char cod[8];

     // packet structure (%%)
     // %% - peer node

     res = sprintf((char*)cod, "%d\0", cn->peer_addr +
CR_NODE_MASK);

     return (char*) cod;
}
```

```
inline void sync1394_decode_cfg(char *buff, struct
sync1394_cluster_node_t* cn) {

     char temp[3];

     temp[0] = buff[0]; temp[1] = buff[1]; temp[2] = '\0';
     cn->peer_addr = (int) simple_strtol ((char*)temp, (char **)
NULL, 10) - CR_NODE_MASK;
}


/*
 * SYNC1394 add pad - iso transmit
 */

inline void* sync1394_add_pad(void *buffer, int size) {

     void *buf;

     if (size < ISO_BUFFER) {
          buf = kmalloc(ISO_BUFFER, GFP_ATOMIC);
          if (buf == NULL)
               return NULL;

          memset(buf, 0, ISO_BUFFER);

          memcpy(buf, buffer, size);

          return buf;
     }
     else
          return buffer;

}

/*************************************************************/
/**   These variables and functions execute a busy-wait        **/
/*************************************************************/

inline int _signal_pending(struct task_struct *p)
{
  return (p->sigpending != 0);
}

extern struct task_struct *current;
extern asmlinkage void schedule(void);

inline int sync1394_yield(void)
{
```

```
  int ret = 0;

   if ( _signal_pending(current) )
    return -1;

   schedule();

   return ( ret );
}

/*
 * IEEE-1394 core driver related prototypes
 */

static void sync1394_add_host(struct hpsb_host *host);
static struct host_info *sync1394_find_host_info(struct hpsb_host
*host);
static void sync1394_host_reset(struct hpsb_host *host);
static void sync1394_remove_host(struct hpsb_host *host);
static void sync1394_iso_receive(struct hpsb_host *host, int
channel, quadlet_t *data, unsigned int length);
static void sync1394_async_receive(struct hpsb_host *host, int
nodeid, int direction, int cts, u8 *data, unsigned int length);

static inline int sync1394_write_async(quadlet_t *buffer, int
length, int dest, packet_type_t pt, int status);
static inline int sync1394_write_iso(quadlet_t *buffer, int
length, int channel);

void sync1394_config_node(int l_id, int pr_id);
void sync1394_config_remove_node(int l_id);
void sync1394_remote_register();
void sync1394_remote_unregister();

static int sync1394_exec_config(const void *data);
static int sync1394_exec_msg(const void *data, const unsigned int
length);

static int sync1394_control_msg(packet_type_t tp, int dest);

int sync1394_send(void *buffer, int size, int dest);
int sync1394_recv_size(void *buffer, const int size, const int
dest);
int sync1394_recv_any_size(void* buffer, int size, int *source);

static int sync1394_get_1394id(void);
static int sync1394_get_control_1394id(void);
static int sync1394_sync_1394(void);
static int sync1394_get_node_count(void);
```

```
static int sync1394_reset_1394(void);

EXPORT_SYMBOL_NOVERS(sync1394_send);
EXPORT_SYMBOL_NOVERS(sync1394_recv_size);
EXPORT_SYMBOL_NOVERS(sync1394_recv_any_size);
EXPORT_SYMBOL_NOVERS(sync1394_get_1394id);
EXPORT_SYMBOL_NOVERS(sync1394_get_control_1394id);
EXPORT_SYMBOL_NOVERS(sync1394_sync_1394);
EXPORT_SYMBOL_NOVERS(sync1394_get_node_count);
EXPORT_SYMBOL_NOVERS(sync1394_reset_1394);
```

Arquivo de Implementação do Módulo – sync1394.c

```
/*
                Implementação SYNC1394 – Módulo

            Copyright 2003 Leonardo Soares Paulino.

   This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the  Free  Software  Foundation.  You  should  have
received a copy of the GNU General Public License along with this
program; if not, write to the

 Free  Software  Foundation,  Inc.,  675  Mass  Ave,  Cambridge,  MA
02139, USA.

*/

#include "sync1394_core.h"

/*
 * Module load parameter definitions
 */

int local_node_id;
int control_node; //control node id
int MTU = 2048; //max packet size (lower limit : 4 bytes / upper
limit 2048 bytes)

MODULE_PARM(control_node,"i");
MODULE_PARM_DESC(control_node, "Node attempt to become Control
Node");

MODULE_PARM(MTU,"i");
MODULE_PARM_DESC(MTU, "Define MAX packet size (lower: 8, upper:
2044, best (default):1800)");

static struct hpsb_highlevel *hl_handle = NULL;

/* OHCI1394 highlevel driver functions */
struct hpsb_highlevel_ops sync1394_hl_ops = {
    add_host:           sync1394_add_host,
    remove_host:        sync1394_remove_host,
    host_reset:         sync1394_host_reset,
    iso_receive:        sync1394_iso_receive,
    fcp_request:        sync1394_async_receive
};
```

```
static LIST_HEAD(sync1394_host_info_list);
static int sync1394_host_count = 0;

void *recv_buffer;
unsigned int recv_length;

packet_type_t recv_type = msg;

static int recv_temp_size;
static int recv_atual;

struct hpsb_host *local_host;

int control_node_id = -1;
int register_status = 0; // 0 = need broadcast control address / 1
= OK
int register_init = 1;

cluster_table_t *node_table;

struct semaphore sync1394_recv_sem;
struct semaphore sync1394_process;
struct semaphore sync1394_reg_status;

#define sync1394_spin_lock(lock, flags)
                  spin_lock_irqsave(lock, flags);
#define sync1394_spin_unlock(lock, flags)
               spin_unlock_irqrestore(lock, flags);
static spinlock_t sync1394_host_info_lock = SPIN_LOCK_UNLOCKED;

//-----------


/*****************************************************************/
/** These functions ensure what just one process sends or receive
per time.                                                       **/
/*****************************************************************/

int sync1394_lock(){

     if (down_interruptible(&sync1394_process))
                       return -ERESTARTSYS;
     return 1;
}

void sync1394_unlock(){

     up(&sync1394_process);
}
```

```
/*****************************************************************/
/** This function is called after registering our operations in
sync1394_module_init.          **/
/** We go ahead and allocate some memory for our host info
structure, and                                                  **/
/** init some structures.
/*****************************************************************/
static void sync1394_add_host(struct hpsb_host *host)
{
       struct host_info *hi;
       unsigned long flags;

       printk(KERN_ALERT "sync1394_add_host\n");

      /* Allocate some memory for our host info structure */
      hi = (struct host_info *)kmalloc(sizeof(struct host_info),
GFP_KERNEL);

      if (hi == NULL) {
         printk(KERN_ALERT "out of memory in
sync1394_add_host\n");
         return;
      }

