Exercise 2 Accessing EEPROM via I2C bus

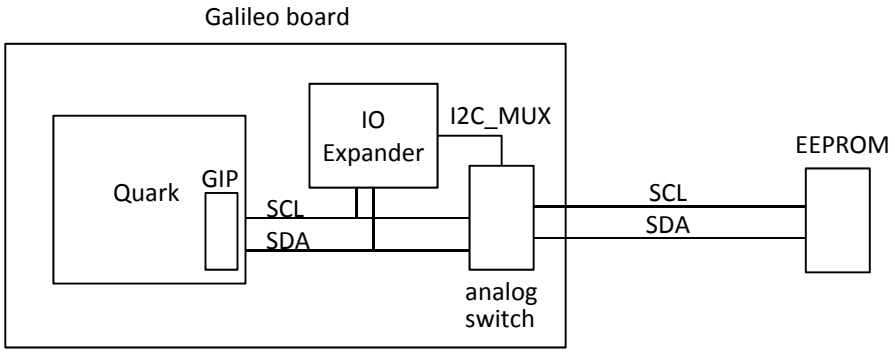
Exercise Objectives

1. To learn the basic programming technique in Linux i2c and gpio kernel modules.
2. To learn i2c and gpio driver architecture and software
3. To apply an application framework to control read/write operations of EEPROM.

Lab Assignment

Most operating systems provide memory mapping facility that allows users to directly access memory device by associating a range of user-space addresses to device memory. When a user program reads from or writes to the assigned address range, the OS performs the function by accessing either a buffered copy in memory or the memory device. This is so called the mmap device or the mmap abstraction which is built upon a device driver to read or write blocks from a memory device (such as a memory card or a frame buffer).

In this assignment, you will develop a driver to perform page read and write operations for an EEPROM. The memory device is a Microchip 24FC256 EEPROM which consists of 512 64-byte pages (or blocks) which can be accessed via an I2C bus. To connect an EEPROM chip, we will use SCL and SDA pins of the digital port on Galileo board. The signals are from the I2C_gpio controller of Quark processor. They are multiplexed and selected with the I2C_MUX signal of the Cypress IO expander on Galileo board. Thus, to enable the I2C bus access to the EEPROM, it is necessary to set up the correct gpio signal on the IO expander. The following diagram shows the connections to an external Microchip 24FC256 EEPROM. Note that the i2c address of the chip is set to 0x54 by connecting its A2 pin to high.



The 1st task of the assignment is to build an i2c client driver for the EEPROM device (*i2c_flash*) such that user programs can invoke read, write, and ioctl driver operations. You can assume the EEPROM chip has a fixed I2C address and is connected to a specific i2c bus. Thus, when the module is initialized, the EEPROM device should be created and named as "i2c_flash". The module should enable device file operations to support the following user-level calls:

1. *int open(const char *pathname, int flags);*
2. *ssize_t read(int fd, void *buf, size_t count); /*to read a sequence of count pages from the EEPROM device into the user memory pointed by buf. The pages to be read start from the current page position of the EEPROM. The page position is then advanced by count and, if reaching the end of pages, wrapped around to the beginning of the EEPROM. */*
3. *ssize_t write(int fd, const void *buf, size_t count); /* to write a sequence of count pages to an EEPROM device starting from the current page position of the EEPROM. The page position is then advanced by count and, if reaching the end of pages, wrapped around to the beginning of the EEPROM. */*

4. *int ioctl(int fd, unsigned long request, ...); /* if request=FLASHGETS, the call returns the status of the EEPROM (busy or not). If a call is with request=FLASHGETP or FLASHSETP, the call gets or sets the current page position with a third argument int *argp. Proper return code should be set for the call, including busy or out of memory. The 3rd request, when request=FLASHERASE and the EEPROM is not busy, is to trigger an erase operation to the EEPROM which write all 1's to all 512 pages. */*
5. *int close(int fd);*

where *count* and *offset* are page number of the EEPROM memory and is ranged from 0 to 511, and the size of the "*buf"* should be *64*count* bytes. Also, the read and write calls are blocking calls, i.e., they will be returned only when the operations are done (return 0) or have an error (return -1), e.g., the EEPROM is busy. Hence, the calling user thread is blocked while the requested operation is in progress.

The 2nd task of the lab assignment is to revise your *i2c_flash* device driver of task 1 such that the read and write calls are non-blocking. To provide the asynchronous processing of EEPROM functions, a work queue will be created when the driver module *i2c-flash* is installed. When a read system call is requested to the EEPROM device, the driver takes one of the following operations:

- If no EEPROM data is ready for read and the EEPROM is not busy, the driver submits a read transfer request to the work queue and returns immediately with -1 and an error code ERRNO=EAGAIN.
- If no EEPROM data is ready for read and the EEPROM is busy, the call returns immediately with -1 and an error code ERRNO=EBUSY.
- If EEPROM data is ready for read, return with 0 after the data of 64*count bytes is copied to the buffer.

On the other hand, when a write system call is requested, the driver:

- Returns immediately with 0 if the EEPROM is not busy and the write operation will be performed by the driver subsequently.
- If the EEPROM is busy, returns immediately with -1 and an error code ERROR=EBUSY.

Note that, instead of using the work queue mechanism, you may choose to use kernel threads to handle non-blocking calls.

In addition to the operations on the EEPROM, your driver should light up a led whenever a read, write, or erase operation is in progress, i.e. the EEPROM is busy. The led should be connected to Galileo's digital IO pin IO8. To test your driver, you will need to develop a user application that can exercise all driver functions.

[Additional requirement for CSE 598 students]

A one-page report should be included in the submission to discuss possible extension of your driver implementation such that it can

1. accept calls from multiple user threads.
2. work on a different EEPROM chip (different number of pages and i2c slave address) which is connected via one of the i2c buses in a target board.

Due Date

This is assignment will be due at 11:59pm on Oct. 10.

What to Turn in for Grading

- Create a working directory to include your source files, makefiles, readme, and a one-page report (for CSE 598). Your source files should include the main program *main_2.c* (the user-level

test program), the driver i2c_*flash.c* , and any other .c or .h files for the assignment. (please clean up all object code from your submission and we will recompile your code using your makefiles.)

- Comment your source files properly and rewrite the readme file to describe how to use your software.
- Compress the directory into a zip archive file named cse438(598)-lastname-assgn02.zip. Note that any object code or temporary build files should not be included in the submission.
- Submit the zip archive to Blackboard by the due date and time.
- Failure to follow these instructions may cause an annoyed and cranky TA or instructor to deduct points while grading your assignment.