KERNEL DRIVER

Existem duas formas de utilizar o display USB:

1 – Criar um *device driver*
2 – Usar um *device driver* padrão

TIPO DE INTERFACE COM O *USER SPACE*:

~~1 – *Block device driver* (armazena dados) → Não~~
2 – *Character device driver* (envia e recebe caracteres) → Sim

Obs.: Um *Character device driver* necessita reservar *major* e *minor numbers*. Por exemplo, se eu quiser conectar 10 displays USB, devo reservar no mínimo 10 *minor numbers*. Mas se eu for conectar apenas um display por máquina, não necessito desperdiçar recursos.

CONTROLE DE TRANSFERÊNCIA (I/O):

Para a primeira etapa de desenvolvimento o PIC não necessita enviar caracteres para o PC, apenas receber dados relativos ao funcionamento do mesmo e eventualmente cadeias de caracteres referentes a mensagens.
O acionamento dos displays (20x2 e 20x4) pode ser controlado de duas formas:

1 - Dentro do PIC (como é feito nos programas de testes)
2 - A outra forma é aproveitar a estrutura dos comandos que são executados hoje no MicroCPD (comandos para a porta paralela) e adaptá-los para enviar via USB.

De qualquer forma, deve existir uma estrutura (script, macro, programa) que capture as informações necessárias do sistema e envie ao ***driver*** do display.

Obs.: Existem comandos **ioctl** no ***character driver*** que possibilitam o envio de dados para o display, provavelmente será melhor criar meus próprios comandos **ioctl** no **Kernel**.

Como todos os dispositivos USB figuram em seu próprio diretório na árvore **sysfs**, por que não usar **sysfs** e criar arquivos no diretório de **dispositivos** USB? Isto possibilita que qualquer programa do ***user-space*** feito em C ou através de script Shell envie mensagens ao display. Isto também evita que tenhamos que escrever um ***character driver*** e implorar por um pedaço de ***minor numbers*** para nosso **dispositivo**.

Para acionar o **USB *driver***, nós precisamos prover 5 itens ao **USB subsystem**:

1 – Um **ponteiro** p/ o módulo proprietário desse ***driver***: Isto habilita o ***USB core*** a controlar o módulo ***reference count*** do ***driver*** apropriadamente;
2 – O **nome** do ***USB driver***;
3 – Uma **lista** de ***USB IDs*** que este ***driver*** provê: Esta tabela é usada pelo ***USB core*** para determinar qual ***driver*** foi escolhido para qual **dispositivo**; os scripts ***hot-plug user-space*** usam esta tabela para carregar o ***driver*** automaticamente quando um **dispositivo** é conectado no sistema.

4 – Uma função sonda **probe( )** chamada pelo ***USB core*** quando um dispositivo é encontrado e figura na ***USB ID table***;
5 – Uma função **disconnect( )** chamada quando o **dispositivo** é removido do sistema.

O ***driver*** recupera esta informação com o seguinte código:

```
static struct usb_driver vader_driver =
{
        .owner =        THIS_MODULE,
        .name           =       “usbvader”,
        .probe          =       vader_probe,
        .disconnect     =       vader_disconnect,
        .id_table       =       id_table,
};
```

A variável **id_table** é definida como:

```
static struct usb_device_id id_table [ ] =
{
        { USB_DEVICE(VENDOR_ID, PRODUCT_ID)},
        { },
};
MODULE_DEVICE_TABLE(usb,id_table);
```

A função **vader_probe( )** e a função **vader_disconnect( )** serão descritas mais adiante.

Quando o módulo do ***driver*** é carregado, a estrutura **vader_driver** deve ser registrada no ***USB core***. Isto é realizado com uma única chamada à função **usb_register( )**:

```
retval = usb_register(&vader_driver);
if(retval)
        err(“usb_register falhou.”
                “Erro nro %d”,retval);
```

Do mesmo modo, quando o ***driver*** é descarregado do sistema, ele deve se de-registrar do ***SUB core***:

```
usb_deregister(&vader_driver);
```

A função **vader_probe( )** é chamada quando o ***USB core*** encontrar nosso dispositivo **VADER**. Tudo que é necessário fazer é inicializar o dispositivo e criar o arquivo no **sysfs** no local apropriado. Isto é feito com o código a seguir:

```
/* Inicializa nossa estrutura local para o dispositivo */
dev = kmalloc(sizeof(struct usb_vader), GFP_KERNEL);
memset (dev, 0x00, sizeof (*dev));
```

```
dev->udev = usb_get_dev(udev);
usb_set_intfdata (interface, dev);

/* Cria o arquivo p/ sysfs no USB * device directory */
device_create_file(&interface->dev, &dev_attr_arquivo);

dev_info(&interface->dev,
   "USB VADER conectado\n");
return 0;
```

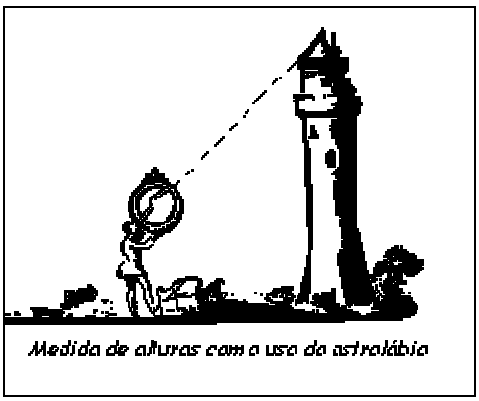
A função **vader_disconnect( )** é igualmente simples, é necessário liberar a memória alocada e remover o arquivo **sysfs**:

```
dev = usb_get_infdata(interface);
usb_set_intfdata(interface, NULL);
device_remove_file(&interface->dev, &dev_attr_arquivo);

usb_put_dev(dev->udev);
kfree(dev);

dev_info(&interface->dev,"USB VADER desconectado\n");
```

Quando o arquivo **sysfs** é lido, mostra uma mensagem vinda do PIC; quando o arquivo for escrito, irá enviar uma mensagem para o PIC. A macro a seguir cria duas funções e declara um arquivo **sysfs** de atributo de dispositivo:

```
#define show_set(value)                                 \
static ssize_t                                          \
show_##value(struct device *dev, char *buf)             \
{                                                       \
  struct usb_interface *intf =                          \
    to_usb_interface(dev);                              \
  struct usb_vader *vader = usb_get_intfdata(intf);     \
                                                        \
  return sprintf(buf, "%d\n", vader->value);            \
}                                                       \
                                                        \
static ssize_t                                          \
set_##value(struct device *dev, const char *buf,        \
        size_t count)                                   \
{                                                       \
  struct usb_interface *intf =                          \
    to_usb_interface(dev);                              \
  struct usb_vader *vader = usb_get_intfdata(intf);     \
  int temp = simple_strtoul(buf, NULL, 10);             \
                                                        \
  vader->value = temp;                                  \
  change_color(led);                                    \
  return count;                                         \
}                                                       \
```

```
static DEVICE_ATTR(value, S_IWUGO | S_IRUGO,
show_##value, set_##value);
show_set(blue);
show_set(red);
show_set(green);
```



Medida de alturas com o uso do astrolábio

Luiz Francisco Bozo - 2008