EE405 Electronic Design Lab - RoboCam Weeks 4 – 5

Lab 2. Light Control

I. Purpose

The purpose of this lab is to control two lights for illumination: hardwired LED lights using GPIO hardware and software of command lines, shell script, C program, and device driver module.

II. Problem Statement

Problem 2. Light Control.
Implement an illumination light controller with two white LEDs with Beaglebone: Wire two LEDs using GPIO and transistor array, and drive these using commands with sys file system (sysfs), shell script, C program, and device driver module.

We start from the simplest and perform step-by-step improvements.

Problem 2A. Light Control Commands with sysfs.
Wire two LEDs using GPIO and transistor array. Test commands with sysfs to control the user LED0 on Bone, and then test commands with sysfs for two hard-wired LED lights.

Problem 2B. Light Control Shell Script.
Control two hard-wired LED lights using shell script.

Problem 2C. Light Control C Program.
Control two hard-wired LED lights using C program.

Problem 2D. Test Example Device Driver Program.
Test an example device driver using Linux module.

Problem 2E. Light Control Application and Module.
Control two hard-wired LED lights using application program and device driver module for lights control.

III. Technical Backgrounds

First Week

Hardware

1. Hardware connection summary

AM3359 CPU → AM3359 GPIO → Bipolar Transistor → User LED with R.
→ P8/P9 → TR array IC → Light LED with R.

2. User LEDs on Beaglebone [1]

Getting Started with BeagleBone

http://beagleboard.org/getting-started

You'll see the PWR LED lit steadily. Within 10 seconds, you should see the other LEDs blinking in their default configurations.

- USR0 is configured at boot to blink in a heartbeat pattern
- USR1 is configured at boot to light during microSD card accesses
- USR2 is configured at boot to light during CPU activity
- USR3 is configured at boot to light during eMMC accesses

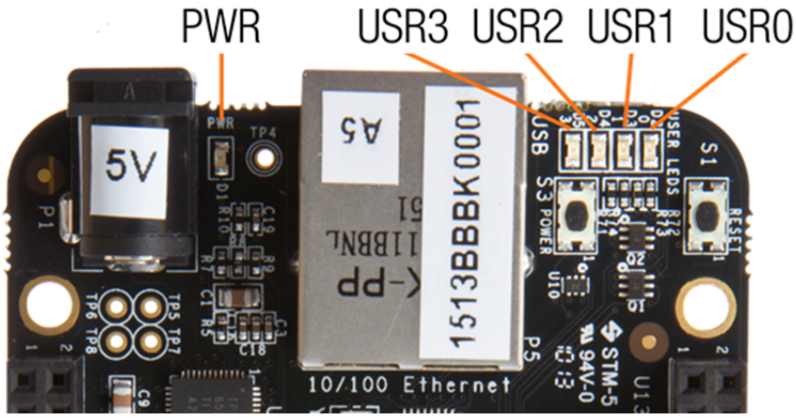


Fig. 2.1. LEDs inBeaglebone

GPIO LED hardware circuit

Four user LEDS are provided via GPIO pins on the processor. Figure 2.2 below shows the LED circuitry. [See p. 44, 7.7.3 User LEDs, Beaglebone System Reference Manual A5].

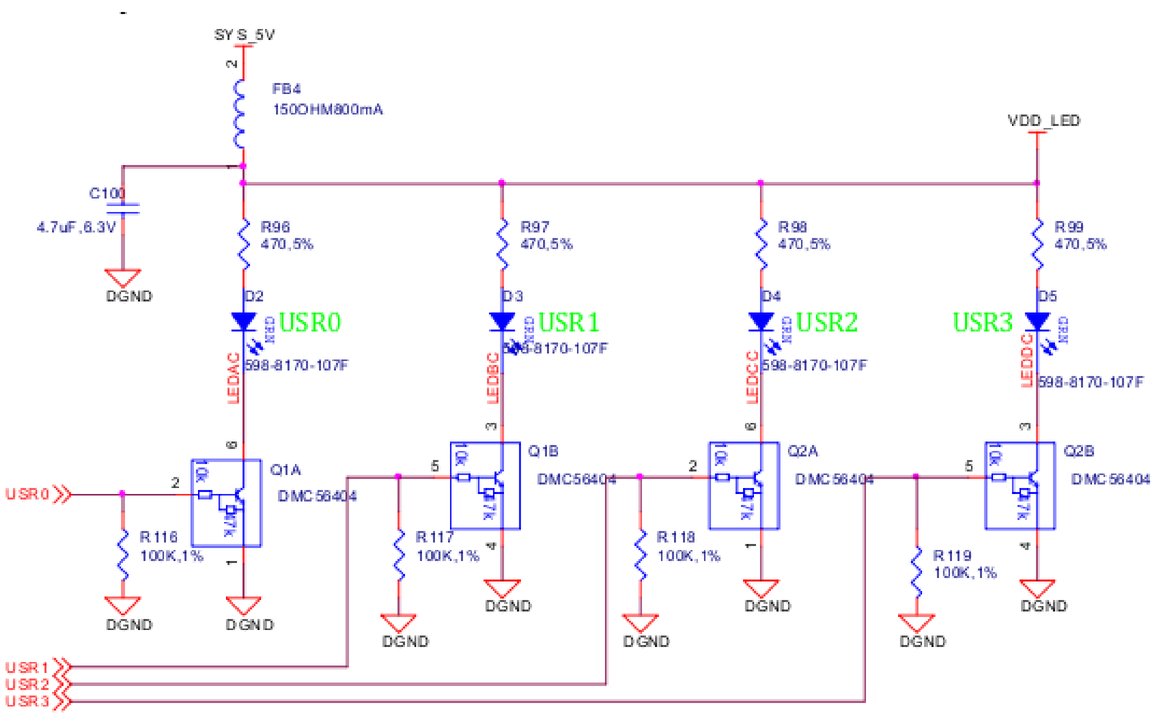


Fig 2.2 User LEDs control on Beaglebone.

