CSE 438 - Assignment 2
Kushal Anil Parmar - 1207686255

Part 1 - User code to access i2c device EEPROM with i2c-dev interface

==========

It contains 4 files:

usermain.c
libmain.h
gpioaccess.h
I2c-dev.h
Makefile

usermain.c -> This is a user space app which uses the library functions to access an EEPROM. It defines the list of operations to perform on the EEPROM. It asks the user to set position of page to read or write, and the number of pages to read or write. It creates a random string to write to the EEPROM of size specified by number of pages entered by user (number of pages * 64bytes per page)

libmain.h -> This library defines the functions/operations to access/perform on the EEPROM. It uses the i2c-dev interface by including i2c-dev.h from the i2c tools package. Slave address is set using the I2C_SLAVE ioctl and read write is performed on a page basis. Each page is of 64 bytes. The read/write functions are blocking calls and return to the user only when the data is completely written and read to/from the EEPROM.

gpioaccess.h -> This is utility library to access the gpio pins. User can provide the gpio number to export the corresponding gpio. The gpio is set using 1 (= out) to gpioDirection input.
GPIO26 acting as direction pin is given the direction as out.
GPIO10 is initialised and set to 0, which is used for turning led ON and OFF.
GPIO74 which acts as a select pin is used set to low to select the GPIO10.
GPIO60 is the i2c mux select pin which is set to low inorder to enable the SDA and SCL lines.

i2c-dev.h -> This is a library is used to access all i2c devices on an adapter from userspace.

Compilation & Usage Steps:

==========

Inorder to compile the user space application:

Method 1:-

1) Change the below SROOT path in the Makefile to your sysroots path
SROOT=/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux/

2) Run make command to compile the user code, after changing the above mentioned path.
make

3) copy the user app "usermain" from pwd to the board using the command:
sudo scp <filename> root@<inet_address>:/home/root

sudo will prompt for password: enter your root password.

4) Run the userapp inside the galileo board:
./usermain

5) **Always choose option 1 first -> EEPROM_init** inorder to correctly set gpio and slave address.
Select options from menu to perform necessary operations.

6) Always set the position of the pointer before reading, inorder to read exact position which was written earlier.

7) Initial position after EEPROM_init and EEPROM_erase will be 0x0000 or page 0 always

OR
Method 2:-
1) export the PATH of your SDK to the present directory:
export PATH="/opt/iot-devkit/1.7.3/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/:$PATH"

replace -> ***/opt/iot-devkit/1.7.3/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/***
with ***<your SDK path>***

2) Cross Compile the user app on the Linux PC to run on Galileo Gen2 board:
i586-poky-linux-gcc --sysroot=/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux usermain.c -o usermain -Wall

replace -> ***/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux***
with ***<you sysroots path>***

3) Follow steps 3 to step 7 from Method 1.

==========
==========

Part 2 - User app + Kernel driver "i2cflash" to access EEPROM

==========

It contains 4 files:

flashapp.c
i2c_flash.c
libioctl.h
Makefile

flashapp.c	->	This is a user space app to test kernel driver i2cflash to access an EEPROM. It defines the operations (read,write,getposition,setposition,erase,getstatus) to perform on The EEPROM. It asks the user to set position of page to read or write, and the number of pages to read or write. It creates a random string to write to the EEPROM of size specified by number of pages entered by user (number of pages * 64bytes per page). The read call waits for return value 0 otherwise the data will be lost.
i2c_flash.c	->	This is a device driver implemented in place of i2c-dev.c inorder to talk with the EEPROM. It implements the ioctl commands as defined above and read/write operations. The read/write are NON Blocking calls which queue the R/W tasks to the workqueue and return to the user. It also sets up gpio60 to low to enable SDA and SCL, gpio 74 to low to enable gpio10 input, gpio 10 initialised to low and gpio 26 direction set to OUT.
libioctl.h	->	This contains the ioctl commands definition.
Makefile	->	To make the target object file of the i2cflash driver.

Compilation & Usage Steps:

==========

Inorder to compile the driver kernel object file:

NOTE: The Galileo Gen2 board should be rebooted inorder for the GPIO pins to function correctly.

Method 1:-

1) Change the following KDIR path to your source kernel path and the SROOT path to your sysroots path in the Makefile

KDIR:=/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux/usr/src/kernel
SROOT=/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux/

2) Run make command to compile both user app and the i2c_flash kernel object.
make

3) copy the file from current dir to board dir:
sudo scp <filename> root@<inet_address>:/home/root

4) On the Galileo Gen2 board, insert the kernel module:
insmod i2c_flash.ko

5) Run the executable flashapp
./flashapp

6) Choose options from the menu to perform the desired operation and observe the LED glow during read, write and erase operations.

7) Remove module after use:
rmmod i2c_flash.ko

OR

Method 2:-
1) export the PATH of your SDK to the present directory:
export PATH="/opt/iot-devkit/1.7.3/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/:$PATH"

2) compile driver
make

3) Repeat steps 3 - step 7 from Method 1.

Do not Follow the below method if using Method 1 for compilation.

Inorder to compile the user space application:

Follow steps from 1 to 4 from above explanation to get the executable flashapp: replace usermain with flashapp

1) ***export PATH="/opt/iot-devkit/1.7.3/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/:$PATH"***
2) ***i586-poky-linux-gcc --sysroot=/opt/iot-devkit/1.7.3/sysroots/i586-poky-linux flashapp.c -o flashapp***
3) ***sudo scp <filename> root@<inet_address>:/home/root***
4) ***./flashapp***

==========
==========

Whether your library (part 1) and your driver (part 2) can be invoked by multiple user threads.

1) User Library

Yes the user library can be invoked by multiple user threads by doing the following.

- We can create multiple i2c-dev modules using mknod command **i2c-dev0, i2c-dev1, i2c-dev2**… which will have same Major number and different Minor numbers. There are 256 minor numbers reserved for i2c.
- Different user threads can open different i2c adapters as named above, and access the eeprom.
- The **read**() and **write**() calls must be wrapped around with a locking mechanism using mutex/semaphore.
- The current_page pointer updation must also be accessed using a lock, to protect from multiple users.

2) Driver

a) Yes the driver can be invoked by multiple user threads by implementing a locking mechanism for the shared resources.

- We declare the mutex/semaphore lock in the device structure.
- Each user need to lock this device structure while accessing the device which ensures that no member is shared between different users.
- We initialize the mutex in the __init module function.
- We add mutex lock and unlock around the following calls to protect the read and write from Multiple users, and also enqueueing and dequeueing work should be protected.
  i2c_master_send();
  i2c_master_read();
  write_work->page_count = count;
  queue_work(my_wq,(struct work_struct*)write_work);
  read_work->page_count = count;
  queue_work(my_wq,(struct work_struct*)read_work);

b) There can be multiple EEPROM chips that can be used by our driver.
They will have different page size and slave address.
So in our code we can keep a global variable which has the page size and another variable which holds the slave address. This can be changed only for different eeproms to be compatible, and then we can all these EEPROM to the device list.