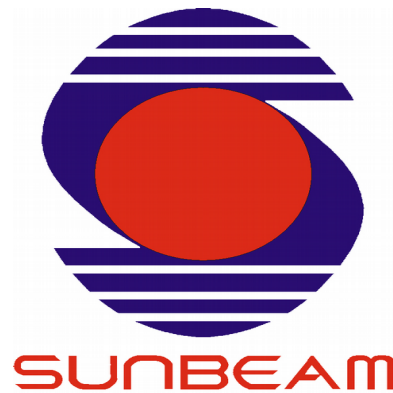


A  
PROJECT REPORT ON

# Embedded Linux Device Driver on BeagleBone Black to control GPIO

SUBMITTED IN  
PARTIAL FULFILLMENT OF  
**DIPLOMA IN EMBEDDED SYSTEM DESIGN (PG-DESD)**



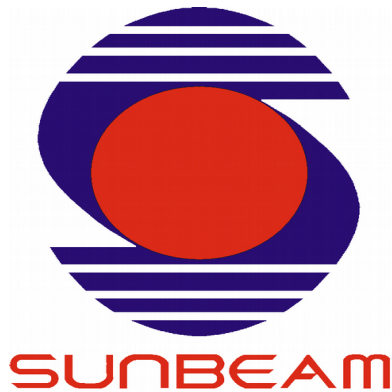
BY

**Bipin Dangwal**

AT

**SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY,  
HINJAWADI**

**SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY,  
HINJAWADI.**



**CERTIFICATE**

This is to certify that the project

**Embedded Linux Device Driver on  
BeagleBone Black to control GPIO**

Has been submitted by

**Bipin Dangwal**

In partial fulfillment of the requirement for the Course of **PG Diploma in  
Embedded System Design (PG-DESD AUG2019)** as prescribed by The  
CDAC ACTS, PUNE.

Place: Hinjawadi

Date: 30-JAN-2020

**Authorized Signature**

# ACKNOWLEDGMENT

I am grateful to my guide, Ms. Radhika Sahastrabudhe for sharing his pearls of wisdom and assisting me throughout with my project “Embedded Linux Device Driver on Beagle Bone Black to control GPIO”. Words shall never be able to describe the spirit of working together nor shall they be able to express the feeling I felt towards my guide. It has been a greatly enriching experience for me to work under his caring guidance.

I would also like to thank my Course Co-ordinator Mr. Devendra Dhande for making all facilities available. I would also like to thank Mr. Nilesh Ghule Sir, for their co-operation and doing their best in making resources available to us.

I also express my gratitude to all members for their interminable support. Though words have their own limitation I have made a modest effort to acknowledge the support extended.

This platform has also given me an opportunity to thank all the teaching and non-teaching staff of Sunbeam Institute of Information and Technology for their necessary help they gave me directly or indirectly.

# **ABSTRACT**

Contemporary era has become the age of information and high performance computing and an open-source platform like Linux becomes quite handy to fulfill the requirements with minimum memory consumption. Today, most of the gadgets used for information sharing, (e.g, multimedia projector, WiFi router, infotainment system in automobiles, etc.) run on embedded Linux. In an embedded system, mpu(micro-processor unit) requires communicating with several devices for which, the kernel requires device drivers. With raise in number to different types of devices, distributed embedded system has become one of the hot-topics in the tech industry. Thus, an engineer working on BSP (Board Support Packages) has plenty of opportunities both in the academia and the industry. With the fast-growing advancements in the fields of embedded electronics and distributed embedded systems, embedded linux has become a primary requirement for the industry to work on.

