EE414 Embedded Systems
Lab 2. LED Driver

Due Demo & Report: 11:59 PM, Apr 9, Fri. via KLMS

1. Purpose

Understand how to drive the GPIO LEDs in Beaglebone.
Target board: Beaglebone (containing 4 GPIO LEDs) with Linux.
Host computer: PC with Linux, cross compiler, NFS, and minicom.
Note that this experiment is for driving LED device on Beaglebone in user space (not kernel space).

2. Problem Statement

Problem 2 (LED Display)

Write a program for LED display output on the embedded board, to output beats of the metronome visually using LED lamps with fixed tempo of 60 and fixed time-signature of 6/8.

LED Output

Beats (LED lamps on) with respect to time	
Period	1 Hz (Tempo of 60 bpm).
Number of LED lamps	4.
Number of LED lamp output	Controlled by Time signature input.
Time signature 6/8:	3 1 1 2 1 1 ; 3 1 1 2 1 1 ; …

Which means "strong, weak, weak, medium, weak, weak" in one measure.

Duty 50 %

LED display pattern ('O' means on, 'X' means off. One column for 0.5 s):

```
LED 4: X X X X X X X X X X X X   X X X X X X X X X X X X  …
LED 3: O X X X X X X X X X X X   O X X X X X X X X X X X  …
LED 2: O X X X X X O X X X X X   O X X X X X O X X X X X  …
LED 1: O X O X O X O X O X O X   O X O X O X O X O X O X  …
```

Note. The various user-configurable tempo and time-signature will be implemented in the future.

Terminology

Music(al) notation (music sheet, score): Visual representation of music played or sung.
Measure: A unit in music notation separated with "bar".
Musical note: Denote duration of a tone. Whole, half, quarter, eighth, etc.
Time signature: Number of musical notes in a measure / Representative musical note.
Tempo: Absolute duration of the representative musical note.
Metronome: Any device that produces regular, metrical ticks (beats, clicks)

3. Technical Backgrounds

A. Hardware connection

User I/O ←→ PC ←→ Serial/USB connection ←→ Embedded board → GPIO → LEDs

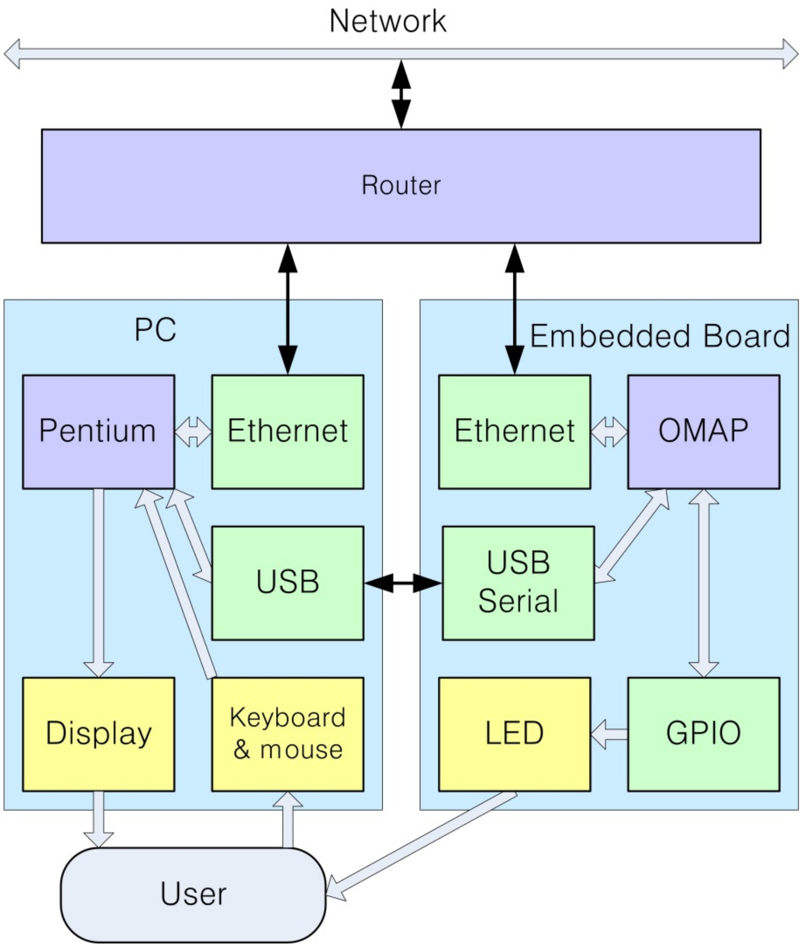


Fig. 2.1 Block Diagram for Lab 2

B. Software Connection

You require two programs:

1) Minicom on the PC: The same as used in Lab 1.
2) Metronome_led on Embedded Board: Application program.

