MODELO DE ACOPLAMENTO DE ELEMENTOS DE SOFTWARE PARA SATELITES VIRTUAIS MULTIPLATAFORMA

Arthur Adriano Ferreira (UBC – Bolsista PIBIC/CNPq)
E-mail: arthuradriano@gmail.com

Dr. Ulisses Thadeu Vieira Guedes (DMC/INPE, Orientador)
E-mail: utvg@zedelho.dmc.inpe.br

Junho de 2009

MODELO DE ACOPLAMENTO DE ELEMENTOS DE SOFTWARE PARA SATELITES VIRTUAIS MULTIPLATAFORMA

Arthur Adriano Ferreira (UBC – Bolsista PIBIC/CNPq)
E-mail: arthuradriano@gmail.com

Dr. Ulisses Thadeu Vieira Guedes (DMC/INPE, Orientador)
E-mail: utvg@zedelho.dmc.inpe.br

Junho de 2009

Temos o destino que merecemos
O nosso destino esta de acordo com nossos méritos.
Albert Einstein

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me dado força quando meu corpo não aguentava mais, e também por ter renovado a esperança, quando esta já não mais se fazia presente.

Agradeço minha família pelo apoio, suporte e pela paciência que em momento algum me fora negada, e em particular a meu irmão, Samuel Ferreira, pelos momentos de descontração que muito me fizeram rir e me ajudaram a relaxar quando os momentos de tensão eram altos.

Agradeço, e muito, à minha amada e sempre companheira Ana Carolina, pelo carinho e amizade a mim demonstrados, por ter suportado minha ausência durante todo esse tempo dedicado a este trabalho, e por ter ajudado a me levantar nas varias vezes em que tropecei em um obstáculo.

Agradeço também ao meu professor e orientador Ulisses Guedes, pela enorme paciência para ensinar em detalhes durante todas as etapas deste projeto, mesmo quando em horários não tão convenientes, ao Fernando Pacheco, por ter me ajudado a pensar no conteúdo e na forma na parte de desenvolvimento gráfico deste trabalho, e aos amigos da faculdade, que sem dúvida deram boas dicas para melhor desempenho do software desenvolvido para este projeto.

MODELO DE ACOPLAMENTO DE ELEMENTOS DE SOFTWARE PARA SATÉLITES VIRTUAIS MULTIPLATAFORMA

RESUMO

Este trabalho, iniciado em agosto de 2009, tem como objetivo a continuidade ao projeto em andamento desde julho de 2002, para o desenvolvimento de módulos de acoplamento de software que permitam o uso de modelos dinâmicos diversos (sensores, atuadores e planta dinâmica), quanto o teste de métodos de transferência de dados através de redes de alta velocidade e enlaces seriais (baixa velocidade). Inicialmente o trabalho realizado em 2002 tratou de um teste conceitual posto em funcionamento/operação, onde se constatou a viabilidade técnica do projeto, sendo implementado até então as aplicações em redes TCP/IP para serviços unicast, anycast e multicast, envolvendo os protocolos de transporte TCP e UDP, e a aplicação em sistemas de I/O (Input/Output) em barramento serial. O trabalho atual trata do aprimoramento dos módulos de aplicação de serviços de rede, utilizando os protocolos TCP, UDP e ICMP, bem como o aprimoramento do módulo de comunicação serial e o desenvolvimento e a implementação de módulos de comunicação com portas paralelas e também portas e/ou dispositivos USB. Através de exaustivos testes de simulação podemos também prever e implementar o tratamento de erros possíveis ocasionados pela oscilação/sobrecarga do sistema operacional, ou por perda de comunicação com um terminal, como a queda de conexão de rede, por exemplo. Para dar continuidade ao projeto existente estão programadas as atividades: Desenvolver, implementar e testar aplicações em rede TCP/IP para serviços unicast, anycast e multicast, envolvendo protocolos de rede TCP, UDP e ICMP; Desenvolver, implementar e testar sistemas de I/O em barramentos seriais e USB multiplataformas; Desenvolver, implementar e testar protocolos de aplicações necessárias para a interconexão dos módulos; Prever a implementação do módulo gráfico para a monitoração e apresentação dos comportamentos da dinâmica do sistema.

MULTIPLATFORM VIRTUAL SATELLITES ELEMENTS COUPLING SOFTWARE MODEL

ABSTRACT

This work, initiated in August 2009, aims to continue the project in progress since July 2002 for the development of a software coupling modules that allow the use of different dynamic models (sensors, actuators and plant dynamics), as the test of methods for transferring data over high-speed networks and serial links (low speed). Initially the work started in 2002 tried to test a concept brought into operation, which found the technical feasibility of the project, being implemented by the applications then on TCP / IP services for unicast, anycast and multicast, involving the TCP and UDP transport protocols, and the application in I / O (Input / Output) serial bus systems. The current work is to improve the application's modules for network services using the TCP, UDP and ICMP as well as improving the module for serial communication and the development and implementation of modules for communication with parallel ports and / or USB devices. Through extensive testing and simulation we can also provide and implement the treatment of possible errors caused by oscillation / overload the operating system, or loss of communication with a terminal, or with a network connection, for example. To continue the existing project activities are planned: to develop, implement and test applications on TCP / IP services for unicast, multicast and anycast, involving the TCP, UDP and ICMP network protocols; develop, implement and test I/O systems in serial buses and USB ports in any OS (operational system); develop, implement and test protocols for applications for the modules intercommunication; provide the implementation of the graphic module for monitoring and reporting of the system's dynamic behavior.

SUMÁRIO

LISTA DE FIGURAS

Pag.

LISTA DE TABELAS

Pag.

LISTA DE ABREVIATURAS E SIGLAS

CB Computador de Bordo

DMC Divisão de Mecânica Espacial e Controle

GPS *Global Positioning System*

ICMP *Internet Control Message Protocol*

INPE Instituto Nacional de Pesquisas Espaciais

IP *Internet Protocol*

PC *Personal Computer*

TCP *Transfer Control Protocol*

UDP *User Datagram Protocol*

USB *Universal Serial Bus*

1 INTRODUÇÃO AO PROJETO LABSIM/SCAO

Este trabalho tem como objetivo dar continuidade ao projeto Laboratório de Simulação e Integração de Sistemas de Controle de Atitude e Órbita (LABSIM/SCAO) em andamento desde julho de 2002 para o desenvolvimento de módulos de acoplamento de software que permitam o uso de modelos dinâmicos diversos, que envolve o uso de sensores, atuadores e planta dinâmica.

O projeto LABSIM/SCAO teve seu inicio dado pela Divisão de Mecânica Espacial e Controle, do Instituto Nacional de Pesquisas Espaciais, (DMC/INPE), com a finalidade de disponibilizar recursos computacionais para a simulação e integração de subsistemas de controle e elementos correlacionados que permitam testar, verificar e simular condições ambientais encontrados nos satélites artificiais, sem a necessidade de montar em um satélite real. Com isso pretende-se que o LABSIM/SCAO permita *hardware in the loop* e *hardware simulado* para os testes de leis de controle de um protótipo virtual (ou real) de um Computador de Bordo (CB), incluindo sensores reais e/ou virtuais (componentes simulados através de software desenvolvido e projetado pelo INPE em ambiente de software livre, utilizando, para isso, o sistema operacional GNU/Linux e a linguagem de programação C).

1.1 CENÁRIO

Faz parte do cenário um conjunto de sensores e atuadores, representados por blocos distintos, interligados ao Computador de Bordo (CB) através de portas analógicas ou digitais, informando a dinâmica do sistema.

O cenário de processamento requisitado pelo LABSIM/SCAO exige a localização exata de uma determinada tarefa, transparência na transferência dos dados, flexibilidade na inserção ou remoção de *hardware in the loop*, e um sistema complexo de aquisição e/ou envio de dados em alta velocidade de aquisição e computação, devendo satisfazer os períodos amostrais inerentes de sensores e atuadores utilizados nas missões, formando assim um cenário extremamente particular. Para atingir estes objetivos, faz-se necessário explorar recursos computacionais e de rede, utilizando-se da distribuição de tarefas.

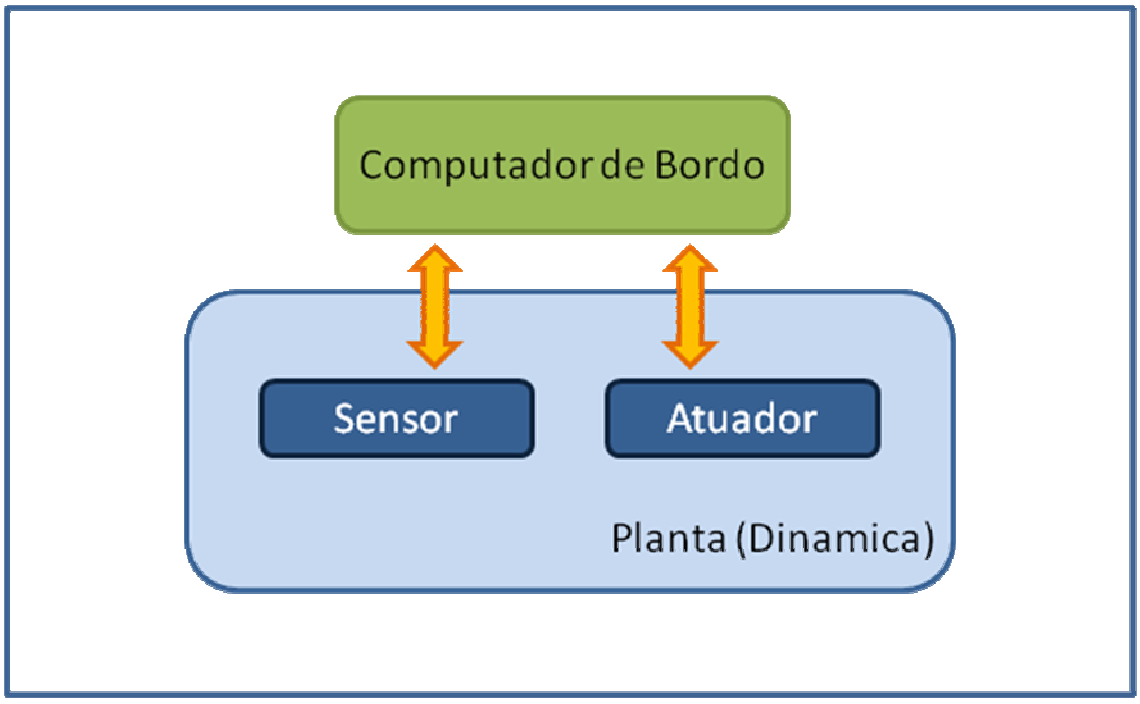

Fig.1 Identificação dos elementos ativos considerados pelo computador de bordo para ler e atuar na dinâmica do satélite.

A forma de comunicação entre o Computador de Bordo e os dispositivos ocorre através de forma direta (hardware) ou através dos barramentos (portas seriais, paralelas e USBs).

1.2 REDE

O protocolo TCP/IP possui um endereçamento de 32 bits, originalmente divididos em poucas estruturas de tamanho fixo denominado “classes de endereço”. Examinando os primeiros bits de um endereço já se torna possível identificar a qual classe pertence o IP em questão, e também a estrutura do endereço:

- Classe A: Primeiro bit é 0 (zero)
- Classe B: Primeiros dois bits são 10 (um, zero)
- Classe C: Primeiros três bits são 110 (um, um, zero)
- Classe D: Primeiros quatro bits são 1110 (um, um, um, zero)
- Classe E: Primeiros cinco bits são 11110 (um, um, um, um, zero)

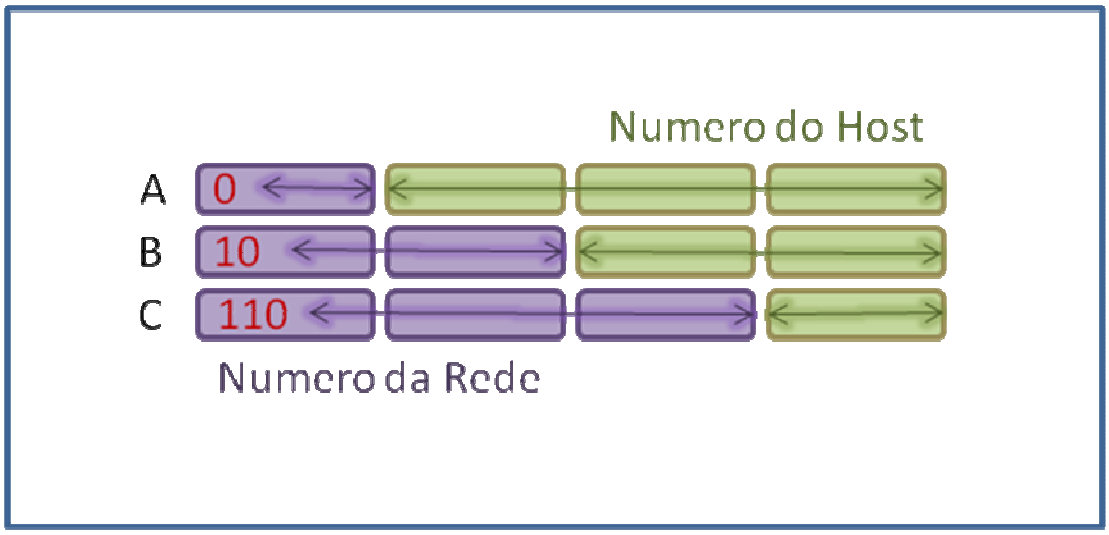

Fig. 2 – Os números de rede e de host para as classes A, B e C

A tabela a seguir contém o intervalo das classes de endereços IPs:

Classe	**Gama de Endereços**	**Nº de endereços por rede**
A	1.0.0.0 a 127.255.255.255	16.777.216
B	128.0.0.0 a 191.255.255.255	65.536
C	192.0.0.0 a 223.255.255.255	256
D	224.0.0.0 a 239.255.255.255	*Multicast*
E	240.0.0.0 a 247.255.255.255	Reservado para testes pela IETF

Tab. 1 Intervalo das classes de endereços IPs

As três principais e mais utilizadas classes são a classe A, classe B e classe C. Para este projeto utilizaremos de inicio as classes C, para uso geral, e a classe D, destinado ao *multicast*. Define-se *multicast* como um endereço tal que um conjunto de máquinas se identifica com aquele endereço, constituindo um grupo selecionado e com um mesmo endereço. A rede pertencente à classe D – faixa de endereçamento de 224.0.0.0 a 239.255.255.255 e máscara 255.255.255.255 – presta-se para esta finalidade e convive com outros endereços públicos, privados e outros *muticast*. Assim, um único datagrama IP enviado com endereço de destino coincidente com algum endereço desta faixa tipo *multicast* será recebido pelo conjunto de máquinas, membros do grupo, configuradas para tal e de forma simultânea.

A circulação dos dados se da em uma rede local, não sendo necessária a confirmação de entrega dentro do ambiente de trabalho deste projeto. Isto, aliado ao uso de endereçamento *multicast* conduz ao uso do protocolo de transporte UDP (User Datagram Protocol). Sendo assim, prováveis certificações de entrega e o controle de envio e/ou recebimento devem ser implementadas pela aplicação.

2 INTRODUÇÃO AOS ELEMENTOS DO SISTEMA

Envolvidos neste sistema estarão a Planta, um conjunto de Sensores e Atuadores, e o Computador de Bordo.

2.1 PLANTA

A Planta é responsável pela propagação de atitude e órbita. Tais dados são enviados através de uma rede de alta velocidade para o grupo de sensores e/ou atuadores. A Planta recebe segundo um endereçamento IP *multicast* os dados dos atuadores. A estrutura do dado enviado pela planta segue as especificações de rede TCP/IP para números inteiros e o padrão IEE para ponto flutuante. Desta forma, um único datagrama UDP/IP pode ser capturado pelos dispositivos acoplados a rede e decodificados.

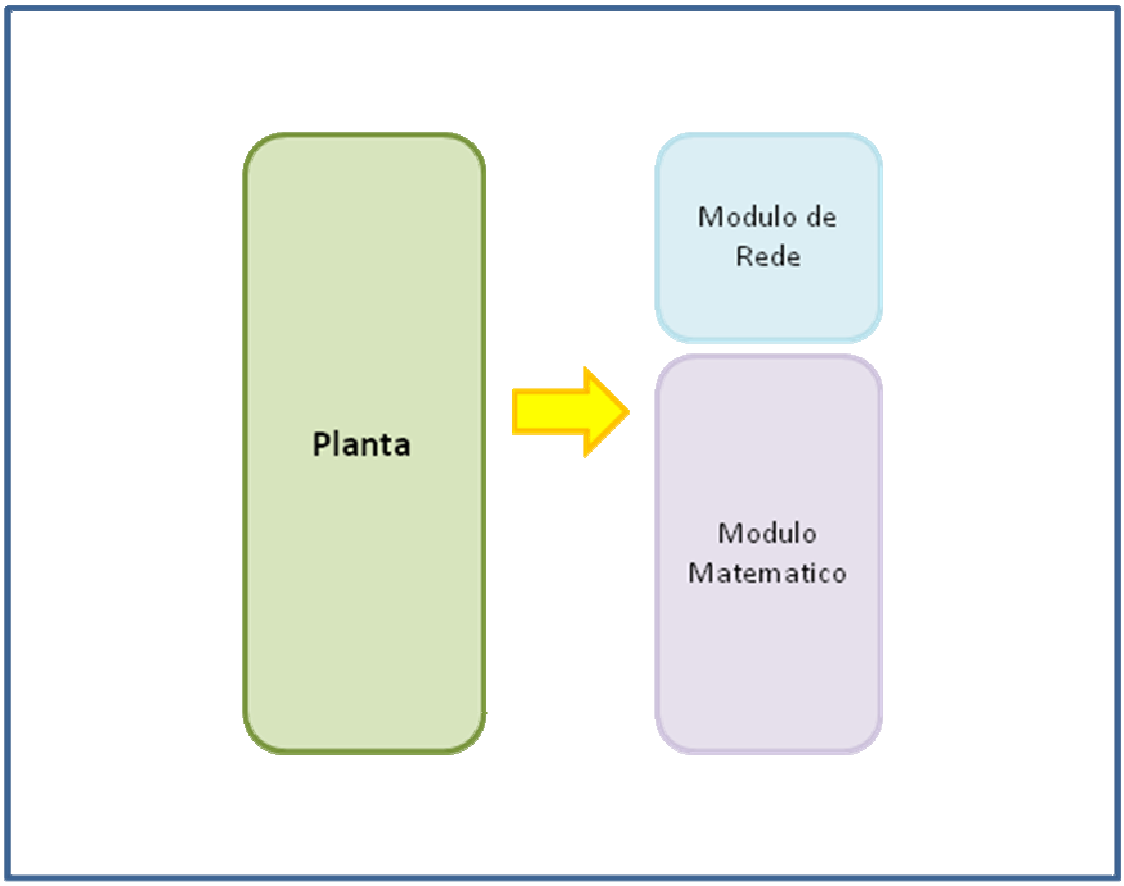

Fig. 3 – Conversão da planta para blocos simulados

A planta é dividida em dois módulos. O primeiro, denominado Módulo de Rede, recebe a função de ler e escrever os dados recebidos e enviados para a rede. Esses dados seguem uma estrutura pré-definida, contendo informações de atitude (vetor de atitude, rotações de giros, volantes de inércia, etc.) e órbita (posição e velocidade). O segundo, o Módulo Matemático, recebe a responsabilidade de cuidar das operações lógico-aritméticas.