# INDEX

<b>1.</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2.</b>	<b>LITERATURE SURVEY</b>	<b>2</b>
	2.1 Microprocessor	2
	2.2 Operating System	2
	2.3 Linux Operating System	3
	2.4 Device Driver	3
	2.5 GPIO	4
	2.6 Beaglebone Black	4
<b>3.</b>	<b>PROJECT ARCHITECTURE</b>	<b>6</b>
<b>4.</b>	<b>HARDWARE DESCRIPTION</b>	<b>7</b>
	4.1 Development board	7
	4.2 Hardware Processor	8
<b>5.</b>	<b>SOFTWARE REQUIREMENTS AND SPECIFICATIONS</b>	<b>10</b>
	5.1 Cross-compilation Toolchain:	10
	5.2 Bootloader	10
	5.2.1 RBL (ROM Bootloader)	11
	5.2.2 MLO (Memory Loader)	11
	5.2.3 U-boot	11
	5.3 Kernel	11
	5.4 Root File-system	11
	5.5 Steps for Setting up Embedded Linux for BeagleBone Black	11
	5.5.1 Compiling Linux For Beagle Bone Board	11
	5.5.2 Building uBoot For Beagle Bone Board	13
	5.5.3 Porting Linux on SD-Card	13
	5.5.4 Mounting Rootfs on SD-Card	14
	5.5.5. Setting up Bootloader in first partiton of disk	15
	5.6 Communication Between HOST machine and Target Machine	15
	5.7 Make File for compilation Linux Device Driver	16
<b>6.</b>	<b>SOURCE CODE EXPLANATION</b>	<b>17</b>
	6.1 Implementation of Per-device Structure	17
	6.2 Implementation of Initialization and Exit Routines	18
	6.3 Implementation of the Entry Points for the GPIO Device Driver	19

	6.4 Implementation of Interrupt Handling Functions	22
<b>7.</b>	<b>TESTING</b>	23
<b>8.</b>	<b>FUTURE SCOPE</b>	25
<b>9.</b>	<b>CONCLUSION</b>	26
<b>10</b>	<b>REFERNCES</b>	27
<b>.</b>		

## LIST OF FIGURES

Serial No.	Figure Title	Page
1	OS Architecture	2
2	Kernel Space Interaction	4
3	Project Architecture	6
4	Beagle Bone Rev C Board	7
5	AM3358BZCZ100 Processor Diagram	8
6	Cross-compilation Type	10
7	BeagleBone Black Booting Sequence	11
8	BeagleBone Version	15
9	Makefile file for Linux device driver compilation	16
10	Per-device data structure for the device driver	17
11	Implementation of exit routine	18
12	Implementation of gpio_open entry point	19
13	Implementation of gpio_close entry point	19
14	Implementation of gpio_read entry point	20
15	Implementation of gpio_write entry point	21
16	Implementation of Interrupt Handling Function	22
17	User Application	23
18	Project Setup	23
19	Interrupt Notification	24
20	GPIO Pin Change	24
21	Read operation	24

# Introduction

The project was aimed at implementing a General Purpose Input/Output (GPIO) device driver for the BeagleBone Black Rev C platform. Specific attention was given to implement the device driver based on the Linux character device driver. Each of the GPIO pins on BeagleBone Black is exposed to userspace for use by a device file in the `/dev` directory. While a dynamically assigned major number was used to identify the device driver associated with the GPIO device, a minor number was used by the kernel to differentiate between GPIO pins that the device driver controls. Four entry points were implemented to handle low-level hardware GPIO resources on behalf of system calls requested by a userspace application. Besides, interrupt handling was implemented as an experimental feature to see how a device driver supports interrupt in an embedded Linux system.

The GPIO device driver could be used on the BeagleBone Black platform by loading it either as a kernel module or as an integral component of the Linux kernel. One or more user space processes could open a device file for reading the logic level of a GPIO pin, controlling the direction of a GPIO pin, and setting its logic level to high state or low state. Besides, interrupt handling feature was tested; the result showed that more than one GPIO pin can be accessed as in our case one is input and another is output.



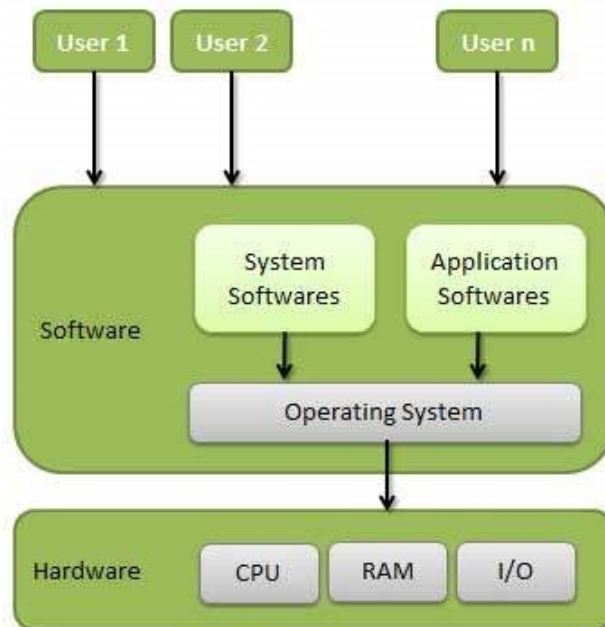
# Literature Survey

## 2.1 Microprocessor

A microprocessor is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together. Microprocessors help to do everything from controlling elevators to searching the Web. Everything a computer does is described by instructions of computer programs, and microprocessors carry out these instructions many millions of times a second.

