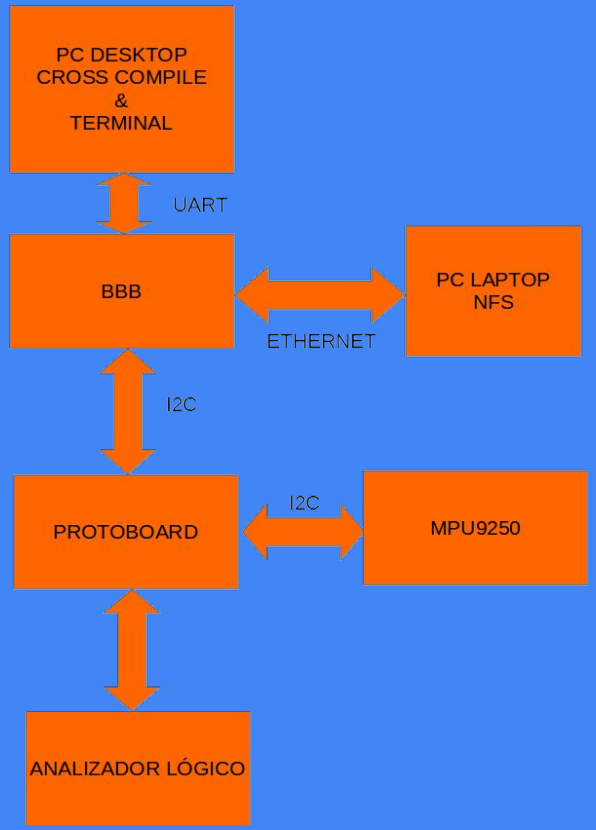
Trabajo Final IMD
Driver I2C Sensor de temperatura MPU9250

Autor: Gonzalo Lavigna
Junio 2019

Configuración del setup

- EL alumno utilizó la placa Beagle Bone Black.
- Por esto las instrucciones de la guía sirven para la compilación del mismo.
- Se realizarón las siguientes instalaciones de dependencias:
  - sudo apt install libssl-dev bison flex
  - sudo apt install picocom
  - sudo adduser $USER dialout
- Se instaló el toolchain para hacer cross compilación:
  - sudo apt install gcc-arm-linux-gnueabi
  - dpkg -L gcc-arm-linux-gnueabi

Setup Utilizado



- Se utilizó otra computadora para cargar el NFS, porque el puerto de red de la que tenemos para compilar y desarrollar esta ocupado.
- Se conecta un analizador lógico para ver las transferencia I2C y poder ver el tráfico.
- El Analizador Lógico se conecta a la PC Desktop.

Compilación del kernel

- Se siguieron las instrucciones de la guía para poder bajar los fuentes
  - git remote add stable git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
  - git fetch stable
  - git checkout -b 4.19.y stable/linux-4.19.y
- En la terminal en la cual se va a realizar la compilación del kernel, se utiliza la configuración por defecto de la beaglebone
  - export ARCH=arm
  - export CROSS_COMPILE=arm-linux-gnueabi-
  - make omap2plus_defconfig
  - make -j4
- Se verifica en el archivo .config que esté configurado la utilización del network file system CONFIG_PROVE_LOCKING = Y

Primer Booteo

- El archivo zimage que utiliza para bootear está en:
  - /home/glavigna/linux-kernel-labs/src/linux/arch/arm/boot
- El archivo de device tree que se utiliza está en:
  - /home/glavigna/linux-kernel-labs/src/linux/arch/arm/boot/dts
- Para la primer booteo se utiliza el dtb por defecto para la beagle bone black → am335x-boneblack.dtb
- Se instala el servidor TFTP para poder hacer el traspaso del zimage y el dtb elegido.
  - Los archivos de zimage y dtb se tienen que copiar en /var/lib/tftpboot.
  - Para copiarlos hay que hacer un sudo cp

Primer booteo

- En la PC server se instala el NFS con:
  - sudo apt install nfs-kernel-server
  - sudo /etc/init.d/nfs-kernel-server restart (Para reiniciar el servicio de kernel server)
- Se afrontó el problema que en la instrucción del booteo y la configuración del NFS tienen que apuntar al mismo path en la PC que hace de servidor.
- El archivo para el NFS se modifica para que contenga la siguientes líneas:
  - sudo nano /etc/exports
  - /home/glavigna/linux-kernel-labs/modules/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)