2.2 SENSORES

Para este projeto estão previstos a implementação de alguns tipos de sensores:

- Sensor Solar: Analógico, Digital (um e dois eixos) e Presença;

- Sensor Magnético: Magnetômetros de um, dois ou três eixos com escala simples e dupla (baixa e alta precisão);
- Estrelas: Modelo estrelar;
- Girômetros: até três eixos;
- Acelerômetros: até três eixos;
- GPS: até quatro receptores com até quatro antenas.

2.3 ATUADORES

Para este projeto estão previstos a implementação de alguns tipos de atuadores:

- Rodas de reação: até quatro rodas;
- Volantes de Inércia: até quatro volantes;
- Bobinas Magnéticas: até seis bobinas;
- Jatos de Gás: até doze propulsores;

Os atuadores são elementos que interagem fortemente com a planta, integrando-se a ela e operando apenas como conversores de um sistema de acionamento – liga, desliga, acelera, desacelera.

2.4 COMPUTADOR DE BORDO

O Computador de Bordo (CB) é um PC com duas interfaces seriais, sendo a primeira delas acoplada a interface serial do Sensor, e a segunda, a interface do Computador que simula o Atuador. Os computadores Sensor e Atuador já estarão acoplados a rede de alta velocidade e passarão a informar e comandar, respectivamente, o CB e a Planta com as informações, estabelecendo um ciclo infinito.

2.5 SISTEMA

O procedimento de interface do CB com os dispositivos de aquisição e controle pode ser simulado utilizando interfaces de software para portas seriais, paralelas, USB, etc., escritos de forma modular e utilizando métodos de compartilhamento de memória para a comunicação entre processos.

A dinâmica real é substituída por uma simulada por software independente, a Planta, que é interligada aos dispositivos (sensores e atuadores) através de uma rede GigaEthernet (alta velocidade). A rede de alta velocidade no prédio do LABSIM/SCAO é de 1 Gbps, o que representa a taxa de transferência de 125MBytes/s. Neste caso, a transferência de um datagrama UDP/IP vazio (20 bytes e sem qualquer dado) consome um tempo de transferência mínimo de 0,16 micro-segundos. Considerando um MTU – *Maximum Transfer Unit* – de 1500, um datagrama IP com 1480 bytes, tem-se um tempo de transferência da ordem de 12 micro-segundos. Neste cenário, a freqüência de amostragem útil limitada pela linha de transmissão seria de 83 KHz até ~6.3 MHz, dependendo do tamanho do quadro (*frame*) transferido completo ou parcial (1 byte), respectivamente. Estes valores satisfazem a operação da maioria dos sensores existentes e normalmente utilizados em satélites artificiais.

Baseado nestas condições, o diagrama funcional do LABSIM/SCAO segue o seguinte modelo:

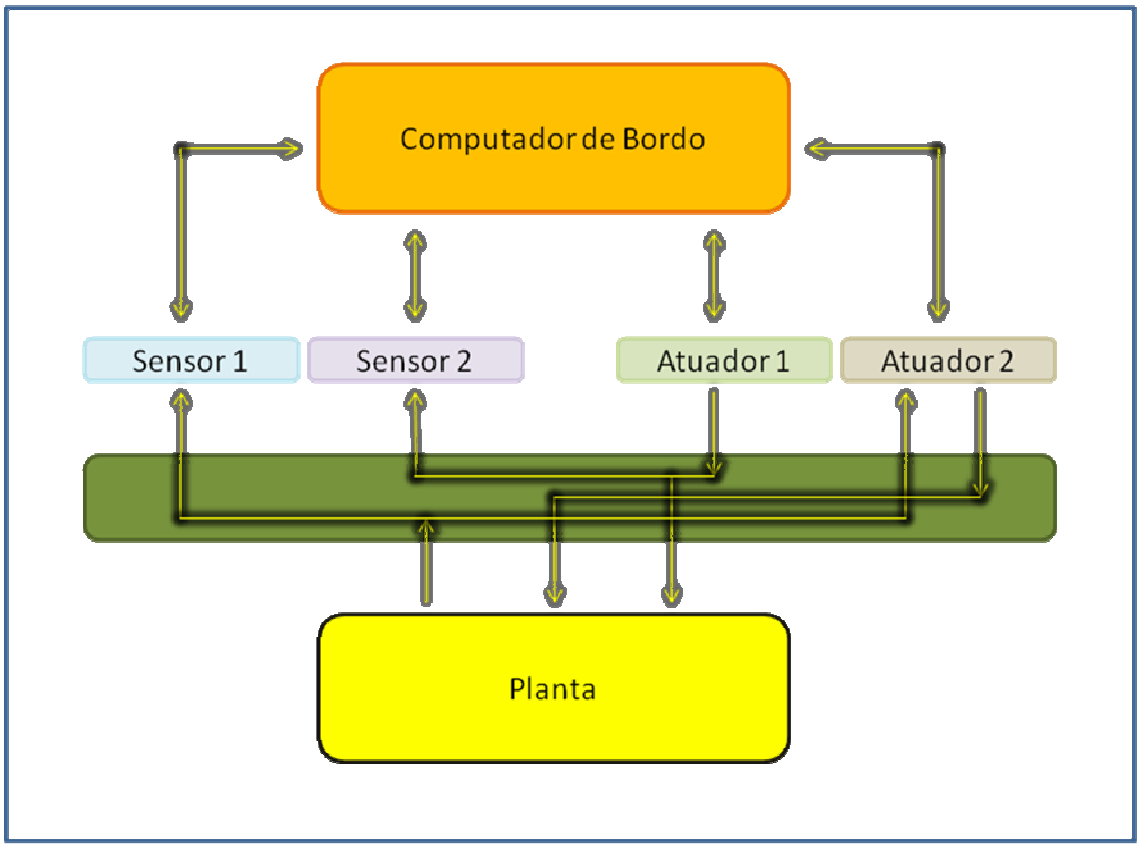

Fig. 4 – Diagrama de blocos do LABSIM/SCAO

Para esta parte do projeto será focado o desenvolvimento de módulos de acoplamento de software responsáveis pela intercomunicação dos elementos apresentados neste capitulo.

3 DESENVOLVIMENTO DOS MÓDULOS

Para esta etapa do projeto foi programado as seguintes atividades:

a) Desenvolver, implementar e testar aplicações em rede TCP/IP para serviços unicast, anycast e multicast, envolvendo protocolos de rede TCP, UDP e ICMP;
b) Desenvolver, implementar e testar sistemas de I/O em barramentos seriais e USB multiplataformas;
c) Desenvolver, implementar e testar protocolos de aplicações necessárias para a interconexão dos módulos;
d) Prever a implementação do módulo gráfico para a monitoração e apresentação dos comportamentos da dinâmica do sistema.
e) Documentação do software e produtos.

Tais atividades respeitam o seguinte cronograma:

Item	Ago – Out/08	Nov – Jan/09	Fev – Abr/09	Mar – Jun/09	Julho/09
a	X				
b		X			
c			X		
d				X	
e	X	X	X	X	X

Tab. 2 – Cronograma de atividades

No desenvolvimento dos módulos foi utilizado a linguagem de programação C, e toda sua implementação e testes foram feitos em um ambiente de software livre, o GNU/Linux, porém foi considerado portabilidade para execução em multiplataforma.

3.1 MÓDULOS DE REDE

Para o desenvolvimento do módulo de rede uma pesquisa sobre os *Berkeley Sockets* (Soquetes de Berkeley).

Berkeley Sockets, também conhecido como BSD Sockets, se origina do sistema operacional 4.2BSD Unix, apresentado em 1983, sendo desenvolvido na Universidade da Califórnia, em Berkeley, e consiste em uma biblioteca para desenvolvimento de

aplicações utilizando a linguagem de programação C para a comunicação inter-processos, mais comumente usados em aplicações de rede (pode-se aplicar o conceito de sockets para a comunicação de processos sendo executados em uma mesma máquina, porém o custo de tempo gasto na transmissão de dados é alto quando comparado a outros métodos mais eficazes, que será discutido no capitulo 3.3.

A interface BSD Socket permite a comunicação entre hosts utilizando o conceito de um socket de internet. Tais interfaces são acessíveis em três níveis diferentes, atuando na camada de Rede ou na camada de Transporte, de acordo com o modelo OSI, mais detalhadas a seguir.

Para a construção da base do módulo de rede foram explorados as funções e argumentos:

- socket()

  Cria um *endpoint* para a comunicação e retorna um descritor. socket() recebe três argumentos:

  - Domínio:
    Especifica a família do protocolo a ser utilizado. Neste projeto somente utilizamos a família PF_INET, onde se incluí o Internet Protocol version 4 (IPv4).

  - Tipo:
    Um dos tipos listados abaixo:

    - SOCK_RAW:
      Protocolo utilizado para acesso direto à camada de Rede.

    - SOCK_STREAM:
      Protocolo com garantia de entrega integra do dado (solicita confirmação de entrega do pacote). Atua na camada de Transporte.

    - SOCK_DGRAM:
      Protocolo sem garantia de entrega integra do dado (não solicita confirmação de entrega do pacote). Atua na camada de Transporte.

  - Protocolo:
    Especifica qual protocolo será utilizado no transporte do dado. Para os tipos listados acima, respectivamente, utilizamos o parâmetro IPPROTO_ICMP, IPPROTO_TCP e IPPROTO_UDP.

- setsockopt()

Esta função ajusta a(s) opção(ões) passada(s) pelo argumento *option_name*, dentro do nível do protocolo especificado no argumento *level*. No módulo de rede foi utilizado a opção SO_REUSEADDR, nível SOL_SOCKET, que torna o endereço IP do socket criado reutilizável, a fim de evitar erros de reutilização do endereço IP em tempo de execução.

Para que o código pudesse ficar enxuto, ou seja, ter o mínimo de linhas possíveis, diminuindo seu tamanho e o tempo de compilação, a criação dos sockets foi dividida em dois módulos, onde um cria um socket "genérico", e o tipo de protocolo será utilizado é definido por passagem de parâmetros, como mostra a figura a seguir contendo um trecho do código:

```
28  /**---------------------------------------------------------------------------
29   * PROCEDURE : socket_init
30   *
31   * PURPOSE   : Cria um socket para comunicacao e retorna um
32   *             descritor.
33   *
34   * INPUT     : domain   - especificacao do dominio de comunicacao
35   *             type     - especificacao do tipo de protocolo
36   *             protocol - especificacao do protocolo
37   *
38   * OUTPUT    : Retorna o numero do socket criado.
39   *
40   * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
41   *---------------------------------------------------------------------------**/
42  int socket_init(int domain, int type, int protocol)
43  {
44      int sockfd;
45      unsigned int on = 1;
46
47      /* criando socket */
48      if ( (sockfd = socket(domain, type, protocol)) < 0) {
49          perror("socket_init: socket()");
50          return (-1);
51      }
52
53      /* ajustando opcao de socket */
54      if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
55          perror("socket_init: setsockopt()");
56  /*      return (-1);
57  */  }
58
59      return sockfd;
60  }
```

Fig. 5 – Trecho do código responsável pela criação do *socket*.

A partir dele foram criados os módulos que definem qual o tipo de socket que será criado (se o socket seria criado para TCP, por exemplo). As figuras seguintes mostram os trechos do código que informa ao módulo socket_init() quais parâmetros utilizar na criação do socket:

```
89
90  /**---------------------------------------------------------------------------
91   * PROCEDURE : socket_udp_init
92   * *
93   * PURPOSE   : Cria um socket em UDP para comunicacao e retorna um
94   *             descritor.
95   *
96   * INPUT     : Nenhum.
97   *
98   * OUTPUT    : Retorna o numero do socket criado.
99   * *
100  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
101  *---------------------------------------------------------------------------**/
102 int socket_udp_init(void)
103 {
104     int sockfd;
105
106     /* criando socket em UDP */
107     sockfd = socket_init(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
108     return sockfd;
109 }
110
```

Fig. 6 – Trecho do código responsável pela criação de socket para UDP

O trecho acima mostra a passagem dos parâmetros SOCK_DGRAM – tipo de protocolo que não exige confirmação de entrega do pacote – e também IPPROTO_UDP – define o UDP como o protocolo a ser utilizado pela aplicação.

```
111 /**-------------------------------------------------------------------------
112  * PROCEDURE : socket_tcp_init
113  *
114  * PURPOSE   : Cria um socket em TCP para comunicacao e retorna um
115  *             descritor.
116  *
117  * INPUT     : Nenhum.
118  *
119  * OUTPUT    : Retorna o numero do socket criado.
120  *
121  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
122  *-------------------------------------------------------------------------**/
123 int socket_tcp_init(void)
124 {
125     int sockfd;
126
127     /* criando socket em TCP */
128     sockfd = socket_init(AF_INET, SOCK_STREAM, IPPROTO_TCP);
129     return sockfd;
130 }
```

Fig. 7 – Trecho do código responsável pela criação de socket para TCP

Na figura acima os parâmetros utilizados foram o SOCK_STREAM – tipo de protocolo que exige a confirmação de entrega do pacote, tornando a comunicação confiável, porem com maior atraso na transferência do dado – e também IPPROTO_TCP, confirmando a utilização do TCP como protocolo.

Desenvolveu-se também um módulo destinado à utilização do protocolo ICMP, que serve basicamente para a detecção de erros em tempo de execução. Porém retirou-se este módulo, e a implementação de tal recurso foi aplicada nos módulos de TCP e UDP.

Para identificar uma máquina na rede, alem de ter o meio por onde receber (ou enviar) o dado – função exercida pelos sockets –, uma máquina deve ter um endereço IP associado ao socket para que este possa ser reconhecido dentro da rede. Para isso, o módulo apresentado abaixo serve para carregar a estrutura *sockaddr_in* com as informações de endereço IP e a porta de destino do host:

```
/**--------------------------------------------------------------------------
 * PROCEDURE : sockaddr_init
 *
 * PURPOSE   : Carregar estrutura sockaddr_in.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip
 *             port     - porta
 *
 * OUTPUT    : Estrutura sockaddr_in carregada. Os valores nao informados serao
 *             preenchidos com zeros.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *--------------------------------------------------------------------------**/
int sockaddr_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
    /* carregando estrutura sockaddr com os valores ipaddr e port */
    memset(sockaddr, 0, sizeof(*sockaddr));
    sockaddr->sin_family = AF_INET;
    sockaddr->sin_port = htons(port);
    if (inet_pton(AF_INET, ipaddr, &sockaddr->sin_addr.s_addr) <= 0) {
        perror("sockaddr_init: inet_pton()");
        return (-1);
    }

    return (1);
}
```

Fig. 8 – Trecho do código responsável por carregar as informações do host

Com estes módulos conseguimos então criar os módulos da aplicação servidora e cliente. Para cada necessidade existe um módulo que corresponde a situação desejada, ou seja, se há a necessidade de uma aplicação cliente-servidora em que não é permitido erros de transmissão de dados, caso em que dados corrompidos atrapalham em muito a analise do cenário, podemos utilizar os módulos que utilizam sockets para TCP, como mostrado a seguir:

```
/**---------------------------------------------------------------------------
 * PROCEDURE : tcp_client_init
 *
 * PURPOSE   : Iniciar um servico de rede. Cria um socket para TCP e carrega a
 *             estrutura cliente com o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - numero do socket para TCP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int tcp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
    int sockfd;

    /* criando socket em TCP
     * obs: erro ocorrido ja apresentado em socket_init()
     */
    sockfd = socket_tcp_init();

    /* carregando estrutura */
    if (sockaddr_init(sockaddr, ipaddr, port) < 0)
        return (-1);

    return sockfd;
}
```

Fig. 9 – Módulo de criação de aplicação cliente com socket para TCP

E para a aplicação servidora temos o seguinte módulo:

```
/**---------------------------------------------------------------------------
 * PROCEDURE : tcp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para TCP e carrega
 *             a estrutura servidora com endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - Estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para TCP criado.
 *
 * ERRORS    : Em caso de falha na criacao do socket ou no preenchimento
 *             da estrutura contendo o endereco o sistema retorna um
 *             numero negativo (-1).
 *---------------------------------------------------------------------------**/
int tcp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
    int sockfd;

    /* criando socket em TCP utilizando tcp_client_init()
     * obs: erro ocorrido ja apresentado em socket_init()
     */
    if ( (sockfd = tcp_client_init(sockaddr, ipaddr, port)) < 0)
        return (-1);

    /* fixando estrutura sockaddr ao socket */
    if (bind(sockfd,
             (const struct sockaddr *) sockaddr,
             sizeof(*sockaddr)
             ) < 0) {
        perror("tcp_server_init: bind()");
        return (-1);
    }

    /* estabelecendo lista de espera ajustado em 5 */
    if (listen(sockfd, 5) < 0) {
        perror("tcp_server_init: listen()");
        return (-1);
    }

    return sockfd;
}
```

Fig. 10 – Módulo de criação da aplicação servidora com socket para TCP

Diferente de aplicações cliente-servidor com sockets para UDP, a aplicação envolvendo o protocolo TCP necessita estabelecer uma conexão permanente para que possa ocorrer a transferência de dados. O módulo que permite esta conexão é mostrado a seguir:

```
231 /**-------------------------------------------------------------------------
232  * PROCEDURE : accept_init
233  *
234  * PURPOSE   : Criar um novo socket para a conexao estabelecida
235  *
236  * INPUT     : sockfd     - socket do servidor a aceitar nova conexao
237  *             peeraddr   - ponteiro para estrutura sockaddr_in
238  *
239  * OUTPUT    : acceptsock - retorna o numero do socket criado.
240  *             peeraddr   - estrutura carregada com informacoes do cliente.
241  *
242  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
243  *-------------------------------------------------------------------------**/
244 int accept_init(int sockfd, struct sockaddr_in *peeraddr)
245 {
246     int acceptsock;
247     socklen_t addrlen;
248
249     memset(peeraddr, 0, sizeof(*peeraddr));
250     addrlen = sizeof(*peeraddr);
251
252     /* criando novo socket para a conexao estabelecida */
253     if ( (acceptsock = accept(sockfd,
254                               (struct sockaddr *) peeraddr,
255                               &addrlen)
256                               ) < 0) {
257         perror("accept_init: accept()");
258         return (-1);
259     }
260
261     return acceptsock;
262 }
```

Fig. 11 – Módulo para estabelecer uma conexão entre cliente e servidor

Com o módulo acima, através da função accept(), a aplicação servidora é capaz de aceitar uma conexão entrante e iniciar a transferencia de dados.