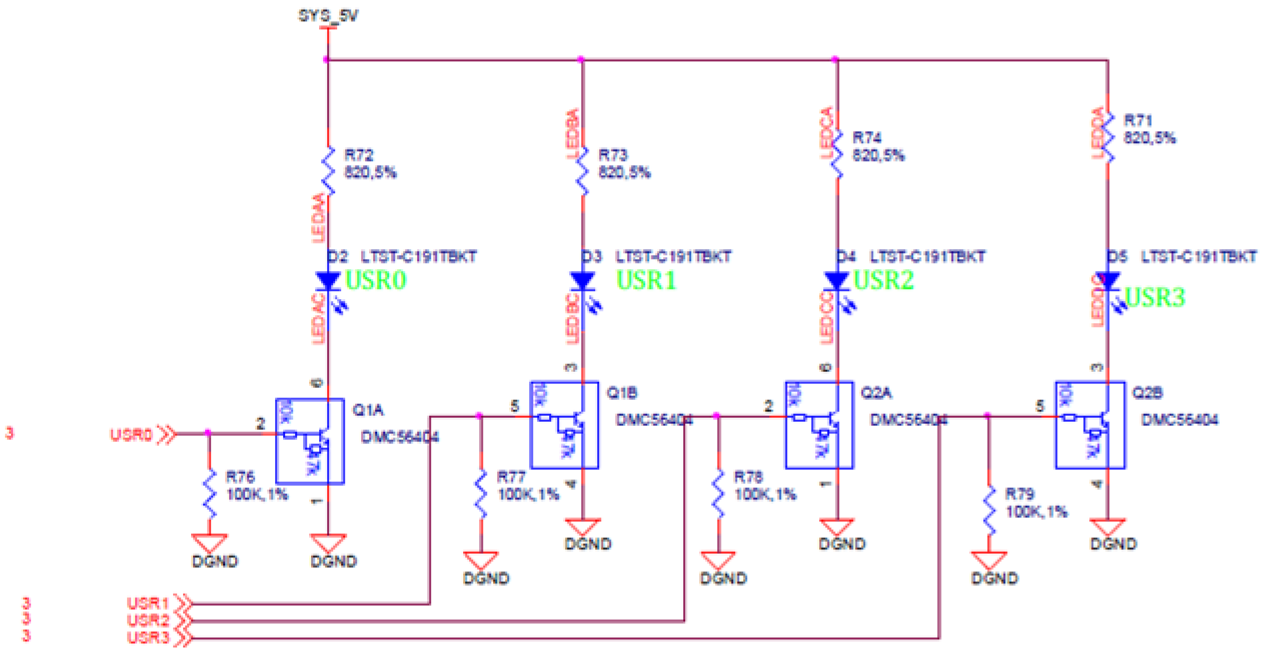
Metronome_led on embedded board controls four user LEDs on Beaglebone.
Minicom on the PC can be used for displaying messages from your application program.

C. LED hardware circuit in Beaglebone

[Refer Beaglebone Black System Reference Manual A5.2 p. 56]

User LEDs

Four user LEDS are provided via GPIO pins on the processor. Figure 2.2 below shows the LED circuitry.



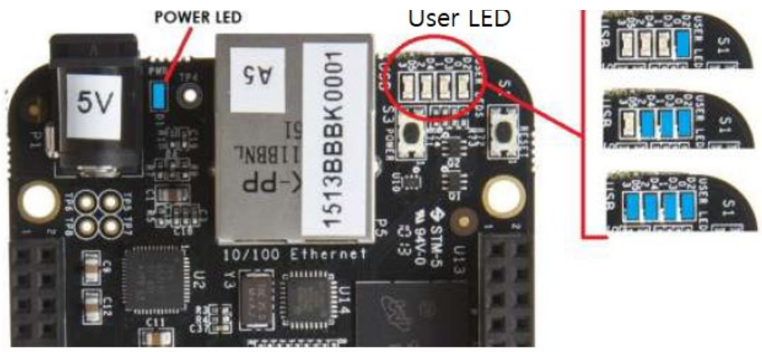


Fig 2.2 User LEDS in Beaglebone

LED	GPIO
User 0	GPIO1_21
User 1	GPIO1_22
User 2	GPIO1_23
User 3	GPIO1_24

Table 5. User LED Control

D. GPIO (General Purpose I/O)

Internal block diagram of AM3359 processor in BeangeBone is shown in Fig. 2.3. You can see "GPIO" under Parallel input/output blocks.