Comandos consola UART

- En la consola UART que conectamos con el picocom con la BBB
  - picocom -b 115200 /dev/ttyUSB0
  - Interrumpir el booteo con space al principio
  - setenv ipaddr 192.168.0.100
  - setenv serverip 192.168.0.1
  - tftp 0x81000000 text.txt (Solo para probar la conexion TFTP)
  - md 0x81000000 (Solo para probar la conexion TFTP)
  - setenv bootargs root=/dev/nfs rw ip=192.168.0.100 console=ttyO0,115200n8 nfsroot=192.168.0.1:/home/glavigna/linux-kernel-labs/modules/nfsroot,nfsvers=3
  - tftp 0x81000000 zImage
  - tftp 0x82000000 am335x-boneblack.dtb
  - bootz 0x81000000 - 0x82000000

Tree

- Se agarra el archivo am335x-boneblack.dts y se crea otro con con la siguiente terminación am335x-boneblacklavigna.dts
- En este se agrega la configuración de los pines, el device address y la frecuencia del I2C. También algo importante es el nombre del driver.

```
&am33xx_pinmux {
    i2c1_pins: pinmux_i2c1_pins {
    pinctrl-single,pins = <
        AM33XX_IOPAD(0x958, PIN_INPUT_PULLUP | MUX_MODE2) /* spi0_d1.i2c1_sda */
        AM33XX_IOPAD(0x95c, PIN_INPUT_PULLUP | MUX_MODE2) /* spi0_cs0.i2c1_scl */
    >;
    };
};

/*Habilitamos el bus I2C1*/
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    status = "okay";
    clock-frequency = <400000>;

    mympu9250: mympu9250@68 {
        compatible = "mse,mympu9250";
        reg = <0x68>;
    };
};
```

Compilación Device Tree

- Agregamos la instrucción para la compilación del device tree
- En el makefile de los DTB en: ~/linux-kernel-labs/src/linux/arch/arm/boot/dts/Makefile
- Debajo de la línea 692 , agregamos el nuevo device tree a compilar
  - dtb-$(CONFIG_SOC_AM33XX) += \
  - Agregamos el nuevo .dtb → am335x-boneblacklavigna.dtb
- Volvemos al path ~/linux-kernel-labs/src/linux y hacemos un make dtbs
- El nuevo dtb va a estar en: ~/linux-kernel-labs/src/linux/arch/arm/boot/dts

```
692  dtb-$(CONFIG_SOC_AM33XX) += \
693      am335x-baltos-ir2110.dtb \
694      am335x-baltos-ir3220.dtb \
695      am335x-baltos-ir5221.dtb \
696      am335x-base0033.dtb \
697      am335x-bone.dtb \
698      am335x-boneblack.dtb \
699      am335x-boneblacklavigna.dtb \
```

Booteo Device Tree Custom

- Con respecto a los comandos de consola del primer booteo hay que cambiar los siguientes comandos:
  - tftp 0x82000000 am335x-boneblacklavigna.dtb
- Una vez realizado el boot con esta configuración ejecutamos el siguiente comando para ver si cargo el device tree custom.
  - tftp 0x82000000 am335x-boneblacklavigna.dtb

```
# find /sys/firmware/devicetree/ -name "*mympu9250*"
/sys/firmware/devicetree/base/ocp/i2c@4802a000/mympu9250@68
```

Driver MPU9250→ Introducción

- Para la escritura del driver se decide solamente obtener la temperatura que indica el MPU9250.
- Si bien este módulo presenta muchas más funcionalidades, nos quedamos con esta a modo de demostración.
- Las funciones del driver están pensadas para obtener datos del dispositivos a través de la interfaz I2C. El detalle fino del driver se decidió no hacerse por cuestiones de tiempo y además porque sería una copia de la SAPI de Eric Pernia.