Four user LEDs are connected to GPIO pins as shown in the following Table.

Table 2.1 User LED Control

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

Hardware summary:

- ✓ Four user LEDs on Beaglebone.
- ✓ Visible output.
- ✓ LEDs are connected to GPIO1_21 to 25 already with patterns in PCB.

We are going to control User 0 LED, which is connected to processor via GPIO1_21.

3. GPIO (General Purpose I/O)

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

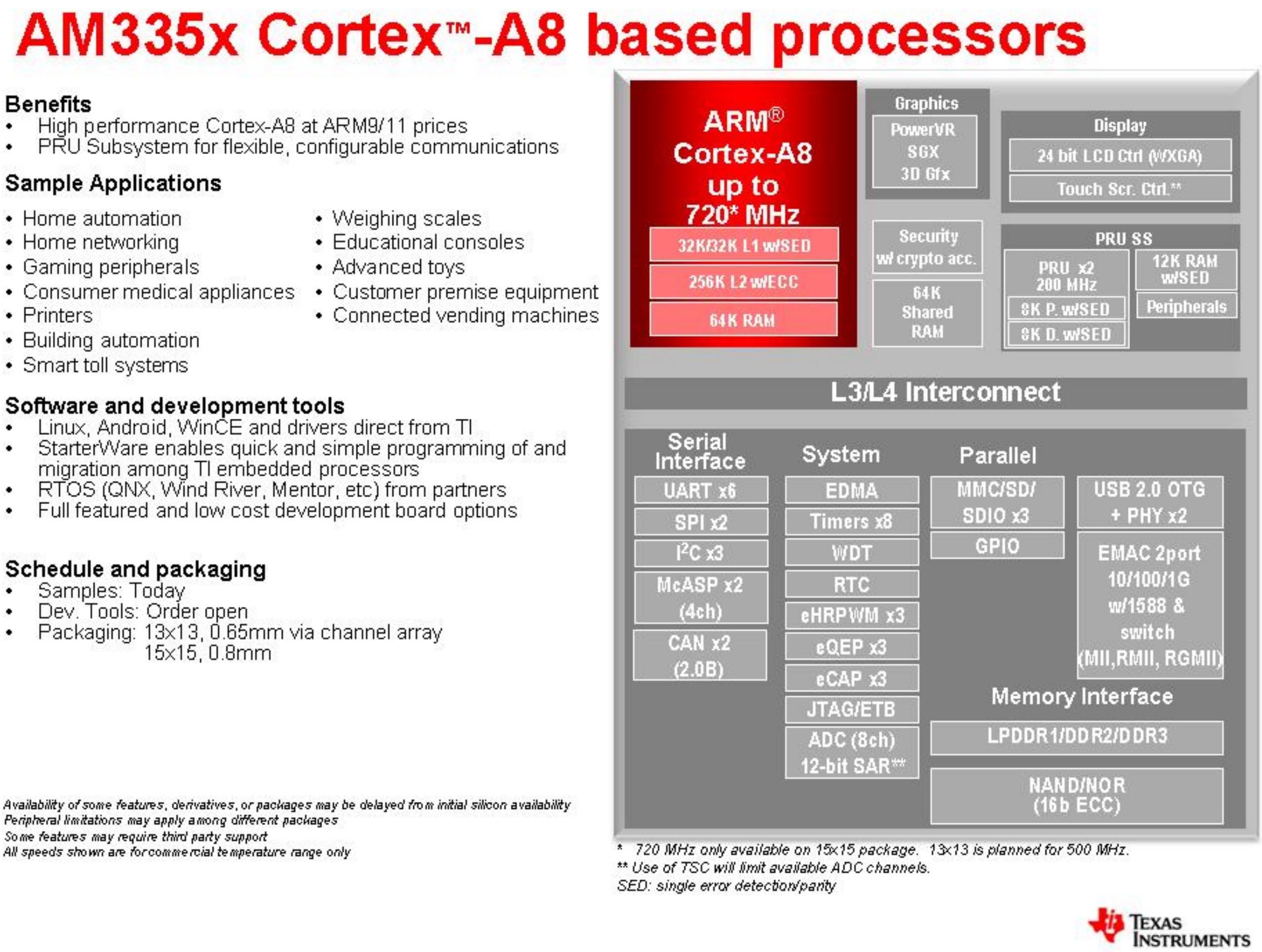


Fig. 2.3 Internal block diagram of AM3359 Processor
[Refer http://www.ti.com/product/am3359].

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.
- Wake-up request generation (in Idle mode) upon the detection of signal transition(s)

4. SAFE GPIO CONNECTION HOW TO

GPIO voltage and current

The GPIO pins on the Beaglebone are ***quite fragile***, compared to the Arduino or Netduino. The Arduino is happy with 5V of input, but can also work with 3.3V. The Netduino prefers 3.3V, but is 5V tolerant meaning you can get away with it. The Beaglebone, on the other hand, is ***decidedly intolerant of 5V***. Hook it up to 5V, even for a split second, and it will ***die***. I wish I could say that I learned this by reading about it, but you can trust me on this one.

The GPIO pins are also more delicate than the Arduino when it comes to current. According to this post by Beaglebone hardware designer Gerald Coley, the recommended max output current through a pin is ***4-6 mA***, and the max input current is 8 mA.

SAFE GPIO OUTPUT CONNECTION HOW TO

GPIO output is 0 V or 3.3 V. We are going to limit the output current to be LESS THAN 1 mA, regardless of connected logic circuit. The connected logic circuit may be operated by +5 V power. In this case,

- GPIO Logic 0 output of 0 V → Connected logic 0 V. Regarded as logic 0.
- GPIO logic 1 output of 3.3 V → Connected logic 3.3 V. Regarded as logic 1 since 3.3 V > threshold (half of 5 V).

