PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Engenharia de Computação

Jonathan Ferreira Bispo
William Azevedo de Paula

DESENVOLVIMENTO DE DEVICE DRIVER USB PARA LINUX

Belo Horizonte
2012

Jonathan Ferreira Bispo
William Azevedo de Paula

DESENVOLVIMENTO DE DEVICE DRIVER USB PARA LINUX

Trabalho apresentado às disciplinas Sistemas Operacionais e Laboratório de Sistemas Operacionais, do núcleo de informática da Pontifícia Universidade Católica de Minas Gerais.
Orientador: Paulo César Amaral

Belo Horizonte
2012

RESUMO

Este trabalho tem por objetivo abordar os processos necessários ao desenvolvimento de device drivers para sistema operacional Linux, apresentando algumas informações a respeito do kernel do sistema operacional e sobre o desenvolvimento de módulos de kernel carregáveis, nos quais os drivers se baseiam. Um foco maior é dado aos drivers de caractere, principalmente aos driers USB, apresentando as ferramentas disponíveis para o desenvolvimento dos mesmos e as formas utilizadas para interagir com eles. É apresentado também o desenvolvimento de um device driver para comunicação via USB com o dispositivo Arduino, baseado no microcontrolador Atmel328, e de uma biblioteca em C++ para ser utilizada no desenvolvimento de firmwares para o mesmo.

Palavras chave: Device Drivers. Arduino. USB. Módulos carregáveis. Kernel

ABSTRACT

This paper aims to present the necessary process for developing device drivers for Linux Operating System, showing some information about the kernel of the operating system and about development of loadable kernel modules, in which the device drivers are based. A higher focus is given to the character drivers, mainly to USB drivers, showing the available tools for developing them and the forms used to Interact with them. The development of a device driver for communication with the Arduino device (based on Atmega328 microcontroller) and a C++ library to be used in Arduino firmware developing will also be presented.

Keywords: Device Drivers. Arduino. USB. loadable modules. Kernel

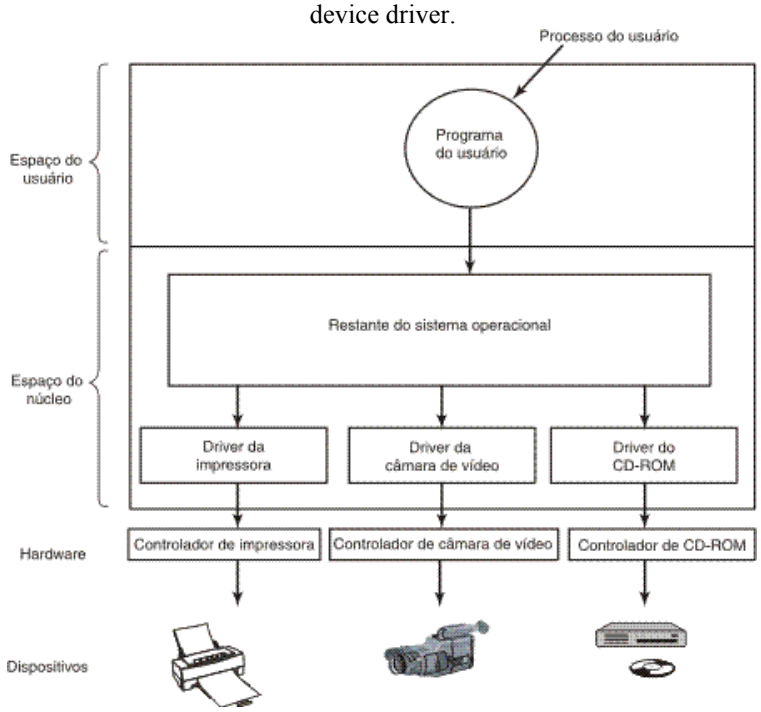
SUMÁRIO

1. INTRODUÇÃO

A comunicação com periféricos é uma parte essencial de um sistema operacional. Nesse contexto, a intermediação entre as chamadas de sistema e os dispositivos é realizada por um software chamado “Device Driver”, abstraindo as operações de baixo nível e tornando as operações no dispositivo mais fáceis de ser realizadas e transparentes ao usuário. Dessa forma, as rotinas necessárias ao correto funcionamento do dispositivo e de interface com o mesmo são controladas exclusivamente pelo device driver.

Esse software é dependente do dispositivo ao qual está relacionado, e do sistema operacional no qual será executado, uma vez que ele engloba as especificidades do dispositivo e as operações para manter a comunicação entre o sistema operacional e o mesmo.

Figura 1: Esquema representando a intermediação entre um processo e os periféricos, por meio de device driver.



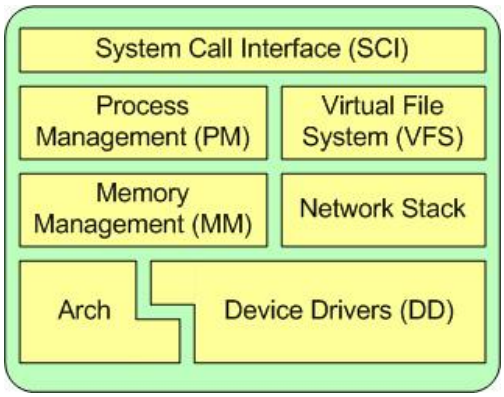
Fonte: SOMTB3ed

Este trabalho irá focar nos drivers de dispositivo para sistemas operacionais Linux, que são baseados em Unix.

2. ARQUITETURA DO KERNEL

O kernel (ou núcleo) do sistema operacional é basicamente o responsável por gerenciar todo o hardware do computador (incluindo periféricos), controlar a execução dos processos e prover uma interface (API) para as aplicações. Ele é dividido em vários "subsistemas", dentre eles uma interface de chamadas de sistema, gerenciamento de memória, device drivers (que é o objetivo de nosso estudo), dentre outras.

Figura 2: Sub Sitemas do Kernel.



Fonte: Sergio Prado

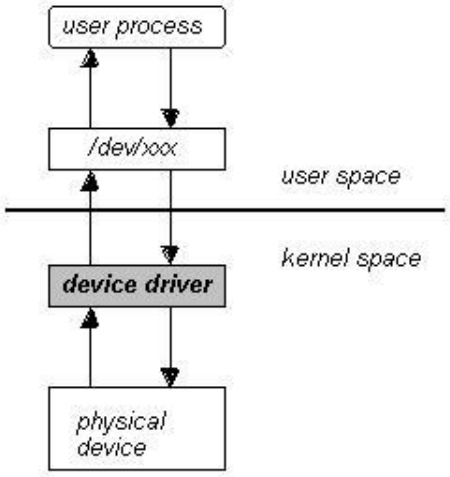
Os subsistemas que compõem o kernel são executados no chamado "kernel space", enquanto os processos pertencentes ao usuário são executados no "user space". Apenas aplicações executadas no kernel space possuem acesso direto ao hardware. Para que uma aplicação em "user space" acesse o hardware, ela precisa se comunicar com o driver do dispositivo (que está em kernel space) para que ele acesse o dispositivo especificado e passe os resultados (se houver) para a aplicação no espaço do usuário.

3. LINUX DEVICE DRIVERS

Nos sistemas baseados em Unix, os drivers de dispositivo são implementados na forma de módulos carregáveis, que são “frações” de código dinamicamente “anexadas” ao kernel, para poderem ser executadas em kernel space. Dessa forma, um determinado driver pode ser carregado no núcleo do sistema operacional apenas quando sua utilização for necessária, para que possa interagir com algum dispositivo, e sem a necessidade de recompilar todo o Kernel.

Para que aplicações no espaço do usuário possam se comunicar com o driver de dispositivo, o mesmo é mapeado em arquivos especiais, localizados no diretório “/dev”. Ao realizar chamadas de sistema sobre esses arquivos (como leitura e/ou escrita) o driver é acionado, realizando a operação mais adequada para a chamada feita.

Figura 3: Mapeamento do Driver em Arquivo.