Driver MPU9250

- Primero se tienen que escribir estas estructuras para poder registrar el driver.
- Tener en cuenta que el device tiene que tener el mismo nombre declarado en el device tree, esto es para que el sistema operativo pueda encontrar el device.

```
static const struct i2c_device_id mympu9250_i2c_id[] = {
{ "mympu9250", 0 },
{ }
};
MODULE_DEVICE_TABLE(i2c, mympu9250_i2c_id);
static const struct of_device_id mympu9250_of_match[] = {
    { .compatible = "mse,mympu9250" },
    { }
};
MODULE_DEVICE_TABLE(of, mympu9250_of_match);
```

Driver MPU9250

- Para la escritura se utilizó la explicación del driver de Derek Malloy en:
  - http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/
- Se escriben las funciones de probe() y remove()
- En la función de probe():
  - Se realiza el ebbchar_init() que registra el dispositivo en /dev/
  - También se hace una copia del cliente I2C para que pueda ser llamado por las funciones de read() y write() por parte del usuario.

```
static struct i2c_driver mympu9250_i2c_driver = {
    .driver = {
        .name = "mympu9250",
        .of_match_table = mympu9250_of_match,
    },
    .probe = mympu9250_probe,
    .remove = mympu9250_remove,
    .id_table = mympu9250_i2c_id,
};
```

Compilacion Driver

- Para compilar el driver hay que hacer:
  - export ARCH=arm
  - export CROSS_COMPILE=arm-linux-gnueabi-
  - make
- Este es el makefile para generar el .ko.

```
ifneq ($(KERNELRELEASE),)
obj-m := mympu9250.o
else
KDIR := $(HOME)/linux-kernel-labs/src/linux
all:
    $(MAKE) -C $(KDIR) M=$$PWD
endif
```

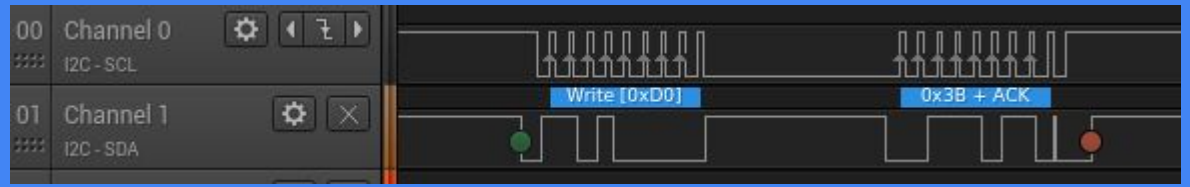
- Para compilar el código de usuario hay que hacer:
  - arm-linux-gnueabi-gcc test_write.c -o test_write

Driver MPU9250→Probe

- En la función de probe() se hace un acceso de lectura de 21 registros como se hace en la SAPI, para poder ver si el dispositivo está funcionando.

```
pr_info("READ 21 REGISTER FROM ADDRES 0x3B\n");
rv = i2c_master_send(client,buf,1);
rv = i2c_master_recv(client,mpu9250_output_buffer,21);
pr_info("Datos Recibido: %0d\n",rv);
for (i = 0; i < 21; i++)
{
    pr_info("REGISTRO=0x%02X --> Valor: 0x%02X\n",buf[0]+i,mpu9250_output_buffer[i]);
}
```

- Se hace un write con el start address para configurar el primer registro desde donde se empiezan a leer todos los registros. (Esto es un write a 0x68)



Driver MPU9250→Probe

- (Esto es un read a 0x68). A partir de aquí se leen 21 registros en rafaga.

- Esto es lo que se lee a través de la consola, vemos que lo datos coinciden entre la terminal y la lectura del analizador lógico.

```
[11755.714276] READ 21 REGISTER FROM ADDRES 0x3B
[11755.720005] Datos Recibido: 21
[11755.723093] REGISTRO=0x3B --> Valor: 0x29
[11755.727231] REGISTRO=0x3C --> Valor: 0xBC
[11755.731271] REGISTRO=0x3D --> Valor: 0x00
[11755.735363] REGISTRO=0x3E --> Valor: 0x00
[11755.739400] REGISTRO=0x3F --> Valor: 0x33
[11755.743434] REGISTRO=0x40 --> Valor: 0x1C
[11755.747535] REGISTRO=0x41 --> Valor: 0x08
[11755.751571] REGISTRO=0x42 --> Valor: 0x50
[11755.755658] REGISTRO=0x43 --> Valor: 0x00
[11755.759696] REGISTRO=0x44 --> Valor: 0x33
[11755.763730] REGISTRO=0x45 --> Valor: 0x01
[11755.767818] REGISTRO=0x46 --> Valor: 0x28
[11755.771854] REGISTRO=0x47 --> Valor: 0x00
[11755.775940] REGISTRO=0x48 --> Valor: 0x82
[11755.779977] REGISTRO=0x49 --> Valor: 0x00
[11755.784010] REGISTRO=0x4A --> Valor: 0x00
[11755.788097] REGISTRO=0x4B --> Valor: 0x00
[11755.792133] REGISTRO=0x4C --> Valor: 0x00
[11755.796220] REGISTRO=0x4D --> Valor: 0x00
[11755.800257] REGISTRO=0x4E --> Valor: 0x00
[11755.804291] REGISTRO=0x4F --> Valor: 0x00
```