      /* Initialize some host stuff */
      memset(hi, 0, sizeof(struct host_info));
      INIT_LIST_HEAD(&hi->list);
      hi->host = host;

      sync1394_spin_lock(&sync1394_host_info_lock, flags);
      list_add_tail(&hi->list, &sync1394_host_info_list);
      sync1394_host_count++;

      local_host = host;

      printk(KERN_ALERT "Node id: %d\n", local_host->node_id &
NODE_MASK);
    printk(KERN_ALERT "Node count: %d\n", local_host->node_count);

      register_status = 0;

      sync1394_spin_unlock(&sync1394_host_info_lock, flags);

      return;
}

/*****************************************************************/
```

```
/** This fuction returns a host info structure from the host
structure, in                                                **/
/** case we have multiple hosts.
/*****************************************************************/

static struct host_info *sync1394_find_host_info(struct hpsb_host
*host)
{
       struct list_head *lh;
       struct host_info *hi;

       list_for_each (lh, &sync1394_host_info_list) {
           hi = list_entry(lh, struct host_info, list);
           if (hi->host == host) {
              return hi;
           }
       }

       return(NULL);
}


/*****************************************************************/
/** This function is called after host reset.                   **/
/*****************************************************************/

static void sync1394_host_reset(struct hpsb_host *host)
{

       local_host = host;

       printk(KERN_ALERT "Node id: %d\n", local_host->node_id &
NODE_MASK);
            printk(KERN_ALERT "Node count: %d\n", local_host-
>node_count);

       register_status = 0;
};

/*****************************************************************/
/** This function is called when a host is removed.             **/
/*****************************************************************/

static void sync1394_remove_host(struct hpsb_host *host)
{
       struct host_info *hi;
       unsigned long flags;

       printk(KERN_ALERT "sync1394_remove_host\n");
```

```
       sync1394_spin_lock(&sync1394_host_info_lock, flags);

       hi = sync1394_find_host_info(host);
       if (hi != NULL) {
             sync1394_host_count--;
             list_del(&hi->list);
             kfree(hi);
       }
       else
             printk(KERN_ALERT "attempt to remove unknown host
%p\n", host);

       local_host = NULL;

      sync1394_spin_unlock(&sync1394_host_info_lock, flags);
}

/*****************************************************************/
/**    This function is called when a iso packet is received.   **/
/*****************************************************************/

static void sync1394_iso_receive(struct hpsb_host *host, int
channel, quadlet_t *data, unsigned int length)
{
       DBGMSG(sync1394_dev_name, "sync1394_iso_received");
}

/*****************************************************************/
/** This function is called when a async request (fcp_request) is
received.                                                       **/
/*****************************************************************/

static void sync1394_async_receive(struct hpsb_host *host, int
nodeid, int direction, int cts, u8 *data, unsigned int length)
{
     int err;

     if (direction == 0) {  // packet_type = config | unconfig |
just_config | resume_send...

         DBGMSG(sync1394_dev_name,
“sync1394_async_control_received");

        sync1394_exec_config((void*)data);
     }
     else {  // packet_type = msg
        sync1394_exec_msg((void*)data, length);
       }
```

```
}

/*****************************************************************/
/**      This function deliver a received packet (command)      **/
/*****************************************************************/

static int sync1394_exec_config(const void *data){

        struct host_info *hi;
        sync1394_packet_t *p;
        struct sync1394_cluster_node_t *node;
        unsigned long flags;

        p = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

        if (p == NULL)
            return -ENOMEM;

        sync1394_decode_msg((char*)data, p);

       node = kmalloc(sizeof(struct sync1394_cluster_node_t),
GFP_KERNEL);

        if (node == NULL) {
            kfree(p);
            return -ENOMEM;
        }

        switch (p->type) {

           case config : {
                 sync1394_decode_cfg(p->data, node);
                 sync1394_config_node(p->node_source, node-
>peer_addr);
                 break;
           }

           case config_control : {
                  control_node_id = p->node_source;
                  register_status = 1;
                  if (register_init == 0){
                       register_init = 1;
                       up(&sync1394_reg_status);
                  }
                  break;
           }

           case config_req_control : {
                  if (control_node == 1)
```

```
                        register_status = 0;
                    break;
            }

            case unconfig : {
                  sync1394_decode_cfg(p->data, node);
                  sync1394_config_remove_node(p->node_source);
                  break;
            }

            case resume_send : {
                  DBGMSG(sync1394_dev_name, "Remote Resume_Send
received from node %d", p->node_source);
                 node_table[p->node_source]->status = s_recv;
                  break;
            }

            case wait_send : {
                  sync1394_spin_lock(&sync1394_host_info_lock,
flags);
                  DBGMSG(sync1394_dev_name, "Remote Wait_Send
received from node %d", p->node_source);
                  node_table[p->node_source]->status = s_send;
                  sync1394_spin_unlock(&sync1394_host_info_lock,
flags);
                  break;
            }

            case wait_receive : {
                  sync1394_spin_lock(&sync1394_host_info_lock,
flags);
                   DBGMSG(sync1394_dev_name, "Remote Wait_Receive
received from node %d", p->node_source);
                   node_table[p->node_source]->status = s_recv;
                   sync1394_spin_unlock(&sync1394_host_info_lock,
flags);
                    break;
            }

          default : DBGMSG(sync1394_dev_name, "Empty instruction");
                    break;
          }

          kfree(node);
          kfree(p);
          return 0;
}

/*****************************************************************/
```

```
/**      This function deliver a received packet (mensage)      **/
/*************************************************************/

static int sync1394_exec_msg(const void *data, const unsigned int
length){

       if (recv_length <= MTU) {
            memcpy(recv_buffer, data, recv_length);
            up(&sync1394_recv_sem);
       }
       else {
            if (recv_temp_size <= MTU) {
                  memcpy(recv_buffer + recv_atual, data,
recv_temp_size);
                  up(&sync1394_recv_sem);
            }
            else {
                 memcpy(recv_buffer + recv_atual, data, MTU);
       }

       recv_atual += MTU;
       recv_temp_size -= MTU;
       }

       return 0;
}

/*************************************************************/
/**   This function send a control packet (command) to "dest"   **/
/*************************************************************/

static int sync1394_control_msg(packet_type_t tp, int dest) {

       int err;
       sync1394_packet_t *p;
       char *q;

       p = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

       if (p == NULL)
           return -ENOMEM;

       p->node_source = local_node_id;
       p->node_target = dest;
       p->type = tp;
       p->protocol = 0;
       p->signal = 0;

       q = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);
```

```
        if (q == NULL) {
            kfree(p);
            return -ENOMEM;
        }

        sync1394_encode_msg(p, q);

        switch (tp) {

             case wait_send : {
                  DBGMSG(sync1394_dev_name, "Send Wait_Send to node
%d", dest);
                  break;
             }

             case wait_receive : {
                  DBGMSG(sync1394_dev_name, "Send Wait_Receive to
node %d", dest);
                  break;
             }

             case resume_send : {
                  DBGMSG(sync1394_dev_name, "Send Resume_Send to
node %d", dest);
                  break;
             }

             default : DBGMSG(sync1394_dev_name, "Empty
instruction");
                  break;
             }

             err = sync1394_write_async((quadlet_t*) q, strlen(q),
dest, tp, 0);

             kfree(q);
             kfree(p);

             return err;
}

/*****************************************************************/
/**       This function send a async packet to node "dest"      **/
/**                 dest = ALL_NODES -> broadcast               **/
/*****************************************************************/