Fonte: Sergio Prado

Para identificar um dispositivo, o kernel utiliza dois números: o “major number” e o “minor number”. O minor number permite ao núcleo determinar qual device driver está mapeado em uma entrada (arquivo) em /dev que está sendo acessada. Todos os dispositivos que recebam mesmo major number são controlados por um mesmo driver. Já o minor number é utilizado para indicar o número do dispositivo. Por exemplo, se um driver controla N dispositivos, todos os N possuirão o mesmo major number, mas cada um terá o seu minor number.

3.1 Tipos de Dispositivo

Os dispositivos podem ser agrupados em três tipos básicos, de acordo com a maneira em que é realizada a entrada e saída: dispositivos de caractere, dispositivos de bloco e dispositivos de rede. Este trabalho abordará com mais detalhes os device drivers para dispositivos de caractere.

3.1.1. Dispositivos de Caractere

Os dispositivos de caractere (ou Char Drivers) transmitem dados através de fluxos sequenciais de bytes. São utilizados para impressoras, teclados, modems, etc. Geralmente, suportam as operações básicas: abrir, fechar, ler e escrever.

Cada dispositivo de caractere é representado por uma estrutura “device_struct”, que contém o nome do driver e um ponteiro para uma estrutura “file_operations”, a qual define as operações que o dispositivo suporta. Para inicializar um dispositivo de caractere, o driver do mesmo registra as operações no sistema de arquivo virtual, que associa uma estrutura “device_struct” ao vetor de drivers registrados, “chardevs”.

3.1.2. Dispositivos de Bloco

Os dispositivos de bloco (ou block drivers) permitem que dados armazenados em blocos de tamanho fixo de bytes sejam acessados a qualquer momento, de forma aleatória (não sequencial). Para isso, um sistema de controle de entrada e saída empregado deve ser muito mais sofisticado que o utilizado em dispositivos de caractere.

Os maiores exemplos deste tipo de dispositivo são os discos rígidos.

3.1.3. Dispositivos de Rede

Dispositivos que fornecem interface para troca de dados de rede entre hosts (hospedeiros). Uma das principais diferenças do dispositivo de rede para os de bloco ou caractere é que o kernel não solicita dados do mesmo. Em vez disso, o dispositivo de rede utiliza interrupções para notificar o processador à medida que recebe pacotes.

3.2 Drivers USB

O barramento serial universal (USB) permite a conexão entre um computador hospedeiro (host) e diversos periféricos, e foi criado com o objetivo de substituir uma grande variedade de barramentos mais lentos, como as conexões paralela, serial e PS/2. O USB se desenvolveu, e hoje é capaz de suportar uma grande quantidade de dispositivos diferentes e a altas velocidades de transmissão, podendo atingir o limite teórico de 480 MBps.

Uma grande vantagem de um dispositivo USB, é que o mesmo é reconhecido pelo seu device driver no exato momento em que é conectado ao barramento, sendo automaticamente configurado. O USB core determina qual driver associar ao dispositivo através de informações armazenadas no mesmo, geralmente o "Vendor ID" (número de identificação do vendedor) e o "Product ID" (Número de identificação do produto do vendedor), e também por meio da estrutura "usb_device_id", que contém uma listagem das características (geralmente vendor id e product id) dos dispositivos aos quais o driver oferece suporte.

Um dispositivo USB é muito complexo, porém o kernel do Linux possui um subsistema chamado "USB core", que abstrai a maior parte da complexidade. Através da biblioteca "linux/usb.h" é possível interagir com tais dispositivos, sem necessidade de manipular o protocolo USB diretamente.

3.2.1 Endpoints

As comunicações USB se baseiam nos "endpoints", que podem ser entendidos como canais de comunicação, por onde os dados são enviados, tanto do hospedeiro para os periféricos (Endpoints de saída) quanto no sentido contrário (Endpoints de entrada). O linux implementa os endpoints por meio da estrutura "USB_host_endpoint".

Existem quatro tipo de endpoint:

3.3.1.1. Controle (Control)

São utilizados para permitir acesso a diferentes partes de um dispositivo. São normalmente utilizados para configurar um dispositivo, obter informações sobre o mesmo, enviar comandos, verificar status do dispositivo, etc. Todo dispositivo USB possui pelo menos um endpoint de controle, o "endpoint zero", utilizado pelo USB core para configurar o dispositivo no momento em que ele é inserido no barramento.

3.3.1.2. Interrupção (Interrupt)

Utilizados para transferir pequenas quantidades de dados em tamanhos fixos, cada vez que o hospedeiro (host) solicita os mesmos.

3.3.1.3. Massa (Bulk)

São utilizados para transferir grandes quantidades de dados. São maiores que os endpoints de interrupção e são utilizados quando quantidades de dados não fixas precisam ser enviadas sem tolerância a perda de dados.

3.3.1.4. Isocrono (Isochronous)

São utilizados para transferir grandes quantidades de dados em intervalos fixos de tempo (periódicos), com tolerância a pequenas perdas de dados, desde que a transferência ocorra no tempo especificado. É utilizado em processamentos de tempo real, como em dispositivos de áudio e vídeo.

3.2.2. Interfaces

Os endpoints são encapsulados em “interfaces”. Cada interface suporta apenas um tipo de conexão USB lógica, como um teclado, um mouse, etc. Mas os dispositivos USB podem possuir várias interfaces. Um exemplo é uma caixa de som, a qual pode possuir duas interfaces: Um painel com botões e controles, e um auto falante. Cada driver controla apenas uma interface, portanto dispositivos com múltiplas interfaces necessitam de múltiplos drivers para operar corretamente.

O Linux implementa interfaces por meio da struct “usb_host_interface”.

3.2.3. Configurações

As interfaces, por sua vez, são encapsuladas em “configurações”. Um dispositivo pode possuir várias configurações, podendo alternar entre elas. Porém, apenas uma configuração estará ativa por vez.

Linux descreve as configurações por meio da struct “usb_host_config”.

3.2.4. URBs (USB Request Blocks)

O kernel do Linux se comunica com todos os dispositivos USB utilizando uma abstração chamada URB (USB Request Block), a qual é utilizada para enviar e/ou receber dados de/para um endpoint específico de um dispositivo USB específico, de maneira assíncrona. Em qualquer transferência de dados, deve-se alocar ao menos uma URB e enviá-la ao endpoint desejado.

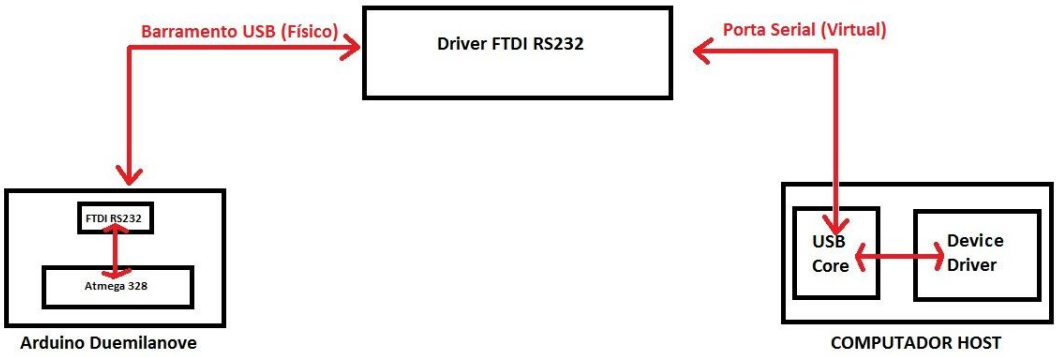
4. DESENVOLVIMENTO DE UM DRIVER USB PARA ARDUINO

O objetivo deste trabalho é desenvolver um device driver de caractere para permitir a comunicação entre um computador e um kit de desenvolvimento Arduino, através do barramento USB.