Driver MPU9250→Device Tree

- Para instalar el driver hay que ejecutar el siguiente comando desde la consola:
  - cd /root/mympu9250
  - Insmod mympu9250.ko

```
# insmod mympu9250.ko
[11755.652908] EBBChar: Initializing the EBBChar LKM
[11755.657977] EBBChar: registered correctly with major number 246
[11755.664010] EBBChar: device class registered correctly
[11755.671486] EBBChar: device class created correctly
[11755.676723] PROBE:mympu9250
```

- En la consola hacemos un ls -al en /dev/ → Aca vemos que se registro el device con 246, que es el mismo número que aparece cuando hacemos insmod.

```
crw-------    1 root     root       89,   0 Jan  1 00:00 i2c-0
crw-------    1 root     root       89,   1 Jan  1 00:00 i2c-1
crw-------    1 root     root       89,   2 Jan  1 00:00 i2c-2
crw-------    1 root     root      246,   0 Jan  1 03:15 i2c_mse
```

Driver MPU9250→Remove

- En el remove(), se hace la destrucción del device, cuando se hace el rmmod mypu9250.

```
static int mympu9250_remove(struct i2c_client *client)
{
    pr_info("REMOVE:mympu9250\n");
    ebbchar_exit();
    return 0;
}
```

- En la consola podemos observar que el dispositivo se remueve correctamente.

```
# rmmod mympu9250
[12587.916268] REMOVE:mympu9250
[12587.919978] EBBChar: Goodbye from the LKM!
```

Driver MPU9250→Char Device

- Para realizar el char device se reescribe la funcionalidad de las siguiente funciones.
- Se agrega la sobreescritura del lseek() para poder escribir y leer en una posición en particular.

```
// The prototype functions for the character driver -- must come before the struct definition
static int     dev_open(struct inode *, struct file *);
static int     dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);
static loff_t  device_lseek(struct file *file, loff_t offset, int orig);
```

```
static struct file_operations fops =
{
    .open = dev_open,
    .read = dev_read,
    .write = dev_write,
    .release = dev_release,
    .llseek = device_lseek,
};
```

Driver MPU9250→llseek()

- Las funciones de open(), release() no se describen porque son muy parecidas a las funciones realizadas en la página de Derek Malloy.
- La función de device_lseek() actualiza el parámetro offset para la funciones de read y de write(), para saber a que registro leer o escribir.

```
static loff_t device_lseek(struct file *file, loff_t offset, int orig) {
    loff_t new_pos = 0;
    printk(KERN_INFO "mympu9250 : lseek function in work\n");
   //No importa la configuracion del offset siempre se configura con el valor del parametro.
   //Esto es porque no tenemos ningun buffer y la idea es accederlo como un mapa de registros.
    switch(orig) {
        case 0 : /*seek set*/
            new_pos = offset;
            break;
        case 1 : /*seek cur*/
            new_pos = offset;
            break;
        case 2 : /*seek end*/
            new_pos = offset;
            break;
    }
    file->f_pos = new_pos;
    return new_pos;
}
```

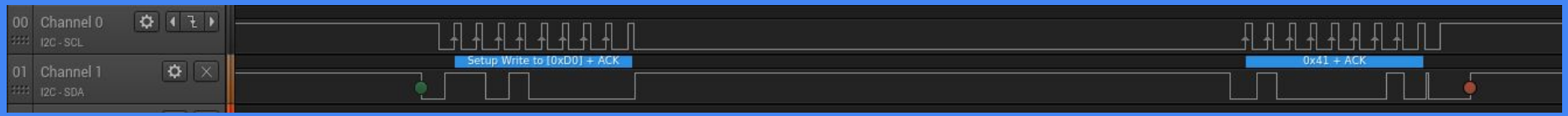
Driver MPU9250→read()