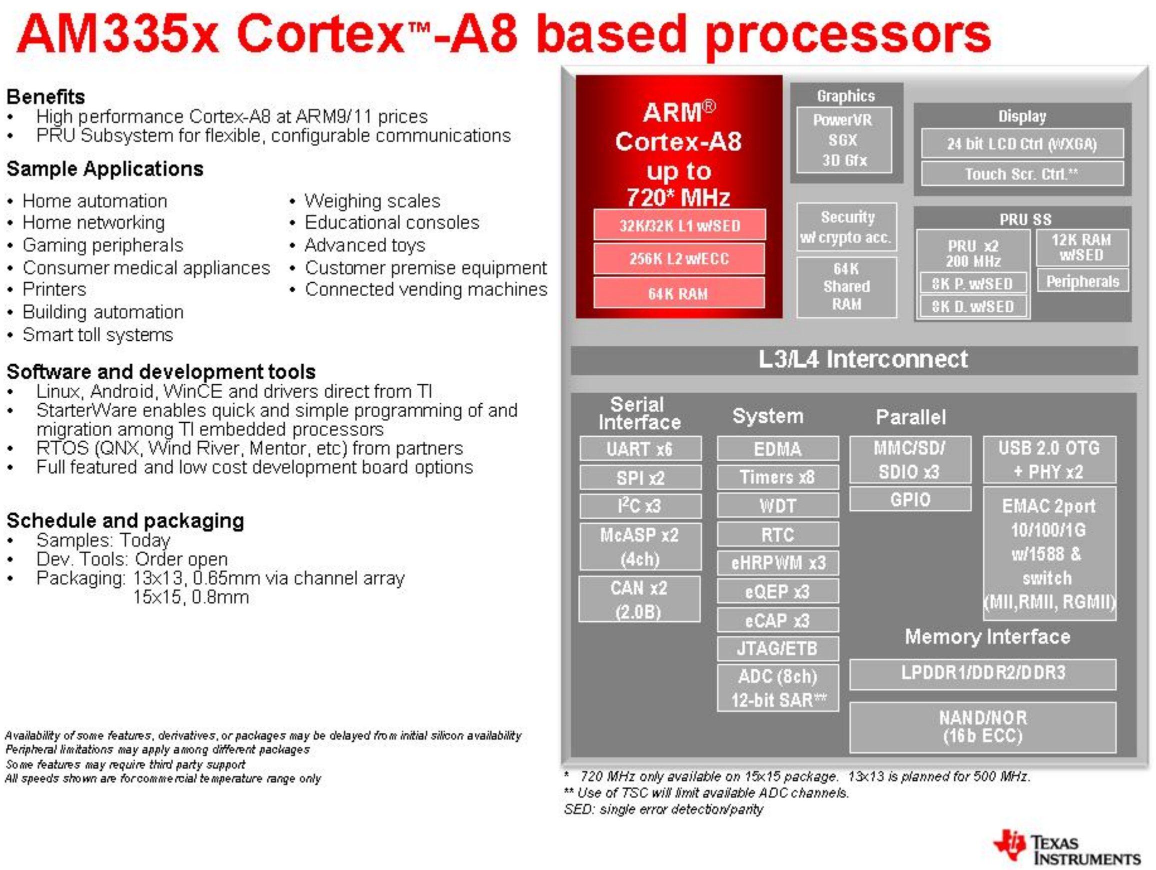


Fig. 2.3 Internal block diagram of AM3359 Processor.
[Refer http://www.ti.com/product/am3359]

Purpose of GPIO peripheral in AM335x processor

The general-purpose interface combines four general-purpose input/output (GPIO) modules. Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities; thus, the general-purpose interface supports up to 128 (4 × 32) pins. These pins can be configured for the following applications:

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell
- Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.

GPIO Features

Each GPIO module is made up of 32 identical channels. Each channel can be configured to be used in the following applications:

- Data input/output
- Keyboard interface with a de-bouncing cell
- Synchronous interrupt generation (in active mode) upon the detection of external events (signal transition(s) and/or signal level(s))
- Wake-up request generation (in Idle mode) upon the detection of signal transition(s)

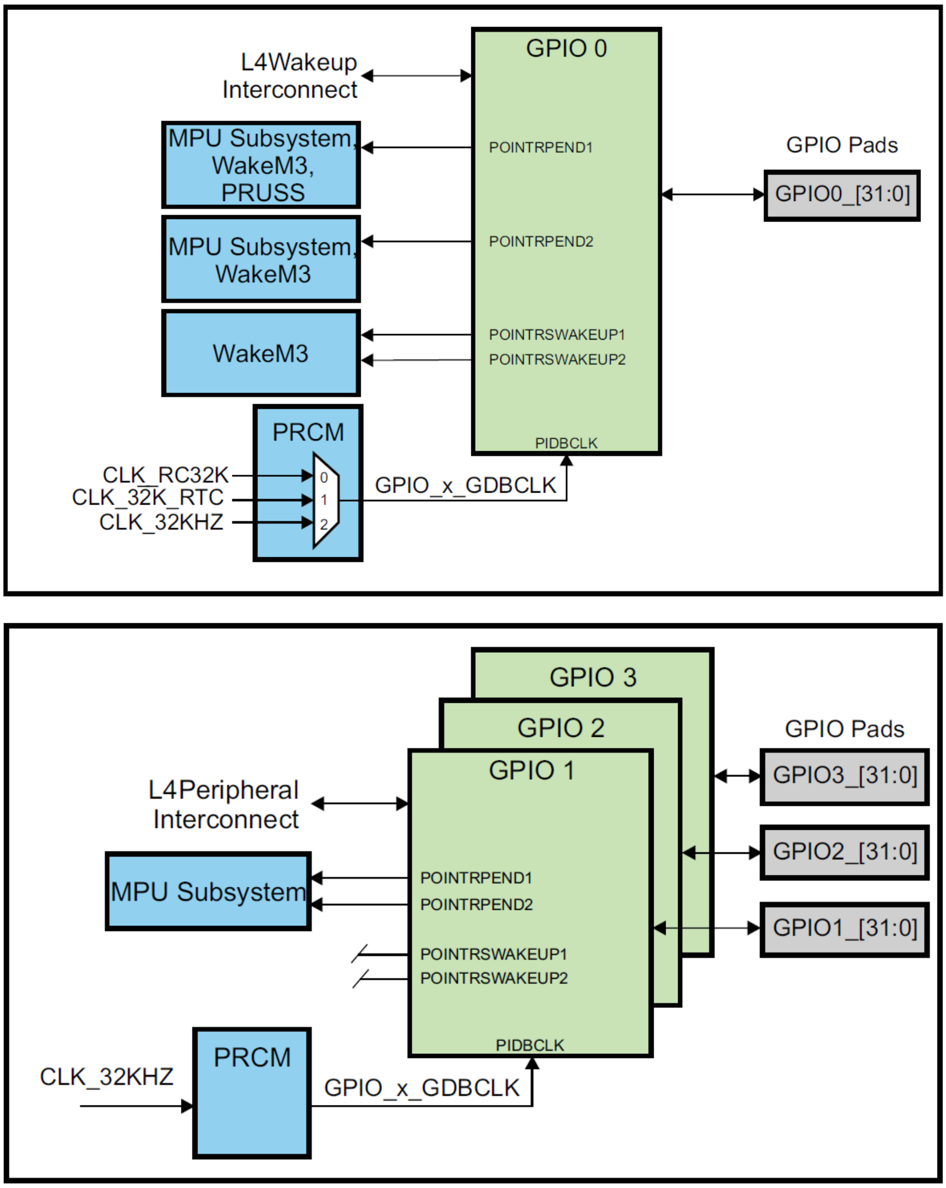
Global features of the GPIO interface are:

- Synchronous interrupt requests from each channel are processed by two identical interrupt generation sub-modules to be used independently by the ARM Subsystem
- Wake-up requests from input channels are merged together to issue one wake-up signal to the system

Integration

The device instantiates four GPIO_V2 modules. Each GPIO module provides the support for 32 dedicated pins with input and output configuration capabilities. Input signals can be used to generate interruptions and wake-up signal. Two Interrupt lines are available for bi-processor operation. Pins can be dedicated to be used as a keyboard controller.