O Arduino é uma plataforma baseada no microcontrolador Atmega238, o qual não possui controlador USB interno, sendo um microcontrolador com suporte apenas a transmissões seriais. Em vez disso, a placa do Arduino possui um conversor USB-Serial, o FTDI RS232, que recebe dados via USB, armazena em um buffer, e por último transfere os mesmos serialmente para o pino rx do microcontrolador. Ao enviar dados do dispositivo para o computador, o princípio adotado é semelhante: Os dados são transmitidos serialmente do pino tx do Atmega para o RS232, o qual vai armazenando-os em um buffer, para depois enviá-los através da porta USB.

Os device drivers desenvolvidos para esta plataforma, normalmente, não se comunicam diretamente com o barramento USB. Em vez disso, eles enviam os dados serialmente para uma porta serial virtual, gerada pelo driver do conversor RS232, enxergando assim o Arduino como se fosse uma porta serial. Esse processo é exemplificado na figura 4:

Figura 4: Diagrama da comunicação entre Arduino e o Host.



Fonte: Autores

A proposta para o trabalho é desenvolver um device driver que não necessite do Driver do FTDI RS232 e nem da porta Serial Virtual, pois esse método deixa o sistema submetido às velocidades de transferência do protocolo de transferência serial, que é lento. Dessa forma, o novo driver permitirá que o USB core acesse diretamente a porta USB, sem mapeá-la em porta serial virtual, permitindo velocidades de transferência muito maiores, pois não possuem o atraso proposital que o método anterior possuía para torna-lo compatível com o padrão serial. O diagrama de representação da comunicação passa a ser o diagrama da figura 5:

Figura 5: Diagrama da comunicação entre Arduino e o Host com o novo driver.

Fonte: Autores

Outra vantagem é que através de drivers USB, pode-se saber instantaneamente se existem dispositivos conectados ao host, simplesmente consultando o diretório "/dev", ou o arquivo "/proc/devices".

Abordaremos então os principais elementos que compõem um driver USB de caractere para Linux. Essa implementação divide-se em três partes: A primeira é o driver propriamente dito, chamado "Arduino_Driver" (implementado no arquivo Arduino.c), o qual é carregado dinamicamente no kernel utilizando o comando "insmod", no Shell do Linux; uma biblioteca para aplicações no Arduino (chamada "Arduino_USB"), a qual pode ser usada para qualquer firmare que deseja comunicar com o computador host através do Arduino_Driver; e por último um firmware simples para testar a biblioteca Arduino_USB e o device driver Arduino_Driver, utilizando display de LCD para permitir a visualização de dados transferidos.

4.1. O Driver

Como foi dito anteriormente, um device driver para Linux é um módulo de kernel carregável. Por isso, além de "linux/usb.h" também são utilizadas no projeto as bibliotecas "linux/module.h" e "linux/kernel.h", dentre outras. Com o auxílio dessas bibliotecas, um device driver USB para Linux pode ser facilmente implementado. Os trechos principais do código fonte do driver serão explicados a seguir:

4.1.1. struct usb_device_id

Esta estrutura é responsável por definir as características que um dispositivo deve possuir para que o driver seja atribuído a ele. Ao conectar um dispositivo à porta USB, o USB core irá procurar por um driver que possua uma entrada em "usb_device_id" que corresponda

às características do dispositivo, e em seguida associa o dispositivo ao driver encontrado (se encontrar).

Em Arduino_Driver (o device driver desenvolvido para o Arduino), como na maioria dos device drivers, foram adotados o "vendor ID" e o "product ID" do conversor usb-serial presente na placa do Arduino. Dessa forma, embora na prática estejamos utilizando um device driver para Arduino, na verdade o Driver é do conversor usb-serial. Como mostra a figura 6.

Figura 6: Definição do Vendor ID e do Product ID, e declaração da estrutura "usb_device_id"

```
#define VENDOR_ID      0x0403   //Future Technonoly Devices International
#define PRODUCT_ID     0x6001   //conversor USB-Serial RS232 (utilizado no Arduino)



//============================================================================

//Tabela de dispositivos aceitos pelo Driver
static struct usb_device_id Arduino_Id[] = {
    {USB_DEVICE(VENDOR_ID , PRODUCT_ID)},
    { }
};
```

Fonte: Autores

4.1.2. struct usb_driver

Esta é uma estrutura que representa um driver USB. Ela contém o nome do driver, uma strutura "usb_device_id", e *callbacks* para vários métodos, sendo ".probe" e ".disconnect" os mais importantes (serão discutidos mais a frente). Nem todos os campos dessa estrutura precisam se inicializado. A figura 7 apresenta a "usb_driver" utilizada.

Figura 7: estrutura usb_driver

```
//Estrutura que define o driver
static struct usb_driver Arduino_Driver = {
   .owner      = THIS_MODULE,
   .name       = "Arduino_Driver",
   .id_table   = Arduino_Id,
   .probe      = Arduino_Probe,
   .disconnect = Arduino_Disconnect,
};
```

Fonte: Autores

O valor passado para ".probe" é um *callback* para o método chamado quando um dispositivo que atenda às especificações de "usb_device_id" é conectado ao computador e associado ao driver. Neste procedimento são feitas as inicializações necessárias ao funcionamento do driver, como identificação dos endpoints e alocação de memória para os

buffers de entrada/saída, utilizando a função “kzalloc” (semelhante a “malloc”, porém para ser executada no kernel space).

Figura 8: protótipo do método probe

```
static int Arduino_Probe(struct usb_interface *interface, const struct usb_device_id *id)
```

Fonte: autores

Por fim, o método probe registra o dispositivo junto ao USB core, utilizando a função “usb_register_dev”, obtendo assim o “minor number do dispositivo”.

O valor passado para “.disconnect” é um *callback* para o método chamado quando o dispositivo é desconectado do host. A principal utilidade deste método é a remoção do registro do dispositivo junto ao USB core, devolvendo assim o minor number, isto é, tornando-o disponível para outro dispositivo que vier a ser conectado.

Figura 9: protótipo do método disconnect

```
static void Arduino_Disconnect(struct usb_interface *interface)
```

Fonte: Autores

4.1.3. Os métodos “__init” e “__exit”

Um módulo de kernel necessita de um método de inicialização, que é chamado quando o mesmo é carregado. É nesse método que é realizado o registro do driver junto ao USB core, através da função “usb_register”. Como um device driver no Linux é na verdade um módulo de kernel carregado dinamicamente, esta regra vale para ele também.

O método de inicialização é definido acrescentando-se o símbolo “__init” à frente do nome do método, e pela utilização da macro “module_init(x)”, onde x é o nome do método de inicialização.

Além de um método de inicialização, é necessário também um método de “finalização”, o qual é chamado quando o módulo (o driver, no caso) é removido (descarregado) do kernel. É neste método que é realizado o “desregistro” do módulo junto ao kernel, utilizando a função “usb_deregister”.

O método de inicialização é definido acrescentando-se o símbolo “__exit” à frente do nome do método, e pela utilização da macro “module_exit(x)”, onde x é o nome do método de finalização.

O método “printk”, constantemente utilizado, possui funcionamento semelhante ao “printf”, com a diferença que as mensagens não são direcionadas ao terminal de saída, mas

sim ao kernel, sendo salvas em um arquivo de log que pode ser visualizado com o comando "dmesg". Esse método pode ser utilizado para depuração do driver (gerando mensagens constantes sobre o status do mesmo em regiões chave do código) e para gerar um log de funcionamento do driver.

Figura 10: Definição dos métodos de inicialização e finalização do driver

```
static int __init Usb_Arduino_init(void){
    int result = 0;

    //Registra este drive com o USB core
    result = usb_register(&Arduino_Driver);
    if(result){
       printk(KERN_INFO "[EERRO] Falha ao registrar o Arduino_Driver. Erro numero %d",result);
    }
    else{
       printk(KERN_INFO "Arduino_Driver registrado com sucesso");
    }
    return result;
}

static void __exit Usb_Arduino_Exit(void){
    //Desregistra o driver do USB Core
  usb_deregister(&Arduino_Driver );
  printk(KERN_INFO "Arduino_Driver Desregistrado\n");
}


module_init(Usb_Arduino_init);
module_exit(Usb_Arduino_Exit);
```

Finte: Autores

4.1.4. struct file_operations

Esta estrutura serve para definir as operações suportadas pelo dispositivo. As operações mais comuns são: "Abrir", "Ler" e "escrever".