#define MSG_ADDR CSR_REGISTER_BASE + CSR_FCP_RESPONSE
#define OTHER_ADDR CSR_REGISTER_BASE + CSR_FCP_COMMAND
```

```
static inline int sync1394_write_async(quadlet_t *buffer, int
length, int dest, packet_type_t pt, int status)
{
        if (pt == msg) {
            return hpsb_write(local_host, 0xffc0 | dest,
MSG_ADDR, (quadlet_t *) buffer, length);
        }
        else
           return hpsb_write(local_host, 0xffc0 | dest,
OTHER_ADDR, (quadlet_t *) buffer, length);
};

#undef MSG_ADDR
#undef OTHER_ADDR

/*****************************************************************/
/**    This function send a iso packet (command) to channel
"channel"                                                       **/
/*****************************************************************/

static inline int sync1394_write_iso(quadlet_t *buffer, int
length, int channel)
{
        int err = 0;

        quadlet_t* lbuf;
        struct hpsb_packet *p;

        p = alloc_hpsb_packet(ISO_BUFFER);
        if (!p) return -ENOMEM;

        p->host = local_host;
        p->generation = get_hpsb_generation(local_host);
        p->speed_code = 2;

        fill_iso_packet(p, ISO_BUFFER, channel, 0, 0);

      lbuf = (quadlet_t*) sync1394_add_pad((void*)buffer, length);

        memcpy(p->data, lbuf, ISO_BUFFER);

        err = hpsb_send_packet(p);

        if (length < ISO_BUFFER)
              kfree(lbuf);

        return !err;
```

```
};

/*************************************************************/
/** This function splits a packet and send each one         **/
/*************************************************************/

inline int send_big_buffer(const void *buffer, int size, const int
dest){

       int err;
       register int temp, atual;
       char buf[2048];

       atual = 0;

       while (size > 0) {

             if (size > MTU)
                  temp = MTU;
             else
                  temp = size;

             memcpy(buf, (buffer + atual), temp);

            err = sync1394_write_async((quadlet_t*) buf, temp,
dest, msg, 1);

              if (err < 0) {
                  return err;
              }

              size -= MTU;
              atual += MTU;
        }

        return err;
}

/*************************************************************/
/** This function is called by __syscall 230                **/
/*************************************************************/

asmlinkage long (*original_call_0)(void);

#define SYSCALL_SEND_PCKT 230 //entrada na sys_call_table para
interceptar (entrada nao usada) - SEND_PCKT

/* send for user-aplications    */
```

```
asmlinkage int send_pckt(void *buffer, const int size, const int
dest)
{
          int err, out;
          unsigned int flags, policy;

#ifdef SYNC1394_BENCHMARK
          unsigned long jif;
#endif

//MOD_INC_USE_COUNT;

          sync1394_lock();

#ifdef SYNC1394_BENCHMARK
          jif = jiffies;
#endif

         if ((control_node == 1) && (register_status == 0)) {
               err = sync_1394();
               if (err < 0)
                           goto exit;
          }

          if (node_table[dest]->status == s_recv) { //peer node
alreary executed

                node_table[dest]->status = s_free;

                //send buffer
                if (size <= MTU) {
                    err = sync1394_write_async((quadlet_t*)
buffer, (int) size, dest, msg, 0);
                     if (err < 0)
                             goto exit;
                }
                else {
                   err = send_big_buffer(buffer, size, dest);
                   if (err < 0)
                             goto exit;
                }

            //wake up the semaphore

         }
         else { //waiting peer node

           err = sync1394_control_msg(wait_send, dest);
           if (err < 0)
```

```
                              goto exit;

            DBGMSG(sync1394_dev_name, "Node %d stoped in SEND
busy-wait", local_node_id);

            out = 0;  // wait for receive-side

            while (out == 0) {

                  if (node_table[dest]->status == s_recv) { //peer
node executed
                      node_table[dest]->status = s_free;
                      out = 1;
                  }
                  else {
                      err = sync1394_yield();

                      if (err < 0) goto exit;
                  }
            }

            //send buffer
           if (size <= MTU) {
                 err = sync1394_write_async((quadlet_t*) buffer,
(int) size, dest, msg, 0);
                 if (err < 0)
                           goto exit;
           }
           else {
                err = send_big_buffer(buffer, size, dest);
                if (err < 0)
                           goto exit;
            }

           //wake up the semaphore

     }

//MOD_DEC_USE_COUNT;

#ifdef SYNC1394_BENCHMARK
       jif = jiffies - jif;
       printk("SYNC1394 Elapsed Time ( HZ = %d) - Send Operation -
%d / HZ\n", HZ, jif);
#endif

exit :     sync1394_unlock();

           return err;
```

```
}

/* send for kernel-aplications  */

int sync1394_send(void *buffer, const int size, const int dest){

         return send_pckt(buffer, size, dest);
}

/*****************************************************************/
/**       This function is called by __syscall 231              **/
/*****************************************************************/

asmlinkage long (*original_call_1)(void);

#define SYSCALL_RECV_SIZE 231 //entrada na sys_call_table para
interceptar (entrada nao usada) - RECEIVE_SIZE

/* receive_size for user-aplications  */

asmlinkage int receive_size(void *buffer, const int size, const
int dest, const int direction)
{
        int err;
        unsigned int flags, policy;
        unsigned long flag;
#ifdef SYNC1394_BENCHMARK
        unsigned long jif;
#endif

//MOD_INC_USE_COUNT;

        sync1394_lock();

#ifdef SYNC1394_BENCHMARK
        jif = jiffies;
#endif

       if ((control_node == 1) && (register_status == 0)) {
              err = sync_1394();
              if (err < 0)
                            goto exit;
        }

        if (direction == 0)
              recv_buffer = vmalloc(size);
        else
              recv_buffer = buffer;
```

```
        if (recv_buffer == NULL) {
              err = -ENOMEM;
              goto exit;
        }

        recv_length = size;

        recv_temp_size = size;

        recv_atual = 0;

     if (node_table[dest]->status == s_send) { //peer node alreary
executed

                node_table[dest]->status = s_free;

                err = sync1394_control_msg(resume_send, dest);

                if (err < 0)
                            goto exit;

                DBGMSG(sync1394_dev_name, "Node %d stoped in
RECEIVE semaphore", local_node_id);

               if (down_interruptible(&sync1394_recv_sem)) {
                        err = -ERESTARTSYS;
                         goto exit;
                       }
        }
        else {

         err = sync1394_control_msg(wait_receive, dest);

          if (err < 0)
                 goto exit;

          DBGMSG(sync1394_dev_name, "Node %d stoped in RECEIVE
semaphore", local_node_id);

           if (down_interruptible(&sync1394_recv_sem)) {
                       err = -ERESTARTSYS;
                        goto exit;
           }
                                }