With four GPIO modules, the device allows for a maximum of 128 GPIO pins. (The exact number available varies as a function of the device configuration and pin muxing.) GPIO0 is in the Wakeup domain and may be used to wakeup the device via external sources. GPIO[1:3] are located in the peripheral domain.



General-Purpose Interface Basic Programming Model

Set and Clear Instructions

The GPIO module implements the set-and-clear protocol register update for the data output and interrupt enable registers. This protocol is an alternative to the atomic test and set operations and consists of writing operations at dedicated addresses (one address for setting bit[s] and one address for clearing bit[s]). The data to write is 1 at bit position(s) to clear (or to set) and 0 at unaffected bit(s).

Registers can be accessed in two ways:

- Standard: Full register read and write operations at the primary register address
- Set and clear (recommended): Separate addresses are provided to set (and clear) bits in registers. Writing 1 at these addresses sets (or clears) the corresponding bit into the equivalent register; writing a 0 has no effect.

Therefore, for these registers, three addresses are defined for one unique physical register. Reading these addresses has the same effect and returns the register value.

Refer Technical Reference Manual, pp. 4513 – 4515 for further details.

E. Device Driver

Device Driver Concepts [4]

One of the fundamental purposes of a *device driver* is to *isolate the user's programs from ready access to critical kernel data structures and hardware devices*. Furthermore, a well-written device driver hides the complexity and variability of the hardware device from the user. For example, a program that wants to write data to the hard disk need not care if the disk drive uses 512-byte or 1024-byte sectors. The user simply opens a file and issues a write command.

The device driver *handles the details and isolates the user from the complexities and perils of hardware device programming*. The *device driver provides a consistent user interface* to a large variety of hardware devices. It provides the basis for the familiar UNIX/Linux convention that everything must be represented as a *file.*

Loadable Modules

Unlike some other operating systems, Linux has the capability to add and remove kernel components at runtime. Linux is structured as a monolithic kernel with a well-defined interface for adding and removing device driver modules dynamically after boot time. This feature not only adds flexibility to the user, but it has proven invaluable to the device driver development effort. Assuming that your device driver is reasonably well behaved, you can insert and remove the device driver from a running kernel at will during the development cycle instead of rebooting the kernel every time a change occurs.

Loadable modules have particular importance to embedded systems. Loadable modules enhance field upgrade capabilities; the module itself can be updated in a live system without the need for a reboot. Modules can be stored on media other than the root (boot) device, which can be space constrained.

Driver File System Operations

Recall that module_init() is used for module initialization (with 'insmod' command), and module_exit() is used for module exit (with 'rmmod' command). Now we need some methods to interface with our device driver from our application program.

After the device driver is loaded into a live kernel, the first action we must take is to prepare the driver for subsequent operations. The open() method is used for this purpose. After the driver has been opened, we need routines for reading and writing to the driver. A release() routine is provided to clean up after operations when complete (basically, a close call). Finally, a special system call 'ioctl()' is provided for nonstandard communication to the driver.

F. GPIO LED

[Hardware Interfacing on the Beaglebone, http://www.nathandumont.com/node/250]

Pin mux

The Beaglebone has far more peripherals than pins, despite being a BGA package. To get around this it assigns different functionality to each pin based on software selections like most other modern microcontrollers. The Beaglebone actually has a more flexible arrangement than previous BeagleBoard variants, allowing run-time customisation of most of the I/O pins from a set of files under the /sys/ folder. If you take a look at the Beaglebone System Reference Manual starting on page 48 is a description of the I/O pins. There are up to 66 3.3V GPIO pins available with several I²C, SPI and UART interfaces as well as timers and PWM etc available as options instead of GPIO on certain pins. There are a group of ADC pins as well but be careful the ADCs are only 1.8V tolerant so watch how many volts you put across them! The initial connector pinout mentions the most likely pin mode for each pin, but on the following pages each pin has details of the up to 8 functions available on each pin. The pin name as far as the kernel is concerned is always the MODE0 function, which, frustratingly, isn't always the "Signal Name" given in the connector pinout.

GPIO Pins

There are a total of 66 GPIO pins available on the Beaglebone making it a very capable controller for a lot of things. Using them without writing a Kernel module relies on toggling individual pins one at a time via control files though, so don't plan on driving big wide parallel busses or anything without significant effort!

GPIO Hardware Overview

Refer AM335X Technical Reference Manual (in pdf form), Ch 25. General-Purpose Input/Output (p. 4505)

Purpose of the Peripheral

The general-purpose interface combines four general-purpose input/output (GPIO) modules. Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities; thus, the general-purpose interface supports up to 128 (4 × 32) pins. These pins can be configured for the following applications:

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell
- Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.
- Wake-up request generation (in Idle mode) upon the detection of signal transition(s)

GPIO Features

Shared registers can be accessed through “Set & Clear” protocol. The wake-up feature of the GPIO modules is only supported on GPIO0.