Figura 11: estrutura "file_operations"

```
//Operações suportadas pelo dispositivo
static const struct file_operations Arduino_Fops = {
    .owner      = THIS_MODULE,
    .open       = Arduino_Open,
    .read       = Arduino_Read,
    .write      = Arduino_Write,
};
```

Fonte: Autores

Os métodos "open", "read" e "write" se inicializado em "file_operations", são chamados toda vez que uma operação de abertura, leitura ou escrita, respectivamente, é realizada sobre o arquivo de caractere no qual o dispositivo está mapeado, em "/dev".

Em “open”, são inicializadas as informações sobre o arquivo, como buffer, endpoints o/ou urbs, para que as demais operações possam ser realizadas. Em “read”, o driver solicita dados do dispositivo, através do método “usb_bulk_msg()”, e em seguida copia o dado lido no kernel space para o user space, utilizando o método “copy_to_user()”. Em “write”, o driver envia um dado para o dispositivo, também através do método “usb_bulk_msg()”, obtido no user space (o dado que o usuário tentou escrever no arquivo de caractere em “/dev”) através do método “copy_from_user()”.

4.1.5. struct usb_class_driver

Esta estrutura representa a classe do dispositivo USB. As três propriedades mais importantes são “name”, que corresponde ao nome do arquivo que será gerado no diretório “/dev”, seguido de um “%d”, que é onde será concatenado o minor number do dispositivo, permitindo gerar um arquivo para cada dispositivo conectado; “fops”, que é uma estrutura “file_operations”; e “minor_base” que é a “base” para determinação dos minor numbers, isto é, o valor a partir do qual se atribuem os minor numbers. Por exemplo, se utilizarmos name = “Arduino%d” e minor_base = 0, o primeiro Arduino conectado ao computador irá receber o minor number 0; o segundo receberá o minor number 1; o terceiro receberá minor number igual a 2, e assim por diante. Da mesma forma, para os três primeiros serão gerados os seguintes arquivos em “/dev”: “Arduino0”, “Arduino1” e “Arduino2”.

A figura 12 ilustra a situação apresentada, onde USB_ARDUINO_MINOR_BASE é uma definição para a minor base desejada. No driver desenvolvido, foi utilizado minor base igual a zero.

Figura 12: estrutura “usb_class_driver”

```
//Classe do dispositivo, para poder registrá-lo no USB-Core
static struct usb_class_driver Arduino_Class = {
    .name       = "Arduino%d",
    .fops       = &Arduino_Fops,
    .minor_base = USB_ARDUINO_MINOR_BASE,
};
```

Fonte: Autores

O método “write” comunica normalmente com o conversor usb-serial presente na placa Arduino. Porém, a transmissão de dados do conversor para o microcontrolador Atmega328 é feita de forma serial. O problema é que, sem utilizar a porta serial virtual e o driver do FTDI RS232, a transferência não obedece aos *baund rates* padrões de uma

comunicação serial, e não foi encontrada nenhuma documentação sobre a velocidade de transmissão sem a utilização do referido driver. Para contornar esse problema, testes foram realizados e constatou-se que o período entre a transferência de cada bit de um dado é de, em média, 4 ciclos de clock do Arduino. Uma biblioteca foi desenvolvida, para ser utilizada nos firmwares do Arduino, permitindo ao Atmega328 se comunicar com o FTDI RS232 dessa maneira.

A comunicação serial necessita de, além dos oito bits transferidos (no caso da transferência de um byte), de outros dois bits: Um para marcar o início da transmissão e outro de paridade, para facilitar detecção de erros. O problema é que os métodos da biblioteca "linux/usb.h" não permitem a transferência de bits isolados, mas apenas de bytes inteiros. Portanto, para atender ao protocolo, dois bytes devem ser enviados: um byte fixo, terminado em 0 (valor que marca o início da transferência) e um byte com o dado transferido. O bit de paridade não foi implementado neste trabalho, ficando como proposta de aperfeiçoamento furo.

Para fornecer mais possibilidades de transferência o Arduino_Driver foi codificado de forma a prosseguir da seguinte maneira, quando o usuário tentar escrever no arquivo de caractere (ou pseudo-arquivo): Verifica os três primeiros caracteres. Se forem 'byt', os caracteres seguintes são transferidos na forma de um byte em uma única transferência (funciona apenas para valores de 0 a 255); se forem 'str', os caracteres seguintes serão considerados uma única string, e serão transmitidos um por vez (um byte para cada caractere). Por exemplo, se o usuário tentar escrever "byt49" no arquivo de caractere, o Atmega receberá o número 49, podendo interpretá-lo como o caractere '1' (valor ASCII); por outro lado, se o usuário tentar escrever "str49", os caracteres '4' e '9' serão recebidos (ou a string "49"). Se os três primeiros caracteres não forem os especificados, nada é feito.

Quanto ao método "read", para ser codificado, foi observado o seguinte comportamento, por meio de testes: O FTDI RS232 envia constantemente dois bytes através do barramento USB. Um byte é sempre o valor 255, enquanto o segundo byte varia de acordo com o nível lógico no pino tx (pino de transferência) do Atmega: Quando o pino está em nível '0' (baixo), o valor 112 é transmitido. Quando está em nível '1' (Alto), o valor 96 é transmitido. Portanto, a leitura é feita da seguinte forma, utilizando um método simplificado de *handshaking*: Para cada um dos oito bits necessários (leitura de um byte), o Arduino_Driver primeiramente envia um byte ao Arduino, solicitando-o que envie o próximo bit. O Atmega328 coloca então o próximo bit no pino tx. O Arduino_Driver recebe dois bytes do Arduino e, dependendo do valor do segundo byte (112 ou 96) determina se é um bit 0 ou 1.

Este é um método não eficiente, uma vez que o overhead gerado nas “trocas de mensagens” é muito alto (maior que a transmissão efetiva do dado), tornando a transferência muito lenta. Uma sugestão de aperfeiçoamento futuro é a realização de testes para tentar encontrar um padrão de transferência semelhante ao encontrado na transferência de dados do driver para o Arduino.

4.2. A Biblioteca para Atmega328

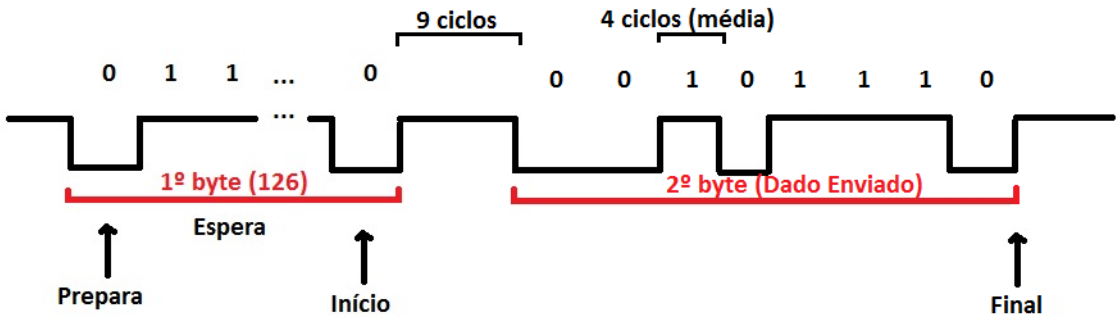
A arquivo de cabeçalho “Arduino_USB.h” foi desenvolvido para permitir o fácil desenvolvimento de firmwares para o Arduino que sejam capazes de se comunicar com o computador hospedeiro (host) sem a necessidade de uma porta serial virtual e nem do driver do conversor usb-serial FTDI RS232. Para isso, a biblioteca implementa um método para inicialização do firmware, o “Usb_Setup”, responsável apenas por inicializar as variáveis e estruturas necessárias ao funcionamento da biblioteca; além de três métodos básicos: “Read_Byte_from_USB”, “Read_String_From_USB” e “Write_To_USB”.