           // BUFFER OK

          sync1394_spin_lock(&sync1394_host_info_lock, flags);
          if (direction == 0)
```

```
             err = copy_to_user(buffer, recv_buffer, recv_length);
          //else
               //memcpy(buffer, recv_buffer, recv_length);
            sync1394_spin_unlock(&sync1394_host_info_lock, flags);

//MOD_DEC_USE_COUNT;

exit: recv_length = 0;

      recv_temp_size = 0;

      recv_atual = 0;

      if ((direction == 0) && (recv_buffer != NULL))
            vfree(recv_buffer);
      else
            recv_buffer = NULL;

#ifdef SYNC1394_BENCHMARK
           jif = jiffies - jif;
           printk("SYNC1394 Elapsed Time ( HZ = %d) - Receive
Operation - %d / HZ\n", HZ, jif);
#endif

       sync1394_unlock();

       if (err < 0)
               return err;
       else
               return recv_length;

}

/* receive_size for kernel-aplications  */

int sync1394_recv_size(void *buffer, const int size, const int
dest){

       return receive_size(buffer, size, dest, 1);
}


/*****************************************************************/
/**    This function is called by __syscall 233                 **/
/*****************************************************************/

asmlinkage long (*original_call_3)(void);
```

```
#define SYSCALL_RECV_ANY_SIZE 233 //entrada na sys_call_table para
interceptar (entrada nao usada) - RECEIVE_ANY_SIZE

/* receive_any_size for user-aplications  */

asmlinkage int receive_any_size(void *buffer, const int size, int
*source, const int direction)
{
         register int err = -1, out;
         register int x;
         int dest = -1;
         unsigned int flags, policy;
#ifdef SYNC1394_BENCHMARK
         unsigned long jif;
#endif

//MOD_INC_USE_COUNT;

         sync1394_lock();

#ifdef SYNC1394_BENCHMARK
         jif = jiffies;
#endif

         if ((control_node == 1) && (register_status == 0)) {
              err = sync_1394();
              if (err < 0)
                             goto exit;
         }

         if (direction == 0)
              recv_buffer = vmalloc(size);
         else
              recv_buffer = buffer;

         if (recv_buffer == NULL) {
              err = -ENOMEM;
              goto exit;
         }

         recv_length = size;

         recv_temp_size = size;

         recv_atual = 0;

         out = 0;
```

```
        DBGMSG(sync1394_dev_name, "Node %d stoped in RECEIVE_ANY
busy-wait", local_node_id);

        while (out == 0) {

             for (x = 0; x < NODE_COUNT; x++) {

                 if (node_table[x]->status == s_send) { //peer
node alreary executed

                     node_table[x]->status = s_free;
                     out = 1;
                     dest = x;
                     break;
                 }

                 if (dest < 0) {
                   err = sync1394_yield();
                   if (err < 0) goto exit;
                 }
             }
        }

        if (dest > -1) {
               if (direction == 0)
                    put_user(source, &dest);
               else
                    memcpy(source, &dest, sizeof(int));
        }
        else {
               err = -EDOM;
               goto exit;
        }

         err = sync1394_control_msg(resume_send, dest);

         if (err < 0)
               goto exit;

        DBGMSG(sync1394_dev_name, "Node %d stoped in RECEIVE_ANY
semaphore", local_node_id);

          if (down_interruptible(&sync1394_recv_sem)) {
               err = -ERESTARTSYS;
               goto exit;
           }

        // BUFFER OK
```

```
        sync1394_spin_lock(&sync1394_host_info_lock, flags);
         if (direction == 0)
              err = copy_to_user(buffer, recv_buffer, recv_length);
         //else
             //memcpy(buffer, recv_buffer, recv_length);
        sync1394_spin_unlock(&sync1394_host_info_lock, flags);

//MOD_DEC_USE_COUNT;

exit: recv_length = 0;

      recv_temp_size = 0;

      recv_atual = 0;

      if ((direction == 0) && (recv_buffer != NULL))
              vfree(recv_buffer);
      else
              recv_buffer = NULL;

#ifdef SYNC1394_BENCHMARK
      jif = jiffies - jif;
      printk("SYNC1394 Elapsed Time ( HZ = %d) - Receive Any
Operation - %d / HZ\n", HZ, jif);
#endif

      sync1394_unlock();

      if (err < 0)
         return err;
      else
         return recv_length;
}


/* receive_any_size for kernel-aplications  */

int sync1394_recv_any_size(void *buffer, int size, int *source) {

        return receive_any_size(buffer, size, source, 1);
}

/*****************************************************************/
/**       This function is called by __syscall 234              **/
/*****************************************************************/

asmlinkage long (*original_call_4)(void);
```

```
#define SYSCALL_GET_1394ID 234 //entrada na sys_call_table para
interceptar (entrada nao usada) - GET_1394ID

/* get_id for user-aplications  */

asmlinkage int get_1394id(void){

       return local_node_id;
}

/* get_id for kernel-aplications  */

static int sync1394_get_1394id(void){

       return get_1394id();
}

/*****************************************************************/
/**     This function is called by __syscall 235                **/
/*****************************************************************/

asmlinkage long (*original_call_5)(void);

#define SYSCALL_GET_CONTROL_1394ID 235 //entrada na sys_call_table
para interceptar (entrada nao usada) - GET_CONTROL_1394ID

/*  get_control_id for user-aplications  */

asmlinkage int get_control_1394id(void){

      return control_node_id;
}

/*  get_control_id for kernel-aplications  */

static int sync1394_get_control_1394id(void){

      return get_control_1394id();
}

/*****************************************************************/
/**          This function is called by __syscall 236          **/
/*****************************************************************/

asmlinkage long (*original_call_6)(void);

#define SYSCALL_SYNC_1394 236 //entrada na sys_call_table para
interceptar (entrada nao usada) - SYNC_1394
```

```
/* init for user-aplications   */

asmlinkage int sync_1394(void){

       int x;

       if (register_status == 0) {

           if (control_node == 1) {
                 control_node_id = local_node_id;
                sync1394_remote_register(config_control);
                 register_status = 1;
           }
           else {
                sync1394_remote_register(config_req_control);
                 register_init = 0;
                 while (register_status == 0) {
                      if (sync1394_yield() < 0)
                          return -EINTR;
                }
          }
       }

        return 0;
}

/* init for kernel-aplications  */

static int sync1394_sync_1394(void){

       return sync_1394();
}

/*****************************************************************/
/**          This function is called by __syscall 237          **/
/*****************************************************************/

asmlinkage long (*original_call_7)(void);

#define SYSCALL_GET_NODE_COUNT 237 //entrada na sys_call_table
para interceptar (entrada nao usada) - GET_NODE_COUNT

