Implementación de manejadores de dispositivos

Presentación Trabajo Final

Matías Alvarez

Agenda

- LCD HD44780
- Driver
  - Device Tree
  - Kernel Space
  - User Space - Char Device

LCD HD44780

- IR Register - Códigos de instrucción
- Data Register - Datos

Selector de Registro		
RS	**R/W**	**Funcion**
0	0	Envía un comando al LCD
0	1	Lee busy flag y address counter del LCD
1	0	Envía data al LCD
1	1	Lee data del LCD

LCD HD44780

- Set de Instrucciones

Clear Display									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	0	0	0	0	0	0	0	1

Home Cursor									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	0	0	0	0	0	0	0	1

Display On/Off Control									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	0	0	0	0	1	**D**	**C**	**B**

LCD HD44780

- Set de Instrucciones

Function Set									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	0	0	1	**DL**	**N**	**F**	0	1

Set DDRAM Address									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
0	0	1	**Address a escribir**						

Write Data									
RS	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
1	0	**Data a escribir**							

Device Tree - AM335x-Boneblack

- Estructura de datos que describe componentes de hardware para que el kernel pueda usarlos.
- Device Tree Source
- Device Tree Blob
- make dtbs

Device Tree - AM335x-Boneblack

- El subsistema pinctrl permite el manejo del pinmuxing dentro del microprocesador.
- El pinmuxing DEBE ser descripto en el device tree
- Par (registro, valor) para configurar los pines

```
&am33xx_pinmux {
        i2c1_pins: pinmux_i2c1_pins {
                pinctrl-single,pins = <
                        AM33XX_IOPAD(0x958, PIN_INPUT_PULLUP | MUX_MODE2)
                        AM33XX_IOPAD(0x95c, PIN_INPUT_PULLUP | MUX_MODE2)
                >;
        };
};
```

Device Tree - AM335x-Boneblack

- **pinctrl-0:** Linkea a una determinada configuración de pines para un dado estado del dispositivo
- **pinctrl-names:** Asocian un nombre a cada estado

```
&i2c1 {
        status = "okay";
        pinctrl-names = "default";
        clock-frequency = <100000>;
        pinctrl-0 = <&i2c1_pins>;
        mylcd: mylcd@3f {
                compatible = "mse,mylcd";
                reg = <0x3f>;
        };
};
```

Driver I2C

- Define los dispositivos que el driver puede manejar

```
static const struct i2c_device_id mylcd_i2c_id[] =
{
   { "mylcd", 0 },
   { }
};
MODULE_DEVICE_TABLE(i2c, mylcd_i2c_id);
```

Driver I2C

- Con un device tree, cada dispositivo es unido a su driver respectivo utilizando el string compatible.
- Lista los strings compatibles soportados.

```
static const struct of_device_id mylcd_of_match[] =
{
    { .compatible = "mse,mylcd" },
    { }
};
MODULE_DEVICE_TABLE(of, mylcd_of_match);
```

Driver I2C

- Todos los drivers i2c deben instanciarla y auto-registrarse al core del i2c mediante esta estructura.
- **probe:** Inicializa el dispositivo.
- **remove:** Apaga el dispositivo.
- **id_table:** Apunta a la tabla de dispositivos que el driver maneja.

```
static struct i2c_driver mylcd_i2c_driver =

{

    .driver =

    {

        .name = "mylcd",

        .of_match_table = mylcd_of_match,

    },

    .probe = mylcd_probe,

    .remove = mylcd_remove,

    .id_table = mylcd_i2c_id

};
```

Character Device

- Permiten a las aplicaciones de espacio de usuario acceder a dispositivos que no son ni de red ni block devices. Se listan en **/dev**.
- Identificados con major y minor number.
- Objetos como archivos - Accedidos mediante **open**, **read**, **write**, etc.
- Driver implementa estas operaciones y las registra mediante la estructura **file_operators**

Character Device

```
static struct file_operations mylcd_fops =
{
    .read = mylcd_fopread,
    .write = mylcd_fopwrite,
    .unlocked_ioctl = mylcd_ioctl,
    .open = mylcd_open,
    .release = mylcd_release,
};
```

Character Device - Inicialización

- **register_chrdev(major, name, fops):** En nuestro caso major es 0, por lo que la función aloca dinámicamente un major number y lo retorna.
- **class_create(owner, name):** Registra la clase del dispositivo.
- **device_create():** Registra el driver del dispositivo y se crea un archivo en **/dev**.

Character Device - FOPS

- **Open:** Llamada cuando el espacio de usuario abre el archivo.
- **Release:** Llamada cuando el espacio de usuario cierra el archivo.
- **Read:** Llamada cuando el espacio de usuario usa la system call **read()**. Debe leer del dispositivo, escribir en el buffer del usuario y actualizar el offset. Devuelve los bytes leídos.
- **Write:** Llamada cuando el espacio de usuario usa la system call **write()**. Debe escribir en el dispositivo, actualizar el offset y devolver los bytes escritos.

UNLOCKED_IOCTL

- Asociada a la system call **ioctl()**.
- Permite aumentar las capacidades del driver más allá del read y write.
- Se le pasa un unsigned int indicando el comando a ejecutar.
- Se le pasa un unsigned long como argumento - Opcional.
- Asignación de nombres simbólicos a los comandos en el header del driver mediante las macros **_IO**, **_IOR**, **_IOW**, **_IOWR** definidas en **<asm/ioctl.h>**.
- Argumentos de las macros: **type**, **number**, **data_type**.

UNLOCKED_IOCTL

```
#define LCD_IOCTL_GETCHAR          _IOR(LCD_IOCTL_BASE, IOCTLC | (0x01 << 2), char *)

#define LCD_IOCTL_SETCHAR          _IOW(LCD_IOCTL_BASE, IOCTLC | (0x01 << 2), char *)

#define LCD_IOCTL_GETPOSITION      _IOR(LCD_IOCTL_BASE, IOCTLB | (0x03 << 2), char *)

#define LCD_IOCTL_SETPOSITION      _IOW(LCD_IOCTL_BASE, IOCTLB | (0x04 << 2), char *)

#define LCD_IOCTL_RESET            _IOW(LCD_IOCTL_BASE, IOCTLC | (0x05 << 2), char *)

#define LCD_IOCTL_HOME             _IOW(LCD_IOCTL_BASE, IOCTLC | (0x06 << 2), char *)

#define LCD_IOCTL_SETBACKLIGHT     _IOW(LCD_IOCTL_BASE, IOCTLC | (0x07 << 2), char *)

#define LCD_IOCTL_GETBACKLIGHT     _IOR(LCD_IOCTL_BASE, IOCTLC | (0x07 << 2), char *)
```

Intercambio de datos

- **get_user(v, p):** Se obtiene en la variable del kernel **v**, el valor apuntado por el puntero **p** de espacio de usuario.
- **put_user(v, p):** El valor apuntado por el puntero **p** del espacio de usuario es seteado con el valor de la variable **v** del kernel.
- **i2c_master_send(client, buf, count):** Envía por el bus I2C, **count** bytes contenidos en **buf**.
- **i2c_master_recv(client, buf, count):** Recibe por el bus I2C, **count** bytes y los almacena en **buf**.

¿Preguntas?

Muchas Gracias