O método “Read_Byte_from_USB” tem por objetivo retornar um valor inteiro correspondente a um byte lido via USB. Como foi dito anteriormente, os dados que chegam via USB à placa do Arduino são transmitidos serialmente ao Atmega328, através do pino rx (pino receptor). Para que um byte possa ser lido, dois bytes devem ser transferidos: o primeiro com valor fixo igual a 126 (01111110, em binário), e o segundo com o dado a ser transmitido.

O pino rx do Atmega328 fica sempre com valor lógico alto. Quando “Read_Byte_from_USB” é chamada, ela fica em loop esperando que um valor zero apareça no pino rx. Em seguida, ela entra em loop novamente, aguardando uma segunda leitura igual a zero, a qual marca o início da transmissão do dado. Portanto, quando o valor 126 é transmitido, a função irá reconhecer os dois zeros (inicial e final) do mesmo, e em seguida ler os oito bits que compõem o dado a ser lido, a intervalos de, em média, 4 ciclos de clock do Atmega. Esses atrasos são tão pequenos (porém necessários) que os métodos de atraso disponíveis nas bibliotecas C++ do Arduino não possuem precisão para gerá-los. Para solucionar esse problema, os atrasos foram gerados utilizando-se instruções NOP, em Assembly, uma vez que cada uma dessas instruções corresponde a um atraso de um ciclo de clock do Atmega. Outro detalhe importante é que o método em C++ para leitura dos pinos (“digitalWrite()”) possui um overhead muito grande (da ordem de milissegundos, enquanto ciclos de clock do Atmega são da ordem de nanosegundos) e não conseguem fazer a leitura a tempo. Para contornar esse problema, a leitura dos valores deve ser feita diretamente no

registrador PORTD do Atmega, processo que quase não gera overhead. À medida que os bits são lidos, uma operação de “OR bit a bit” é feita, concatenando os mesmos e gerando um byte (8 bits) no final, que é retornado pela função na forma de um inteiro.

Figura 13: Trasmissão de um byte para o Arduino



Fonte: Autores

O método “Read_String_From_USB” lê uma string de tamanho máximo definido por “BUFFER_SIZE” (idealmente 64) e retorna o endereço da mesma. O funcionamento do método é simples: Ele faz sucessivas chamadas a “Read_Byte_from_USB” e vai salvando os bytes retornados em um vetor, até atingir o tamanho máximo ou até encontrar a marca de final da string, que é o byte 10. Por fim, retorna o endereço do vetor.

Quanto ao método “Write_To_USB”, ele recebe como parâmetro um número inteiro, que é o byte a ser enviado. Então, para cada um dos oito bits que compõem o inteiro, faz os seguintes passos: Fica em loop chamando o método “Read_Byte_from_USB”, e permanece em loop até que receba o número correspondente ao “índice” do próximo bit (1 para o primeiro bit; 2 para o segundo; 3 para o terceiro; ect.); então ele escreve no pino tx o valor do bit correspondente. O método de escrita da biblioteca C++ do Arduino, “digitalWrite”, possui um overhead muito grande também, levando ao seguinte problema: Após enviar o bit solicitando o envio, o Arduino_Driver lê o próximo bit, porém devido ao overhead da função “digitalWrite”, ele lê o dado que estava no pino tx antes de a função mudá-lo para o valor correto. Para evitar esse problema, a manipulação direta do registrador PORTD do Atmega328 é utilizada no lugar de “digitalWrite”, realizando a mudança quase instantânea do valor do pino, o que diminuiu consideravelmente a taxa de erros de transmissão.

Foi incluído na biblioteca um suporte a *debugging* utilizando um display de LCD. Se a macro “ACTIVE_DEBUG” for definida, seja na própria biblioteca ou no código do firmware (antes de incluir a biblioteca), o Atmega328 irá escrever no display de LCD, na linha inferior, todos os oito bits lidos, e La linha superior o valor decimal referente, durante a execução do

método “Read_Byte_from_USB”. Este recurso deve estar ativo apenas para depuração, devendo ficar desabilitado nas aplicações.

Figura 14: código para depuração, apenas quando ACTIVE_DEBUG estiver definido.

```
#ifdef ACTIVE_DEBUG
    lcd.print(val);
    lcd.setCursor(0, 1);
    lcd.print((val & B10000000) >> 7);
    lcd.print((val & B01000000) >> 6);
    lcd.print((val & B00100000) >> 5);
    lcd.print((val & B00010000) >> 4);
    lcd.print((val & B00001000) >> 3);
    lcd.print((val & B00000100) >> 2);
    lcd.print((val & B00000010) >> 1);
    lcd.print((val & B00000001) >> 0);
#endif
```

Fonte: Autores

4.3. O firmware

Um firmware bem simples foi desenvolvido para permitir a demonstração do driver e da biblioteca. Para tal, o Arduino precisa estar conectado a um display de LCD 16x2, de acordo com a pinagem informada na biblioteca “LiquidCrystal.h”. O mesmo realiza a tarefa básica de, primeiramente, esperar pelo envio de uma string via USB. Após receber a mesma, o dispositivo escreve a mesma no display e entra no “modo de escrita”, acendendo o LED conectado ao pino 13 do Atmega328 para indicar essa situação. A partir de então, o Arduino irá enviar bytes pela porta USB, começando com o valor ‘65’, que corresponde ao caractere ‘A’ (maiúsculo) na tabela Unicode. Cada vez que o Arduino_Driver “buscar” um dado (O dado somente é efetivamente enviado quando o usuário efetuar a leitura no arquivo de caractere em “/dev”), o Atmega328 irá incrementar o mesmo por 1, indo para o próximo caractere (irá percorrer todo o alfabeto maiúsculo, e mais alguns caracteres).

Para permitir tal funcionamento, primeiramente a biblioteca “Arduino_USB.h” deve ser copiada para o diretório “libraries/Arduino_USB” (A pasta deve ser criada dentro de “libaries”), dentro da pasta onde está o software de desenvolvimento “Arduino alpha”, juntamente com o arquivo “keywords.txt” (Arquivo com definição das palavras chave da biblioteca, permitindo ao editor do Arduino destacar tais palavras). Em seguida, ao criar um novo arquivo fonte, deve-se realizar a inclusão da mesma, uilizando “#include<Arduino_USB.h>”, ou acessar o menu “Sketch/Import Library.../ Arduino_USB”.

Feito isso, o firmware pode acessar todos os métodos definidos na biblioteca, para poder interagir com Arduino_Driver via USB.

5. MONTAGEM E REALIZAÇÃO DE TESTES

Três etapas serão apresentadas para montar o dispositivo e realizar na prática os testes do funcionamento do Arduino_Driver:

5.1. Compilação e instalação do Driver

Para realização dos testes, a primeira coisa a se fazer é compilar driver, que está codificado no arquivo "Arduino.c". O arquivo "Makefile", o qual possui os comandos de compilação do driver e inclusão das dependências, deve estar na mesma pasta. Para compilar, basta acessar o diretório pelo terminal do Linux e digitar o comando (o aplicativo "make" deve estar instalado):

```
$ make
```

Ao fazer isso, o aplicativo "make" irá acessar o makefile e compilar o código do driver de acordo com as instruções, gerando vários arquivos de saída, dentre eles o "Arduino.ko", que é o módulo de kernel carregável.

Para carregar o módulo, deve-se utilizar o comando (estando posicionado no mesmo diretório que o arquivo ".ko"):

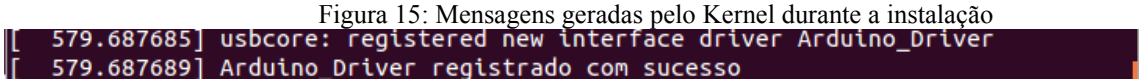
```
$ insmod Arduino.ko
```

Para verificar se o Driver foi corretamente carregado, deve-se digitar o comando abaixo, para exibir as mensagens geradas no kernel, para assim saber o status do carregamento:

```
$ dmesg
```

Fito isso, as duas mensagens indicadas na figura 15 devem aparecer. A primeira é lançada pelo próprio USB core, indicando que um novo driver USB foi instalado. A segunda, é lançada Arduino_Driver, no método "__init", para indicar que a instalação foi realizada com sucesso.