Microprocessors were invented in the 1970s for use in embedded systems. The majority are still used that way, in such things as mobile phones, cars, military weapons, and home appliances. Some microprocessors are microcontrollers, so small and inexpensive that they are used to control very simple products like flashlights and greeting cards that play music when you open them. A few especially powerful microprocessors are used in personal computers.

## 2.2 Operating System



**Figure 1 OS Architecture**

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and

controlling peripheral devices such as disk drives and printers. Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Following are some of important functions of an operating System:

1. Memory Management
2. Processor Management
3. Device Management
4. File Management
5. Security
6. Control over system performance
7. Job accounting
8. Error detecting aids
9. Coordination between other software and users

## **2.3 Linux Operating System**

Linux is a family of open source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds. Linux is typically packaged in a Linux distribution.

Distributions include the Linux kernel and supporting system software and libraries, many of which are provided by the GNU Project. Popular Linux distributions include Debian, Fedora, and Ubuntu. Commercial distributions include Red Hat Enterprise Linux and SUSE Linux Enterprise Server.

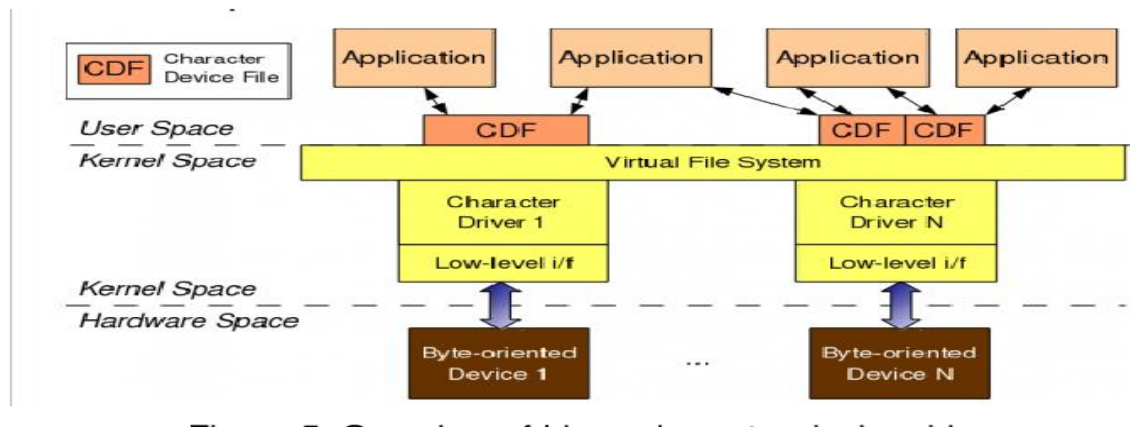
Linux was originally developed for personal computers based on the Intel x86 architecture, but has since been ported to more platforms than any other operating system.

## **2.4 Device Driver**

In Linux there are essentially three kinds of devices: network devices, block devices and, character devices. Network devices are represented as network interfaces and are visible when issuing ifconfig command from userspace. Block devices provide userspace applications with access to raw storage hardware devices (e.g. hard disks).

These are visible to userspace as device files in /dev directory. The last type is character devices. They provide userspace applications with access to other types of devices such as input, serial, graphics, or sound, to name just a few. The design of the Linux GPIO device driver for BeagleBone Black was based on the Linux character device driver.

For that reason, this section will look at the internals of the Linux character device driver. Linux character device drivers are byte-oriented operations or character-oriented operations in the C lingo. Because a large number of hardware devices are character oriented, a majority of Linux device drivers are character device drivers. Linux character device drivers are the kernel code that gets access to data from a hardware device sequentially. They are capable of capturing raw data from several kinds of devices such as serial ports, mice, tapes, or memory.