/* init for user-aplications  */

asmlinkage int get_node_count(void){

      return NODE_COUNT;
}
```

```
/* init for kernel-aplications  */

static int sync1394_get_node_count(void){

      return get_node_count();
}

/*****************************************************************/
/**  This function is called by __syscall 238                   **/
/*****************************************************************/

asmlinkage long (*original_call_8)(void);

#define SYSCALL_RESET_1394 238 //entrada na sys_call_table para
interceptar (entrada nao usada) - RESET_1394

/* reset for user-aplications  */

asmlinkage int reset_1394(void){

         int x;
         recv_length = 0;
         recv_type = msg;

         if (recv_buffer != NULL)
                        vfree(recv_buffer);

         lock_kernel();
         init_MUTEX_LOCKED(&sync1394_recv_sem);
         init_MUTEX(&sync1394_process);
         init_MUTEX_LOCKED(&sync1394_reg_status);
         unlock_kernel();

         sync1394_remote_register(config);

         return 0;
}

/* reset for kernel-aplications  */

static int sync1394_reset_1394(void){

         return reset_1394();
}

/*****************************************************************/
/**   Config procedures                                         **/
/*****************************************************************/
```

```
void sync1394_config_node(int l_id, int pr_id)
{
        unsigned long flags;

       sync1394_spin_lock(&sync1394_host_info_lock, flags);

        node_table[l_id]->status = s_free;
        node_table[l_id]->peer_addr = pr_id;

        if (control_node == 1)
             DBGMSG(sync1394_dev_name, "Control node configured");
       else
             DBGMSG(sync1394_dev_name, "Node %d configured",l_id);

      sync1394_spin_unlock(&sync1394_host_info_lock, flags);
}

void sync1394_config_remove_node(int l_id)
{
       unsigned long flags;

       sync1394_spin_lock(&sync1394_host_info_lock, flags);

       node_table[l_id]->status = s_off;
       node_table[l_id]->peer_addr = 65;

       if (control_node == 1)
          DBGMSG(sync1394_dev_name, "Control node removed");
       else
          DBGMSG(sync1394_dev_name, "Node %d removed", l_id);

      sync1394_spin_unlock(&sync1394_host_info_lock, flags);
}

void sync1394_remote_register(packet_type_t type) {

       int err;
       sync1394_packet_t *p;
       struct sync1394_cluster_node_t* node;
       char* q;

       node = kmalloc(sizeof(struct sync1394_cluster_node_t),
GFP_KERNEL);

      node->peer_addr = node_table[local_node_id]->peer_addr;

        p = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

      if (p == NULL)
```

```
         DBGMSG(sync1394_dev_name,"No memory avaiable %s",
SYNC1394_DRIVER_NAME);

         p->node_source = local_node_id;
         p->node_target = 64;
         p->type = type;
         p->protocol = 0;
         p->signal = 0;

        q = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

         if (q == NULL) {
              kfree(p);
              DBGMSG(sync1394_dev_name,"No memory avaiable %s",
SYNC1394_DRIVER_NAME);
         }

         sync1394_encode_msg(p, q);
       strcat(q, sync1394_encode_cfg(node)); //append encoded info

        switch (type) {

             case config : {
                  DBGMSG(sync1394_dev_name, "Node %d Broadcast
Configuration", local_node_id);
                   break;
             }

             case config_control : {
                   DBGMSG(sync1394_dev_name, "Control Node %d
Broadcast Address", local_node_id);
                    break;
             }

             case config_req_control : {
                   DBGMSG(sync1394_dev_name, "Work Node %d Request
Control Address", local_node_id);
                    break;
             }
        }

   err   =   sync1394_write_async((quadlet_t*)   q,   strlen(q),
ALL_NODES, config, 0);

         kfree(q);
         kfree(p);
         kfree(node);
}
```

```
void sync1394_remote_unregister() {

        int err;
        sync1394_packet_t *p;
        struct sync1394_cluster_node_t* node;
                              char* q;

      node = kmalloc(sizeof(struct sync1394_cluster_node_t),
GFP_KERNEL);

       node->peer_addr = 65;

        p = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

       if (p == NULL)
         DBGMSG(sync1394_dev_name,"No memory avaiable %s",
SYNC1394_DRIVER_NAME);

       p->node_source = local_node_id;
       p->node_target = 64;
       p->type = unconfig;
       p->protocol = 0;
       p->signal = 0;

        q = kmalloc(sizeof(sync1394_packet_t), GFP_KERNEL);

       if (q == NULL) {
               kfree(p);
               DBGMSG(sync1394_dev_name,"No memory avaiable %s",
SYNC1394_DRIVER_NAME);
        }

        sync1394_encode_msg(p, q);
        strcat(q, sync1394_encode_cfg(node)); //append encoded
info

         DBGMSG(sync1394_dev_name, "Node %d Broadcast Unregister",
local_node_id);

         err = sync1394_write_async((quadlet_t*) q, strlen(q),
ALL_NODES, unconfig, 0);

         kfree(q);
         kfree(p);
         kfree(node);
}

/*************************************************************/
/**                        Init procedure                   **/
```

```
/*****************************************************************/

static int sync1394_module_init(void)
{
       int x;
       extern long sys_call_table[];

      /* Ieee1394 highlevel driver registering */
       hl_handle = hpsb_register_highlevel(SYNC1394_DRIVER_NAME,
&sync1394_hl_ops);
       if (hl_handle == NULL) {
           DBGMSG(sync1394_dev_name,"No more memory for driver
%s", SYNC1394_DRIVER_NAME);
           return -ENOMEM;
       }

         local_node_id = local_host->node_id & NODE_MASK;
    DBGMSG(sync1394_dev_name,"Local Node Id = %d", local_node_id);

     if ((MTU < 4) || (MTU > 2048)) {
          printk("MTU (%d) out of bounds. Setting default
value\n", MTU);
          MTU = 2048;
     }

     DBGMSG(sync1394_dev_name,"Defined MTU = %d bytes", MTU);

   //hpsb_listen_channel(hl_handle, local_host, local_node_id);
   //DBGMSG(sync1394_dev_name,"Local Node Listen Channel = %d",
local_node_id);

    // self configuration
    node_table = kmalloc(sizeof(cluster_table_t), GFP_KERNEL);
    for (x = 0; x < NODE_COUNT; x++){
                 node_table[x]->status = s_off;
                 node_table[x]->peer_addr = 65;
    }
    sync1394_config_node(local_node_id, 64);

    //remote configuration
    if (control_node == 1)    {
             control_node_id = local_node_id;
            sync1394_remote_register(config_control);
             register_status = 1;
    }

     sync1394_remote_register(config);
    //semaphores initialization
    lock_kernel();
```

```
      init_MUTEX_LOCKED(&sync1394_recv_sem);
      init_MUTEX(&sync1394_process);
      init_MUTEX_LOCKED(&sync1394_reg_status);
      unlock_kernel();