We attach resistor of 5 Kohm between GPIO output and external logic input. In this case, the maximum current is 3.3 V / 5 Kohm = 0.66 mA.

Hence the solution is

GPIO output (Right end Output in Fig. 2.3) → 5 Kohm resistor → External 5 V logic input.

SAFE GPIO INPUT CONNECTION HOW TO

External logic may use +5 V power. The external logic output (0 V/5 V) is to be connected to the GPIO input, which is NOT 5V TOLERANT.

- External logic output 0 (0 V) → GPIO input 0 V. Regarded as logic 0. OK.
- External logic output 1 (5 V) → GPIO input 5 V. Beaglebone uses 3.3V. **BURNS BEAGLEBONE!**

Why external 5 V to GPIO input burns Beaglebone? In Fig 2.4, suppose the Input (in the left side) is driven by 5 V logic output. The voltages in pins of protection diode D1 are anode voltage of 5 V and cathode voltage of 3.3 V (Vcc), and hence producing voltage drop of 1.7 V. The current through D1 became very large and it will burn out!

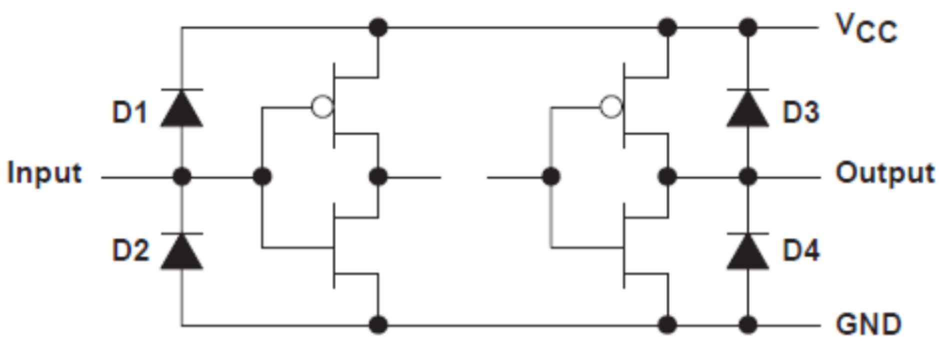


Fig 2.4 Diode paths in COMS circuits

How to make GPIO input 5V tolerant?

We attach resistor of 5 Kohm between external logic output and GPIO input. In this case, the diode D1 is forward biased with 0.7 V voltage drop with current (5 – 3.3 – 0.7) V/5 Kohm = 0.2 mA. The voltage at the Input (GPIO input pin) became Vcc + 0.7 = 4 V. This will in effect make the pin 5V-tolerant for digital I/O.

Hence the solution is

External 5 V logic output → 5 Kohm resistor → GPIO input (Left end Input in Fig. 2.3)

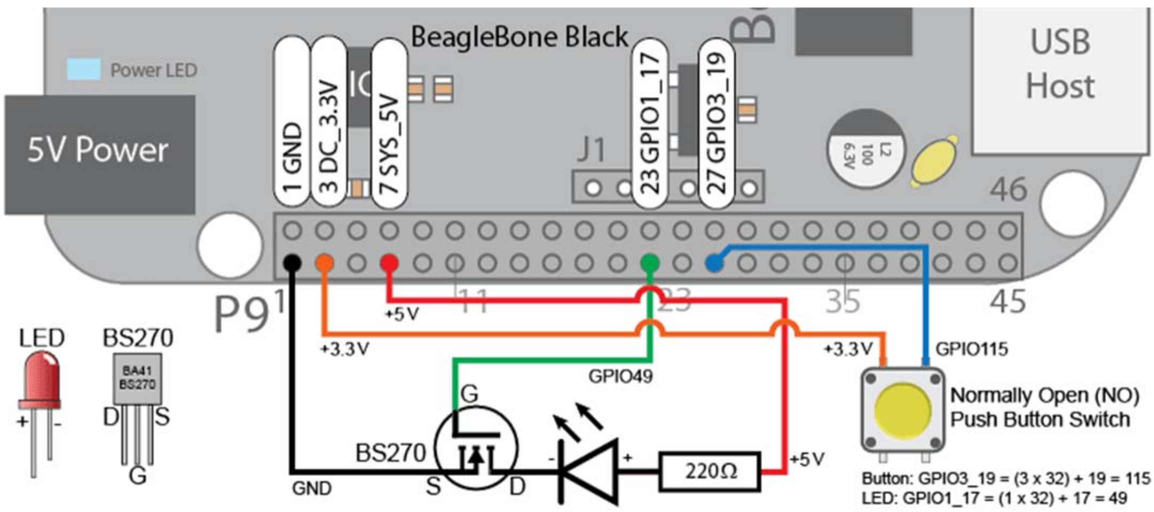
5. GPIO to LEDs Circuit

Similar to User LEDs circuit in Beaglebone, we are going to construct two Light LEDs circuit for RoboCam on Breadboard.

Two Light control LEDs are driven by GPIO also, and connection from Beaglebone to Breadboard can be done using two I/O connectors P8/P9 as follows:

AM3359 CPU → GPIO → P8/P9 → TR array IC → Light LEDs with R

Typical external ED circuit for Beaglebone can be found in the Web [http://derekmolloy.ie/kernel-gpio-programming-buttons-and-leds/] as follows:



R.

Fig. 2.5 Typical External LED circuit for Beaglebone

Instead of discrete transistor, we are going to use uLN2803AN, which is 8-unit Darlington transistor array up to 500 mA in DIP type package.