- Este es el código de usuario para poder leer cierta cantidad de registros. Con el lseek() nos ubicamos en el registro en el cual se encuentra la temperatura.

```
//Lectura de la temperatura
lseek(fd,TEMPREG,SEEK_SET);
cant_lectura = 2;
printf("Se leeran %d bytes \n",cant_lectura);
printf("Leyendo desde el dispositivo \n");
ret = read(fd,buffer,cant_lectura);
```

```
[13664.480004] mympu9250 : lseek function in work
Se leeran 2 bytes [13664.485001] i2c: Start Reg Address MPU9250: 0x41

Leyendo desde el dispositivo
[13664.492432] i2c: Cantidad de bytes configurados para leer = 2
[13664.492691] EBBChar: Sent 2 characters to the user
REGISTRO=0x41 --> Valor: 0x08
REGISTRO=0x42 --> Valor: 0x20
Temperatura:27.17
```



Driver MPU9250→read()

- Este es el código del driver, el offset es el que se setea con el lseek(), de esta manera podemos configurar el primer adress desde donde hay que levantar los datos de la temperatura.

```
static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset){
   int error_count = 0;
   int ret;
   //El offset esta configurado cuando se hace la funcion de lseek.
   pr_info("i2c: Start Reg Address MPU9250: 0x%02X \n",(char)(*offset));
   //Esto es para tenerlo sin el correspondiente casteo//pr_info("Offset: %lld \n",(*offset));
   //Con el read siempre me llevo el primer address-->Recordar que el address siempre es de un byte para este dispositivo.
   ADDRESS[0] = (char)(*offset);
   //Seteo el Address a partir del cual quiero escribir
   ret = i2c_master_send(modClient,ADDRESS,1);
   if(ret < 0){
      printk(KERN_INFO "i2C: No se pudo configurar el registro para leer los datos desde el MPU9250\n");
   }

   pr_info("i2c: Cantidad de bytes configurados para leer = %d",len);
   ret = i2c_master_recv(modClient,message,len);
   if(ret != len){
      printk(KERN_INFO "i2C: No se pudieron recibir %0d bytes desde el MPU9250\n", len);
   }
```

Driver MPU9250→write()

- Este es el código de usuario para poder escribir cierta cantidad de registros. Con el lseek() nos ubicamos en el registro en el cual se encuentra la temperatura.

```
//Offset para configurar la escritura de un registro
offset = PWRMGMNTREG_1;
lseek(fd,offset,SEEK_SET);
wr_buff[0] = 0x1;
cant_escritura = 1;
printf("Se escribiran %d bytes al registro: 0x%02X \n",cant_escritura,PWRMGMNTREG_1);
printf("Escribiendo desde el dispositivo \n");
ret = write(fd,wr_buff,cant_escritura);
```

```
[13875.907467] mympu9250 : lseek function in work
Se escribiran 1 bytes al registro: 0x6B [13875.912140] i2c: Start Reg Address MPU9250: 0x6B

Escribiendo desde el dispositivo
[13875.920377] i2c: Cantidad de bytes a escribir: 1
[13875.928366] i2c: Bytes recibidos desde el usuario:2
[13875.934467] i2c: Se escribieron 2 bytes a través del I2C
```

Driver MPU9250→write()

- Este es el código del driver, el offset es el que se setea con el lseek() en el código de usuario , de esta manera podemos configurar el primer adress desde donde hay que empezar a escribir los datos.

```
static ssize_t dev_write(struct file *filep, const char *buffer, size_t len, loff_t *offset){
    int ret;
    char buf[256];
    pr_info("i2c: Start Reg Address MPU9250: 0x%02X \n",(char)(*offset));
    pr_info("i2c: Cantidad de bytes a escribir: %0d \n",len);
    //Cargo el address que viene en el offset
    buf[0] = (char)(*offset);
    //Agarro los datos provenientes del usuario
    ret = copy_from_user(message,buffer,len);
    size_of_message = strlen(message);
    pr_info("i2c: Bytes recibidos desde el usuario:%d\n",size_of_message);
    //Copio el mensaje proveniente del usuario en el mensaje a enviar al usuario
    //Recordar que el address que hay que poner al principio hay que ponerlo y por eso va el +1.
    strcpy(buf+1,message);
    //Escribo por el I2C con el address al principio.
    ret = i2c_master_send(modClient,buf,size_of_message);
    pr_info("i2c: Se escribieron %d bytes a través del I2C\n",ret);
```