Para os módulos da aplicação cliente-servidora com socket para UDP, este módulo não se faz necessário.

```
264 /**-------------------------------------------------------------------------
265  * PROCEDURE : udp_client_init
266  *
267  * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega a
268  *             estrutura contendo o endereco ip e porta de destino.
269  *
270  * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
271  *             ipaddr   - endereco ip de destino
272  *             port     - porta de destino
273  *
274  * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
275  *                        preenchidos com zero.
276  *             sockfd   - Socket para UDP criado.
277  *
278  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
279  *-------------------------------------------------------------------------**/
280 int udp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
281 {
282     int sockfd;
283 
284     /* criando socket em UDP
285      * obs: erro ocorrido ja apresentado em socket_init()
286      */
287     sockfd = socket_udp_init();
288 
289     /* carregando estrutura */
290     if (sockaddr_init(sockaddr, ipaddr, port) < 0)
291         return (-1);
292 
293     return sockfd;
294 }
```

Fig. 12 – Módulo de criação da aplicação cliente com socket para UDP.

```
296 /**-------------------------------------------------------------------------
297  * PROCEDURE : udp_server_init
298  *
299  * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega
300  *             a estrutura servidora com endereco IP e porta de destino.
301  *
302  * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
303  *             ipaddr   - endereco ip de destino
304  *             port     - porta de destino
305  *
306  * OUTPUT    : sockaddr  - Estrutura carregada. Os campos vazios sao
307  *                         preenchidos com zero.
308  *             sockfd    - Socket para UDP criado.
309  *
310  * ERRORS    : Em falha na criacao ou no preenchimento da estrutura que
311  *             contem o endereco IP de servico e a porta de destino o
312  *             sistema retorna um valor negativo (-1)
313  *-------------------------------------------------------------------------**/
314 int udp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
315 {
316     int sockfd;
317
318     /* criando socket em UDP
319      * obs: erro ocorrido ja apresentado em socket_init()
320      */
321     sockfd = udp_client_init(sockaddr, ipaddr, port);
322
323     /* fixando estrutura sockaddr_in ao socket */
324     if (bind(sockfd,
325              (const struct sockaddr *) sockaddr,
326              sizeof(*sockaddr)
327              ) < 0) {
328         perror("udp_server_init: bind()");
329         return (-1);
330     }
331
332     return sockfd;
333 }
```

Fig. 13 – Módulo de criação da aplicação servidora com socket para UDP

Percebe-se que o que diferencia o módulo servidor do cliente tanto em TCP como em UDP é a chamada da função bind(), que relaciona o socket criado ao endereço IP fornecido (além do módulo accept_init() utilizado somente pela aplicação servidora com socket para TCP).

Para a construção do módulo da aplicação servidora *multicast*, usa-se os módulos da aplicação servidora com socket para UDP (*udp_server_init()*) e adiciona-o ao grupo de *multicast* desejado. Desta forma a aplicação servidora *multicast* é capaz de ser carregada com um endereço IP próprio do servidor e também com um endereço IP *multicast*, fazendo parte do grupo e recebendo os dados conforme as capacidades de uma rede *multicast*, descrita no capitulo 1.2.

Os módulos desta aplicação são mostrados a seguir:

```
334
335 /**---------------------------------------------------------------------------
336  * PROCEDURE : udp_server_mult_init
337  *
338  * PURPOSE   : Iniciar servico de rede. Criar um socket para UDP e carregar a
339  *             estrutura contendo o endereco IP de servico em modo multicast
340  *             e a porta de destino.
341  *
342  * INPUT     : sockaddr    - ponteiro para estrutura sockaddr_in
343  *             ipaddr      - endereco ip de destino
344  *             port        - porta de destino
345  *             grpaddr     - ponteiro para estrutura im_req
346  *             mult_ipaddr - endereco ip de destino
347  *
348  * OUTPUT    : sockaddr    - Estrutura carregada. Os campos vazios sao
349  *                           preenchidos com zero.
350  *             svaddr      - Estrutura carregada. Os campos vazios sao
351  *                           preenchidos com zero.
352  *             grpaddr     - Estrutura carregada. Os campos vazios sao
353  *                           preenchidos com zero.
354  *             sockfd      - Socket para UDP multicast criado.
355  *
356  * ERRORS    : Em falha o sistema retorna um numero negativo (-1)
357  *---------------------------------------------------------------------------**/
358 int udp_server_mult_init(struct sockaddr_in *sockaddr,
359                          char ipaddr[15],
360                          int port,
361                          struct ip_mreq *grpaddr,
362                          char mult_ipaddr[16]
363                          )
364 {
365     int sockfd;
366
367     /* criando socket em UDP */
368     if ( (sockfd = udp_server_init(sockaddr, ipaddr, port)) < 0)
369         return (-1);
370
371     /* fixando estrutura sockaddr_in ao socket */
372     if (bind(sockfd,
373              (const struct sockaddr *) sockaddr,
374              sizeof(*sockaddr)
375              ) < 0) {
376         perror("udp_server_mult_init: bind()");
377         return (-1);
378     }
```

Fig. 14 – Módulo da aplicação servidora *multicast*

```
380     /* preenchendo estrutura im_req com endereco de multicast */
381     if (inet_pton(AF_INET, mult_ipaddr, &grpaddr->imr_multiaddr.s_addr) <= 0) {
382         perror("udp_server_mult_init: inet_pton()");
383         return (-1);
384     }
385
386     /* preenchendo estrutura im_req com endereco de interface */
387     if (inet_pton(AF_INET, ipaddr, &grpaddr->imr_interface.s_addr) <= 0) {
388         perror("udp_server_mult_init: inet_pton(IPADDR) ");
389         return (-1);
390     }
391
392     /* adicionando socket ao grupo informado na estrutura im_req */
393     if (setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, grpaddr,
394                    sizeof(*grpaddr)) < 0) {
395         perror("udp_server_mult_init: setsockopt()");
396         return (-1);
397     }
398
399     return sockfd;
400 }
```

Fig. 15 – Continuação do módulo da aplicação servidora *multicast*

Os módulos mostrados acima servem para iniciar um serviço de rede. Para o encerramento do serviço utilizamos os módulos de encerramento mostrados a seguir:

```
401
402 /**-------------------------------------------------------------------------
403  * PROCEDURE : udp_server_mult_close
404  *
405  * PURPOSE   : Encerrar o servico de rede
406  *
407  * INPUT     : sockfd  - socket a ser encerrado
408  *             grpaddr - apontador para estrutura ip_mreq
409  *
410  * OUTPUT    : Retira sockfd da lista de membros de multicast e fecha o socket.
411  *
412  * ERRORS    : Em falha o sistema retorna um numero negativo (-1).
413  *-------------------------------------------------------------------------**/
414 int udp_server_mult_close(int sockfd, struct ip_mreq *grpaddr)
415 {
416     int status;
417
418     /* retirando membro do grupo de multicast */
419     if ( (status = setsockopt(sockfd,
420                               IPPROTO_IP,
421                               IP_DROP_MEMBERSHIP,
422                               grpaddr,
423                               sizeof(*grpaddr)
424                               )) < 0) {
425         perror("udp_server_mult_close: setsockopt()");
426         return status;
427     }
428
429     /* encerrando socket */
430     if (fd_close(sockfd) < 0)
431         return (-1);
432
433     return status;
434 }
```

Fig. 16 – Encerramento do serviço de rede *multicast*

No módulo acima o servidor *multicast* é retirado do grupo de *multicast* previamente cadastrado através do argumento IP_DROP_MEMBERSHIP, e para o encerramento do socket é chamado o módulo *fd_close()*. Este módulo, mostrado a seguir, encerra qualquer descritor passado pelo argumento *fd*, não se limitando ao encerramento de descritores de *sockets*, mas se estende a qualquer descritor que tenha sido criado. Tal observação existe para cobrir também os encerramentos necessários para os módulos de criação de *pipes* e descritores para comunicação com portas seriais e paralelas, alem do módulo de interface com dispositivos USB.

```
/**-------------------------------------------------------------------------
 * PROCEDURE : fd_close *
 * PURPOSE   : Encerra servico de rede, ou de comunicacao por porta serial
 *             ou paralela, ou comunicacao por canal pipe.
 *
 * INPUT     : fd - descritor a ser encerrado
 *
 * OUTPUT    : Retorna o status de encerramento
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int fd_close(int fd)
{
    int status;

    /* encerrando descritor */
    if ( (status = close(fd)) < 0)
        perror("fd_close: close()");

    return status;
}
```

Fig. 17 – Módulo de encerramento de descritores.

3.2 MÓDULOS DE INTERFACE COM PORTAS SERIAIS E PARALELAS

No desenvolvimento dos módulos de interface com portas seriais e portas paralelas uma pesquisa sobre opções de *flags* de abertura de portas foi realizada em detalhes para habilitar a comunicação. Para que se possa estabelecer uma comunicação com as portas, precisa-se da abertura de cada porta como um arquivo descritor (*file descriptor*) e, então, ajusta-se as configurações de *flags*.

```
629 /**-------------------------------------------------------------------------
630  * PROCEDURE : serial_init
631  *
632  * PURPOSE   : Iniciar servico de serial
633  *
634  * INPUT     : serial - caminho da porta serial a ser iniciada
635  *             ispeed - velocidade de INPUT
636  *             ospeed - velocidade de OUTPUT
637  *
638  * OUTPUT    : Retorna o numero do serial criado
639  *
640  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
641  *-------------------------------------------------------------------------**/
642 int serial_init(char *serial, int ispeed, int ospeed)
643 {
644     int serialfd;
645     struct termios options;
646
647     /* abre o caminho apontado por *serial em leitura e escrita, sem obter o
648      * controle total do terminal (controle permanece com o SO) */
649     if ( (serialfd = open(serial, O_RDWR | O_NOCTTY | O_NONBLOCK)) < 0) {
650         perror("serial_init: open()");
651         return (-1);
652     }
653
654     /* ajuste de flags */
655     if (fcntl(serialfd, F_SETFL, FNDELAY) < 0) {
656         perror("serial_init: fcntl()");
657 /*      return (-1);
658 */  }
659
660     /* verifica flags existentes */
661     memset(&options, 0, sizeof(options));
662     tcgetattr(serialfd, &options);
663
664     /* ajustando opcoes de flags para modo raw */
665     /* opcoes de output */
666     options.c_oflag &= ~OPOST;    /* habilita modo raw */
667     /* opcoes locais */
668     options.c_lflag &= ~ECHO;     /* desabilita "echo" de caracteres */
669     options.c_lflag &= ~ECHOE;
670     options.c_lflag &= ~ECHONL;
671     options.c_lflag &= ~ICANON;   /* habilita modo raw */
672     options.c_lflag &= ~ISIG;     /* desabilita sinais */
673     options.c_lflag &= ~IEXTEN;   /* desabilita funcoes extras */
```

Fig. 18 – Módulo de interface com porta serial

```
674     /* opcoes de controle */
675     options.c_cflag |= CLOCAL;    /* nao altera o owner */
676     options.c_cflag |= CREAD;     /* habilita leitura */
677     options.c_cflag |= CS8;       /* 8 bits */
678     options.c_cflag &= ~CSIZE;    /* desabilita bit de mascara */
679     options.c_cflag &= ~CSTOPB;   /* 1 bit de stop */
680     options.c_cflag &= ~CRTSCTS;  /* desabilita controle de hardware */
681     options.c_cflag &= ~PARENB;   /* desabilita bit de paridade */
682     /* opcoes de input */
683     options.c_iflag &= ~IGNBRK;   /* aceita condicoes de parada  */
684     options.c_iflag &= ~BRKINT;   /* nao envia sinais ao condicoes de parada */
685     options.c_iflag &= ~PARMRK;   /* ignora erros de paridade */
686     options.c_iflag &= ~ISTRIP;
687     options.c_iflag &= ~INLCR;    /* mapeia de NL para CR */
688     options.c_iflag &= ~IGNCR;    /* aceita CR */
689     options.c_iflag &= ~ICRNL;    /* permanesce mapeado para CR */
690     options.c_iflag &= ~IXON;    /* desabilita controle de software */
691
692     cfsetispeed(&options, ispeed);
693     cfsetospeed(&options, ospeed);
694
695     /* ajustando flags aa porta aberta
696      * descarregando buffers de input e
697      * de output e aplica a mudanca
698      */
699     tcsetattr(serialfd, TCSAFLUSH, &options);
700
701     return serialfd;
702 }
```

Fig. 19 – Continuação do Módulo de Interface com porta serial

Este módulo, em semelhança com os módulos de rede, prepara o caminho para a comunicação com a porta serial através das opções de flags, e retorna um descritor.

```
703
704 /**-------------------------------------------------------------------------
705  * PROCEDURE : parallel_init
706  *
707  * PURPOSE   : Iniciar servico de comunicacao com porta paralela *
708  * INPUT     : parallel - caminho da porta paralela a ser iniciada
709  *             ispeed   - velocidade de INPUT
710  *             ospeed   - velocidade de OUTPUT
711  *
712  * OUTPUT    : Retorna o numero do serial criado
713  *
714  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
715  *-------------------------------------------------------------------------**/
716 int parallel_init(char *parallel, int ispeed, int ospeed)
717 {
718     int parallelfd;
719     struct termios options;
720
721     /* abre o caminho apontado por *serial em leitura e escrita, sem obter o
722      * controle total do terminal (controle permanece com o SO) */
723     if ( (parallelfd = open(parallel, O_RDWR | O_NOCTTY | O_NONBLOCK)) < 0) {
724         perror("parallel_init: open()");
725         return (-1);
726     }
727
728     /* ajuste de flags */
729     if (fcntl(parallelfd, F_SETFL, FNDELAY) < 0) {
730         perror("parallel_init: fcntl()");
731 /*      return (-1);
732 */  }
733
734     /* verifica flags existentes */
735     memset(&options, 0, sizeof(options));
736     tcgetattr(parallelfd, &options);
737
738     /* ajustando opcoes de flags para modo raw */
739     /* opcoes de output */
740     options.c_oflag &= ~OPOST;    /* habilita modo raw */
741     /* opcoes locais */
742     options.c_lflag &= ~ECHO;     /* desabilita "echo" de caracteres */
743     options.c_lflag &= ~ECHONL;   /* desabilita "echo" de newline */
744     options.c_lflag &= ~ICANON;   /* habilita modo raw */
745     options.c_lflag &= ~ISIG;     /* desabilita sinais */
746     options.c_lflag &= ~IEXTEN;   /* desabilita funcoes extras */
```

Fig. 20 – Módulo de interface com portas paralelas

```
747     /* opcoes de controle */
748     options.c_cflag |= CLOCAL;    /* nao altera o owner */
749     options.c_cflag |= CREAD;     /* habilita leitura */
750     options.c_cflag &= ~CSIZE;    /* desabilita bit de mascara */
751     options.c_cflag &= ~CRTSCTS;  /* desabilita controle de hardware */
752     options.c_cflag &= ~PARENB;   /* desabilita bit de paridade */
753     /* opcoes de input */
754     options.c_iflag &= ~IGNBRK;   /* aceita condicoes de parada  */
755     options.c_iflag &= ~BRKINT;   /* nao envia sinais ao condicoes de parada */
756     options.c_iflag &= ~PARMRK;   /* ignora erros de paridade */
757     options.c_iflag &= ~ISTRIP;
758     options.c_iflag &= ~INLCR;    /* mapeia de newline para carriage */
759     options.c_iflag &= ~IGNCR;    /* aceita carriage */
760     options.c_iflag &= ~ICRNL;    /* permanesce mapeado para carriage */
761     options.c_iflag &= ~IXON;    /* desabilita controle de software */
762
763     cfsetispeed(&options, ispeed);
764     cfsetospeed(&options, ospeed);
765
766     /* ajustando flags */
767     tcsetattr(parallelfd, TCSAFLUSH, &options);
768
769     return parallelfd;
770 }
```

Fig. 21 – Continuação do Módulo de interface com portas paralelas

Este módulo assemelha-se do anterior em sua estrutura pela forma de abertura do canal de comunicação com as portas paralelas, valendo-se de ajustes de *flags* e, também, retornando um descritor ao sistema.

3.3 MÓDULOS DE INTERFACE COM DISPOSITIVOS USB

Na criação deste módulo foi utilizado a biblioteca *libusb*, disponível em <http://libusb.sourceforge.net/>, e seu desenvolvimento e testes foram feitos exclusivamente em ambiente de software livre utilizando o sistema operacional GNU/Linux, porém respeitado a portabilidade para outros ambientes operacionais.

Para a interface com dispositivos USB, dois módulos foram desenvolvidos: *_usb_init()* e *_usb_close()*.

Nos nomes destes módulos foi necessário a adição de um sublinhado, "_", pois faz parte da biblioteca do *libusb* as funções *usb_init()* e *usb_close()*.

No primeiro módulo, de inicio de serviço de comunicação com interfaces USB, realiza-se uma busca por dispositivos conectados ao barramento, e retorna-se ao sistema o numero de dispositivos encontrados.

```
793
794 /**-------------------------------------------------------------------------
795  * PROCEDURE : _usb_init
796  *
797  * PURPOSE   : Iniciar um servico para USB
798  *
799  * INPUT     : Nenhum
800  *
801  * OUTPUT    : Retorna o numero de dispositivos encontrados
802  *
803  * ERROS     : Retorna um valor negativo (-1)
804  *-------------------------------------------------------------------------**/
805 int _usb_init(void)
806 {
807     int numdev;
808     /* iniciando usb */
809     usb_init();
810
811     /* verificacao de bus */
812     if (usb_find_busses() < 0) {
813         perror("_usb_init: usb_find_busses()");
814         return (-1);
815     }
816
817     /* verificacao de devices */
818     if ( (numdev = usb_find_devices()) < 0) {
819         perror("_usb_init: usb_find_devices()");
820         return (-1);
821     }
822
823     return numdev;
824 }
```

Fig. 22 – Módulo de inicio de serviço de comunicação com dispositivos USB

No segundo módulo, o módulo de encerramento do serviço de comunicação com interfaces USB, entra como argumento o dispositivo USB que se deseja encerrar a comunicação, o argumento **udh*, um ponteiro para uma estrutura *usb_dev_handle*.

```
826 /**-------------------------------------------------------------------------
827  * PROCEDURE : _usb_close
828  *
829  * PURPOSE   : Encerrar servico USB
830  *
831  * INPUT     : udh - apontador para estrutura usb_dev_handle
832  *
833  * OUTPUT    : Retorna um valor positivo (1)
834  *
835  * ERROS     : Retorna um valor negativo (-1)
836  *-------------------------------------------------------------------------**/
837 int _usb_close(usb_dev_handle *udh)
838 {
839     if (usb_close(udh) < 0) {
840         perror("_usb_close: usb_close()");
841         return (-1);
842     }
843
844     return (1);
845 }
```

Fig. 23 – Módulo de encerramento de serviço de comunicação com dispositivo USB

3.4 MÓDULOS DE INTERCOMUNICAÇÃO ENTRE PROCESSOS

Este módulo serve para a intercomunicação entre os processos, a fim de que possa ocorrer a transferência de dados na mesma máquina. Este é o caso comentado no capitulo 3.1, sobre a comunicação entre processos em uma mesma máquina. Com este módulo busca-se ganhar tempo na transferência dos dados. Três módulos foram desenvolvidos, sendo o primeiro responsável pela criação do *pipe*, os outros dois responsáveis pela abertura do *pipe* para leitura e escrita, conforme segue:

```
/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_init
 *
 * PURPOSE   : Verificar e/ou criar um canal de comunicacao entre processos
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser criado
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_init(char *path)
{
    struct stat status;
    memset(&status, 0, sizeof(status));

    /* recebendo status do arquivo pipe */
    if (stat(path, &status) < 0) {

        /* arquivo pipe nao existe */
        if (EBADF) {

            /* cria o arquivo pipe */
            if (mknod(path, S_IFIFO, 0) < 0) {
                perror("pipe_read_init: mknod(1)");
                return (-1);
            }
        }

        else {

            /* erro nao esperado */
            perror("pipe_init: stat()");

            if (mknod(path, S_IFIFO, 0) < 0) {
                perror("pipe_read_init: mknod(2)");
                return (-1);
            }
        }
    }