Use two GPIOs to control two lights each. We selected GPIO0_30 and GPIO0_31 as follows:

P9.11	GPIO0_30	Changed to LightOut1
P9.13	GPIO0_31	Changed to LightOut2

Wiring for Light 1:

P9.11 GPIO0_30 → R1 → TR array B input;

TR array OC output → R2 → Cathode of LED;

Anode of LED → 5V

Wiring for Light 2:

P9.13 GPIO0_31 → R1 → TR array B input;
TR array OC output → R2 → Cathode of LED;
Anode of LED → 5V

We select R1 as 5 Kohm for safety. Since GPIO output can be either 0 V or 3.3 V, even when the other side of resistor is shorted to ground or Vcc of 5V, the output current of GPIO is less than 1 mA.

We need to select the value of R2 to limit the current of LED to be less than the specified. Search on voltage drop and desired current of white LED. Determine the value of R2.

SOFTWARE

1. User LED control using command (Problem 2A)

Refer: EBC Exercise 10 Flashing an LED
http://elinux.org/EBC_Exercise_10_Flashing_an_LED

This page is for the Bone (Black or White) running the 3.8 Kernel.

Four User LEDs does appear as directories in /sys/class/leds as
beaglebone:green:usr0@
beaglebone:green:usr1@
beaglebone:green:usr2@
beaglebone:green:usr3@

In the directory beaglebone:green:usr0, you can control User LED0 by writing to each file:
trigger, brightness, delay_on, and delay_off.

For details, read the following sections (in the above web page) and test by yourself:
A. gpio via the Shell Command Line and sysfs.
B. Flashing the user LEDs.

2. GPIO LED control using command and sysfs (Problem 2A also)

gpio-sysfs
gpio-sysfs is the preferred method of gpio interfacing from user space. Use this unless you absolutely need to update multiple gpios simultaneously or you must use a kernel version before 2.6.27. The full documentation is alongside the gpio-framework documentation:
https://www.kernel.org/doc/Documentation/gpio/sysfs.txt and
https://www.kernel.org/doc/Documentation/gpio/gpio.txt.

Overview of sysfs directories and files:
How to use GPIO signals
https://developer.ridgerun.com/wiki/index.php/How_to_use_GPIO_signals

For detailed procedure, see Lab Procedures.

3. LED Control with Shell Script (Problem 2B)

The *shell* provides you with an interface to the UNIX system. It reads input from you and executes the programs you specified. While the programs are executing, it displays their output. The real power of the shell lies in the fact that it is much more than a command interpreter. It is also a powerful programming language, complete with conditional statements, loops, and functions.

Read and test:
Beginners – Bash Scripting, https://help.ubuntu.com/community/Beginners/BashScripting.
Especially, Sections "Scripting" to "Functions".

4. LED Control using C and sysfs (Problem 2C)

Refer: Access GPIO from Linux user space
http://falsinsoft.blogspot.kr/2012/11/access-gpio-from-linux-user-space.html

Manage GPIO from application

All these same operations can be made using a software application. Follow short lines of C code showing how the reproduce the same steps as above (remember to change XX with the GPIO number you want to use).

Reserve (export) GPIO:

```
int fd;
char buf[MAX_BUF];
int gpio = XX;

fd = open("/sys/class/gpio/export", O_WRONLY);
sprintf(buf, "%d", gpio);
write(fd, buf, strlen(buf));
close(fd);
```

Set the direction of GPIO:

```
sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);

fd = open(buf, O_WRONLY);

// Set out direction
write(fd, "out", 3);
// Set in direction
write(fd, "in", 2);

close(fd);
```

Note. Change sprint to snprintf.

In case of *out* direction set the value of GPIO:

```
sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);

fd = open(buf, O_WRONLY);

// Set GPIO high status
write(fd, "1", 1);
// Set GPIO low status
write(fd, "0", 1);

close(fd);
```

In case of *in* direction get the current value of GPIO:

```
char value;

sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);

fd = open(buf, O_RDONLY);

read(fd, &value, 1);

if(value == '0')
{
    …. // Current GPIO status low
}
else
{
    …. // Current GPIO status high
}

close(fd);
```

Once finished free (unexport) the GPIO:

```
fd = open("/sys/class/gpio/unexport", O_WRONLY);

sprintf(buf, "%d", gpio);

write(fd, buf, strlen(buf));

close(fd);
```

We are going to modify these to functions in the Design section.

5. Device Driver Module Example (Problem 4D)

Kernel Modules Versus Applications
[http://www.xml.com/ldd/chapter/book/ch02.html#t1]

Whereas an application performs a single task from beginning to end, a module registers itself in order to serve future requests, and its "main" function terminates immediately. In other words, the task of the function *init_module* (the module's entry point) is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do." The second entry point of a module, *cleanup_module*, gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." The ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. *printf* is one of those callable functions and is defined in *libc*. A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The *printk* function used in *hello.c* earlier, for example, is the version of *printf* defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.[6]

Device Driver

https://en.wikipedia.org/wiki/Device_driver

In computing, a **device driver** (commonly referred to as a *driver*) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.[2]