Figura 15: Mensagens geradas pelo Kernel durante a instalação



Fonte: Autores

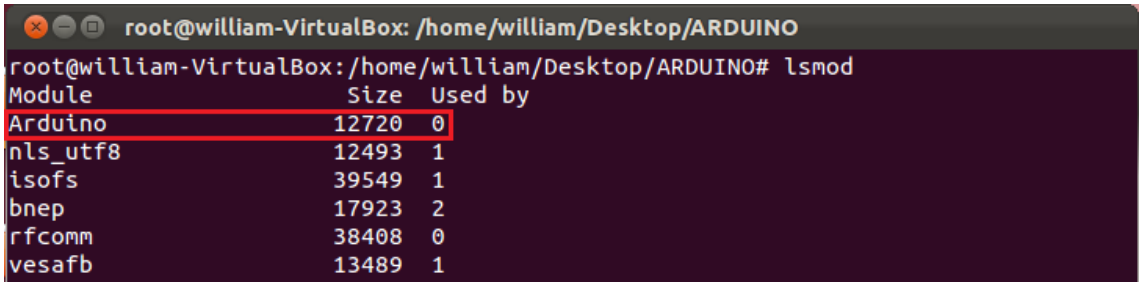
Outra forma de verificar se o driver foi instalado corretamente é listar os módulos que estão instalados no kernel, e verificar se "Arduino" (nome do arquivo ".ko") está entre eles. Para isso, deve-se aplicar o comando:

```
$ lsmod
```

Ou então pode-se utilizar o comando abaixo para filtrar o Arduino dentre os módulos carregados:

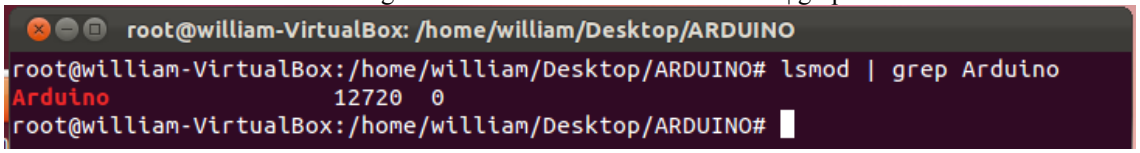
```
$ lsmod | grep Arduino
```

Figura 16: listagem de alguns módulos carregados. "Arduino" está na lista, indicando que a carga ocorreu com sucesso.



Fonte: Autores

Figura 17: Resultado utilizando "lsmod | grep Arduino"



Fonte: Autores

Caso seja necessário remover o driver do kernel, deve-se utilizar o comando abaixo, para efetuar a desinstalação do modulo:

```
$ rmmod Arduino.ko
```

Ao fazer isso, o método “__exit” do driver é chamado, desregistrando o driver juto ao USB Core. Dessa forma, outras duas mensagens são geradas no kernel (Figura 18), podendo ser visualizadas através do comando “dmesg”. A primeira mensagem é lançada pelo USB core, indicando que o driver USB foi removido (desregistrado). A segunda mensagem foi lançada pelo próprio Arduino_Driver, após chamar o método “usb_deregister”.

Figura 18: Mensagens geradas pelo Kernel durante a remoção

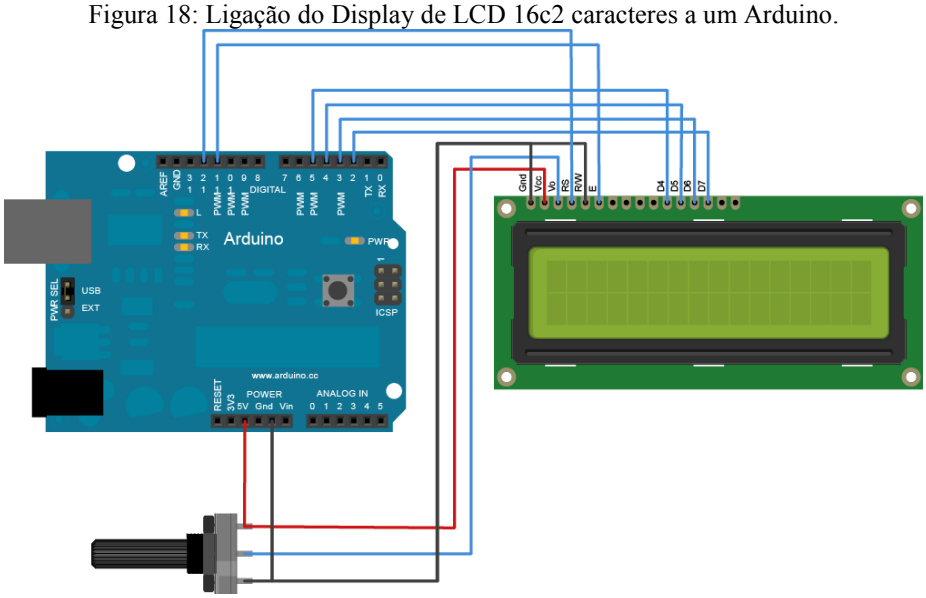
```
[ 2458.269091] usbcore: deregistering interface driver Arduino_Driver
[ 2458.269889] Arduino_Driver Desregistrado
```

Fonte: Autores

5.2. Montagem e gravação do dispositivo

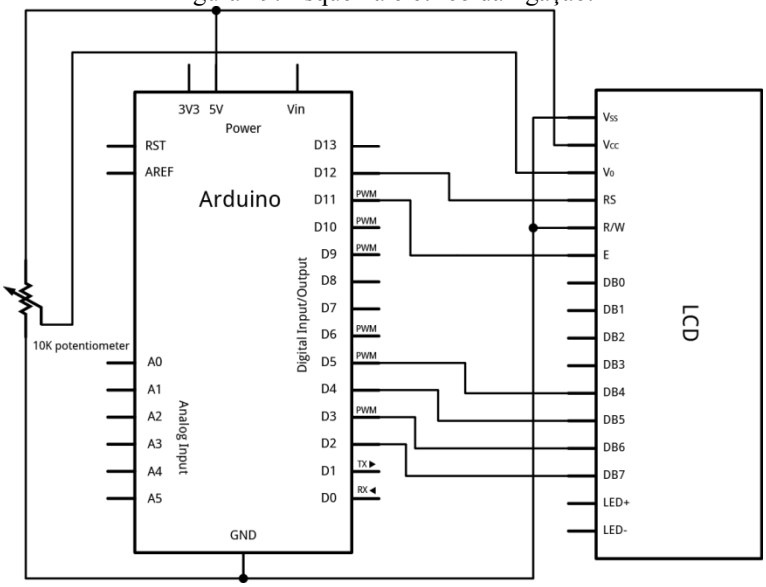
O próximo passo a realizar é conectar um display de LCD 16x2 caracteres ao Arduino, para que este possa exibir os dados recebidos do usuário. A montagem deve ser feita respeitando a pinagem exigida pela biblioteca “LiquidCrystal.h”, do arduino. O esquema elétrico indicando a montagem é exibido a seguir:

Figura 18: Ligação do Display de LCD 16c2 caracteres a um Arduino.



Fonte: ARDUINO. Acesso em 7 jun. 2012

Figura 19: Esquema elétrico da ligação.



Fonte: ARDUINO, acesso em 7 jun. 2012

Para realização dos testes, duas montagens semelhantes foram feitas, isto é, dois Arduinos modelo *Duemilanove* e dois displays de LCD 16x2 foram utilizados.