Integration

See Figs in p. 4507, TRM

Clocks

GPIO module runs using two clocks:

- The debouncing clock is used for the debouncing sub-module logic (without the corresponding configuration registers). This module can sample the input line and filters the input level using a programmed delay.
- The interface clock provided by the peripheral bus (OCP compatible system interface). It is used through the entire GPIO module (except within the debouncing sub-module logic). It clocks the OCP interface and the internal logic. Clock gating features allow adapting the module power consumption to the activity.

Set and Clear Instructions

The GPIO module implements the set-and-clear protocol register update for the data output and interrupt enable registers. This protocol is an alternative to the atomic test and set operations and consists of writing operations at dedicated addresses (one address for setting bit[s] and one address for clearing bit[s]). The data to write is 1 at bit position(s) to clear (or to set) and 0 at unaffected bit(s).

Registers can be accessed in two ways:

- Standard: Full register read and write operations at the primary register address
- Set and clear (recommended): Separate addresses are provided to set (and clear) bits in registers.

Writing 1 at these addresses sets (or clears) the corresponding bit into the equivalent register; writing a 0 has no effect.

Therefore, for these registers, three addresses are defined for one unique physical register. Reading these addresses has the same effect and returns the register value.

Clear Data Output Register (GPIO_CLEARDATAOUT):

- A write operation in the clear data output register clears the corresponding bit in the data output register when the written bit is 1; a written bit at 0 has no effect.
- A read of the clear data output register returns the value of the data output register.

Set Data Output Register (GPIO_SETDATAOUT):

- A write operation in the set data output register sets the corresponding bit in the data output register when the written bit is 1; a written bit at 0 has no effect.
- A read of the set data output register returns the value of the data output register.

Data Input (Capture)/Output (Drive)

The output enable register (GPIO_OE) controls the output/input capability for each pin. At reset, all the GPIO-related pins are configured as input and output capabilities are disabled. This register is not used within the module; its only function is to carry the pads configuration.

<u>When configured as an output (the desired bit reset in GPIO_OE)</u>, the value of the corresponding bit in the GPIO_DATAOUT register is driven on the corresponding GPIO pin. Data is written to the data output register synchronously with the interface clock. This register can be accessed with read/write operations or by using the alternate set and clear protocol register update feature. This feature lets you set or clear specific bits of this register with a single write access to the set data output register (GPIO_SETDATAOUT) or to the clear data output register (GPIO_CLEARDATAOUT) address. If the application uses a pin as an output and does not want interrupt generation from this pin, the application must properly configure the interrupt enable registers.

<u>When configured as an input (the desired bit set to 1 in GPIO_OE)</u>, the state of the input can be read from the corresponding bit in the GPIO_DATAIN register. The input data is sampled synchronously with the interface clock and then captured in the data input register synchronously with the interface clock. When the GPIO pin levels change, they are captured into this register after two interface clock cycles (the required cycles to synchronize and to write data). If the application uses a pin as an input, the application must properly configure the interrupt enable registers to the interrupt as needed.

G. How to control user LEDs?

We test two methods:

1) Use sysfs with either commands or shell script
2) Use C with mmap()

   #include <sys/mman.h>

   mmap()/munmap()

   Map or unmap files or devices into memory.

   void *mmap(void ****addr*, size_t** *length***, int** *prot***, int** *flags***, int** *fd***, off_t** *offset***);**

   int munmap(void ****addr*, size_t** *length***);**

4. Design and Preparation

1. Search and summarize GPIO with sysfs.

2. Design: Control user LEDs with mmap
   Study example program: pushLEDmmap.c & pushLEDmmap.h
   Change to userLEDmmap.c & userLEDmmap.h

   In userLEDmmap.h, you need to include GPIO definitions for user LEDs.

3. Design: Metronome_led.c to play metronome using user LEDs.
   You may reuse userLEDsmmap.h.
   You may use the following algorithm for userLEDmmap.c:

Algorithm for userLEDmmap

- Set global P = [7, 1, 1, 3, 1, 1]; // LED bit pattern to display 3, 2, or 1 lamps
- Set sginal callback for Ctrl+C
- Init mmap - user_leds_setup()
- Set parameters
  - Let Time_signature = 6/8.
  - Let Tempo = 60.
- Repeat the following
  - Repeat the following 6 times (i=0, … , 5)
    - Write_GPIO_LED(P(i)) // Turn LED with pattern P(i)
    - Sleep n ms // Depending on Tempo
    - Write_GPIO_LED(0) // Turn all LEDs off
    - Sleep n ms // Depending on Tempo
  - End repeat i
- End repeat
- Print Quit message
- Release mmap - release_user_leds()

5. Lab Procedures

Contents

Step 0. Setup cross development environment for module

Note. Step 0 is already done in Lab 1, but should be repeated also after power on.

0.1 Connection

PC --- Ethernet cable --- Router – Ethernet cable --- Beaglebone --- Power adaptor

After power on Beaglebone:

PC ------------------------ USB cable --------------------- Beaglebone

0.2 Start PC NFS server

To start the NFS server, you can run the following command at a terminal prompt:

```
$ sudo /etc/init.d/nfs-kernel-server start
$ sudo service nfs-kernel-server start
```

Check if nfs is started.

```
$ ps -aux | grep nfs
…...
root      3005  0.0  0.0     0    0 ?     S    17:51   0:00 [nfsd]
```

0.3 Connect to Beaglebone

Use Minicom

```
$ sudo minicom –w
```

or ssh

```
$ ssh debian@192.168.7.2         // ssh via USB
$ ssh debian@192.168.0.4         // If Bone IP is known
```

Log in to Beaglebone with id 'debian' and passwd 'temppwd'. You should use your own user id/passwd created in the Lab 1.