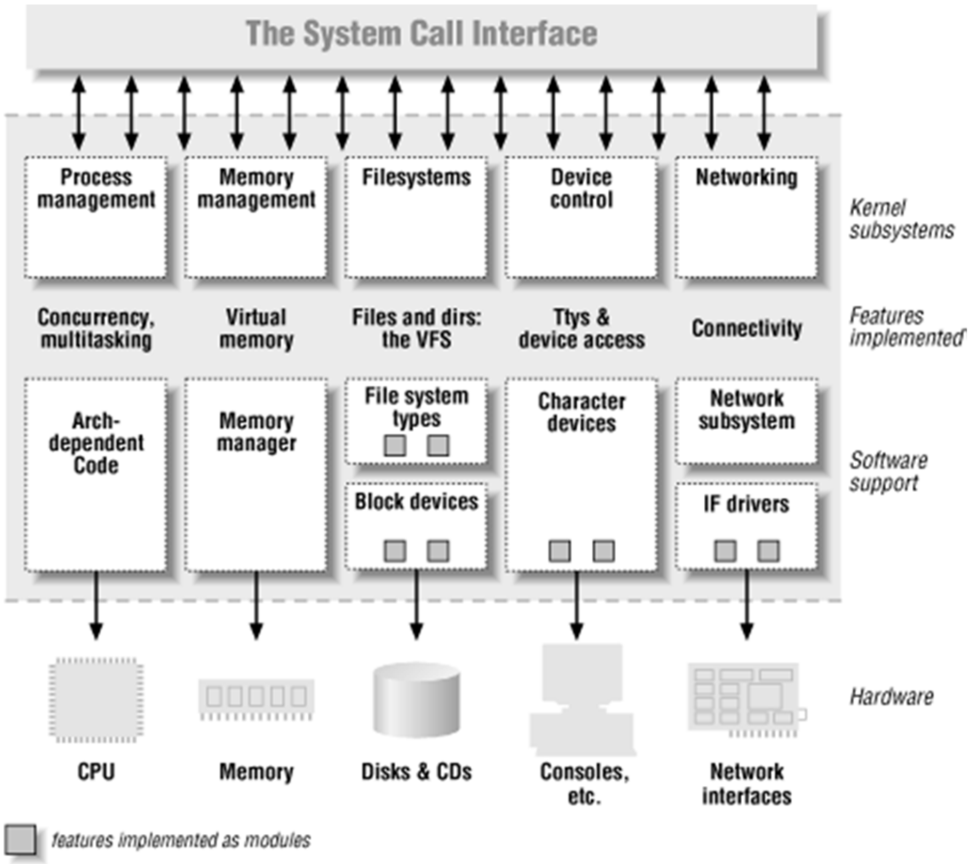


Fig 2.6. Device Drivers in Kernel [Lubuini, Linux Device Driver].

Example device driver source

HelloDev.c
Test_HelloDev.c
Makefile

HelloDev.c

```
/*  File HelloDev.c
 *  A simple character device with file system operations
 */

#include <linux/module.h>       /* Needed by all modules */
#include <linux/fs.h>

# define HELLO_MAJOR    234

static int debug_enable = 0;
module_param(debug_enable, int, 0);
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");

struct file_operations HelloDev_fops;

static int HelloDev_open(struct inode *inode, struct file *file)
{
        printk("HelloDev_open: successful\n");
        return 0;
```

```
}

static int HelloDev_release(struct inode *inode, struct file *file)
{
        printk("HelloDev_release: successful\n");
        return 0;
}

static ssize_t HelloDev_read(struct file *file, char *buf, size_t count,
        loff_t *ptr)
{
        printk("HelloDev_read: returning zero bytes\n");
        return 0;
}

static ssize_t HelloDev_write(struct file *file, const char *buf, size_t count,
        loff_t *ppos)
{
        printk("HelloDev_write: accepting zero bytes\n");
        return 0;
}

static long HelloDev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        printk("HelloDev_ioctl: cmd=%d, arg=%ld\n", cmd, arg);
        return 0;
}

static int __init HelloDev_init(void)
{
        int ret;
        printk("HelloDev Init - debug mode is %s\n",
                debug_enable ? "enabled" : "disabled" );
        ret = register_chrdev(HELLO_MAJOR, "hellodev", &HelloDev_fops);
        if (ret < 0) {
                printk("Error registering HelloDev fops\n");
                return ret;
        }
        printk("HelloDev: registered successfully!\n");

        /* Init processing here ...     */

        return 0;
}

static void __exit HelloDev_exit(void)
{
        unregister_chrdev(HELLO_MAJOR, "hellodev");

        printk("Goodbye, HelloDev\n");
}

struct file_operations HelloDev_fops = {
        owner:          THIS_MODULE,
        read:           HelloDev_read,
        write:          HelloDev_write,
        compat_ioctl:   HelloDev_ioctl, // ioctl --> compat_ioctl Changed 2.6.36
        open:           HelloDev_open,
        release:HelloDev_release,
};

module_init(HelloDev_init);
module_exit(HelloDev_exit);

MODULE_AUTHOR("Christoper Hallinan");
```

```
MODULE_DESCRIPTION("HelloDev Example");
MODULE_LICENSE("GPL");
```

Makefile

```
# Embedded Bone cross-compile module makefile

ifneq ($(KERNELRELEASE),)
    obj-m := HelloDev.o
else
    SUBDIRS := $(shell pwd)

default:
ifeq ($(strip $(KERNELDIR)),)
	$(error "KERNELDIR is undefined!")
else
	$(MAKE) -C $(KERNELDIR) M=$(SUBDIRS) modules
endif

app:
	arm-linux-gnueabihf-gcc -o Test_HelloDev Test_HelloDev.c

clean:
	rm -rf *~ *.ko *.o *.mod.c modules.order Module.symvers .pwm* .tmp_versions
	rm use_hellodev1

endif
```

Test_HelloDev.c

```
/*	File Test_HelloDev.c  */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
	/* Our file descriptor	*/
	int fd;
	int rc = 0;
	char *rd_buf[16];

	printf("%s: entered\n", argv[0]);

	// Open the device
	fd = open("/dev/HelloDev", O_RDWR);
	if (fd == -1) {
		perror("open failed");
		rc = fd;
		exit(-1);
	}
	printf("%s: open successful\n", argv[0]);
```

```
        // Issue a read
        rc = read(fd, rd_buf, 0);
        if (rc == -1) {
                perror("read failed");
                close(fd);
                exit(-1);
        }
        printf("%s: read: returning %d bytes!\n", argv[0], rc);

        // Close device
        close(fd);
        return 0;
}
```

This example device driver module and application program will be tested in Lab Procedure section.

7. Light Control Device Driver Module (Problem 2E)

See the Design section.

IV. Equipment and Parts

1. Lab equipment

Router.
IBM PC with Windows and Linux (Dual boot).
Embedded board Beaglebone with cables and 4GB SD.
Breadboard.