**Figure 2 Kernel Space Interaction**

As illustrated by figure 1, in order for a userspace application to use a character-oriented device in hardware space, it must use a corresponding character device driver in kernel space. A character device file appeared in /dev directory acts as a communication channel between a userspace application and a character device driver. A userspace application performs usual file operations (e.g. open, read, write, close) on the character device file. Those operations are then mapped into the entry points in the linked character device driver by the virtual file system (VFS). Those entry points finally perform low-level accesses to the actual device to get desired result.

## 2.5 GPIO

A GPIO (general-purpose I/O) is a single electrical signal that the CPU can either set to one of two values — zero or one, naturally — or read one of those values from (or both). Either way, a GPIO does not seem like a particularly expressive device. But, at their simplest, GPIOs can be used to control LEDs, reset lines, or pod-bay door locks. With additional "bit-banging" logic, GPIOs can be combined to implement higher-level protocols like i2c or DDC — a frequent occurrence on contemporary systems. GPIOs are thus useful in a lot of contexts. GPIO lines seem to be especially prevalent in embedded systems; even so, there never seems to be enough of them. As one might expect, a system with dozens (or even hundreds) of GPIOs needs some sort of rational abstraction for managing them.

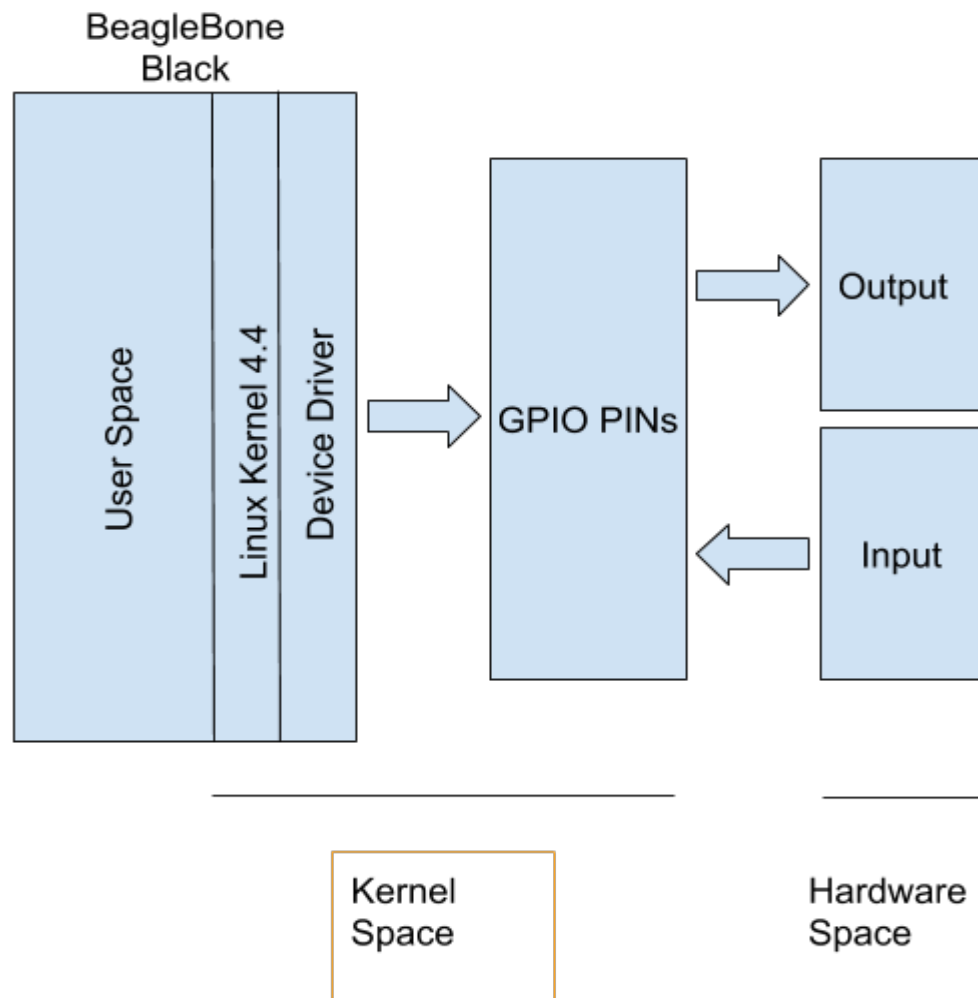
## 2.6 Beaglebone Black

The BeagleBone Black is the latest addition to the BeagleBoard.org family and like its predecessors, is designed to address the Open Source Community, early adopters, and anyone interested in a low cost ARM Cortex-A8 based processor. It has been equipped with a minimum set of features to allow the user to experience the power of the processor