0.4 Start nfs client on Beaglebone

Start nfs-client on Bealgebone

```
# sudo bash bone_nfs_client.sh
```

Check mounted folder

```
# ls /<user_id>/nfs_client
```

Step 1. Test LEDs using sysfs and commands

1.1 Select LEDs for test

Note that four LEDs on Beaglebone are connected to GPIOs as follows:

LED	GPIO
User 0	GPIO1_21
User 1	GPIO1_22
User 2	GPIO1_23
User 3	GPIO1_24

Table 5. User LED Control

Selection: Test User 0 LED.

Since LED0 is used as heart beat signal for Ubuntu, we are going to stop the heartbeat, and perform our experiment.

1.2 Check sysfs file

The control files are all contained in Beaglebone file system:

```
# ls -F /sys/class/gpio
export   gpio13@     gpiochip32@  gpiochip96@
gpio12@  gpiochip0@  gpiochip64@  unexport
```

Browse the contents of gpiochip32 (containing GPIO1_0 to 31)

```
# cd /sys/class/gpio/gpiochip32
# ls -F
base  device@  label  ngpio  power/  subsystem@  uevent
```

Note

```
# ls -la /sys/class/gpio/gpiochip32
lrwxrwxrwx 1 root gpio 0 Sep 10 02:54 /sys/class/gpio/gpiochip32 ->
       ../../devices/platform/ocp/4804c000.gpio/gpio/gpiochip32
# ls -F /sys/devices/platform/ocp/4804c000.gpio/gpio/gpiochip32
base  device@  label  ngpio  power/  subsystem@  uevent
```

Hence the actual location of gpiochip32 is /sys/devices/platform/ocp/4804c000.gpio/gpio/gpiochip32.

1.3 Find LED sysfs

Search /sys/class:

```
# ls -F /sys/class
ata_device/     dvb/           mbox/           rc/            tpm/
ata_link/       extcon/        mdio_bus/       regulator/     tpmrm/
ata_port/       firmware/      mem/            remoteproc/    tty/
backlight/      gpio/          misc/           rfkill/        typec/
bdi/            graphics/      mmc_host/       rtc/           ubi/
block/          hidraw/        mtd/            scsi_device/   udc/
bsg/            hwmon/         net/            scsi_disk/     uio/
devcoredump/    i2c-adapter/   phy/            scsi_host/     vc/
devfreq/        i2c-dev/       power_supply/   sound/         video4linux/
devfreq-event/  input/         pps/            spidev/        vtconsole/
```

```
dma/              iommu/        ptp/          spi_master/   watchdog/
drm/              leds/         pwm/          thermal/
```

Found “leds”!

Search /sys/class/leds:

```
# ls -F /sys/class/leds
beaglebone:green:usr0@  beaglebone:green:usr2@
beaglebone:green:usr1@  beaglebone:green:usr3@
```

Here you see the directories for controlling each of the user LEDs. By default, usr0 flashes a heartbeat pattern and usr1 flashes when the micro SD card is accessed. Let's control usr0.

1.4 Get access right to usr0 LED

Go to the directory /sys/class/leds

```
# cd /sys/class/leds
# cd beaglebone\:green\:usr0
```

Note that '\ ' should be included before each ':'.

```
# ls -F
brightness  invert         power/      trigger
device@     max_brightness subsystem@  uevent
```

See what's in trigger

```
# cat trigger
none rc-feedback rfkill-any kbd-scrolllock kbd-numlock kbd-capslock kbd-
kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shif
tllock kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock usb-gadget usb-host mmc0
mmc1 timer oneshot disk-activity ide-disk mtd nand-disk [heartbeat]
backlight gpio cpu cpu0 default-on panic
```

This shows trigger can have many values. The present value is **heartbeat** (enclosed with '[]'). Check the LED, is it beating?

You can stop the heartbeat via:

```
# echo none > trigger
```

Heartbeat is stopped! Check:

```
# cat trigger
[none] rc-feedback kbd-scrollock kbd-numlock kbd-capslock kbd-kanalock
kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shiftllock kbd-
shiftrlock kbd-ctrlllock kbd-ctrlrlock nand-disk usb-gadget usb-host timer
oneshot heartbeat backlight gpio default-on mmc0
```

1.5 Control on/off of usr0 LED

Turn on/off usr0 LED

```
# echo 1 > brightness
# echo 0 > brightness
```

Usr0 LED is turned on and off!

1.6 Control periodic on/off of usr0 LED

LED trigger with timer and 10% duty:

```
# echo timer > trigger
# echo 100 > delay_on
# echo 900 > delay_off
```

Observe period and duty of User LED0.