```

Fig. 24 – Módulo de inicio de serviço de intercomunicação entre processos

```
547
548     else {
549
550         /* verifica se arquivo e' do tipo pipe */
551         if (!S_ISFIFO(status.st_mode)) {
552             if (remove(path) < 0) {
553                 perror("pipe_read_init: remove()");
554                 return (-1);
555             }
556
557             if (mknod(path, S_IFIFO, 0) < 0) {
558                 perror("pipe_read_init: mknod(3)");
559                 return (-1);
560             }
561         }
562     }
563
564     return (1);
565 }
```

Fig. 25 – Continuação do módulo de intercomunicação entre processos

O próximo módulo é responsável pela abertura do canal de comunicação para somente leitura, utiliza-se para isso o argumento O_RDONLY:

```
/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_read_init
 *
 * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para leitura
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser aberto para
 *                    leitura
 *
 * OUTPUT    : Retorna o numero do pipe criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_read_init(char *path)
{
    int pipefd;

    /* cria arquivo pipe */
    if (pipe_init(path) < 0) {
        printf("Falha ao criar arquivo pipe");
        return (-1);
    }

    /* abre pipe para somente leitura */
    if ( (pipefd = open(path, O_RDONLY)) < 0) {
        perror("pipe_read_init: open()");
        return (-1);
    }

    return pipefd;
}
```

Fig. 26 – Módulo de abertura de um *pipe* para somente leitura

Neste módulo, a abertura do canal de comunicação para somente escrita, com o argumento O_WRONLY:

```
598 /**-------------------------------------------------------------------------
599  * PROCEDURE : pipe_write_init
600  *
601  * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para escrita
602  *
603  * INPUT     : path - caminho/nome do arquivo do tipo pipe a ser aberto para
604  *                    escrita
605  *
606  * OUTPUT    : Retorna o numero do pipe criado
607  *
608  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
609  *-------------------------------------------------------------------------**/
610 int pipe_write_init(char *path)
611 {
612     int pipefd;
613
614     /* cria arquivo pipe */
615     if (pipe_init(path) < 0) {
616         printf("Falha ao criar arquivo pipe");
617         return (-1);
618     }
619
620     /* abre pipe para somente escrita */
621     if ( (pipefd = open(path, O_WRONLY)) < 0) {
622         perror("pipe_write_init: open()");
623         return (-1);
624     }
625
626     return pipefd;
627 }
```

Fig. 27 – Módulo de abertura de um *pipe* para somente escrita

3.5 MÓDULOS DE AQUISIÇÃO/ENVIO E TRANSFERÊNCIA DE DADOS

Os módulos seguintes foram desenvolvidos para disponibilizar a transferência de dados, em todas as combinações possíveis dentro dos grupos:

1. Aquisição, envio;
2. Rede, pipes, portas seriais, paralelas e dispositivos USB.

São eles:

a) Aquisição de dados da Rede:

```
851 /**-----------------------------------------------------------------------------
852  * PROCEDURE : network_recv
853  *
854  * PURPOSE   : Receber dado da rede
855  *
856  * INPUT     : from_sockfd - socket "fonte" do dado recebido
857  *             buffer      - buffer de armazenamento
858  *             buffer_size - tamanho maximo do buffer de leitura
859  *
860  * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
861  *
862  * ERRORS    : Em falha o sistema retorna um valor negativo
863  *-----------------------------------------------------------------------------**/
864 int network_recv(int from_sockfd, char *buffer, int buffer_size)
865 {
866     struct sockaddr_in from_addr;
867     socklen_t addrlen;
868     int nbytes;
869
870     /* alocando espaco em memoria para buffer */
871     if ( (buffer = (char *) malloc(buffer_size)) == NULL) {
872         perror("network_recv: malloc()");
873         return (-1);
874     }
875
876     memset(buffer, 0, buffer_size);
877     memset(&from_addr, 0, sizeof(from_addr));
878
879     /* recebendo dados da rede */
880     addrlen = sizeof(from_addr);
881     if ( (nbytes = recvfrom(from_sockfd,
882                             buffer,
883                             buffer_size,
884                             0,
885                             (struct sockaddr *) &from_addr,
886                             &addrlen
887                             ))< 0)
888         perror("network_recv: recvfrom()");
889
890     return nbytes;
891 }
```

Fig. 28 – Módulo de aquisição de dados da Rede

b) Envio de dados para a Rede:

```
/**--------------------------------------------------------------------------
 * PROCEDURE : network_send
 *
 * PURPOSE   : Enviar dado para rede
 *
 * INPUT     : to_sockfd   - socket "fonte" do dado recebido
 *             to_sockaddr - apontador para estrutura sockaddr_in carregada com
 *                           o endereco ip e a porta de destino
 *             buffer      - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *--------------------------------------------------------------------------**/
int network_send(int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer)
{
    int nbytes;
    /* enviando dados para rede */
    if ( (nbytes = sendto(to_sockfd,
                          buffer,
                          strlen(buffer),
                          0,
                          (const struct sockaddr *) to_sockaddr,
                          sizeof(*to_sockaddr)
                          )) < 0)
        perror("network_send: sendto()");

    /* liberando espaco em memoria */
    free(buffer);

    return nbytes;
}
```

Fig. 29 – Módulo de envio de dados para a Rede

c) Aquisição de dados de um Descritor:

Obs: Neste módulo foram incluídos as aquisições recebidas de um canal pipe, ou de portas seriais, ou paralelas, pois em todos estes casos a aquisição é feita da mesma maneira, envolvendo um descritor.

```
926 /**-----------------------------------------------------------------------------
927  * PROCEDURE : fd_recv
928  *
929  * PURPOSE   : Receber dado de um canal de pipe, de uma porta serial ou paralela
930  *
931  * INPUT     : from_fd     - descritor pipe de destino
932  *             buffer      - buffer de armazenamento
933  *             buffer_size - tamanho do buffer de leitura
934  *
935  * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
936  *
937  * ERRORS    : Em falha o sistema retorna um valor negativo
938  *-----------------------------------------------------------------------------**/
939 int fd_recv(int from_fd, char *buffer, int buffer_size)
940 {
941     int nbytes;
942
943     /* alocando espaco em memoria para buffer */
944     if ( (buffer = (char *) malloc(buffer_size)) == NULL) {
945         perror("fd_recv: malloc()");
946         return (-1);
947     }
948
949     /* recebendo dados de pipe */
950     if ( (nbytes = read(from_fd, buffer, buffer_size)) < 0)
951         perror("fd_recv: read()");
952
953     return nbytes;
954 }
```

Fig. 30 – Módulo de aquisição de dados de um Descritor

d) Módulo de envio de dados para um Descritor:

Obs: Neste módulo, assim como explicado no item (c) deste capitulo, a comunicação se da através de descritores.

```
955
956 /**-------------------------------------------------------------------------
957  * PROCEDURE : fd_send
958  *
959  * PURPOSE   : Enviar dado para um canal de pipe, uma porta serial ou paralela
960  *
961  * INPUT     : to_fd  - descritor pipe de destino
962  *             buffer - buffer de armazenamento
963  *
964  * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
965  *
966  * ERRORS    : Em falha o sistema retorna um valor negativo
967  *-------------------------------------------------------------------------**/
968 int fd_send(int to_fd, char *buffer)
969 {
970     int nbytes;
971
972     /* enviando dados para pipe */
973     if ( (nbytes = write(to_fd, buffer, strlen(buffer))) < 0)
974         perror("pipe_send: write()");
975
976     /* liberando espaco em memoria */
977     free(buffer);
978
979     return nbytes;
980 }
```

Fig. 31 – Módulo de envio de dados para um Descritor

e) Aquisição de dados de um dispositivo USB:

```
982 /**-------------------------------------------------------------------------
983  * PROCEDURE : _usb_recv
984  *
985  * PURPOSE   : Receber dados de um dispositivo usb
986  *
987  * INPUT     : from_udh      - apontador para estrutura usb_dev_handle
988  *             from_endpoint - endpoint do dispositivo fonte
989  *             timeout       - tempo limite de aquisicao do dado
990  *             buffer        - buffer de armazenamento
991  *             buffer_size   - tamanho do buffer de leitura
992  *
993  * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
994  *
995  * ERRORS    : Em falha o sistema retorna um valor negativo
996  *-------------------------------------------------------------------------**/
997 int _usb_recv(usb_dev_handle *from_udh,
998               int from_endpoint,
999               int timeout,
1000              char *buffer,
1001              int buffer_size
1002              )
1003 {
1004     int nbytes;
1005
1006     /* alocando espaco em memoria */
1007     if ( (buffer = (char *) malloc(buffer_size)) == NULL) {
1008         perror("_usb_recv: malloc()");
1009         return (-1);
1010     }
1011
1012     memset(buffer, 0, buffer_size);
1013
1014     /* recebendo dado de usb */
1015     if ( (nbytes = usb_bulk_read(from_udh,
1016                                  from_endpoint,
1017                                  buffer,
1018                                  buffer_size,
1019                                  timeout
1020                                  )) < 0)
1021         perror("_usb_recv: usb_bulk_read()");
1022
1023     return nbytes;
1024 }
```

Fig. 32 – Módulo de aquisição de dados de um dispositivo USB

f) Envio de dados para um dispositivo USB

```
1026 /**-------------------------------------------------------------------------
1027  * PROCEDURE : _usb_send
1028  *
1029  * PURPOSE   : Enviar dados para um dispositivo usb
1030  *
1031  * INPUT     : to_udh      - apontador para estrutura usb_dev_handle
1032  *             to_endpoint - endpoint do dispositivo de destino
1033  *             timeout     - tempo maximo de tentativa de envio
1034  *             buffer      - buffer de armazenamento
1035  *
1036  * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
1037  *
1038  * ERRORS    : Em falha o sistema retorna um valor negativo
1039  *-------------------------------------------------------------------------**/
1040 int _usb_send(usb_dev_handle *to_udh,
1041               int to_endpoint,
1042               int timeout,
1043               char *buffer
1044               )
1045 {
1046     int nbytes;
1047
1048     /* enviando dado para usb */
1049     if ( (nbytes = usb_bulk_write(to_udh,
1050                                   to_endpoint,
1051                                   buffer,
1052                                   strlen(buffer),
1053                                   timeout
1054                                   )) < 0)
1055         perror("_usb_send: usb_bulk_write()");
1056
1057     /* liberando memoria */
1058     free(buffer);
1059
1060     return nbytes;
1061 }
```

Fig. 33 – Módulo de envio de dados para um dispositivo USB

Com estes módulos de aquisição/envio de dados fica possível desenvolver as combinações para que a transferência do dado possa ocorrer. São descritos a seguir:

g) Dado recebido da Rede e enviado para a Rede:

```
1065 /**---------------------------------------------------------------------------
1066  * PROCEDURE : network_to_network
1067  *
1068  * PURPOSE   : Receber dado da rede e enviar para a rede
1069  *
1070  * INPUT     : from_sockfd - socket "fonte" do dado recebido
1071  *             to_sockfd   - socket de destino do dado a ser enviado
1072  *             to_sockaddr - apontador para estrutura sockaddr_in de destino
1073  *             buffer      - buffer de armazenamento
1074  *             buffer_size - tamanho maximo do buffer de leitura
1075  *
1076  * OUTPUT    : Retorna a quantidade de bytes enviados.
1077  *
1078  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1079  *---------------------------------------------------------------------------**/
1080 int network_to_network(int from_sockfd, int to_sockfd,
1081  struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size)
1082 {
1083     int nbytes;
1084
1085     /* recebendo dados da rede */
1086     if (network_recv(from_sockfd, buffer, buffer_size) < 0)
1087         return (-1);
1088
1089     /* enviando dados para rede */
1090     nbytes = network_send(to_sockfd, to_sockaddr, buffer);
1091
1092     return nbytes;
1093 }
```

Fig. 34 – Módulo de transferência de dados Rede-Rede

h) Dado recebido da Rede e enviado para um Descritor:

Obs: Descritores possíveis foram discutidos no item (c) desde capitulo

```
1095 /**-------------------------------------------------------------------------
1096  * PROCEDURE : network_to_fd
1097  *
1098  * PURPOSE   : Receber dado da rede e enviar para um canal de pipe, ou uma
1099  *             porta serial ou paralela.
1100  *
1101  * INPUT     : from_sockfd - socket "fonte" do dado recebido
1102  *             to_fd       - socket de destino do dado a ser enviado
1103  *             buffer      - buffer de armazenamento
1104  *             buffer_size - tamanho maximo do buffer de leitura
1105  *
1106  * OUTPUT    : Retorna a quantidade de bytes enviados.
1107  *
1108  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1109  *-------------------------------------------------------------------------**/
1110 int network_to_fd(int from_sockfd, int to_fd, char *buffer, int buffer_size)
1111 {
1112     int nbytes;
1113
1114     /* recebendo dados da rede */
1115     if (network_recv(from_sockfd, buffer, buffer_size) < 0)
1116         return (-1);
1117
1118     /* enviando dados para FD */
1119     nbytes = fd_send(to_fd, buffer);
1120
1121     return nbytes;
1122 }
```

Fig. 35 – Módulo de transferência de dados Rede-Descritor

i) Dado recebido da rede e enviado para um dispositivo USB

```
1124 /**-------------------------------------------------------------------------
1125  * PROCEDURE : network_to_usb
1126  *
1127  * PURPOSE   : Receber dado da rede e enviar para dispositivo usb
1128  *
1129  * INPUT     : from_sockfd - socket "fonte" do dado recebido
1130  *             to_udh      - apontador para estrutura usb_dev_handle
1131  *             to_endpoint - endpoint do dispositivo de destino
1132  *             timeout     - tempo limite de transferencia de dados
1133  *             buffer      - buffer de armazenamento
1134  *             buffer_size - tamanho maximo do buffer de leitura
1135  *
1136  * OUTPUT    : Retorna a quantidade de bytes enviados.
1137  *
1138  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1139  *-------------------------------------------------------------------------**/
1140 int network_to_usb(int from_sockfd, usb_dev_handle *to_udh, int to_endpoint,
1141  int timeout, char *buffer, int buffer_size)
1142 {
1143     int nbytes;
1144
1145     /* recebendo dados da rede */
1146     if (network_recv(from_sockfd, buffer, buffer_size) < 0)
1147         return (-1);
1148
1149     /* enviando dados para usb */
1150     nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);
1151
1152     return nbytes;
1153 }
```

Fig. 36 – Módulo de transferência de dados Rede-USB

j) Dado recebido de um Descritor e enviado para Descritor:

```
1154
1155 /**-------------------------------------------------------------------------
1156  * PROCEDURE : fd_to_fd
1157  *
1158  * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
1159  *             ou de uma porta paralela e enviar para um canal de pipe, ou
1160  *             uma porta serial ou paralela.
1161  *
1162  * INPUT     : from_fd     - FD fonte
1163  *             to_fd       - FD de destino
1164  *             buffer      - buffer de armazenamento
1165  *             buffer_size - tamanho maximo do buffer de leitura
1166  *
1167  * OUTPUT    : Retorna a quantidade de bytes enviados.
1168  *
1169  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1170  *-------------------------------------------------------------------------**/
1171 int fd_to_fd(int from_fd, int to_fd, char *buffer, int buffer_size)
1172 {
1173     int nbytes;
1174
1175     /* recebendo dados de FD */
1176     if (fd_recv(from_fd, buffer, buffer_size) < 0)
1177         return (-1);
1178
1179     /* enviando dados para FD */
1180     nbytes = fd_send(to_fd, buffer);
1181
1182     return nbytes;
1183 }
```

Fig. 37 – Módulo de transferência Descritor-Descritor

k) Dado recebido de um Descritor e enviado para a Rede:

```
1185 /**-------------------------------------------------------------------------
1186  * PROCEDURE : fd_to_network
1187  *
1188  * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
1189  *             ou de uma porta paralela e enviar para a rede.
1190  *
1191  * INPUT     : from_fd     - FD fonte
1192  *             to_sockfd   - socket de destino
1193  *             to_sockaddr - apontador para estrutura sockaddr_in
1194  *             buffer      - buffer de armazenamento
1195  *             buffer_size - tamanho maximo do buffer de leitura
1196  *
1197  * OUTPUT    : Retorna a quantidade de bytes enviados.
1198  *
1199  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1200  *-------------------------------------------------------------------------**/
1201 int fd_to_network(int from_fd, int to_sockfd, struct sockaddr_in *to_sockaddr,
1202  char *buffer, int buffer_size)
1203 {
1204     int nbytes;
1205
1206     /* recebendo dados de FD */
1207     if (fd_recv(from_fd, buffer, buffer_size) < 0)
1208         return (-1);
1209
1210     /* enviando dados para rede */
1211     nbytes = network_send(to_sockfd, to_sockaddr, buffer);
1212
1213     return nbytes;
1214 }
```

Fig. 38 – Módulo de transferência Descritor-Rede

l) Dado recebido de um descritor e enviado para dispositivo USB:

```
1215
1216 /**-------------------------------------------------------------------------
1217  * PROCEDURE : fd_to_usb
1218  *
1219  * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
1220  *             ou de uma porta paralela e enviar para dispositivo usb.
1221  *
1222  * INPUT     : from_fd     - FD fonte
1223  *             to_udh      - apontador para estrutura usb_dev_handle
1224  *             to_endpoint - endpoint do dispositivo usb de destino
1225  *             timeout     - tempo maximo de transferencia de dados
1226  *             buffer      - buffer de armazenamento
1227  *             buffer_size - tamanho maximo do buffer de leitura
1228  *
1229  * OUTPUT    : Retorna a quantidade de bytes enviados.
1230  *
1231  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1232  *-------------------------------------------------------------------------**/
1233 int fd_to_usb(int from_fd, usb_dev_handle *to_udh, int to_endpoint, int timeout,
1234  char *buffer, int buffer_size)
1235 {
1236     int nbytes;
1237
1238     /* recebendo dados de FD */
1239     if (fd_recv(from_fd, buffer, buffer_size) < 0)
1240         return (-1);
1241
1242     /* enviando dados para USB */
1243     nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);
1244
1245     return nbytes;
1246 }
```

Fig. 39 – Módulo de transferência Descritor-USB

m) Dado recebido de dispositivo USB e enviado para dispositivo USB:

```
1248 /**-------------------------------------------------------------------------
1249  * PROCEDURE : _usb_to_usb
1250  *
1251  * PURPOSE   : Receber dado de dispositivo usb e enviar para dispositivo usb.
1252  *
1253  * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
1254  *             from_endpoint - endpoint do dispositivo usb fonte
1255  *             timeout       - tempo maximo de transferencia de dados
1256  *             to_udh        - apontador para estrutura usb_dev_handle destino
1257  *             to_endpoint   - endpoint do dispositivo usb de destino
1258  *             buffer        - buffer de armazenamento
1259  *             buffer_size   - tamanho maximo do buffer de leitura
1260  *
1261  * OUTPUT    : Retorna a quantidade de bytes enviados.
1262  *
1263  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1264  *-------------------------------------------------------------------------**/
1265 int _usb_to_usb(usb_dev_handle *from_udh, int from_endpoint, int timeout,
1266  usb_dev_handle *to_udh, int to_endpoint, char *buffer, int buffer_size)
1267 {
1268     int nbytes;
1269
1270     /* recebendo dados de usb */
1271     if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
1272         return (-1);
1273
1274     /* enviando dados para usb */
1275     nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);
1276
1277     return nbytes;
1278 }
```