     /* Save the original routine address */
     original_call_0 = (long
(*)(void))(sys_call_table[SYSCALL_SEND_PCKT]);
     original_call_1 = (long
(*)(void))(sys_call_table[SYSCALL_RECV_SIZE]);
     original_call_3 = (long
(*)(void))(sys_call_table[SYSCALL_RECV_ANY_SIZE]);
     original_call_4 = (long
(*)(void))(sys_call_table[SYSCALL_GET_1394ID]);
     original_call_5 = (long
(*)(void))(sys_call_table[SYSCALL_GET_CONTROL_1394ID]);
     original_call_6 = (long
(*)(void))(sys_call_table[SYSCALL_SYNC_1394]);
     original_call_7 = (long
(*)(void))(sys_call_table[SYSCALL_GET_NODE_COUNT]);
     original_call_8 = (long
(*)(void))(sys_call_table[SYSCALL_RESET_1394]);

    /* Replace the syscall */
   sys_call_table[SYSCALL_SEND_PCKT]  = (unsigned long)send_pckt;
 sys_call_table[SYSCALL_RECV_SIZE]  = (unsigned long)receive_size;
  sys_call_table[SYSCALL_RECV_ANY_SIZE]  = (unsigned
long)receive_any_size;
 sys_call_table[SYSCALL_GET_1394ID]  = (unsigned long)get_1394id;
  sys_call_table[SYSCALL_GET_CONTROL_1394ID]  = (unsigned
long)get_control_1394id;
  sys_call_table[SYSCALL_SYNC_1394]  = (unsigned long)sync_1394;
   sys_call_table[SYSCALL_GET_NODE_COUNT]  = (unsigned
long)get_node_count;
  sys_call_table[SYSCALL_RESET_1394]  = (unsigned long)reset_1394;

     DBGMSG(sync1394_dev_name, "syscall(s) intercepted");

    printk("Succesfully load %s driver\n", sync1394_dev_name);

    return 0;
}

/*****************************************************************/
/**                          Exit procedure                     **/
/*****************************************************************/

static void __exit sync1394_module_exit(void)
```

```
{
      extern long sys_call_table[];

      //self unregister
      if (node_table) kfree(node_table);
          sync1394_config_remove_node(local_node_id);

     //remote unregister
     sync1394_remote_unregister();

    //hpsb_unlisten_channel(hl_handle, local_host, local_node_id);

     /* restaura endereco original */
     sys_call_table[SYSCALL_SEND_PCKT] = (unsigned
long)original_call_0;
     DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_SEND_PCKT);

     sys_call_table[SYSCALL_RECV_SIZE] = (unsigned
long)original_call_1;
     DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_RECV_SIZE);

     sys_call_table[SYSCALL_RECV_ANY_SIZE] = (unsigned
long)original_call_3;
     DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_RECV_ANY_SIZE);

      sys_call_table[SYSCALL_GET_1394ID] = (unsigned
long)original_call_4;
      DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_GET_1394ID);

      sys_call_table[SYSCALL_GET_CONTROL_1394ID] = (unsigned
long)original_call_5;
      DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_GET_CONTROL_1394ID);

      sys_call_table[SYSCALL_SYNC_1394] = (unsigned
long)original_call_6;
      DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_SYNC_1394);

        sys_call_table[SYSCALL_GET_NODE_COUNT] = (unsigned
long)original_call_7;
       DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_GET_NODE_COUNT);
```

```
        sys_call_table[SYSCALL_RESET_1394] = (unsigned
long)original_call_8;
        DBGMSG(sync1394_dev_name, "Syscall numero %d
restored",SYSCALL_RESET_1394);

        hpsb_unregister_highlevel(hl_handle);
        hl_handle = NULL;

       printk("Succesfully unload %s driver\n", sync1394_dev_name);

}

module_init(sync1394_module_init);
module_exit(sync1394_module_exit);


MODULE_LICENSE("GPL");
MODULE_AUTHOR("LSP - soares@inf.ufsc.br");
MODULE_DESCRIPTION("SYNC1394 - Synchronous Interface to
IEEE1394");
//EXPORT_NO_SYMBOLS;
```

Anexo B – Exemplo de Aplicação Produtor/Consumidor em nível de Usuário

Este programa produz uma coleção de números e os transmite ao nó consumidor, que utiliza a primitiva *receive any* dentro de um *loop while.*

```
#include <stdlib.h>
#include <sync1394/sync1394.h>

void no0(){

   int total = 0;

   int origem;

   int *p_origem = &origem;

   int r_buf;

   printf("Executando nó 0\n\n");

   while (0==0){

      printf("Esperando...\n\n");

      total = receive_any((void*)&r_buf, sizeof(int), &p_origem);

      printf("Valor recebido = %d\n\n", r_buf));

      printf("Bytes recebidos = %d / Nó origem: %d\n\n", total,
origem);

   }

   printf("Finalizando nó 0\n\n");
}

void no1(){

   int erro = 0;

   printf("Executando nó 1\n\n");

   erro = send((void*)&arg, sizeof(int), 0);
```

```
   printf("Finalizando nó 1\n\n");
}
int main(){
   int id = get_1394id();
   printf("ID do nó: %d\n", id);
   switch (id){
      case 0 : {
               no0();
               break;
      }
      case 1 : {
               no1();
               break;
      }
   }
_exit(0);
}
```

Anexo C – Exemplo de Aplicação Produtor/Consumidor em nível de Kernel

Este programa produz um número inteiro e o transmite ao nó consumidor, que utiliza a primitiva *receive* dentro de um *loop while*.

```
#define MODULE

#include <linux/errno.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>

#include <sync1394/sync1394_mod.h>

int produtor() {

    int t1 = 12345678;

    int i = sync1394_send((void*)&t1, sizeof(int), 1);
}

int consumidor() {

    int t2;

    int i = sync1394_receive(&t2, sizeof(int), 0);

    printk("Valor recebido: %d\n\n", t2);

}

int init_module() {

    int id, dest;

    id = sync1394_get_1394id();

    if (id == 0)

        produtor();