Step 2. Test LEDs with shell script

2.1 Make a subdirectory b_gpio_led_shell.

Make a subdirectory

```
$ mkdir -p ~/Embedded/lab2/d_gpio_led_shell
$ cd ~/Embedded/lab2/d_gpio_led_shell
```

2.2 Edit control_led0.sh

You can edit a shell script filename.sh for a sequence of commands.
Edit control_led0.sh.

```
#!/bin/bash
# Shell program to control LED0 with on_time & off_time
#
# Check for two arguments
# $# means the number of arguments
if [ $# -lt 2 ]
  then
    echo "Usage: control_led0.sh on_time off_time"
    exit
fi

# Control LED0
echo "Control LED0 with on_time $1 and off_time $2"
cd /sys/class/leds
cd beaglebone\:green\:usr0
echo none > trigger
echo timer > trigger
echo $1 > delay_on
echo $2 > delay_off
```

Make this shell script executable with 'chmod'.

```
# cd /root/nfs_client/lab2/d_gpio_led_shell
# chmod a+x control_led0.sh
```

2.3 Run shell script.

Run shell script on Bone. We need two arguments: on_time and off_time (in ms).

```
# ./control_led0.sh 500 1000
```

Note on shell.
"echo $SHELL" returns
/bin/bash // on Ubuntu PC & Bone.
Hence we are using bash.

Step 3. Control LED with C and mmap

3.1 Make a working directory

$ mkdir -p ~/Embedded/lab2/i_mmap
$ cd ~/Embedded/lab2/i_mmap

3.2 Build shell script to stop & restore user LEDs

Edit shell script "stop_user_leds.sh" to stop user LEDs used by kernel

```
#!/bin/bash
# stop_user_leds.sh
# Shell program to stop user LEDs

# Stop user LED0
cd /sys/class/leds
cd beaglebone\:green\:usr0
echo none > trigger
echo "Stop user LED0"

cd ../beaglebone\:green\:usr1
echo none > trigger
echo "Stop user LED1"

cd ../beaglebone\:green\:usr2
echo none > trigger
echo "Stop user LED2"

cd ../beaglebone\:green\:usr3
echo none > trigger
echo "Stop user LED3"
```

Edit shell script "restore_user_leds.sh" to restore user LEDs to be used by kernel

```
#!/bin/bash
# restore_user_leds.sh
# Shell program to stop user LEDs

# Stop user LED0
cd /sys/class/leds
cd beaglebone\:green\:usr0
echo heartbeat > trigger
echo "Restore user LED0"

cd ../beaglebone\:green\:usr1
echo mmc0 > trigger
echo "Restore user LED1"

cd ../beaglebone\:green\:usr2
echo cpu0 > trigger
echo "Restore user LED2"

cd ../beaglebone\:green\:usr3
echo mmc1 > trigger
echo "Restore user LED3"
```

Make them executable.

 # chmod a+x stop_user_leds.sh

```
# chmod a+x restore_user_leds.sh
```

Stop user LEDs

```
# ./stop_user_leds.sh
```

Restore user LEDS

```
# ./restore_user_leds.sh
```

3.3 Test example program: pushLEDmmap.c with pushLEDmmap.h

Hardware

Pushbutton (additional device, not on the board) > ECAPPWM0 as GPIO_07
UART4 TXD as GPIO0_31 > LED (additional device, not on the board)

About pushLEDmmap code

If you push the button (connected through GPIO_07), the LED (connected through GPIO0_31) will turn on. But in our experiment, it will not work because GPIO0_07 and GPIO0_31 are not connected to external devices like push button and LED.

Edit pushLEDmmap.h [Beaglebone Cookbook, Yoder, O'Reily 2015]

```
// From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
// -through-dev-mem
// user contributions licensed under cc by-sa 3.0 with attribution required
// http://creativecommons.org/licenses/by-sa/3.0/
// http://blog.stackoverflow.com/2009/06/attribution-required/
// Author: madscientist159
(http://stackoverflow.com/users/3000377/madscientist159)

#ifndef _BEAGLEBONE_GPIO_H_
#define _BEAGLEBONE_GPIO_H_

#define GPIO0_START_ADDR 0x44e07000
#define GPIO0_END_ADDR   0x44e08000
#define GPIO0_SIZE (GPIO0_END_ADDR - GPIO0_START_ADDR)

#define GPIO1_START_ADDR 0x4804C000
#define GPIO1_END_ADDR   0x4804D000
#define GPIO1_SIZE (GPIO1_END_ADDR - GPIO1_START_ADDR)

#define GPIO2_START_ADDR 0x41A4C000
#define GPIO2_END_ADDR   0x41A4D000
#define GPIO2_SIZE (GPIO2_END_ADDR - GPIO2_START_ADDR)

#define GPIO3_START_ADDR 0x41A4E000
#define GPIO3_END_ADDR   0x41A4F000
#define GPIO3_SIZE (GPIO3_END_ADDR - GPIO3_START_ADDR)

#define GPIO_DATAIN 0x138
#define GPIO_SETDATAOUT 0x194
#define GPIO_CLEARDATAOUT 0x190

#define GPIO_07  (1<<7)
#define GPIO_31  (1<<31)
#endif
```