Fig. 40 – Módulo de transferência USB-USB

n) Dado recebido de dispositivo USB e enviado para a Rede:

```
1279
1280 /**-------------------------------------------------------------------------
1281  * PROCEDURE : _usb_to_network
1282  *
1283  * PURPOSE   : Receber dado de dispositivo usb e enviar para a rede.
1284  *
1285  * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
1286  *             from_endpoint - endpoint do dispositivo usb fonte
1287  *             timeout       - tempo maximo de transferencia de dados
1288  *             to_sockfd     - socket de destino
1289  *             to_sockaddr   - apontador para estrutura sockaddr_in destino
1290  *             buffer        - buffer de armazenamento
1291  *             buffer_size   - tamanho maximo do buffer de leitura
1292  *
1293  * OUTPUT    : Retorna a quantidade de bytes enviados.
1294  *
1295  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1296  *-------------------------------------------------------------------------**/
1297 int _usb_to_network(usb_dev_handle *from_udh, int from_endpoint, int timeout,
1298  int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size)
1299 {
1300     int nbytes;
1301
1302     /* recebendo dados de usb */
1303     if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
1304         return (-1);
1305
1306     /* enviando dados para rede */
1307     nbytes = network_send(to_sockfd, to_sockaddr, buffer);
1308
1309     return nbytes;
1310 }
```

Fig. 41 – Módulo de transferência USB-Rede

o) Dado recebido de dispositivo USB e enviado para um Descritor:

```
1312 /**-------------------------------------------------------------------------
1313  * PROCEDURE : _usb_to_fd
1314  *
1315  * PURPOSE   : Receber dado de dispositivo usb e enviar para um canal de pipe,
1316  *             ou uma porta serial ou paralela
1317  *
1318  * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
1319  *             from_endpoint - endpoint do dispositivo usb fonte
1320  *             timeout       - tempo maximo de transferencia de dados
1321  *             to_fd         - FD de destino
1322  *             buffer        - buffer de armazenamento
1323  *             buffer_size   - tamanho maximo do buffer de leitura
1324  *
1325  * OUTPUT    : Retorna a quantidade de bytes enviados.
1326  *
1327  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1328  *-------------------------------------------------------------------------**/
1329 int _usb_to_fd(usb_dev_handle *from_udh, int from_endpoint, int timeout,
1330  int to_fd, char *buffer, int buffer_size)
1331 {
1332     int nbytes;
1333
1334     /* recebendo dados de usb */
1335     if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
1336         return (-1);
1337
1338     /* enviando dados para FD */
1339     nbytes = fd_send(to_fd, buffer);
1340
1341     return nbytes;
1342 }
```

Fig. 42 – Módulo de transferência USB-Descritor

3.6 MÓDULO DE MONITORAÇÃO GRÁFICA

Na construção do módulo gráfico exigiu-se a criatividade, pois as opções para atingir o objetivo não faltavam. Levou-se em consideração a portabilidade, a leveza *versus* quantidade de dados na atualização, e a facilidade no entendimento e no desenvolvimento do módulo gráfico capaz da integração entre a linguagem de programação C com a linguagem JAVA para futuras implementações e/ou alterações no código.

A forma escolhida envolvia o uso do programa Adobe Flex Builder 3, um software *opensource* voltado para aplicações em *web* da Adobe Systems Incorporated, pois facilita na geração de gráficos para futura visualização e análise em um *browser*.

Este software faz a leitura de arquivos em formato *XML* e os representa em gráficos, que para este projeto optou-se pelo gráfico tipo “linha”, pois assim torna-se claro acompanhar o histórico do movimento dos dados coletados. O módulo *gmonitor()*, apresentado logo a seguir, gera o arquivo *XML* que será lido pela aplicação:

```
1344 /**-------------------------------------------------------------------------
1345  * PROCEDURE : gmonitor
1346  *
1347  * PURPOSE   : Atualizar arquivo que gera um grafico para monitoracao do
1348  *             comportamento do sistema
1349  *
1350  * INPUT     : sockaddr      - apontador para estrutura sockaddr_in
1351  *             ipaddr        - endereco IP do servidor de monitoracao
1352  *             port          - porta de servico do servidor de monitoracao
1353  *             grpaddr       - ponteiro para estrutura ip_mreq
1354  *             mult_ipaddr   - endereco IP de multicast de servico
1355  *
1356  * OUTPUT    : Retorna um valor positivo (1)
1357  *
1358  * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
1359  *-------------------------------------------------------------------------**/
1360 int gmonitor(struct sockaddr_in *sockaddr, char ipaddr[16], int port,
1361              struct ip_mreq *grpaddr, char mult_ipaddr[16])
1362 {
1363     /* iniciando servico de monitoracao
1364      */
1365     int pid;
1366     if ( (pid = fork()) < 0) {
1367         perror("gmonitor: fork()");
1368         return(-1);
1369     }
1370
1371     /* liberando processo PAI e mantendo processo FILHO dentro de laco para
1372      * funcao select
1373      */
1374     if (pid != 0) {
1375         printf("GMonitor() em execucao.\n");
1376         return (1);
1377     }
1378
1379     /* preenchendo o arquivo xml
1380      */
1381     char *xml_start = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<items>\n";
1382     char *xml_front = "\t<item time=\"";
1383     char *xml_middle = "\" value=\"";
1384     char *xml_end = "\"/>\n";
1385     char *xml_finish = "</items>";
```

Fig. 43 – Módulo de Monitoração Gráfica

```
1386
1387      struct xml_values {
1388          char xv_time[8];
1389          char xv_value[255];
1390      } xml_values[10];
1391
1392      char xml[strlen(xml_start)
1393               + strlen(xml_front)*10
1394               + strlen(xml_middle)*10
1395               + strlen(xml_end)*10
1396               + sizeof(xml_values)*10
1397               + strlen(xml_finish)
1398               ];
1399
1400      int xml_file;
1401      char xml_fname[15];
1402
1403      char received_value[255];
1404      char received_value_aux[sizeof(received_value)-3];
1405      int nfds;
1406      fd_set rdset;
1407
1408      struct tm TM;
1409      struct timeval TV;
1410      memset(&TM, 0, sizeof(TM));
1411      memset(&TV, 0, sizeof(TV));
1412
1413      /* criando socket de para receber os dados da rede
1414       */
1415      int gmonitor_socket;
1416      if ( (gmonitor_socket = udp_server_mult_init(sockaddr,
1417          ipaddr, port, grpaddr, mult_ipaddr)) < 0)
1418      {
1419          perror("gmonitor: udp_server_mult_init()");
1420          return(-1);
1421      }
```

Fig. 44 – Continuação do Módulo de Monitoração Gráfica

No próximo trecho de código deste módulo ira evidenciar o uso da função *select()* com o objetivo de tornar a atualização do gráfico instantânea, ou seja, no momento em que um novo dado trafegue pela rede a função *select()* avisa ao aplicativo que um novo dado esta disponível para a atualização, e desta forma ela cria um novo arquivo XML com os novos dados de leitura.

```
1422
1423      /* iniciando laco para funcao select
1424       */
1425      int i;
1426      for (;;) {
1427          /* ajustando parametro de leitura de select
1428           */
1429          nfds = 0;
1430          FD_ZERO(&rdset);
1431          FD_SET(gmonitor_socket, &rdset);
1432          nfds = gmonitor_socket + 1;
1433
1434          /* iniciando funcao select.
1435           * Nao ha a necessidade de prever um tempomaximo, pois se nao houver
1436           * dados para atualizacao nao devera ser alterado o grafico de
1437           * monitoracao, nao havendo a necessidade de se dar refresh na pagina de
1438           * visualizacao do grafico. Neste caso a funcao select esta aqui somente
1439           * para que seja executadoa atualizacao assim que houver a necessidade.
1440           * Caso contrario o laco ficara em BLOCK/ON HOLD.
1441           */
1442          if (select(nfds, &rdset, NULL, NULL, NULL) < 0) {
1443              perror("gmonitor: select()");
1444  /*          return(-1);
1445  */      }
1446
1447          /* ajustando tempo de recebimento do dado
1448          */
1449          if (gettimeofday(&TV, NULL) < 0) {
1450              perror("gmonitor: gettimeofday()");
1451  /*          return (-1);
1452  */      } else {
1453              localtime_r(&TV, &TM);
1454          }
1455
1456          for (i = 0; i < 9; i++)
1457          {
1458              strcpy(xml_values[i].xv_time, xml_values[i+1].xv_time);
1459              strcpy(xml_values[i].xv_value, xml_values[i+1].xv_value);
1460          }
1461          sprintf(xml_values[9].xv_time, "%.2d:%.2d:%.2d", TM.tm_hour,
1462              TM.tm_min, TM.tm_sec);
1463
1464          /* efetuando leitura de novos dados
1465          */
1466          if (network_recv(gmonitor_socket, &received_value, sizeof(xml)) < 0) {
1467              printf("gmonitor: network_recv(): failed to receive data.\n");
1468  /*          return(-1);
1469  */      }
```

Fig. 45 – Uso da função *select()* para atualizações instantâneas

Podemos notar que o descritor que esta sendo usado no argumento *received_value* eh um socket para UDP, criado através de um servidor *multicast*, como evidenciado na figura 47. Outro detalhe da função *select()* é o bloqueio do programa ate que o valor

(neste caso, *received_value*) aguardado seja recebido pela função. Para que isso não bloqueie também os outros módulos foi utilizado a função *fork()*, mostrado na figura 46, para que o programa possa criar uma copia idêntica de si, e assim ficar disponível para executar outras tarefas.

A seguir é mostrado o trecho de geração do arquivo:

No treco de código a seguir eh mostrado o formato do dado a ser recebido pelo gmonitor() para que não seja confundido qual gráfico será atualizado:

```
1470
1471        /* para que seja atualizado o arquivo correto,
1472         * os 3 primeiros caracteres do valor recebido pelo socket
1473         * "gmonitor_socket" devera conter o tipo da estrutura a ser passada.
1474         * Por exemplo, se desejamos atualizar o ultimo valor lido pelo
1475         * sensor de temperatura T (765 graus) numero 1, entao o dado a
1476         * ser recebido sera: T01765
1477         * Sendo:
1478         * T   - sensor de temperatura
1479         * 01  - sensor numero 1
1480         * 765 - valor medido
1481         *
1482         * O arquivo de configuracao que irar gerar o grafico correspondente
1483         * tera o formato "sensorXXX.xml". No exemplo acima o arquivo
1484         * tera o nome "sensorT01.xml".
1485         *
1486         * O valor lido podera conter ate 32 caracteres, sendo possivel
1487         * sua alteracao em sua declaracao no inicio deste modulo,
1488         * conforme a necessidade.
1489         */
1490        strncat(xml_fname, "sensor", 6);
1491        strncat(xml_fname, received_value, 3);
1492        strncat(xml_fname, ".xml", 4);
1493
1494        for (i = 0; i < sizeof(received_value_aux); i++)
1495            received_value_aux[i] = received_value[i+3];
1496        sprintf(xml_values[9].xv_value, "%d", received_value_aux);
1497
1498        /* formatando arquivo XML */
1499        strncat(xml, xml_start, strlen(xml_start));
1500
1501        for (i = 0; i < 10; i++)
1502        {
1503            strncat(xml, xml_front, strlen(xml_front));
1504            strncat(xml, xml_values[i].xv_time, strlen(xml_values[i].xv_time));
1505            strncat(xml, xml_middle, strlen(xml_middle));
1506            strncat(xml, xml_values[i].xv_value,strlen(xml_values[i].xv_value));
1507            strncat(xml, xml_end, strlen(xml_end));
1508        }
1509
1510        strncat(xml, xml_finish, strlen(xml_finish));
```

Fig. 46 – Formato do dado a ser recebido pelo *gmonitor()*

No final do trecho acima e também no próximo trecho a ser mostrado o arquivo XML começa a ser receber o seu formato padrão, utilizando para isto concatenar em um único *array* de caracteres, para que possa ser escrito/atualizado rapidamente ao abrir o arquivo XML correspondente.

```
1512          /* Abrindo arquivo XML para atualizacao de dados
1513           * No caso, o arquivo sera criado, caso nao exista no sistema/diretorio
1514           * atual. A flag O_TRUNC certifica de que o arquivo sera limpo antes de
1515           * receber um novo dado/valor para escrita.
1516           */
1517          if ( (xml_file = open(xml_fname, O_CREAT | O_TRUNC | O_WRONLY) ) < 0) {
1518              perror("gmonitor: open()");
1519  /*          return(-1);
1520  */      }
1521
1522          /* atualizando arquivo */
1523          if (write(xml_file, xml, strlen(xml)) < 0) {
1524              perror("gmonitor: write()");
1525  /*          return(-1);
1526  */      }
1527
1528          /* Encerrando atualizacao de dados no arquivo XML */
1529          if (close(xml_file) < 0) {
1530              perror("gmonitor: close()");
1531  /*          return(-1);
1532  */      }
1533      }
1534
1535      /* encerrando servico de monitoracao de rede */
1536      if (close(gmonitor) < 0) {
1537          perror("gmonitor: close()");
1538          return(-1);
1539      }
1540
1541      return(1);
1542  }
```

Fig. 47 – Trecho final do Módulo de Monitoração Gráfica

Um exemplo de arquivo XML preenchido gerado por este módulo esta ilustrado na figura a seguir:

```xml
<?xml version="1.0" encoding="utf-8"?>
<items>
    <item time="22:15:00" value="1"/>
    <item time="22:16:30" value="30"/>
    <item time="22:17:01" value="60"/>
    <item time="22:17:56" value="899"/>
    <item time="22:18:04" value="50"/>
    <item time="22:18:36" value="6"/>
    <item time="22:18:57" value="30"/>
    <item time="22:19:51" value="1"/>
    <item time="22:19:52" value="4000"/>
    <item time="22:19:53" value="50"/>
</items>
```

Fig. 48 – Arquivo XML gerado pelo Módulo *gmonitor()*

A leitura deste arquivo irá gerar o gráfico:

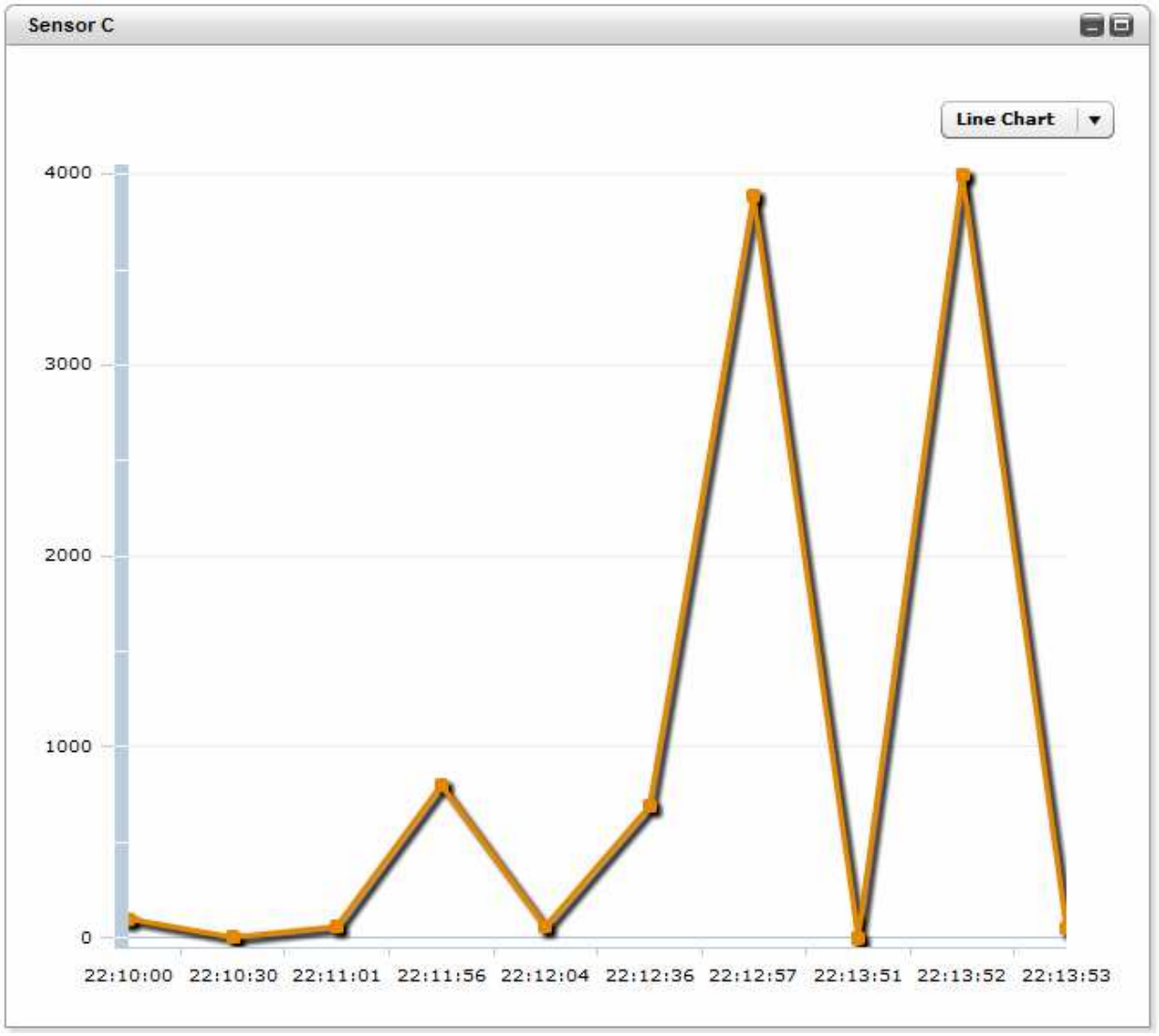

Fig. 49 – Gráfico exemplo gerado pelo aplicativo Adobe Flex Builder

O código em linguagem JAVA utilizado para a construção dos gráficos esta disponível em HTTP://www.adobe.com/.

As possibilidades de se construir um módulo gráfico para a monitoração da dinâmica do sistema são inúmeras. Esta é uma solução de rápido entendimento do código, sendo possível sua expansão e/ou aprimoramento, além de possuir uma interface com o usuário bastante agradável.

4 ANÁLISES E RESULTADOS

Durante o desenvolvimento dos módulos, realizou-se uma bateria exaustiva de testes para se chegar à melhor forma de atingir o objetivo: portabilidade, tempo de processamento (execução dos aplicativos e de compilação) e facilidade para futuras expansões e implementações.

Nos primeiros programas, tudo era implementado dentro de um único módulo, que utilizou-se omo uma ferramenta no teste das funções a fim de analisar os possíveis erros em tempo de execução. Desde ponto de vista, muito da implementação do programa surgiu desta etapa, a programação passou a analisar erros em tempo de execução, e desta forma ganhou -se robustez.

Percebe-se, porém, que desta forma perdia-se a portabilidade e a possibilidade de escolher qual módulo deveria ser executado naquele momento. E a fim de resolver esta questão separou-se o programa em vários módulos, cada módulo responsável por uma determinada tarefa (como mostrado no Capitulo 3).