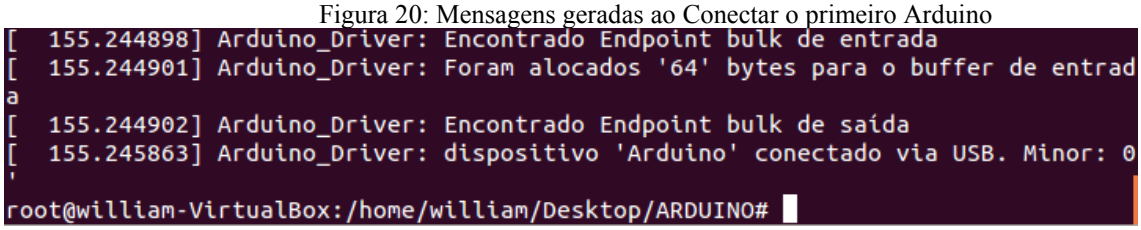
Em seguida, o firmware desenvolvido (“Firmware_Arduino_USB.pde”) deve ser gravado nos dois dispositivos, utilizando cabos USB e o software de desenvolvimento e gravação “Arduino alpha”.

5.3. Conectando e desconectando o(s) dispositivo(s) ao Computador

Após realizar a montagem e a gravação de ambos os dispositivos, estando o Arduino_Driver carregado no kernel, basta conectar um dos Arduinos a uma porta USB do computador. É muito importante que o Arduino não seja conectado antes que o Arduino_Driver esteja carregado, caso contrário o kernel pode atribuir outro driver ao mesmo, o qual passará a ter precedência sobre Arduino_Driver, tornando necessário reiniciar o Sistema Operacional para que este possa ser usado novamente.

Após conectar o dispositivo, o USB core imediatamente irá identificar o Arduino (Na verdade ele detecta o FTDI RS232 conectado à placa do Arduino) e o atribui ao Arduino_Driver. O driver então executará o método “Arduino_Probe”. Algumas mensagens serão geradas no kernel pelo driver, indicando que o dispositivo e seus endpoints foram reconhecidos, e informando o minor number reservado ara ele. Para visualizar as mensagens, utilizar o comando “dmesg”.

Figura 20: Mensagens geradas ao Conectar o primeiro Arduino

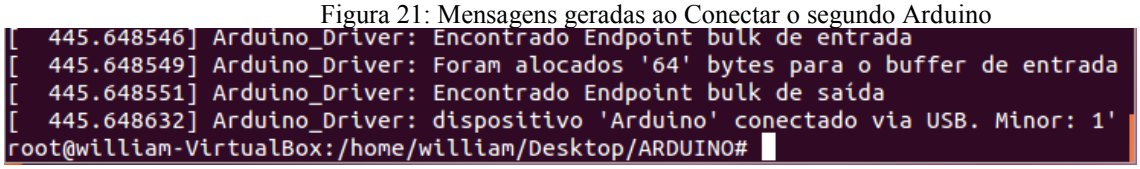


Fonte: Autores

Como a “minor base” utilizada no driver é zero, o minor number atribuído ao primeiro dispositivo será zero, e o arquivo de caractere “Arduino0” é automaticamente criado no diretório “/dev”.

Ao conectar o segundo dispositivo, o Arduino_Driver irá reconhecê-lo também, a atribuir o próximo minor base, que é 1. Dessa forma, o arquivo de caracteres “Arduino1” é criado no diretório “/dev”.

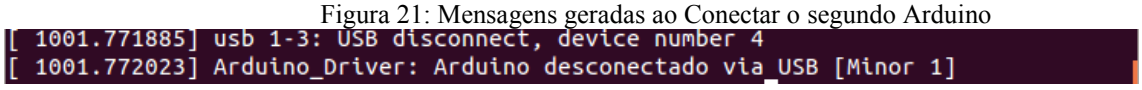
Figura 21: Mensagens geradas ao Conectar o segundo Arduino



Fonte: Autores

Caso um dos dispositivos seja removido do USB, o driver imediatamente percebe, e executa o método “Arduino_Disconnect”, para desregistrar o dispositivo, dentre outras operações. As seguintes mensagens mostradas na figura 22 são geradas no Kernel (uma pelo USB core, e a outra pelo Arduino_Driver), o minor number é “devolvido”, podendo ser utilizado por outro dispositivo, e o arquivo de caracteres correspondente é removido do diretório “/dev”.

Figura 21: Mensagens geradas ao Conectar o segundo Arduino



Fonte: Autores

5.4. Operações de Entrada e Saída com o Dispositivo

Como Arduino_Driver é um dispositivo de caractere, as operações de entrada e saída sobre cada dispositivo associado ao driver é realizada através de operações sobre os

respectivos arquivos de caractere, localizados em "/dev". Desta forma, a leitura e escrita em cada dispositivo pode ser feita de maneira independente dos demais.

Como o firmware gravado em cada dispositivo estará apenas esperando a chegada de uma string, devemos primeiramente escrever a string "strSTRING", onde "STRING" é a string que se deseja enviar, no arquivo de caractere pertencente ao dispositivo almejado. No caso, "/dev/Arduino0" ou "/dev/Arduino1". Qualquer forma de entrada e saída no dispositivo pode ser usada, como por exemplo, através de um programa que manipule arquivos. No caso, optou-se por utilizar entrada e saída através do terminal do Linux.

Primeiramente, deve-se permitir o acesso aos arquivos de caractere. Para os nossos exemplos, fornecemos acesso total (dono, grupo e outros) e a todas as operações nos dispositivos, mas outras permissões mais específicas podem ser dadas.

```
$ chmod 777 /dev/Arduino0
$chmod 777/dev/Arduino1
```

Feito isso, podemos enviar, por exemplo, a string "Teste do" para o primeiro dispositivo (Arduino0), usando:

```
$ echo strTeste do >> /dev/Arduino0
```

O Arduio_Driver irá executar o método "Arduino_Write", enviando os caracteres escritos para o Arduino0, o qual irá capturar a mensagem e exibi-la no display, e também acender o LED conectado ao pino 13 (fixo na placa do Arduino) para indicar que está preparado para enviar um caractere (modo de escrita). O driver foi programado para deixar o seguinte log no kernel, ao realizar a escrita (visualizar com o comando "dmesg"):

Figura 22: Log dos dados enviados. O byte 10 indica o final da string

```
[ 2848.782233] ==============Escrita no Arduino0=========================
[ 2848.782237] A: 0      ENVIANDO: T     84
[ 2848.795487] A: 1      ENVIANDO: e     101
[ 2848.798496] A: 2      ENVIANDO: s     115
[ 2848.801811] A: 3      ENVIANDO: t     116
[ 2848.804597] A: 4      ENVIANDO: e     101
[ 2848.807609] A: 5      ENVIANDO:       32
[ 2848.810806] A: 6      ENVIANDO: d     100
[ 2848.814080] A: 7      ENVIANDO: o     111
[ 2848.816891] A: 8      ENVIANDO:
[ 2848.816892]      10
```

Fonte: Autores

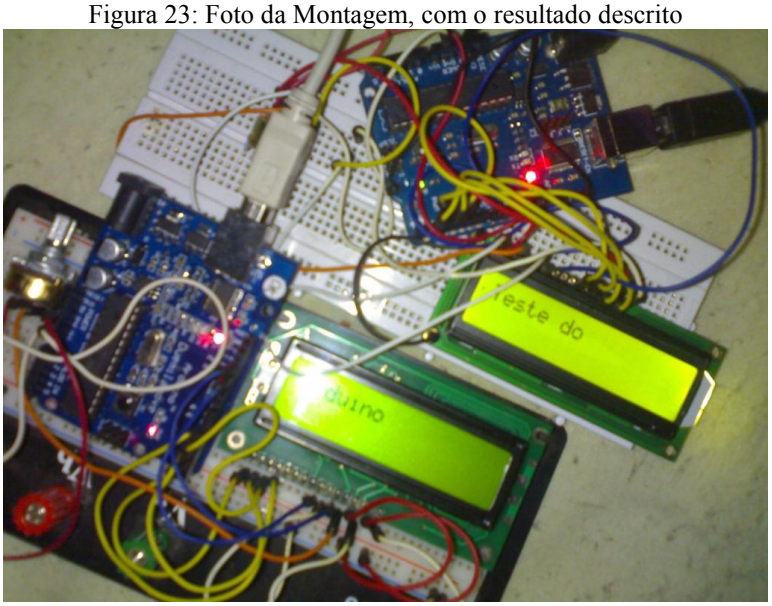
Podemos enviar a string, “Arduino” para o outro Arduino, o qual ainda está esperando, utilizando:

```
$ echo strArduino >> /dev/Arduino1
```

O outro dispositivo, Arduino1, irá então exibir a string na tela, e também entrará em modo de escrita.