2. Electronic parts

ID	Part No	Description	Qty/ group	Unit price
21	CP41B-WES-CK0P0154	DIP LED White, 30 mA	2	500
22	uLN2803AN	Darlington TR array, 18 pin DIP	1	1,480

V. Design

Pre-report of first week

1. Design a hardware circuit for the Light control of Problem 2.

GPIOs to use
P9.11 GPIO0_30 changed to LightOut1

P9.13 GPIO0_31 changed to LightOut2

Circuit connections

P9.11 GPIO0_30 → R1 → TR array B input; TR array OC output → R2 → 1W LED → 5V
P9.13 GPIO0_31 → R1 → TR array B input; TR array OC output → R2 → 1W LED → 5V

Component values

R1 = 5.1 Kohm
For safety (< 1 mA)
R2 = 50 ohm
Limit LED current to 30 mA
(5 – 3.5)/0.03 = 1.5*33 = 50

Test on Breadboard.

2. Summarize commands to control User LEDs, and also commands to control GPIO LEDs (Problem 2A).

Refer Lab Procedures.

3. Design Shell script for light control (Problem 2B).

a. Design shell program ui_control_lights.sh

Objective

Loop for user input and control Light LEDs.

Suggested Algorithm

0. Print title
1. Export: Get access permission for GPIO30 & 31.
2. Set directions of GPIO 30 & 31 as output
5. User Interface Infinite loop
   A. Get user input of light_idd and onoff_str
   B. Check user input lid. Break if < 1. Check if valid.
   C. Check valid on/off string
   D. Action for correct input

8. Set directions as input
9. UnExport: Release access permission for GPIO30 & 31.

How To write Shell

while loop in shell

```
while true; do
  ……
done
```

Get user input of int & str with prompt

```
read -p "Type integer and string: " int str;
```

Check valid integer lid in [1, 2]

```
if [ $lid -lt 1  -o  $lid -gt 2 ];
then
  ……
fi
```

Check valid on/off string

```
if [  $onoff_str != "on"  -a  $onoff_str != "off"  ];
```

b. Design loop_control_lights.sh

Objective

Test speed: Fastest shell loop to on/off lights 1 & 2

Algorithm

0. Print title
1. Export: Get access permission for GPIO30 & 31.
2. Set directions of GPIO 30 & 31 as output
3. Get Start time (ns)
5. Finite loop Many times
   A. Turn on light 1
   B. Turn on Light 2
   C. Turn off light 1
   D. Turn off light 2
6. Get End time (ns)
7. Echo End/Start time
8. Set directions as input
9. UnExport: Release access permission for GPIO30 & 31

How To Get start time in shell

```
start=$(date +%s.%N);
```

Get start time string with sec & ns portion separated by ‘.’: Looks like a floating point number.

Pre-report of second week

4. **Design test-light-control in C for Problem 2C.**

Design and program to control two hard-wired LED lights using C program.
Refer Backgrounds and relevant materials in the Web.

We design three files:

- gpio_control.h: Define gpio control functions

- gpio_control.c: Actual body of gpio control functions
- test_light_control.c: Test light control along with user input loop.

Of course, you may add Makefile.

First gpio_control.h can be defined as follows:

```
// File gpio_control.h
// Function definitions for gpio-control.c and app.

#define MAX_BUF 64     /* For the max length of string */

int gpio_export(unsigned int gpio);     // gpio means gpio number (0 to 127)
int gpio_unexport(unsigned int gpio);
int gpio_set_dir(unsigned int gpio, unsigned int out);  // out = 0: in. out = 1: out.

int gpio_fd_open(unsigned int gpio);    // Returns file descriptor fd
int gpio_fd_set_value(int fd, unsigned int value);    // value can be 0 or 1
int gpio_fd_get_value(int fd, unsigned int *value);   // *value will be 0 or 1
int gpio_fd_close(int fd);
```

Next design seven functions defined as above, referring Backgrounds.

Finally, design test_light_control.c

Suggested algorithm of test_light_control

0. Print title
1. Set variables: light_id = 1;
2. Export GPIO 30 & 31
3. Set direction of GPIO 30 & 31 to out. Open gpio30 & GPIO31.
5. Loop while light_id > 0
    A. Prompt output “"Enter light_id and on_off_str: "
    Get user input of light_id and on_off_str
    B. Check light_id. Break if <= 0.
    C. Check on_off_str and set on_off_int
    D. Control action
8. Close GPIO 30 & 31 and Set direction to input
9. Unexport GPIO 30 & 31

loop_light_control

Also Design loop_light_control.c: The same function as loop_control_lights.sh

5. Summarize what is device driver in Linux (Problem 2D).

Refer relevant Web pages.

6. Design Lights control device driver module and test application for Problem 2E.

We provide template device driver for light control:

- am33xx.h
- gpio.h
- Template_LightLEDs_Module.c

Design LightLEDs_Module.c

Fill in to the given template to make a complete device driver module for light control.
Note that LightLEDs_Module.c contains five functions (in the calling order):

```
1. LightLEDs_init_module (void)
2. LightLEDs_open(struct inode *inode, struct file *filp)
3. LightLEDs_write (struct file *filp, const char *wbuf, size_t wcount, loff_t *f_pos)
4. LightLEDs_release(struct inode *inode, struct file *filp)
5. LightLEDs_cleanup_module (void)
```

Here init_module() and cleanup_module() are complementary,
and also open() and close().