Utilizando a separação dos módulos e da prevenção de possíveis erros em tempo de execução, atingiu-se o objetivo deste trabalho.

O tempo de testes do módulo gráfico foi relativamente curto, pois não foi dada ênfase neste tópico, mas sem causar dano para análises, pois qualquer erro de execução do aplicativo é previsto nos módulos. O teste, efetuado com sucesso, limitou-se em gerar o arquivo em formato XML sem que houvessem erros de leitura dentro do ambiente do Adobe Flex.

Pode-se entçao dizer que os módulos estão prontos, e aptos, para implementação no projeto do LABSIM/SCAO, e sua dinâmica de sistema vista através do módulo de monitoração gráfica, o *gmonitor().*

REFERENCIAS BIBLIOGRAFICAS

STEVENS, W. R., "**TCP/IP Illustrated. Vol. 1: The protocols**", Addison Wesley, 1994, (ISBN 0-201-63346-9).

STEVENS, W. R., "**UNIX Network Programming**", Prentice Hall, 1990, (ISBN 0-13-949876-1).

COMER, D. E., STEVENS, D. L., "**Internetworking with TCP/IP. Vol.3: Client-server programming and applications BSD socket version**", Prentice Hall, 1993, (ISBN 0-13-020272-X).

PEACOCK, C., "**USB in a nutshell**", Beyond Logic, 2002.

FLIEGL, D., "**Programming Guide for Linux USB Device Drivers**", 2000, disponivel em <http://usb.cs.edu/usbdoc>.

SWEET, M. R., "**Serial Programming Guide for POSIX Operating Systems**", 2005.

MESQUITA, R. C., "**Apostila do Curso de Linguagem C / UFMG**", Universidade Federal de Minas Gerais, 1998.

PEREIRA, S. L., "**Linguagem C – Curso Completo**", FATEC-SP, 1997.

JONES, M. T., "**BSD Sockets Programming from a Multi-Language Perspective**", Charles River Media, 2004.

Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, ST-NXP Wireless, Texas Instruments, "**Universal Serial Bus 3.0 Specification**", 2008, disponivel em <http://www.usb.org/>.

"**Sourceforge.net: libusb home**", disponivel em <http://libusb.wiki.sourceforge.net/>

"**The Linux Documentation Project**", disponivel em <http://tldp.org/>

"**CCSDS.org - The Consultative Committee for Space Data Systems**", disponivel em http://www.ccsds.org/

"**MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More**", disponivel em <http://msdn.microsoft.com/>