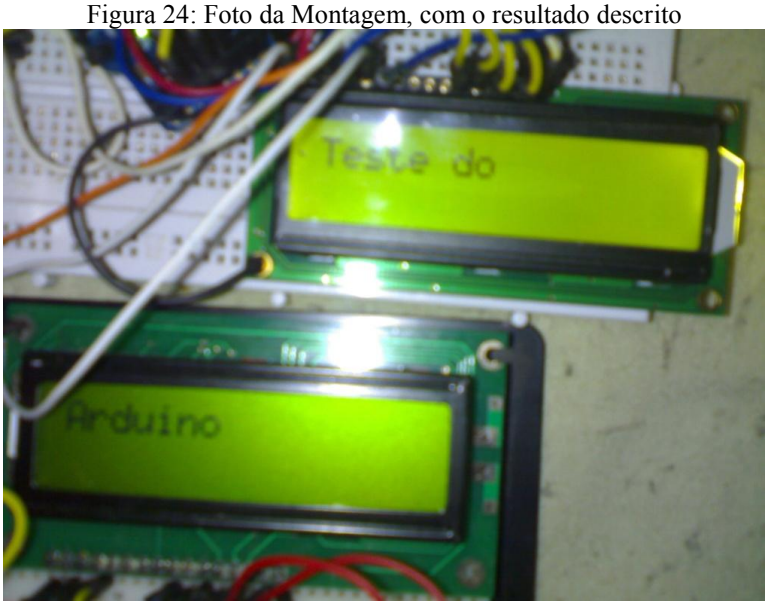
As figuras 23 e 24, mostradas abaixo, são fotos da montagem apresentando o resultado descrito no processo acima, isto é, as mensagens “Teste do” e “Arduino” exibidas por Arduino0 e Arduino1, respectivamente.

Figura 23: Foto da Montagem, com o resultado descrito



Fonte: Autores

Figura 24: Foto da Montagem, com o resultado descrito



Fonte: Autores

Para receber os dados enviados pelo Arduino, podemos ler o arquivo de caracteres. Cada vez que o arquivo for acessado para leitura, um byte enviado pelo Arduino é capturado. Por exemplo, podemos acessar o arquivo através do terminal, utilizando o comando:

```
$ cat /dev/Arduino0
```

Cada vez que um caractere é lido, começando pela letra ‘A’, o Atmega328 incrementa o mesmo, passando para o próximo caractere Unicode.

Figura 25: Saída após ler seguidos caracteres utilizando cat

```
root@william-VirtualBox:/home/william/Desktop/ARDUINO# cat /dev/Arduino0
ABCDEFGHIJKLMNOPQRSTUVWXY^CZ[\]^_`ab^C
```

Fonte: Autores

O driver foi programado para, ao realizar a leitura de um caractere, gerar um log da operação de escrita no kernel, o qual pode ser visualizado utilizando o comando “dmesg”.

Figura 26: Log gerado no Kernel ao ler o caractere ‘A’ (byte 65)

```
[  444.488918] ==============Leitura do Arduino0=========================
[  444.503426] BIT : 1
[  444.503427]
[  444.510068] BIT : 0
[  444.510069]
[  444.520967] BIT : 0
[  444.520969]
[  444.537829] BIT : 0
[  444.537830]
[  444.554048] BIT : 0
[  444.554049]
[  444.569705] BIT : 0
[  444.569706]
[  444.587031] BIT : 1
[  444.587032]
[  444.595416] BIT : 0
[  444.595417]
[  444.595419] VALOR: 65
```

Fonte: Autores

6. CONCLUSÃO

Os Device Drivers são aplicações extremamente necessárias em sistemas computacionais, e seu desenvolvimento mostrou-se não trivial. No entanto, o Sistema operacional Linux, por ser uma plataforma de código aberto baseada em Unix, oferece grande suporte ao desenvolvimento de device drivers e informações sobre o seu componente mais importante: o kernel. Embora bibliotecas desenvolvidas para suporte a aplicações no kernel space abstraem os conceitos de mais baixo nível empregados, conhecimentos sobre o funcionamento e a arquitetura do kernel mostraram-se indispensáveis.

Além disso, o Linux oferece uma biblioteca que facilita o desenvolvimento de drivers USB, abstraindo os principais conceitos envolvidos nesse protocolo de comunicação.

A maior dificuldade encontrada, no entanto, não foi no desenvolvimento do driver, mas sim na comunicação serial entre o Atmega328 e o FTDI RS232, o qual não oferece muito suporte e nem informações a respeito de seu funcionamento sem o driver padrão, de forma que tais informações tiveram que ser obtidas por meio de testes exaustivos e em baixo nível.

O Resultado obtido é satisfatório, uma vez que a comunicação no sentido do computador para o dispositivo está bem eficiente e praticamente sem erros de comunicação. Melhoras podem ser feitas futuramente, de forma a implementar um código de verificação de erros, aproveitando assim os bits desperdiçados na transferência. Outra melhoria pode ser a implementação de uma rotina capaz de ler uma string diretamente, sem precisar fazer chamadas sucessivas a “Read_String_from_USB”.

A comunicação no sentido contrário, porém, está ainda muito ineficiente, seja em tempo de execução (overhead muito grande) seja na quantidade de erros.

Trabalhos futuros podem focar na otimização do funcionamento, que pode ser de grande utilidade a sistemas desenvolvidos utilizando microcontrolador Atmega328, além do fato de permitir, de forma prática, entender os protocolos de transmissão e o desenvolvimento de módulos para o kernel do Linux.

REFERÊNCIAS

ARDUINO**. LiquidCrystal** – "Hello World!". Disponível em: <http://arduino.cc/en/Tutorial/LiquidCrystal>. Acesso em 06 jun. 2012

D'ASSUMPÇÂO, Arthur. **Introdução aos Linux Device Drivers:** Capítulo 1. Disponível em: <http://www.adassumpcao.net/files/ildd_cp1_pt.pdf>. Acesso em 07 jun. 2012

D'ASSUMPÇÂO, Arthur. **Introdução aos Linux Device Drivers:** Capítulo 2. Disponível em: <http://www.adassumpcao.net/files/ildd_cp2_pt.pdf>. Acesso em 07 jun. 2012

D'ASSUMPÇÂO, Arthur. **Introdução aos Linux Device Drivers:** Capítulo 3. Disponível em: <http://www.adassumpcao.net/files/ildd_cp3_pt.pdf>. Acesso em 07 jun. 2012

DEITEL, Harvey M.; DEITEL, Paul M.; CHOFFNES, David R. **Sistemas Operacionais**. 3. ed. São Paulo: Pearson, 2005. P. 634-640. Título Original: Operating Systems.

PRADO, Sérgio. **Linux Device Drivers – Parte 1**. Disponível em: <http://sergioprado.org/linux-device-drivers-parte-1/>. Acesso em: 07 jun. 2012.

PRADO, Sérgio. Linux **Device Drivers – Parte 2**. Disponível em: <http://sergioprado.org/linux-device-drivers-parte-2/>. Acesso em: 07 jun. 2012.

RUBINI, Alessandro. **Linux Device Drivers**. 1. ed. São Paulo: Market Books, 1999. P. 1-76.

RUBINI, Alessandro; CORBET, Jonathan; KROAH-HARTMAN, Greg**. Linux Device Drivers**. 3th ed. Sebastopol, CA: O'Reilly Media, 2005. P. 327-359.

VALLENTIN, Matthias**. Writing a Linux Device Driver for na Unknown USB Device**. Disponível em: < http://matthias.vallentin.net/blog/2007/04/writing-a-linux-kernel-driver-for-an-unknown-usb-device/>. Acesso em 07 jun. 2012.