    else

        consumidor();
```

```
    return 0;
}

void cleanup_module()
{}


MODULE_LICENSE("GPL");
```

Anexo D – GNU General Public License

Tradução não oficial. http://lie-br.conectiva.com.br/licenca_gnu.html.

Licença pública GNU
GNU GENERAL PUBLIC LICENSE Version 2, June 1991

This is an unofficial translation of the GNU General Public License into Portuguese. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU GPL -- only the original English text of the GNU GPL does that. However, we hope that this translation will help Portuguese speakers understand the GNU GPL better.

F.3.1 Introdução

As licenças de muitos softwares são desenvolvidas para cercear a liberdade de uso, compartilhamento e mudanças. A GNU Licença Pública Geral ao contrário, pretende garantir a liberdade de compartilhar e alterar softwares de livre distribuição - tornando-os de livre distribuição também para quaisquer usuários. A Licença Pública Geral aplica-se à maioria dos softwares da Free Software Foundation e a qualquer autor que esteja de acordo de utilizá-la (alguns softwares da FSF são cobertos pela GNU Library General Public License).

Quando nos referimos a softwares de livre distribuição, referimo-nos à liberdade e não ao preço. Nossa Licença Pública Geral foi criada para garantir a liberdade de distribuição de cópias de softwares de livre distribuição (e cobrar por isso caso seja do interesse do distribuidor), o qual recebeu os códigos fonte, o qual pode ser alterado ou utilizado em parte em novos programas.

Para assegurar os direitos dos desenvolvedores, algumas restrições são feitas, proibindo a todas as pessoas a negação desses direitos ou a solicitação de sua abdicação. Essas restrições aplicam-se ainda a certas responsabilidades sobre a distribuição ou modificação do software.

Por exemplo, ao se distribuir cópias de determinado programa, por uma taxa determinada ou gratuitamente, deve-se informar sobre todos os direitos incidentes sobre esse programa, assegurando-se que as fontes estejam disponíveis assim como a Licença Pública Geral GNU.

A proteção dos direitos envolve dois passos: (1) copyright do software e (2) licença que dá permissão legal para cópia, distribuição e/ou modificação dos softwares.

Ainda para a proteção da FSF e do autor é importante que todos entendam que não há garantias para softwares de livre distribuição. Caso o software seja modificado por alguém e passado adiante, este software não mais refletirá o trabalho original do autor não podendo, portanto ser garantido por aquele.

Finalmente, qualquer programa de livre distribuição é constantemente ameaçado pelas patentes de softwares. Buscamos evitar o perigo de que distribuidores destes programas obtenham patentes individuais, tornado-se seus donos efetivos. Para evitar isso foram feitas declarações expressas de que qualquer solicitação de patente deve ser feita permitindo o uso por qualquer indivíduo, sem a necessidade de licença de uso.

Os termos e condições precisas para cópia, distribuição e modificação seguem abaixo:

F.3.2 Licença Pública Geral GNU

TERMOS E CONDIÇÕES PARA CÓPIA, DISTRIBUIÇÃO E MODIFICAÇÃO

0. Esta licença se aplica a qualquer programa ou outro trabalho que contenha um aviso colocado pelo detentor dos direitos autorais dizendo que aquele poderá ser distribuído nas condições da Licença Pública Geral. O Programa, abaixo se refere a qualquer software ou trabalho e a um trabalho baseado em um Programa e significa tanto o Programa em si como quaisquer trabalhos derivados de acordo com a lei de direitos autorais, o que significa dizer, um trabalho que contenha o Programa ou uma parte deste, na sua forma original ou com modificações ou traduzido para uma outra língua (tradução está incluída sem limitações no termo *modificação*).

Atividades distintas de cópia, distribuição e modificações não estão cobertas por esta Licença, estando fora de seu escopo. O ato de executar o Programa não está restringido e a saída do Programa é coberta somente caso seu conteúdo contenha trabalhos baseados no Programa (independentemente de terem sido gerados pela execução do Programa). Se isso é verdadeiro depende das funções executadas pelo Programa.

1. O código fonte do Programa, da forma como foi recebido, pode ser copiado e distribuído, em qualquer media, desde que seja providenciados uns avisos adequados sobre os copyrights e a negação de garantias, e todos os avisos que se referem à Licença Pública Geral e à ausência de garantias estejam inalterados e que quaisquer produtos oriundos do Programa esteja acompanhado desta Licença Pública Geral.

É permitida a cobrança de taxas pelo ato físico de transferência ou gravação de cópias, e podem ser dados garantias e suporte em troca da cobrança de valores.

2. Pode-se modificar a cópia ou cópias do Programa de qualquer forma que se deseje, ou ainda criar-se um trabalho baseado no Programa, e copiá-la e distribuir tais modificações sob os termos da seção 1 acima e do seguinte:

1. Deve existir aviso em destaque de que os dados originais foram alterados nos arquivos e as datas das mudanças;
2. Deve existir aviso de que o trabalho distribuído ou publicado é, de forma total ou em parte derivado do Programa ou de alguma parte sua, e que pode ser licenciado totalmente sem custos para terceiros sob os termos desta Licença.
3. Caso o programa modificado seja executado de forma interativa, é obrigatório, no início de sua execução, apresentar a informação de copyright e da ausência de garantias (ou de que a garantia corre por conta de terceiros), e que os usuários podem redistribuir o programa sob estas condições, indicando ao usuário como acessar esta Licença na sua íntegra.

Esses requisitos aplicam-se a trabalhos de modificação em geral. Caso algumas seções identificáveis não sejam derivadas do Programa, e podem ser consideradas como partes independentes, então esta Licença e seus Termos não se aplicam àquelas seções quando distribuídas separadamente. Porém ao distribuir aquelas seções como parte de um trabalho baseado no Programa, a distribuição como um todo deve conter os termos desta Licença, cujas permissões estendem-se ao trabalho como um todo, e não a cada uma das partes independentemente de quem os tenha desenvolvido.

Mais do que tencionar contestar os direitos sobre o trabalho desenvolvido por alguém, esta seção objetiva propiciar a correta distribuição de trabalhos derivados do Programa.

Adicionalmente, a mera adição de outro trabalho ao Programa, porém não baseado nele nem a um trabalho baseado nele, a um volume de armazenamento ou media de distribuição não obriga a utilização desta Licença e de seus termos ao trabalho.

3. São permitidas a cópia e a distribuição do Programa (ou a um trabalho baseado neste) na forma de código objeto ou executável de acordo com os termos das Seções 1 e 2 acima, desde que atendido o seguinte:

1. Esteja acompanhada dos códigos fonte legível, os quais devem ser distribuídos na forma das Seções 1 e 2 acima, em mídia normalmente utilizada para manuseio de softwares ou
2. Esteja acompanhado de oferta escrita, válida por, no mínimo 3 anos, de disponibilizar a terceiros, por um custo não superior ao custo do meio físico de armazenamento , uma cópia completa dos códigos fonte em meio magnético, de acordo com as Seções 1 e 2 acima.
3. Esteja acompanhada com a mesma informação recebida em relação à oferta da distribuição do código fonte correspondente. (esta alternativa somente é permitida para distribuições não comerciais e somente se o programa recebido na forma de objeto ou executável tenha tal oferta, de acordo com a sub-seção 2 acima).