APENDICE A – CODIGOS-FONTE

Nas próximas paginas será exibido os códigos-fonte desenvolvidos utilizando o editor Edit Plus. Primeiramente teremos o código-fonte principal, intitulado de *INPE.c*, seguido de sua biblioteca, intitulada *INPE.h*, sendo esta utilizada para armazenar as bibliotecas padrão, como a *stdlib.h*, *stdio.h*, como para armazenar também as chamadas (sintaxes) das funções criadas no arquivo *INPE.c*.

```
/**---------------------------------------------------------------------------
 * MODULE NAME : inpe.c
 *
 * PROCEDURE   : Application
 *
 * PURPOSE     : It implements a network application server using both
 *               UDP and TCP protocols and pipes for interprocess
 *               communication.
 *
 * DESCRIPTION : This application was created to implement and test
 *               server functions, to identify and respond
 *               multiple sockets, and/or pipes and/or serial communications
 *               at the same time.
 *
 * RELEASE     : May 19th, 2009.
 *
 *               Arthur Adriano Ferreira, UBC, Bolsista PIBIC/CNPq
 *                E-mail: arthuradriano@gmail.com
 *
 *               Prof. Dr. Ulisses Thadeu Vieira Guedes, DMC/INPE, Orientador
 *                E-mail: utvg@dem.inpe.br
 *-------------------------------------------------------------------------**/

#include "inpe.h"

/*==================== prototipo das funcoes ================================*/

/**---------------------------------------------------------------------------
 * PROCEDURE : socket_init
 *
 * PURPOSE   : Cria um socket para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : domain   - especificacao do dominio de comunicacao
 *             type     - especificacao do tipo de protocolo
 *             protocol - especificacao do protocolo
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_init(int domain, int type, int protocol)
{
        int sockfd;
        unsigned int on = 1;

        /* criando socket */

        if ( (sockfd = socket(domain, type, protocol)) < 0) {

                perror("socket_init: socket()");

                return (-1);

        }

        /* ajustando opcao de socket */

        if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {

                perror("socket_init: setsockopt()");

/*              return (-1);
*/      }


        return sockfd;
}

/**---------------------------------------------------------------------------
```

```
 * PROCEDURE : sockaddr_init
 *
 * PURPOSE   : Carregar estrutura sockaddr_in.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip
 *             port     - porta
 *
 * OUTPUT    : Estrutura sockaddr_in carregada. Os valores nao informados serao
 *             preenchidos com zeros.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int sockaddr_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
       /* carregando estrutura sockaddr com os valores ipaddr e port */
       memset(sockaddr, 0, sizeof(*sockaddr));
       sockaddr->sin_family = AF_INET;
       sockaddr->sin_port = htons(port);
       if (inet_pton(AF_INET, ipaddr, &sockaddr->sin_addr.s_addr) <= 0) {
              perror("sockaddr_init: inet_pton()");
              return (-1);
       }

       return (1);
}

/**-----------------------------------------------------------------------------
 * PROCEDURE : socket_udp_init
 *
 *
 * PURPOSE   : Cria um socket em UDP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int socket_udp_init(void)
{
       int sockfd;

       /* criando socket em UDP */
       sockfd = socket_init(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
       return sockfd;
}

/**-----------------------------------------------------------------------------
 * PROCEDURE : socket_tcp_init
 *
 * PURPOSE   : Cria um socket em TCP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int socket_tcp_init(void)
{
       int sockfd;

       /* criando socket em TCP */
       sockfd = socket_init(AF_INET, SOCK_STREAM, IPPROTO_TCP);
       return sockfd;
}
```

```
/**-------------------------------------------------------------------------
 * PROCEDURE : socket_icmp_init
 *
 * PURPOSE   : Cria um socket em ICMP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_icmp_init(void)
{
       int sockfd;

       /* criando socket em ICMP */
       sockfd = socket_init(AF_INET, SOCK_RAW, IPPROTO_ICMP);
       return sockfd;
}

/**-------------------------------------------------------------------------

 * PROCEDURE : tcp_client_init
 *
 * PURPOSE   : Iniciar um servico de rede. Cria um socket para TCP e carrega a
 *             estrutura cliente com o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - numero do socket para TCP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/

int tcp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{

       int sockfd;

       /* criando socket em TCP
        * obs: erro ocorrido ja apresentado em socket_init()
        */
       sockfd = socket_tcp_init();

       /* carregando estrutura */
       if (sockaddr_init(sockaddr, ipaddr, port) < 0)
              return (-1);

       return sockfd;

}


/**-------------------------------------------------------------------------
 * PROCEDURE : tcp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para TCP e carrega
 *             a estrutura servidora com endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - Estrutura carregada. Os campos vazios sao
```

```
 *                       preenchidos com zero.
 *             sockfd   - Socket para TCP criado.
 *
 * ERRORS    : Em caso de falha na criacao do socket ou no preenchimento
 *             da estrutura contendo o endereco o sistema retorna um
 *             numero negativo (-1).
 *-------------------------------------------------------------------------**/
int tcp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{

        int sockfd;

        /* criando socket em TCP utilizando tcp_client_init()
         * obs: erro ocorrido ja apresentado em socket_init()
         */
        if ( (sockfd = tcp_client_init(sockaddr, ipaddr, port)) < 0)
                return (-1);

        /* fixando estrutura sockaddr ao socket */

        if (bind(sockfd,
                        (const struct sockaddr *) sockaddr,
                        sizeof(*sockaddr)
                        ) < 0) {

                perror("tcp_server_init: bind()");

                return (-1);

        }

        /* estabelecendo lista de espera ajustado em 5 */

        if (listen(sockfd, 5) < 0) {

                perror("tcp_server_init: listen()");

                return (-1);

        }


        return sockfd;

}


/**-------------------------------------------------------------------------
 * PROCEDURE : accept_init
 *
 * PURPOSE   : Criar um novo socket para a conexao estabelecida
 *
 * INPUT     : sockfd     - socket do servidor a aceitar nova conexao
 *             peeraddr   - ponteiro para estrutura sockaddr_in
 *
 * OUTPUT    : acceptsock - retorna o numero do socket criado.
 *             peeraddr   - estrutura carregada com informacoes do cliente.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int accept_init(int sockfd, struct sockaddr_in *peeraddr)
{

        int acceptsock;

        socklen_t addrlen;


        memset(peeraddr, 0, sizeof(*peeraddr));
```

```
        addrlen = sizeof(*peeraddr);

        /* criando novo socket para a conexao estabelecida */

        if ( (acceptsock = accept(sockfd,
                                                (struct sockaddr *) peeraddr,
                                                &addrlen)
                                                ) < 0) {
                perror("accept_init: accept()");
                return (-1);
        }

        return acceptsock;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : udp_client_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega a
 *             estrutura contendo o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para UDP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int udp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{

        int sockfd;

        /* criando socket em UDP
         * obs: erro ocorrido ja apresentado em socket_init()
         */
        sockfd = socket_udp_init();

        /* carregando estrutura */
        if (sockaddr_init(sockaddr, ipaddr, port) < 0)
                return (-1);

        return sockfd;
}


/**-------------------------------------------------------------------------
 * PROCEDURE : udp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega
 *             a estrutura servidora com endereco IP e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr  - Estrutura carregada. Os campos vazios sao
 *                         preenchidos com zero.
 *             sockfd    - Socket para UDP criado.
 *
 * ERRORS    : Em falha na criacao ou no preenchimento da estrutura que
 *             contem o endereco IP de servico e a porta de destino o
 *             sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int udp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
```

```
        int sockfd;

        /* criando socket em UDP
         * obs: erro ocorrido ja apresentado em socket_init()
         */
        sockfd = udp_client_init(sockaddr, ipaddr, port);

        /* fixando estrutura sockaddr_in ao socket */

        if (bind(sockfd,
                        (const struct sockaddr *) sockaddr,
                        sizeof(*sockaddr)
                        ) < 0) {

                perror("udp_server_init: bind()");

                return (-1);

        }

        return sockfd;
}


/**-------------------------------------------------------------------------
 * PROCEDURE : udp_server_mult_init
 *
 * PURPOSE   : Iniciar servico de rede. Criar um socket para UDP e carregar a
 *             estrutura contendo o endereco IP de servico em modo multicast
 *             e a porta de destino.
 *
 * INPUT     : sockaddr    - ponteiro para estrutura sockaddr_in
 *             ipaddr      - endereco ip de destino
 *             port        - porta de destino
 *             grpaddr     - ponteiro para estrutura im_req
 *             mult_ipaddr - endereco ip de destino
 *
 * OUTPUT    : sockaddr    - Estrutura carregada. Os campos vazios sao
 *                           preenchidos com zero.
 *             svaddr      - Estrutura carregada. Os campos vazios sao
 *                           preenchidos com zero.
 *             grpaddr     - Estrutura carregada. Os campos vazios sao
 *                           preenchidos com zero.
 *             sockfd      - Socket para UDP multicast criado.
 *
 * ERRORS    : Em falha o sistema retorna um numero negativo (-1)
 *-------------------------------------------------------------------------**/
int udp_server_mult_init(struct sockaddr_in *sockaddr,
                                             char ipaddr[15],
                                             int port,
                                             struct ip_mreq *grpaddr,
                                             char mult_ipaddr[16]
                                             )
{

        int sockfd;

        /* criando socket em UDP */
        if ( (sockfd = udp_server_init(sockaddr, ipaddr, port)) < 0)
                return (-1);

        /* fixando estrutura sockaddr_in ao socket */

        if (bind(sockfd,
                        (const struct sockaddr *) sockaddr,
                        sizeof(*sockaddr)
                        ) < 0) {

                perror("udp_server_mult_init: bind()");
```

```
                return (-1);

        }

        /* preenchendo estrutura im_req com endereco de multicast */

        if (inet_pton(AF_INET, mult_ipaddr, &grpaddr->imr_multiaddr.s_addr) <= 0) {

                perror("udp_server_mult_init: inet_pton()");

                return (-1);

        }


        /* preenchendo estrutura im_req com endereco de interface */

        if (inet_pton(AF_INET, ipaddr, &grpaddr->imr_interface.s_addr) <= 0) {

                perror("udp_server_mult_init: inet_pton(IPADDR) ");

                return (-1);

        }


        /* adicionando socket ao grupo informado na estrutura im_req */

        if (setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, grpaddr,
                                sizeof(*grpaddr)) < 0) {

                perror("udp_server_mult_init: setsockopt()");

                return (-1);

        }


        return sockfd;

}


/**-------------------------------------------------------------------------
 * PROCEDURE : udp_server_mult_close
 *
 * PURPOSE   : Encerrar o servico de rede
 *
 * INPUT     : sockfd  - socket a ser encerrado
 *             grpaddr - apontador para estrutura ip_mreq
 *
 * OUTPUT    : Retira sockfd da lista de membros de multicast e fecha o socket.
 *
 * ERRORS    : Em falha o sistema retorna um numero negativo (-1).
 *-------------------------------------------------------------------------**/
int udp_server_mult_close(int sockfd, struct ip_mreq *grpaddr)
{
        int status;

        /* retirando membro do grupo de multicast */
        if ( (status = setsockopt(sockfd,
                                                IPPROTO_IP,
                                                IP_DROP_MEMBERSHIP,
                                                grpaddr,
                                                sizeof(*grpaddr)
                                                )) < 0) {

                perror("udp_server_mult_close: setsockopt()");
```

```
                return status;

        }

        /* encerrando socket */

        if (fd_close(sockfd) < 0)
                return (-1);


        return status;

}

/**---------------------------------------------------------------------------
 * PROCEDURE : icmp_client_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para ICMP e carrega a
 *             estrutura contendo o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para UDP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int icmp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{

        int sockfd;

        /* criando socket em ICMP
         * obs: erro ocorrido ja apresentado em socket_init()
         */
        sockfd = socket_icmp_init();

        /* carregando estrutura */
        if (sockaddr_init(sockaddr, ipaddr, port) < 0)
                return (-1);

        return sockfd;

}

/**---------------------------------------------------------------------------
 * PROCEDURE : icmp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para ICMP e carrega
 *             a estrutura servidora com endereco IP e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr  - Estrutura carregada. Os campos vazios sao
 *                         preenchidos com zero.
 *             sockfd    - Socket para UDP criado.
 *
 * ERRORS    : Em falha na criacao ou no preenchimento da estrutura que
 *             contem o endereco IP de servico e a porta de destino o
 *             sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int icmp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port)
{
        int sockfd;
```

```
        /* criando socket em ICMP
         * obs: erro ocorrido ja apresentado em socket_init()
         */
        sockfd = icmp_client_init(sockaddr, ipaddr, port);
        /* fixando estrutura sockaddr_in ao socket */
        if (bind(sockfd,
                        (const struct sockaddr *) sockaddr,
                        sizeof(*sockaddr)
                        ) < 0) {
                perror("icmp_server_init: bind()");
                return (-1);
        }

        return sockfd;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_init
 *
 * PURPOSE   : Verificar e/ou criar um canal de comunicacao entre processos
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser criado
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_init(char *path)
{
        struct stat status;
        memset(&status, 0, sizeof(status));
        /* recebendo status do arquivo pipe */
        if (stat(path, &status) < 0) {
                /* arquivo pipe nao existe */
                if (EBADF) {
                        /* cria o arquivo pipe */
                        if (mknod(path, S_IFIFO, 0) < 0) {
                                perror("pipe_read_init: mknod(1)");
                                return (-1);
                        }
                }

                else {
                        /* erro nao esperado */
                        perror("pipe_init: stat()");

                        if (mknod(path, S_IFIFO, 0) < 0) {
                                perror("pipe_read_init: mknod(2)");
                                return (-1);
                        }
                }
        }

        else {
                /* verifica se arquivo e' do tipo pipe */
                if (!S_ISFIFO(status.st_mode)) {
                        if (remove(path) < 0) {
                                perror("pipe_read_init: remove()");
                                return (-1);
                        }

                        if (mknod(path, S_IFIFO, 0) < 0) {
                                perror("pipe_read_init: mknod(3)");
                                return (-1);
                        }
                }
        }

        return (1);
}
```

```
/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_read_init
 *
 * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para leitura
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser aberto para
 *                    leitura
 *
 * OUTPUT    : Retorna o numero do pipe criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_read_init(char *path)
{
        int pipefd;

        /* cria arquivo pipe */
        if (pipe_init(path) < 0) {
                printf("Falha ao criar arquivo pipe");
                return (-1);
        }

        /* abre pipe para somente leitura */
        if ( (pipefd = open(path, O_RDONLY)) < 0) {
                perror("pipe_read_init: open()");
                return (-1);
        }

        return pipefd;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_write_init
 *
 * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para escrita
 *
 * INPUT     : path - caminho/nome do arquivo do tipo pipe a ser aberto para
 *                    escrita

 *
 * OUTPUT    : Retorna o numero do pipe criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_write_init(char *path)
{
        int pipefd;

        /* cria arquivo pipe */
        if (pipe_init(path) < 0) {
                printf("Falha ao criar arquivo pipe");
                return (-1);
        }

        /* abre pipe para somente escrita */
        if ( (pipefd = open(path, O_WRONLY)) < 0) {
                perror("pipe_write_init: open()");
                return (-1);
        }

        return pipefd;
}



/**-------------------------------------------------------------------------
 * PROCEDURE : serial_init
 *
 * PURPOSE   : Iniciar servico de serial
```

```
 *
 * INPUT     : serial - caminho da porta serial a ser iniciada
 *             ispeed - velocidade de INPUT
 *             ospeed - velocidade de OUTPUT
 *
 * OUTPUT    : Retorna o numero do serial criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int serial_init(char *serial, int ispeed, int ospeed)
{
        int serialfd;
        struct termios options;


        /* abre o caminho apontado por *serial em leitura e escrita, sem obter o
         * controle total do terminal (controle permanece com o SO) */
        if ( (serialfd = open(serial, O_RDWR | O_NOCTTY | O_NONBLOCK)) < 0) {
                perror("serial_init: open()");
                return (-1);
        }

        /* ajuste de flags */
        if (fcntl(serialfd, F_SETFL, FNDELAY) < 0) {
                perror("serial_init: fcntl()");
/*              return (-1);
*/      }

        /* verifica flags existentes */
        memset(&options, 0, sizeof(options));
        tcgetattr(serialfd, &options);

        /* ajustando opcoes de flags para modo raw */
        /* opcoes de output */
        options.c_oflag &= ~OPOST;    /* habilita modo raw */
        /* opcoes locais */
        options.c_lflag &= ~ECHO;     /* desabilita "echo" de caracteres */
        options.c_lflag &= ~ECHOE;
        options.c_lflag &= ~ECHONL;
        options.c_lflag &= ~ICANON;   /* habilita modo raw */
        options.c_lflag &= ~ISIG;     /* desabilita sinais */
        options.c_lflag &= ~IEXTEN;   /* desabilita funcoes extras */
        /* opcoes de controle */
        options.c_cflag |= CLOCAL;    /* nao altera o owner */
        options.c_cflag |= CREAD;     /* habilita leitura */
        options.c_cflag |= CS8;       /* 8 bits */
        options.c_cflag &= ~CSIZE;    /* desabilita bit de mascara */
        options.c_cflag &= ~CSTOPB;   /* 1 bit de stop */
        options.c_cflag &= ~CRTSCTS;  /* desabilita controle de hardware */
        options.c_cflag &= ~PARENB;   /* desabilita bit de paridade */
        /* opcoes de input */
        options.c_iflag &= ~IGNBRK;   /* aceita condicoes de parada  */
        options.c_iflag &= ~BRKINT;   /* nao envia sinais ao condicoes de parada */
        options.c_iflag &= ~PARMRK;   /* ignora erros de paridade */
        options.c_iflag &= ~ISTRIP;
        options.c_iflag &= ~INLCR;    /* mapeia de NL para CR */
        options.c_iflag &= ~IGNCR;    /* aceita CR */
        options.c_iflag &= ~ICRNL;    /* permanesce mapeado para CR */
        options.c_iflag &= ~IXON;    /* desabilita controle de software */

        cfsetispeed(&options, ispeed);
        cfsetospeed(&options, ospeed);

        /* ajustando flags aa porta aberta
         * descarregando buffers de input e
         * de output e aplica a mudanca
         */
        tcsetattr(serialfd, TCSAFLUSH, &options);
```

```
        return serialfd;

}

/**----------------------------------------------------------------------------
 * PROCEDURE : parallel_init
 *
 * PURPOSE   : Iniciar servico de comunicacao com porta paralela
 *
 * INPUT     : parallel - caminho da porta paralela a ser iniciada
 *             ispeed   - velocidade de INPUT
 *             ospeed   - velocidade de OUTPUT
 *
 * OUTPUT    : Retorna o numero do serial criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *----------------------------------------------------------------------------**/
int parallel_init(char *parallel, int ispeed, int ospeed)
{
        int parallelfd;
        struct termios options;


        /* abre o caminho apontado por *serial em leitura e escrita, sem obter o
         * controle total do terminal (controle permanece com o SO) */
        if ( (parallelfd = open(parallel, O_RDWR | O_NOCTTY | O_NONBLOCK)) < 0) {
                perror("parallel_init: open()");
                return (-1);
        }

        /* ajuste de flags */
        if (fcntl(parallelfd, F_SETFL, FNDELAY) < 0) {
                perror("parallel_init: fcntl()");
/*              return (-1);
*/      }

        /* verifica flags existentes */
        memset(&options, 0, sizeof(options));
        tcgetattr(parallelfd, &options);

        /* ajustando opcoes de flags para modo raw */
        /* opcoes de output */
        options.c_oflag &= ~OPOST;    /* habilita modo raw */
        /* opcoes locais */
        options.c_lflag &= ~ECHO;     /* desabilita "echo" de caracteres */
        options.c_lflag &= ~ECHONL;   /* desabilita "echo" de newline */
        options.c_lflag &= ~ICANON;   /* habilita modo raw */
        options.c_lflag &= ~ISIG;     /* desabilita sinais */
        options.c_lflag &= ~IEXTEN;   /* desabilita funcoes extras */
        /* opcoes de controle */
        options.c_cflag |= CLOCAL;    /* nao altera o owner */
        options.c_cflag |= CREAD;     /* habilita leitura */
        options.c_cflag &= ~CSIZE;    /* desabilita bit de mascara */
        options.c_cflag &= ~CRTSCTS;  /* desabilita controle de hardware */
        options.c_cflag &= ~PARENB;   /* desabilita bit de paridade */
        /* opcoes de input */
        options.c_iflag &= ~IGNBRK;   /* aceita condicoes de parada  */
        options.c_iflag &= ~BRKINT;   /* nao envia sinais ao condicoes de parada */
        options.c_iflag &= ~PARMRK;   /* ignora erros de paridade */
        options.c_iflag &= ~ISTRIP;
        options.c_iflag &= ~INLCR;    /* mapeia de newline para carriage */
        options.c_iflag &= ~IGNCR;    /* aceita carriage */
        options.c_iflag &= ~ICRNL;    /* permanesce mapeado para carriage */
        options.c_iflag &= ~IXON;    /* desabilita controle de software */

        cfsetispeed(&options, ispeed);
        cfsetospeed(&options, ospeed);

        /* ajustando flags */
        tcsetattr(parallelfd, TCSAFLUSH, &options);
```

```
        return parallelfd;

}


/**---------------------------------------------------------------------------
 * PROCEDURE : fd_close
 *
 * PURPOSE   : Encerra servico de rede, ou de comunicacao por porta serial
 *             ou paralela, ou comunicacao por canal pipe.
 *
 * INPUT     : fd - descritor a ser encerrado
 *
 * OUTPUT    : Retorna o status de encerramento
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *---------------------------------------------------------------------------**/
int fd_close(int fd)
{
        int status;

        /* encerrando descritor */
        if ( (status = close(fd)) < 0)
                perror("fd_close: close()");

        return status;
}

/**---------------------------------------------------------------------------
 * PROCEDURE : _usb_init
 *
 * PURPOSE   : Iniciar um servico para USB
 *
 * INPUT     : Nenhum
 *
 * OUTPUT    : Retorna o numero de dispositivos encontrados
 *
 * ERROS     : Retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int _usb_init(void)
{
        int numdev;
        /* iniciando usb */
        usb_init();

        /* verificacao de bus */
        if (usb_find_busses() < 0) {
                perror("_usb_init: usb_find_busses()");
                return (-1);
        }

        /* verificacao de devices */
        if ( (numdev = usb_find_devices()) < 0) {
                perror("_usb_init: usb_find_devices()");
                return (-1);
        }

        return numdev;
}

/**---------------------------------------------------------------------------
 * PROCEDURE : _usb_close
 *
 * PURPOSE   : Encerrar servico USB
 *
 * INPUT     : udh - apontador para estrutura usb_dev_handle
 *
 * OUTPUT    : Retorna um valor positivo (1)
```

```
 *
 * ERROS     : Retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_close(usb_dev_handle *udh)
{
        if (usb_close(udh) < 0) {
                perror("_usb_close: usb_close()");
                return (-1);
        }

        return (1);
}

/*==================== AQUISICAO/ENVIO DE DADOS ==============================*/

/**-------------------------------------------------------------------------
 * PROCEDURE : network_recv
 *
 * PURPOSE   : Receber dado da rede
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int network_recv(int from_sockfd, char *buffer, int buffer_size)
{

        struct sockaddr_in from_addr;
        socklen_t addrlen;
        int nbytes;

        /* alocando espaco em memoria para buffer */
        if ( (buffer = (char *) malloc(buffer_size)) == NULL) {
                perror("network_recv: malloc()");
                return (-1);
        }

        memset(buffer, 0, buffer_size);
        memset(&from_addr, 0, sizeof(from_addr));

        /* recebendo dados da rede */
        addrlen = sizeof(from_addr);
        if ( (nbytes = recvfrom(from_sockfd,
                                                buffer,
                                                buffer_size,
                                                0,
                                                (struct sockaddr *) &from_addr,
                                                &addrlen
                                                ))< 0)

                perror("network_recv: recvfrom()");

        free(buffer);

        return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : network_send
 *
 * PURPOSE   : Enviar dado para rede
 *
 * INPUT     : to_sockfd   - socket "fonte" do dado recebido
 *             to_sockaddr - apontador para estrutura sockaddr_in carregada com
 *                           o endereco ip e a porta de destino
 *             buffer      - buffer de armazenamento
```

```
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-----------------------------------------------------------------------------**/
int network_send(int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer)
{
       int nbytes;
       /* enviando dados para rede */
       if ( (nbytes = sendto(to_sockfd,
                                              buffer,
                                              strlen(buffer),
                                              0,
                                              (const struct sockaddr *) to_sockaddr,
                                              sizeof(*to_sockaddr)
                                              )) < 0)

              perror("network_send: sendto()");

       /* liberando espaco em memoria */
       free(buffer);

       return nbytes;

}

/**-----------------------------------------------------------------------------
 * PROCEDURE : fd_recv
 *
 * PURPOSE   : Receber dado de um canal de pipe, de uma porta serial ou paralela
 *
 * INPUT     : from_fd     - descritor pipe de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-----------------------------------------------------------------------------**/
int fd_recv(int from_fd, char *buffer, int buffer_size)
{

       int nbytes;


       /* alocando espaco em memoria para buffer */

       if ( (buffer = (char *) malloc(buffer_size)) == NULL) {

              perror("fd_recv: malloc()");

              return (-1);

       }


       /* recebendo dados de pipe */

       if ( (nbytes = read(from_fd, buffer, buffer_size)) < 0)

              perror("fd_recv: read()");

       return nbytes;
}

/**-----------------------------------------------------------------------------
 * PROCEDURE : fd_send
 *
 * PURPOSE   : Enviar dado para um canal de pipe, uma porta serial ou paralela
 *
```

```
 * INPUT     : to_fd  - descritor pipe de destino
 *             buffer - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int fd_send(int to_fd, char *buffer)
{
        int nbytes;

        /* enviando dados para pipe */

        if ( (nbytes = write(to_fd, buffer, strlen(buffer))) < 0)

                perror("pipe_send: write()");


        /* liberando espaco em memoria */

        free(buffer);


        return nbytes;

}

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_recv
 *
 * PURPOSE   : Receber dados de um dispositivo usb
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle
 *             from_endpoint - endpoint do dispositivo fonte
 *             timeout       - tempo limite de aquisicao do dado
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int _usb_recv(usb_dev_handle *from_udh,
                        int from_endpoint,
                        int timeout,
                        char *buffer,
                        int buffer_size
                        )
{

        int nbytes;


        /* alocando espaco em memoria */
        if ( (buffer = (char *) malloc(buffer_size)) == NULL) {
                perror("_usb_recv: malloc()");
                return (-1);
        }

        memset(buffer, 0, buffer_size);

        /* recebendo dado de usb */
        if ( (nbytes = usb_bulk_read(from_udh,
                                                        from_endpoint,
                                                        buffer,
                                                        buffer_size,
                                                        timeout
                                                        )) < 0)
                perror("_usb_recv: usb_bulk_read()");
```

```
	return nbytes;
}

/**---------------------------------------------------------------------------
 * PROCEDURE : _usb_send
 *
 * PURPOSE   : Enviar dados para um dispositivo usb
 *
 * INPUT     : to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo de destino
 *             timeout     - tempo maximo de tentativa de envio
 *             buffer      - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *---------------------------------------------------------------------------**/
int _usb_send(usb_dev_handle *to_udh,
                        int to_endpoint,
                        int timeout,
                        char *buffer
                        )
{
	int nbytes;

	/* enviando dado para usb */
	if ( (nbytes = usb_bulk_write(to_udh,
                                                                to_endpoint,
                                                                buffer,
                                                                strlen(buffer),
                                                                timeout
                                                                )) < 0)
		perror("_usb_send: usb_bulk_write()");

	/* liberando memoria */
	free(buffer);

	return nbytes;
}

/*==================== TRANSFERENCIA DE DADOS ================================*/

/**---------------------------------------------------------------------------
 * PROCEDURE : network_to_network
 *
 * PURPOSE   : Receber dado da rede e enviar para a rede
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_sockfd   - socket de destino do dado a ser enviado
 *             to_sockaddr - apontador para estrutura sockaddr_in de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int network_to_network(int from_sockfd, int to_sockfd,
 struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size)
{
	int nbytes;

	/* recebendo dados da rede */
	if (network_recv(from_sockfd, buffer, buffer_size) < 0)
		return (-1);

	/* enviando dados para rede */
	nbytes = network_send(to_sockfd, to_sockaddr, buffer);

	return nbytes;
```