Edit pushLEDmmap.c [Beaglebone Cookbook, Yoder, O'Reily 2015]

```
// From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
// -through-dev-mem
```

```
// user contributions licensed under cc by-sa 3.0 with attribution required
// http://creativecommons.org/licenses/by-sa/3.0/
// http://blog.stackoverflow.com/2009/06/attribution-required/
// Author: madscientist159
(http://stackoverflow.com/users/3000377/madscientist159)
//
// Read one gpio pin and write it out to another using mmap.
// Be sure to set -O3 when compiling.
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>    // Defines signal-handling functions (i.e. trap Ctrl-C)
#include <unistd.h>        // close()
#include "pushLEDmmap.h"

// Global variables
volatile int keepgoing = 1;    // Set to 0 when Ctrl-c is pressed

// Callback called when SIGINT is sent to the process (Ctrl-C)
void signal_handler(int sig) {
    printf( "\nCtrl-C pressed, cleaning up and exiting...\n" );
       keepgoing = 0;
}

int main(int argc, char *argv[]) {
    volatile void *gpio_addr;
    volatile unsigned int *gpio_datain;
    volatile unsigned int *gpio_setdataout_addr;
    volatile unsigned int *gpio_cleardataout_addr;

    // Set the signal callback for Ctrl-C
    signal(SIGINT, signal_handler);

    int fd = open("/dev/mem", O_RDWR);

    printf("Mapping %X - %X (size: %X)\n", GPIO0_START_ADDR, GPIO0_END_ADDR,
                                           GPIO0_SIZE);

    gpio_addr = mmap(0, GPIO0_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
                        GPIO0_START_ADDR);

    gpio_datain            = gpio_addr + GPIO_DATAIN;
    gpio_setdataout_addr   = gpio_addr + GPIO_SETDATAOUT;
    gpio_cleardataout_addr = gpio_addr + GPIO_CLEARDATAOUT;

    if(gpio_addr == MAP_FAILED) {
        printf("Unable to map GPIO\n");
        exit(1);
    }
    printf("GPIO mapped to %p\n", gpio_addr);
    printf("GPIO SETDATAOUTADDR mapped to %p\n", gpio_setdataout_addr);
    printf("GPIO CLEARDATAOUT mapped to %p\n", gpio_cleardataout_addr);

    printf("Start copying GPIO_07 to GPIO_31\n");
    while(keepgoing) {
       if(*gpio_datain & GPIO_07) {          //Check whether button is pushed
             *gpio_setdataout_addr= GPIO_31; //LED on
       } else {
```

```
            *gpio_cleardataout_addr = GPIO_31; //LED off
       }
         //usleep(1);
    }

    munmap((void *)gpio_addr, GPIO0_SIZE);
    close(fd);
    return 0;
}
```

Edit Makefile, and make (cross-compile) on PC

$ make m1

arm-linux-gnueabihf-gcc -o pushLEDmmap pushLEDmmap.c

Run on Bone via NFS

```
# ./pushLEDmmap
Mapping 44E07000 - 44E08000 (size: 1000)
GPIO mapped to 0xb6f23000
GPIO SETDATAOUTADDR mapped to 0xb6f23194
GPIO CLEARDATAOUT mapped to 0xb6f23190
Start copying GPIO_07 to GPIO_31
^C
Ctrl-C pressed, cleaning up and exiting…
```

Press Ctrl+C to stop.

3.4 Test prepared userLEDmmap.c

Objective: Control four user LEDs on Beaglebone.

Copy&Paste pushLEDmmap.c and pushLEDmmap.h to create userLEDmmap.c and userLEDmmap.h

Change both .c and .h files to control four user LEDs, instead of the push button and the external LED, used in the tutorial example. You may decide not to change the .h file.

Check four user LEDs can be controlled (in an arbitrary manner). For example, you may try to blink the first user LED. **In this step, just check the mapping works.**

Make on PC

$ make m2

arm-linux-gnueabihf-gcc -o userLEDmmap userLEDmmap.c

Run on Bone Debian

 # ./stop_user_leds.sh

Stop user LED0

Stop user LED1

Stop user LED2

Stop user LED3

 # ./userLEDmmap

GPIO1 at 4804c000 is mapped to 0xb6f6d000

…………… // Are user LEDs working correctly?

 # ./restore_user_leds.sh

Restore user LED0

Restore user LED1

Restore user LED2
Restore user LED3

Is it working correctly?

Step 4. Test Metronome_led application program

4.1 Make a working directory

Make a suitable working directory, for example:

```
$ mkdir ~/Embedded/lab2/k_metro_user_leds
$ cd ~/Embedded/lab2/k_metro_user_leds
```

Copy stop_user_leds.sh & restore_user_leds.sh to this directory.

4.2 Edit prepared Metronome_led.c

Copy & Paste the userLEDmmap.c to create Metronome_led.c

Make sure that you set the correct header file: userLEDmmap.h.

In this step, you have to make the pattern, which we've specified in the [2. Problem Statement] section.

You may find [4. Design and Preparation] section helpful for the implementation.

4.3. Make

Edit prepared Makefile and make

```
$ make m3
arm-linux-gnueabihf-gcc -o Metronome_led Metronome_led.c
```

4.4 Test on Beaglebone.

Stop user LEDS

```
# ./stop_user_leds.sh
```

Run Metronome_led

```
# ./Metronome_led
```

Is it working correctly?

Don't' forget to restore user LEDs:

```
# ./restore_user_leds.sh
```

6. Demonstration

Take a Demo video, following the Step 4.4 and it's result to TA. It should be less than a minute.

7. Report

Each student should prepare his own report containing:

Purpose
Experiment sequence
Experimental results
Discussion
References

You should provide your own thoughts on the purpose/experiment sequence. Simply copying the purpose/steps we provide in this document won't get a good score.

Additional Discussion Item:

A. We have two methods for LED drive:
   1) A method using sysfs in shell script.
   2) A method using mmap in application.

   Compare these two methods. Describe advantages and disadvantages of these methods.

B. Since user LED is a device, it should be controlled by device driver module in kernel level. Discuss advantages and disadvantages of this method.

8. References

[1] Beaglebone System Reference manual A6.

[2] Beaglebone Black System Reference manual A5.2

[2] AM335x ARM Cortex-A8 Microprocessors (MPUs) Technical Reference Manual (Rev. F), Texas Instruments,

[3] AM335x ARM Cortex-A8 Microprocessors (MPUs) (Rev. D) , Texas Instruments,
http://www.ti.com/lit/ds/symlink/am3359.pdf