O código fonte de um trabalho é a melhor forma de produzirem-se alterações naquele trabalho. Códigos fontes completos significam todos os fontes de todos os módulos, além das definições de interfaces associadas, arquivos, scripts utilizados na compilação e instalação do executável. Como uma exceção excepcional, o código fonte distribuído poderá não incluir alguns componentes que não se encontrem em seu escopo, tais como compilador, kernel, etc... para o SO onde o trabalho seja executado.

Caso a distribuição do executável ou objeto seja feita através de acesso a um determinado ponto, então oferta equivalente de acesso deve ser feita aos códigos fonte, mesmo que terceiros não sejam obrigados a copiarem os fontes juntos com os objetos simultaneamente.

4. Não é permitida a cópia, modificação, sublicenciamento ou distribuição do Programa, exceto sob as condições expressas nesta Licença. Qualquer tentativa de cópia, modificação, sublicenciamento ou distribuição do Programa é proibida, e os direitos

descritos nesta Licença cessarão imediatamente. Terceiros que tenham recebido cópias ou direitos na forma desta Licença não terão seus direitos cessados desde que permaneçam dentro das cláusulas desta Licença.

5. Não é necessária aceitação formal desta Licença, apesar de que não haverá documento ou contrato que garanta permissão de modificação ou distribuição do Programa ou seus trabalhos derivados. Essas ações são proibidas por lei, caso não se aceitem as condições desta Licença. A modificação ou distribuição do Programa ou qualquer trabalho baseado neste implica na aceitação desta Licença e de todos os termos desta para cópia, distribuição ou modificação do Programa ou trabalhos baseados neste.

6. Cada vez que o Programa seja distribuído (ou qualquer trabalho baseado neste), o recipiente automaticamente recebe uma licença do detentor original dos direitos de cópia, distribuição ou modificação do Programa objeto deste termos e condições. Não podem ser impostas outras restrições nos recipientes.

7. No caso de decisões judiciais ou alegações de uso indevido de patentes ou direitos autorais, restrições sejam impostas que contradigam esta Licença, estes não isentam da sua aplicação. Caso não seja possível distribuir o Programa de forma a garantir simultaneamente as obrigações desta Licença e outras que sejam necessárias, então o Programa não poderá ser distribuído.

Caso esta Seção seja considerada inválida por qualquer motivo particular ou geral, o seu resultado implicará na invalidação geral desta licença na cópia, modificação, sublicenciamento ou distribuição do Programa ou trabalhos baseados neste.

O propósito desta seção não é, de forma alguma, incitar quem quer que seja a infringir direitos reclamados em questões válidas e procedentes, e sim proteger as premissas do sistema de livre distribuição de software. Muitas pessoas têm feito

contribuições generosas ao sistema, na forma de programas, e é necessário garantir a consistência e credibilidade do sistema, cabendo a estes e não a terceiros decidirem a forma de distribuição dos softwares.

Esta seção pretende tornar claro os motivos que geraram as demais cláusulas desta Licença.

8. Caso a distribuição do Programa dentro dos termos desta Licença tenha restrições em algum País, quer por patentes ou direitos autorais, o detentor original dos direitos autorais do Programa sob esta Licença pode adicionar explicitamente limitações geográficas de distribuição, excluindo aqueles Países, fazendo com que a distribuição somente seja possível nos Países não excluídos.

9. A Fundação de Software de Livre Distribuição (FSF - Free Software Foundation) pode publicar versões revisadas ou novas versões desta Licença Pública Geral de tempos em tempos. Estas novas versões manterão os mesmos objetivos e o espírito da presente versão, podendo variar em detalhes referentes a novas situações encontradas.

A cada versão é dada um número distinto. Caso o Programa especifique um número de versão específico desta Licença a qual tenha em seu conteúdo a expressão - ou versão mais atualizada-, é possível optar pelas condições daquela versão ou de qualquer versão mais atualizada publicada pela FSF.

10. Caso se deseje incorporar parte do Programa em outros programas de livre distribuição de softwares é necessária autorização formal do autor. Para softwares que a FSF detenha os direitos autorais, podem ser abertas exceções desde que mantido o espírito e objetivos originais desta Licença.

11. AUSÊNCIA DE GARANTIAS

UMA VEZ QUE O PROGRAMA É LICENCIADO SEM ÔNUS, NÃO HÁ QUALQUER GARANTIA PARA O PROGRAMA. EXCETO QUANDO TERCEIROS EXPRESSEM-SE FORMALMENTE O PROGRAMA É DISPONIBILIZADO EM SEU FORMATO ORIGINAL, SEM GARANTIAS DE QUALQUER NATUREZA, EXPRESSAS OU IMPLÍCITAS, INCLUINDO, MAS NÃO LIMITADAS, A GARANTIAS COMERCIAIS E DO ATENDIMENTO DE DETERMINADO FIM. A QUALIDADE E A PERFORMANCE SÃO DE RISCO EXCLUSIVO DOS USUÁRIOS, CORRENDO POR SUAS CONTA OS CUSTOS NECESSÁRIOS A EVENTUAIS ALTERAÇÕES, CORREÇÕES E REPAROS JULGADOS NECESSÁRIOS.

EM NENHUMA OCASIÃO, A MENOS QUE REQUERIDO POR DECISÃO JUDICIAL OU POR LIVRE VONTADE, O AUTOR OU TERCEIROS QUE TENHAM MODIFICADO O PROGRAMA, SERÃO RESPONSÁVEIS POR DANOS OU PREJUÍZOS PROVENIENTES DO USO OU DA FALTA DE HABILIDADE NA SUA UTILIZAÇÃO (INCLUINDO, MAS NÃO LIMITADA A PERDA DE DADOS OU DADOS ERRÔNEOS), MESMO QUE TENHA SIDO EMITIDO AVISO DE POSSÍVEIS ERROS OU DANOS.

FIM DA LICENÇA

F.3.3 Apêndice

Como aplicar estes termos a novos softwares?

Caso se tenha desenvolvido um novo programa e se deseje a sua ampla distribuição para o público, a melhor forma de consegui-lo é torná-lo um software de livre distribuição, o qual qualquer um possa distribuí-lo nas condições desta Licença.

Para tanto basta anexar este aviso ao programa. É aconselhável indicar ainda no início de cada arquivo fonte a ausência de garantias e um apontamento para um arquivo contendo o texto geral desta Licença, como por exemplo:

<nome do programa e função> Copyright (C) 199X <Autor>

Este programa é um software de livre distribuição, que pode ser copiado e distribuído sob os termos da Licença Pública Geral GNU, conforme publicada pela Free Software Foundation, versão 2 da licença ou (a critério do autor) qualquer versão posterior.

Este programa é distribuído na expectativa de ser útil aos seus usuários, porém NÃO TEM NENHUMA GARANTIA, EXPLÍCITAS OU IMPLÍCITAS , COMERCIAIS OU DE ATENDIMENTO A UMA DETERMINADA FINALIDADE. Consulte a Licença Pública Geral GNU para maiores detalhes.

Deve haver uma cópia da Licença Pública Geral GNU junto com este software em inglês ou português. Caso não haja escreva para Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Autor@mail.com.br
Endereço

Caso o programa seja interativo, apresente na sua saída um breve aviso quando de seu início como por exemplo:

Gnomovision versão 69, Copyright ©199a Yoyodine

Softwares NÃO POSSUI NENHUMA GARANTIA; para detalhes digite **mostre garantia**. Este é um software de livre distribuição e você está autorizado a distribui-lo dentro de certas condições. Digite **mostre condição** para maiores detalhes.

Os comandos hipotéticos **mostre garantia** e **mostre condição** apresentarão as partes apropriadas da Licença Pública Geral GNU. Evidentemente os comandos podem variar ou serem acionado por outras interfaces como clique de mouse, etc...