```
}

/**---------------------------------------------------------------------------
 * PROCEDURE : network_to_fd
 *
 * PURPOSE   : Receber dado da rede e enviar para um canal de pipe, ou uma
 *             porta serial ou paralela.
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_fd       - socket de destino do dado a ser enviado
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int network_to_fd(int from_sockfd, int to_fd, char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados da rede */
        if (network_recv(from_sockfd, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para FD */
        nbytes = fd_send(to_fd, buffer);

        return nbytes;
}

/**---------------------------------------------------------------------------
 * PROCEDURE : network_to_usb
 *
 * PURPOSE   : Receber dado da rede e enviar para dispositivo usb
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo de destino
 *             timeout     - tempo limite de transferencia de dados
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int network_to_usb(int from_sockfd, usb_dev_handle *to_udh, int to_endpoint,
 int timeout, char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados da rede */
        if (network_recv(from_sockfd, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para usb */
        nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);

        return nbytes;
}

/**---------------------------------------------------------------------------
 * PROCEDURE : fd_to_fd
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para um canal de pipe, ou
 *             uma porta serial ou paralela.
 *
 * INPUT     : from_fd     - FD fonte
```

```
 *             to_fd       - FD de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int fd_to_fd(int from_fd, int to_fd, char *buffer, int buffer_size)
{
       int nbytes;

       /* recebendo dados de FD */
       if (fd_recv(from_fd, buffer, buffer_size) < 0)
              return (-1);

       /* enviando dados para FD */
       nbytes = fd_send(to_fd, buffer);

       return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_to_network
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para a rede.
 *
 * INPUT     : from_fd     - FD fonte
 *             to_sockfd   - socket de destino
 *             to_sockaddr - apontador para estrutura sockaddr_in
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int fd_to_network(int from_fd, int to_sockfd, struct sockaddr_in *to_sockaddr,
 char *buffer, int buffer_size)
{
       int nbytes;

       /* recebendo dados de FD */
       if (fd_recv(from_fd, buffer, buffer_size) < 0)
              return (-1);

       /* enviando dados para rede */
       nbytes = network_send(to_sockfd, to_sockaddr, buffer);

       return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_to_usb
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para dispositivo usb.
 *
 * INPUT     : from_fd     - FD fonte
 *             to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo usb de destino
 *             timeout     - tempo maximo de transferencia de dados
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
```

```
int fd_to_usb(int from_fd, usb_dev_handle *to_udh, int to_endpoint, int timeout,
 char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados de FD */
        if (fd_recv(from_fd, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para USB */
        nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);

        return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_usb
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para dispositivo usb.
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_udh        - apontador para estrutura usb_dev_handle destino
 *             to_endpoint   - endpoint do dispositivo usb de destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_usb(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 usb_dev_handle *to_udh, int to_endpoint, char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados de usb */
        if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para usb */
        nbytes = _usb_send(to_udh, to_endpoint, timeout, buffer);

        return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_network
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para a rede.
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_sockfd     - socket de destino
 *             to_sockaddr   - apontador para estrutura sockaddr_in destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_network(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados de usb */
```

```
        if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para rede */
        nbytes = network_send(to_sockfd, to_sockaddr, buffer);

        return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_fd
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para um canal de pipe,
 *             ou uma porta serial ou paralela
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_fd         - FD de destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_fd(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 int to_fd, char *buffer, int buffer_size)
{
        int nbytes;

        /* recebendo dados de usb */
        if (_usb_recv(from_udh, from_endpoint, timeout, buffer, buffer_size) < 0)
                return (-1);

        /* enviando dados para FD */
        nbytes = fd_send(to_fd, buffer);

        return nbytes;
}

/**-------------------------------------------------------------------------
 * PROCEDURE : gmonitor
 *
 * PURPOSE   : Atualizar arquivo que gera um grafico para monitoracao do
 *             comportamento do sistema
 *
 * INPUT     : sockaddr      - apontador para estrutura sockaddr_in
 *             ipaddr        - endereco IP do servidor de monitoracao
 *             port          - porta de servico do servidor de monitoracao
 *             grpaddr       - ponteiro para estrutura ip_mreq
 *             mult_ipaddr   - endereco IP de multicast de servico
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int gmonitor(struct sockaddr_in *sockaddr, char ipaddr[16], int port,
                        struct ip_mreq *grpaddr, char mult_ipaddr[16])
{
        /* iniciando servico de monitoracao
         */
        int pid;
        if ( (pid = fork()) < 0) {
                perror("gmonitor: fork()");
                return(-1);
        }

        /* liberando processo PAI e mantendo processo FILHO dentro de laco para
         * funcao select
```

```
         */
        if (pid != 0) {
                printf("GMonitor() em execucao.\n");
                return (1);
        }

        /* preenchendo o arquivo xml
         */
        char *xml_start = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<items>\n";
        char *xml_front = "\t<item time=\"";
        char *xml_middle = "\" value=\"";
        char *xml_end = "\"/>\n";
        char *xml_finish = "</items>";

        struct xml_values {
                char xv_time[8];
                char xv_value[255];
        } xml_values[10];


        char xml[strlen(xml_start)
                    + strlen(xml_front)*10
                    + strlen(xml_middle)*10
                    + strlen(xml_end)*10
                    + sizeof(xml_values)*10
                    + strlen(xml_finish)
                    ];

        int xml_file;
        char xml_fname[15];

        char received_value[255];
        char received_value_aux[sizeof(received_value)-3];
        int nfds;
        fd_set rdset;

        struct tm TM;
        struct timeval TV;
        memset(&TM, 0, sizeof(TM));
        memset(&TV, 0, sizeof(TV));

        /* criando socket de para receber os dados da rede
         */
        int gmonitor_socket;
        if ( (gmonitor_socket = udp_server_mult_init(sockaddr,
                ipaddr, port, grpaddr, mult_ipaddr)) < 0)
        {
                perror("gmonitor: udp_server_mult_init()");
                return(-1);
        }

        /* iniciando laco para funcao select
         */
        int i;
        for (;;) {
                /* ajustando parametro de leitura de select
                 */
                nfds = 0;
                FD_ZERO(&rdset);
                FD_SET(gmonitor_socket, &rdset);
                nfds = gmonitor_socket + 1;

                /* iniciando funcao select.
                 * Nao ha a necessidade de prever um tempomaximo, pois se nao houver
                 * dados para atualizacao nao devera ser alterado o grafico de
                 * monitoracao, nao havendo a necessidade de se dar refresh na pagina
de
                 * visualizacao do grafico. Neste caso a funcao select esta aqui
somente
```

```
                 * para que seja executadoa atualizacao assim que houver a
necessidade.
                 * Caso contrario o laco ficara em BLOCK/ON HOLD.
                 */
                if (select(nfds, &rdset, NULL, NULL, NULL) < 0) {
                        perror("gmonitor: select()");
/*                      return(-1);
*/              }

                /* ajustando tempo de recebimento do dado
                */
                if (gettimeofday(&TV, NULL) < 0) {
                        perror("gmonitor: gettimeofday()");
/*                      return (-1);
*/              } else {
                        localtime_r(&TV, &TM);
                }

                for (i = 0; i < 9; i++)
                {
                        strcpy(xml_values[i].xv_time, xml_values[i+1].xv_time);
                        strcpy(xml_values[i].xv_value, xml_values[i+1].xv_value);
                }
                sprintf(xml_values[9].xv_time, "%.2d:%.2d:%.2d", TM.tm_hour,
                        TM.tm_min, TM.tm_sec);

                /* efetuando leitura de novos dados
                */
                if (network_recv(gmonitor_socket, &received_value, sizeof(xml)) < 0) {
                        printf("gmonitor: network_recv(): failed to receive data.\n");
/*                      return(-1);
*/              }

                /* para que seja atualizado o arquivo correto,
                 * os 3 primeiros caracteres do valor recebido pelo socket
                 * "gmonitor_socket" devera conter o tipo da estrutura a ser passada.
                 * Por exemplo, se desejamos atualizar o ultimo valor lido pelo
                 * sensor de temperatura T (765 graus) numero 1, entao o dado a
                 * ser recebido sera: T01765
                 * Sendo:
                 * T   - sensor de temperatura
                 * 01  - sensor numero 1
                 * 765 - valor medido
                 *
                 * O arquivo de configuracao que irar gerar o grafico correspondente
                 * tera o formato "sensorXXX.xml". No exemplo acima o arquivo
                 * tera o nome "sensorT01.xml".
                 *
                 * O valor lido podera conter ate 32 caracteres, sendo possivel
                 * sua alteracao em sua declaracao no inicio deste modulo,
                 * conforme a necessidade.
                 */
                strncat(xml_fname, "sensor", 6);
                strncat(xml_fname, received_value, 3);
                strncat(xml_fname, ".xml", 4);

                for (i = 0; i < sizeof(received_value_aux); i++)
                        received_value_aux[i] = received_value[i+3];
                sprintf(xml_values[9].xv_value, "%d", received_value_aux);

                /* formatando arquivo XML */
                strncat(xml, xml_start, strlen(xml_start));

                for (i = 0; i < 10; i++)
                {
                        strncat(xml, xml_front, strlen(xml_front));
                        strncat(xml, xml_values[i].xv_time,
strlen(xml_values[i].xv_time));
                        strncat(xml, xml_middle, strlen(xml_middle));
```

```
                        strncat(xml,
xml_values[i].xv_value,strlen(xml_values[i].xv_value));
                        strncat(xml, xml_end, strlen(xml_end));
                }

                strncat(xml, xml_finish, strlen(xml_finish));

                /* Abrindo arquivo XML para atualizacao de dados
                 * No caso, o arquivo sera criado, caso nao exista no
sistema/diretorio
                 * atual. A flag O_TRUNC certifica de que o arquivo sera limpo antes
de
                 * receber um novo dado/valor para escrita.
                 */
                if ( (xml_file = open(xml_fname, O_CREAT | O_TRUNC | O_WRONLY) ) < 0)
{
                        perror("gmonitor: open()");
/*                      return(-1);
*/              }

                /* atualizando arquivo */
                if (write(xml_file, xml, strlen(xml)) < 0) {
                        perror("gmonitor: write()");
/*                      return(-1);
*/              }

                /* Encerrando atualizacao de dados no arquivo XML */
                if (close(xml_file) < 0) {
                        perror("gmonitor: close()");
/*                      return(-1);
*/              }
        }

        /* encerrando servico de monitoracao de rede */
        if (close(gmonitor) < 0) {
                perror("gmonitor: close()");
                return(-1);
        }

        return(1);
}
```

```
/**-------------------------------------------------------------------------
 * MODULE NAME : inpe.h
 *
 * PROCEDURE   : Header file
 *
 * PURPOSE     : To reduce, simplify and make more clear the codes which
 *               has a similar header file. Another use is to group the
 *               commun used libraries into a single header.
 *
 * DESCRIPTION : In this header file we will put every header and
 *               function calls not only to allow us to reduce the
 *               source code, but to make the source code more
 *               clear and understandable.
 *
 * RELEASE     : May 19th, 2009.
 *
 *               Arthur Adriano Ferreira, UBC, Bolsista PIBIC/CNPq
 *                E-mail: arthuradriano@gmail.com
 *
 *               Prof. Dr. Ulisses Thadeu Vieira Guedes, DMC/INPE, Orientador
 *                E-mail: utvg@dem.inpe.br
 *-------------------------------------------------------------------------**/

/*==================== bibliotecas ==========================================*/

#include <sys/select.h>
#include <sys/stat.h>
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <netdb.h>
#include <usb.h>
#include <sys/ioctl.h>
#include <sys/io.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

#ifdef WIN32
       #include <winsock2.h>
       #include <conio.h>
#endif

#ifdef DOS
       #include <dos.h>
       #include <conio.h>
#endif

/* Prototipo das funcoes */

/**-------------------------------------------------------------------------
 * PROCEDURE : socket_init
 *
 * PURPOSE   : Cria um socket para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : domain   - especificacao do dominio de comunicacao
 *             type     - especificacao do tipo de protocolo
 *             protocol - especificacao do protocolo
 *
```

```
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_init(int domain, int type, int protocol);

/**-------------------------------------------------------------------------
 * PROCEDURE : sockaddr_init
 *
 * PURPOSE   : Carregar estrutura sockaddr_in.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip
 *             port     - porta
 *
 * OUTPUT    : Estrutura sockaddr_in carregada. Os valores nao informados serao
 *             preenchidos com zeros.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int sockaddr_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-------------------------------------------------------------------------
 * PROCEDURE : socket_udp_init
 *
 *
 * PURPOSE   : Cria um socket em UDP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_udp_init(void);

/**-------------------------------------------------------------------------
 * PROCEDURE : socket_tcp_init
 *
 * PURPOSE   : Cria um socket em TCP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_tcp_init(void);

/**-------------------------------------------------------------------------
 * PROCEDURE : socket_icmp_init
 *
 * PURPOSE   : Cria um socket em ICMP para comunicacao e retorna um
 *             descritor.
 *
 * INPUT     : Nenhum.
 *
 * OUTPUT    : Retorna o numero do socket criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int socket_icmp_init(void);

/**-------------------------------------------------------------------------
 * PROCEDURE : tcp_client_init
 *
 * PURPOSE   : Iniciar um servico de rede. Cria um socket para TCP e carrega a
```

```
 *             estrutura cliente com o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - numero do socket para TCP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int tcp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-----------------------------------------------------------------------------
 * PROCEDURE : tcp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para TCP e carrega
 *             a estrutura servidora com endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - Estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para TCP criado.
 *
 * ERRORS    : Em caso de falha na criacao do socket ou no preenchimento
 *             da estrutura contendo o endereco o sistema retorna um
 *             numero negativo (-1).
 *-----------------------------------------------------------------------------**/
int tcp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-----------------------------------------------------------------------------
 * PROCEDURE : accept_init
 *
 * PURPOSE   : Criar um novo socket para a conexao estabelecida
 *
 * INPUT     : sockfd     - socket do servidor a aceitar nova conexao
 *             peeraddr   - ponteiro para estrutura sockaddr_in
 *
 * OUTPUT    : acceptsock - retorna o numero do socket criado.
 *             peeraddr   - estrutura carregada com informacoes do cliente.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int accept_init(int sockfd, struct sockaddr_in *peeraddr);

/**-----------------------------------------------------------------------------
 * PROCEDURE : udp_client_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega a
 *             estrutura contendo o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para UDP criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int udp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-----------------------------------------------------------------------------
 * PROCEDURE : udp_server_init
 *
```

```
 * PURPOSE   : Iniciar servico de rede. Cria um socket para UDP e carrega
 *             a estrutura servidora com endereco IP e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr  - Estrutura carregada. Os campos vazios sao
 *                         preenchidos com zero.
 *             sockfd    - Socket para UDP criado.
 *
 * ERRORS    : Em falha na criacao ou no preenchimento da estrutura que
 *             contem o endereco IP de servico e a porta de destino o
 *             sistema retorna um valor negativo (-1)
 *-----------------------------------------------------------------------------**/
int udp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-----------------------------------------------------------------------------
 * PROCEDURE : udp_server_mult_init
 *
 * PURPOSE   : Iniciar servico de rede. Criar um socket para UDP e carregar a
 *             estrutura contendo o endereco IP de servico em modo multicast
 *             e a porta de destino.
 *
 * INPUT     : sockaddr    - ponteiro para estrutura sockaddr_in
 *             ipaddr      - endereco ip de destino
 *             port        - porta de destino
 *             svaddr      - ponteiro para estrutura sockaddr_in
 *             grpaddr     - ponteiro para estrutura im_req
 *             mult_ipaddr - endereco ip de destino
 *
 * OUTPUT    : sockaddr    - Estrutura carregada. Os campos vazios sao
 *                           preenchidos com zero.
 *             grpaddr     - Estrutura carregada. Os campos vazios sao
 *                           preenchidos com zero.
 *             sockfd      - Socket para UDP multicast criado.
 *
 * ERRORS    : Em falha o sistema retorna um numero negativo (-1)
 *-----------------------------------------------------------------------------**/
int udp_server_mult_init(struct sockaddr_in *sockaddr, char ipaddr[15],
 int port, struct ip_mreq *grpaddr,
  char mult_ipaddr[16]);

/**-----------------------------------------------------------------------------
 * PROCEDURE : udp_server_mult_close
 *
 * PURPOSE   : Encerrar o servico de rede
 *
 * INPUT     : sockfd  - socket a ser encerrado
 *             grpaddr - apontador para estrutura ip_mreq
 *
 * OUTPUT    : Retira sockfd da lista de membros de multicast e fecha o socket.
 *
 * ERRORS    : Em falha o sistema retorna um numero negativo (-1).
 *-----------------------------------------------------------------------------**/
int udp_server_mult_close(int sockfd, struct ip_mreq *grpaddr);

/**-----------------------------------------------------------------------------
 * PROCEDURE : icmp_client_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para ICMP e carrega a
 *             estrutura contendo o endereco ip e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr - estrutura carregada. Os campos vazios sao
 *                        preenchidos com zero.
 *             sockfd   - Socket para UDP criado.
```

```
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int icmp_client_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-------------------------------------------------------------------------
 * PROCEDURE : icmp_server_init
 *
 * PURPOSE   : Iniciar servico de rede. Cria um socket para ICMP e carrega
 *             a estrutura servidora com endereco IP e porta de destino.
 *
 * INPUT     : sockaddr - ponteiro para estrutura sockaddr_in
 *             ipaddr   - endereco ip de destino
 *             port     - porta de destino
 *
 * OUTPUT    : sockaddr  - Estrutura carregada. Os campos vazios sao
 *                         preenchidos com zero.
 *             sockfd    - Socket para UDP criado.
 *
 * ERRORS    : Em falha na criacao ou no preenchimento da estrutura que
 *             contem o endereco IP de servico e a porta de destino o
 *             sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int icmp_server_init(struct sockaddr_in *sockaddr, char ipaddr[16], int port);

/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_init
 *
 * PURPOSE   : Verificar e/ou criar um canal de comunicacao entre processos
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser criado
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_init(char *path);

/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_read_init
 *
 * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para leitura
 *
 * INPUT     : path - caminho/nome do arquivo tipo pipe a ser aberto para
 *                    leitura
 *
 * OUTPUT    : Retorna o numero do pipe criado.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_read_init(char *path);

/**-------------------------------------------------------------------------
 * PROCEDURE : pipe_write_init
 *
 * PURPOSE   : Iniciar um canal de comunicacao (pipe) somente para escrita
 *
 * INPUT     : path - caminho/nome do arquivo do tipo pipe a ser aberto para
 *                    escrita
 *
 * OUTPUT    : Retorna o numero do pipe criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int pipe_write_init(char *path);

/**-------------------------------------------------------------------------
 * PROCEDURE : serial_init
 *
 * PURPOSE   : Iniciar servico de comunicacao com porta serial
```

```
 *
 * INPUT     : serial - caminho da porta serial a ser iniciada
 *             ispeed - velocidade de INPUT
 *             ospeed - velocidade de OUTPUT
 *
 * OUTPUT    : Retorna o numero do serial criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int serial_init(char *serial, int ispeed, int ospeed);

/**---------------------------------------------------------------------------
 * PROCEDURE : parallel_init
 *
 * PURPOSE   : Iniciar servico de comunicacao com porta paralela
 *
 * INPUT     : parallel - caminho da porta paralela a ser iniciada
 *             ispeed   - velocidade de INPUT
 *             ospeed   - velocidade de OUTPUT
 *
 * OUTPUT    : Retorna o numero do serial criado
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int parallel_init(char *parallel, int ispeed, int ospeed);

/**---------------------------------------------------------------------------
 * PROCEDURE : fd_close
 *
 * PURPOSE   : Encerra servico de rede, ou de comunicacao por porta serial
 *             ou paralela, ou comunicacao por canal pipe.
 *
 * INPUT     : fd - descritor a ser encerrado
 *
 * OUTPUT    : Retorna o status de encerramento
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *---------------------------------------------------------------------------**/
int fd_close(int fd);

/**---------------------------------------------------------------------------
 * PROCEDURE : _usb_init
 *
 * PURPOSE   : Iniciar um servico para USB
 *
 * INPUT     : Nenhum
 *
 * OUTPUT    : Retorna o numero de dispositivos encontrados
 *
 * ERROS     : Retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int _usb_init(void);

/**---------------------------------------------------------------------------
 * PROCEDURE : _usb_close
 *
 * PURPOSE   : Encerrar servico USB
 *
 * INPUT     : udh - apontador para estrutura usb_dev_handle
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERROS     : Retorna um valor negativo (-1)
 *---------------------------------------------------------------------------**/
int _usb_close(usb_dev_handle *udh);

/*=================== funcoes de aquisicao|envio de dados ==================*/

/**---------------------------------------------------------------------------
 * PROCEDURE : network_recv
```

```
 *
 * PURPOSE   : Receber dado da rede
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int network_recv(int from_sockfd, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : network_send
 *
 * PURPOSE   : Enviar dado para rede
 *
 * INPUT     : to_sockfd   - socket "fonte" do dado recebido
 *             to_sockaddr - apontador para estrutura sockaddr_in carregada com
 *                           o endereco ip e a porta de destino
 *             buffer      - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int network_send(int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer);

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_recv
 *
 * PURPOSE   : Receber dado de um canal de pipe, de uma porta serial ou paralela
 *
 * INPUT     : from_fd     - descritor pipe de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int fd_recv(int from_fd, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_send
 *
 * PURPOSE   : Enviar dado para um canal de pipe, uma porta serial ou paralela
 *
 * INPUT     : to_fd  - descritor pipe de destino
 *             buffer - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int fd_send(int to_fd, char *buffer);

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_recv
 *
 * PURPOSE   : Receber dados de um dispositivo usb
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle
 *             from_endpoint - endpoint do dispositivo fonte
 *             timeout       - tempo limite de aquisicao do dado
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente recebida.
```

```
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int _usb_recv(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_send
 *
 * PURPOSE   : Enviar dados para um dispositivo usb
 *
 * INPUT     : to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo de destino
 *             timeout     - tempo maximo de tentativa de envio
 *             buffer      - buffer de armazenamento
 *
 * OUTPUT    : Retorna a quantidade de bytes efetivamente enviada.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo
 *-------------------------------------------------------------------------**/
int _usb_send(usb_dev_handle *to_udh, int to_endpoint, int timeout,
 char *buffer);

/*==================== funcoes de transferencia de dados ====================*/

/**-------------------------------------------------------------------------
 * PROCEDURE : network_to_network
 *
 * PURPOSE   : Receber dado da rede e enviar para a rede
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_sockfd   - socket de destino do dado a ser enviado
 *             to_sockaddr - apontador para estrutura sockaddr_in de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int network_to_network(int from_sockfd, int to_sockfd,
 struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : network_to_fd
 *
 * PURPOSE   : Receber dado da rede e enviar para um canal de pipe, ou uma
 *             porta serial ou paralela.
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_fd       - socket de destino do dado a ser enviado
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int network_to_fd(int from_sockfd, int to_fd, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : network_to_usb
 *
 * PURPOSE   : Receber dado da rede e enviar para dispositivo usb
 *
 * INPUT     : from_sockfd - socket "fonte" do dado recebido
 *             to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo de destino
 *             timeout     - tempo limite de transferencia de dados
 *             buffer      - buffer de armazenamento
```

```
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int network_to_usb(int from_sockfd, usb_dev_handle *to_udh, int to_endpoint,
 int timeout, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_to_fd
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para um canal de pipe, ou
 *             uma porta serial ou paralela.
 *
 * INPUT     : from_fd     - FD fonte
 *             to_fd       - FD de destino
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int fd_to_fd(int from_fd, int to_fd, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_to_network
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para a rede.
 *
 * INPUT     : from_fd     - FD fonte
 *             to_sockfd   - socket de destino
 *             to_sockaddr - apontador para estrutura sockaddr_in
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int fd_to_network(int from_fd, int to_sockfd, struct sockaddr_in *to_sockaddr,
 char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : fd_to_usb
 *
 * PURPOSE   : Receber dado de um canal de pipe, ou de uma porta serial
 *             ou de uma porta paralela e enviar para dispositivo usb.
 *
 * INPUT     : from_fd     - FD fonte
 *             to_udh      - apontador para estrutura usb_dev_handle
 *             to_endpoint - endpoint do dispositivo usb de destino
 *             timeout     - tempo maximo de transferencia de dados
 *             buffer      - buffer de armazenamento
 *             buffer_size - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int fd_to_usb(int from_fd, usb_dev_handle *to_udh, int to_endpoint, int timeout,
 char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_usb
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para dispositivo usb.
```

```
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_udh        - apontador para estrutura usb_dev_handle destino
 *             to_endpoint   - endpoint do dispositivo usb de destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_usb(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 usb_dev_handle *to_udh, int to_endpoint, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_network
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para a rede.
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_sockfd     - socket de destino
 *             to_sockaddr   - apontador para estrutura sockaddr_in destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_network(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 int to_sockfd, struct sockaddr_in *to_sockaddr, char *buffer, int buffer_size);

/**-------------------------------------------------------------------------
 * PROCEDURE : _usb_to_fd
 *
 * PURPOSE   : Receber dado de dispositivo usb e enviar para um canal de pipe,
 *             ou uma porta serial ou paralela
 *
 * INPUT     : from_udh      - apontador para estrutura usb_dev_handle fonte
 *             from_endpoint - endpoint do dispositivo usb fonte
 *             timeout       - tempo maximo de transferencia de dados
 *             to_fd         - FD de destino
 *             buffer        - buffer de armazenamento
 *             buffer_size   - tamanho maximo do buffer de leitura
 *
 * OUTPUT    : Retorna a quantidade de bytes enviados.
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int _usb_to_fd(usb_dev_handle *from_udh, int from_endpoint, int timeout,
 int to_fd, char *buffer, int buffer_size);

/*==================== módulo grafico para monitoracao ======================*/

/**-------------------------------------------------------------------------
 * PROCEDURE : gmonitor
 *
 * PURPOSE   : Atualizar arquivo que gera um grafico para monitoracao do
 *             comportamento do sistema. Este sistema alimentara um outro módulo
 *             utilizando Java que exibira um grafico do comportamento do
 *             sistema.
 *
 * INPUT     : sockaddr      - apontador para estrutura sockaddr_in
 *             ipaddr        - endereco IP do servidor de monitoracao
 *             port          - porta de servico do servidor de monitoracao
 *             grpaddr       - ponteiro para estrutura ip_mreq
```

```
 *             mult_ipaddr   - endereco IP de multicast de servico
 *
 * OUTPUT    : Retorna um valor positivo (1)
 *
 * ERRORS    : Em falha o sistema retorna um valor negativo (-1)
 *-------------------------------------------------------------------------**/
int gmonitor(struct sockaddr_in *sockaddr, char ipaddr[16], int port,
 struct ip_mreq *grpaddr, char mult_ipaddr[16]);
```