Actions in five functions are summarized as follows:

```
1. LightLEDs_init_module (void)
	// A. Register LightLEDs as a character device
	major = register_chrdev( LightLEDs_MAJOR, "LightLEDs", &LightLEDs_fops );

struct file_operations LightLEDs_fops =
{
	owner:	THIS_MODULE,
	open:		LightLEDs_open,
	write:	LightLEDs_write,
	release:	LightLEDs_release,
};

2. static int LightLEDs_open(struct inode *inode, struct file *filp)
	B0. Check and set LightLEDs_usage
	B1. Check_mem_region of GPIO0, and request_mem_region()
	B2. ioremap GPIO0
	C. Set Mux: Assume done by Kernel correctly.
	D. Set direction of GPIOs as output

3. static ssize_t LightLEDs_write (struct file *filp, const char *wbuf, size_t wcount,
loff_t *f_pos)
	E. Get user_ata from app program
	F. Read GPIO0, modify, and write bits 30 & 31 according to user_data

4. static int LightLEDs_release(struct inode *inode, struct file *filp)
```

```
        -D. Set direction of GPIOs as input
        -C. Reset Mux: None.
        -B2. Iounmap(gpio0)
        -B1. release_mem_region(gpio0)
        -B0. Clear LightLEDs_usage

5. LightLEDs_cleanup_module (void)
        // -A. Unregister LightLEDs device
```

Note

Especially, Steps D, F, and -D should be filled in using ioread32(), modify (with AND/OR), and then iowrite32() to suitable registers.

For example, Step -D can be programmed as follows:

```
oe0 = ioread32(gpio0_vbase + OMAP4_GPIO_OE);
oe0new = oe0 | (0x03 << 30);         // Set bits 30 & 31
iowrite32(oe0new, gpio0_vbase + OMAP4_GPIO_OE);
```

Design Test_LightLEDs_Module.c

Design simple application program to test the device driver module for Lights.

Suggested Algorithm

1. Open LightLEDs device open("/dev/LightLEDs", …)
    // Module LightLEDs_open() works.
2. User interface loop
    A. Get user data (0 to 3, neg to exit)
        // Bit 1: Light 2 on/off. Bit 0: Light 1 on/off.
    Exit loop if user_data < 0
    B. write(dev, user_data)
        // Module LightLEDs_write() works.
3. Close LightLEDs device: close()
    // Module LightLEDs_release() works.

VI. Lab procedures

First Week

Problem 2A

Step 1. User LED0 control using sysfs and command line

10. Turn on

Refer Startup_Shutdown_Sequence.docx for startup and shutdown sequence.

11. Select LEDs for test

Selection: Test User 0 LED.

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

12. Check sysfs file for User LEDs (as superuser)

Search /sys/class:

```
# su
# ls -F /sys/class
backlight/  hwmon/        mmc_host/      scsi_disk/     uio/
bdi/        i2c-adapter/  mtd/           scsi_host/     usbmon/
block/      i2c-dev/      net/           sound/         vc/
bsg/        input/        power_supply/  spidev/        video4linux/
dma/        lcd/          pps/           spi_master/    virtio-ports/
drm/        leds/         pwm/           thermal/       vtconsole/
dvb/        mbox/         rc/            timed_output/  watchdog/
firmware/   mdio_bus/     regulator/     tty/
gpio/       mem/          rtc/           ubi/
graphics/   misc/         scsi_device/   udc/
```

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.

13. 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 ':'.

List directory

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

See what's in trigger

```
# cat trigger
none nand-disk mmc0 timer oneshot [heartbeat] backlight gpio cpu0 default-on
```

```
transient
```

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] nand-disk mmc0 timer oneshot heartbeat backlight gpio cpu0 default-on
transient
```

14. Control on/off of usr0 LED

Turn on/off usr0 LED

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

Usr0 LED is turned on and off!

15. 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
```

16. Return to heartbeat

```
# echo heartbeat > trigger
```

Success?

Step 2. Lights Control Commands

20. Wire Two GPIOs to Two Light LEDs on breadboard.

Connect Bone P8/P9 to R, TR array, R, and LEDs on Breadboard.
GPIO0_30 is connected to Light 1, and GPIO0_31 is connected to Light 2 LED.

Note. Use separated power for Beaglebone and Light LEDs if dual power supply is available: You can check currents for each.

21. Access GPIO0_30 for Light 1 as root

Check sysfs for gpio

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

Export GPIO0_30. Note that GPIO number = 0*32 + 30 = 30. Hence GPIO30:

```
# cd /sys/class/gpio
# echo 30 > export
# ls  -F
export gpio30 gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

Note that "gpio30" directory is created.

22. Control Light 1 via GPIO0_30 [Right LED]

Go to GPIO30 directory

```
# cd gpio30
```

Measure voltage of GPIO0_30 pin P9.11.
Set GPIO direction to output

```
# echo out > direction
# cat direction
out
```

Measure voltage of GPIO0_30 pin P9.11.

Turn on your own LED Light 1 (Right LED)

```
# echo 1 > value
# cat value
1
```

Turn off your own LED Light 1.

```
# echo 0 > value
# cat value
0
```

Check voltages of P9.11, 2803 input, 2803 output, and LED pins in each case.

23. Free GPIO0_30

```
# cd /sys/class/gpio
# echo 30 > unexport
```

24. Control Light 2 via GPIO0_31 [Left LED]

Repeat the same procedure for GPIO0_31 for Light 2 (Left light).

Step 3. Light Control Shell Script (Problem 2B)

31. Make a subdirectory b_GPIO_LED_Shell.

Make a subdirectory

```
$ mkdir -p ~/DesignLab/2_LightControl/b_GPIO_LED_Shell
```

32. Test basic scripting

Read and test:

Beginners – Bash Scripting, https://help.ubuntu.com/community/Beginners/BashScripting.
Especially, Sections "Scripting" to "Functions".

33. Test ui_control_lights.sh

Edit prepared ui_control_lights.sh.

Make this shell script executable with 'chmod'.

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

Run

```
$ ./ui_control_lights.sh
```

Enter user input repeatedly.
Does Light LEDs operate as commanded?

34. Test loop_control_lights.sh

Edit loop_control_lights.sh: Add get start and end times.

Make this shell script executable with 'chmod'.

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

Run

```
$ ./loop_control_lights.sh
```

Record elapsed time for M (many) loops.
Record the result for the final report.

Second Week

Step 4. C Program for Two Lights (Problem 2C)

41. Make a subdirectory c_LightControl_C.

Make a subdirectory

```
$ mkdir -p ~/DesignLab/2_LightControl/c_LightControl_C
```

42. Edit files

We need three files:

- gpio_control.h: Define gpio control functions [Given already]
- gpio_control.c: Actual body of gpio control functions
- test_light_control.c: Test light control along with user input loop.

Edit prepared gpio_control.c & test_light_control.c.
Also edit Makefile.

43. Compile

```
$ make
```

44. Run on Bone as root (using NFS).

Does Light LEDs operate as expected?

45. Test loop-light-control

Edit, make.
Run on Bone as root.
Record the result for the final report.

Step 5. Test HelloDev (problem 2D)

51. Make a working directory

```
$ mkdir -p DesignLab/2_LightControl/d_HelloDev
$ cd DesignLab/2_LightControl/d_HelloDev
```

52. Edit Files

Edit or check given HelloDev.c.
Edit or check given Test_HelloDev.c

53. Make

Edit or check given Makefile.

Make

```
$ make clean
$ make
$ make app
```

54. Test on Beaglebone using nfs as root

Insert module HelloDev.ko

```
# insmod HelloDev.ko

# dmesg | tail
…...
[ 3105.006283] HelloDev Init - debug mode is disabled
[ 3105.006376] HelloDev: registered successfully!
```

Run app.

```
# ./Test_HelloDev
./Test_HelloDev: entered
open failed: No such file or directory
```

Oops! Make node.

```
# mknod /dev/HelloDev c 234 0
```

Run app again

```
# ./Test_HelloDev
./Test_HelloDev: entered
./Test_HelloDev: open successful
./Test_HelloDev: read: returning 0 bytes!

# dmesg | tail
…...
[ 3161.387420] HelloDev_open: successful
[ 3161.388787] HelloDev_read: returning zero bytes
[ 3161.393050] HelloDev_release: successful
```

Remove module

```
# rmmod HelloDev

# dmesg | tail
……
 [15389.309311] Goodbye, HelloDev
```

Success?

Do Second trial: Any problem?

Step 6. Compile and run Light control with LED driver module (Problem 2E)

61. Set include directory in DesignLab

Edit (or just check) two include files in DesignLab/include:

am33xx.h
gpio.h

62. Edit prepared LightLEDs_Module.c

Given template Template_LightLEDs_Module.c.
Edit LightLEDs_Module.c

```
$ gedit LightLEDs_Module.c
```

63. Make

Edit prepared Makefile

```
$ gedit Makefile
```

Make

```
$ Make
```

64. Edit prepared app – Test_LightLEDs_Module.c

```
$ gedit  Test_LightLEDs_Module.c
```

65. Make app.

```
$ make app
```

66. Test on Beaglebone via NFS as root.

Insert module

```
# insmod LightLEDs-Module.ko
```

Check dmesg

```
# dmesg | tail
…...
[ 7759.285710]     A. Init LightLEDs: The major number is 238
```

Note that major number of device is assigned as 238.

Make node!

```
# mknod /dev/LightLEDs c 238 0
```

Run app.

```
# ./Test_LightLEDs_Module
LightLEDs device open success.
Enter data for LightLEDs (0 to 3): 3
Write LightLEDs with data 3.        // Two lights on
Enter data for LightLEDs (0 to 3): 1
Write LightLEDs with data 1.        // Right light on
Enter data for LightLEDs (0 to 3): 2
Write LightLEDs with data 2.        // Left right on
Enter data for LightLEDs (0 to 3): 0
Write LightLEDs with data 0.        // Two lights off
Enter data for LightLEDs (0 to 3): -1
LightLEDs device closed.
```

Check dmesg

```
# dmesg | tail
…...
[ 7759.285710]     A. Init LightLEDs: The major number is 238
[ 7924.854708]   B1. Warning: LightLEDs GPIO0 check_mem_region failed...
[ 7924.854787]   D. LightLEDs opened.
[ 7930.952653] LightLEDs GPIO0_DOUT: 00000000 to c0000000
[ 7934.061649] LightLEDs GPIO0_DOUT: c0000000 to 40000000
[ 7936.022798] LightLEDs GPIO0_DOUT: 40000000 to 80000000
[ 7937.479080] LightLEDs GPIO0_DOUT: 80000000 to 00000000
[ 7941.342182]   -D. LightLEDs released.
```

Remove gpio-led device

```
# rmmod LightLEDs_Module

# dmesg | tail
…...
[ 8135.696791]   -A. Cleanup LightLEDs: Unregistered.
```

If successful, *demonstrate to TA!*

VI. Final Report

Discussion for the following question should be included in the report.

1) Compare Shell script, C program, and device driver module. Discuss advantages and disadvantages of each.

2) Why we require transistor array to drive LEDs? Can you drive 3W LEDs using uLN2803?

3) What is color LED? Can you control color LED? Can you control intensity? How many colors can be displayed?

4) Suppose you are going to display one Hangul character with 24 x 24 LED matrix. How can you control this? Can you reduce the number of GPIO pins to drive this?

5) Set your own topic to discuss related to Lab 2, and explain summary of your search result.

VII. References

[1] Beaglebone Rev A6 System Reference Manual, http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SRM.pdf

[2] AM3359 Datasheet, AM335x ARM Cortex-A8 Microprocessors (MPUs) (Rev. J), http://www.ti.com/lit/ds/symlink/am3359.pdf

[3] Technical Reference Manual - AM335x ARM Cortex-A8 Microprocessors (MPUs) Technical Reference Manual (Rev. O), http://www.ti.com/lit/ug/spruh73o/spruh73o.pdf