and is not intended as a full development platform as many of the features and interfaces supplied by the processor are not accessible from the BeagleBone Black via onboard support of some interfaces. It is not a complete product designed to do any particular function. It is a foundation for experimentation and learning how to program the processor and to access the peripherals by the creation of your own software and hardware. It also offers access to many of the interfaces and allows for the use of add-on boards called capes, to add many different combinations of features. A user may also develop their own board or add their own circuitry.

BeagleBone Black is manufactured and warranted by Circuitco LLC in Richardson Texas for the benefit of the community and its supporters. In addition, Circuit provides the RMA support for the BeagleBone Black.

# Project Architecture



**Figure 3 Project Architecture**

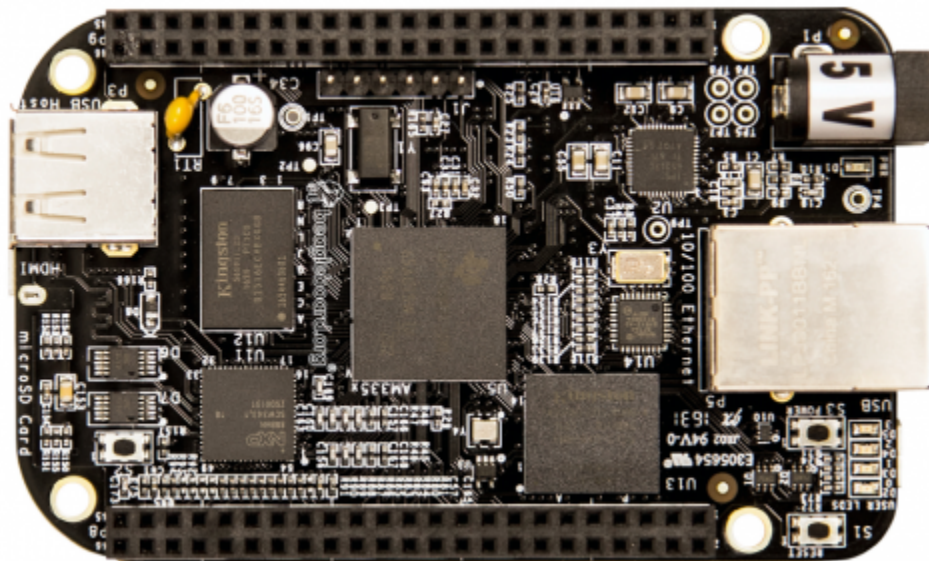
The architecture is defining how our project is accessing the GPIO pins of BeagleBone Black through device driver. GPIO pins of Beagle Bone are configured as I/P and O/P pins. The project has two parts: First Part is Compiling the Linux Kernel version 4.4 and customized it for ARM architecture & ported on BBB. The GPIO device driver has been cross compiled using ARM cross compilation tool chain. The second part was writing a device driver for GPIO pins, which will expose Hardware resource GPIO pins to userspace through /dev directory. For this we have created two device files in /dev. First *GPIOIN* to test input and second *GPIOOUT* to test output. Any of the GPIO pins can be selected as *GPIOIN* and *GPIOOUT* by writing on these device files. An user-space app is used to test the LED & switch connected to the GPIO pins.

# Hardware Description

## 4.1 Development board

The Beagle Bone Black rev C platform was chosen for the project to implement the Linux GPIO device driver. This is due to the fact that Beagle Bone Black is a Open source Linux computer. The platform is widely used by hobbyists and professional embedded developers around the world. Therefore, there is a large support from the community when it comes to getting help or information. For convenience, Beagle Bone Black will be used when referring to the Beagle Bone Black rev C platform throughout this report. Figure 1 shows the Beagle Bone Black platform used in the project.

The BeagleBone Black is the newest member of the BeagleBoard family. It is a lower-cost, high-expansion focused BeagleBoard using a low cost Sitara AM3359AZCZ100 Cortex A8 ARM processor from Texas Instruments.



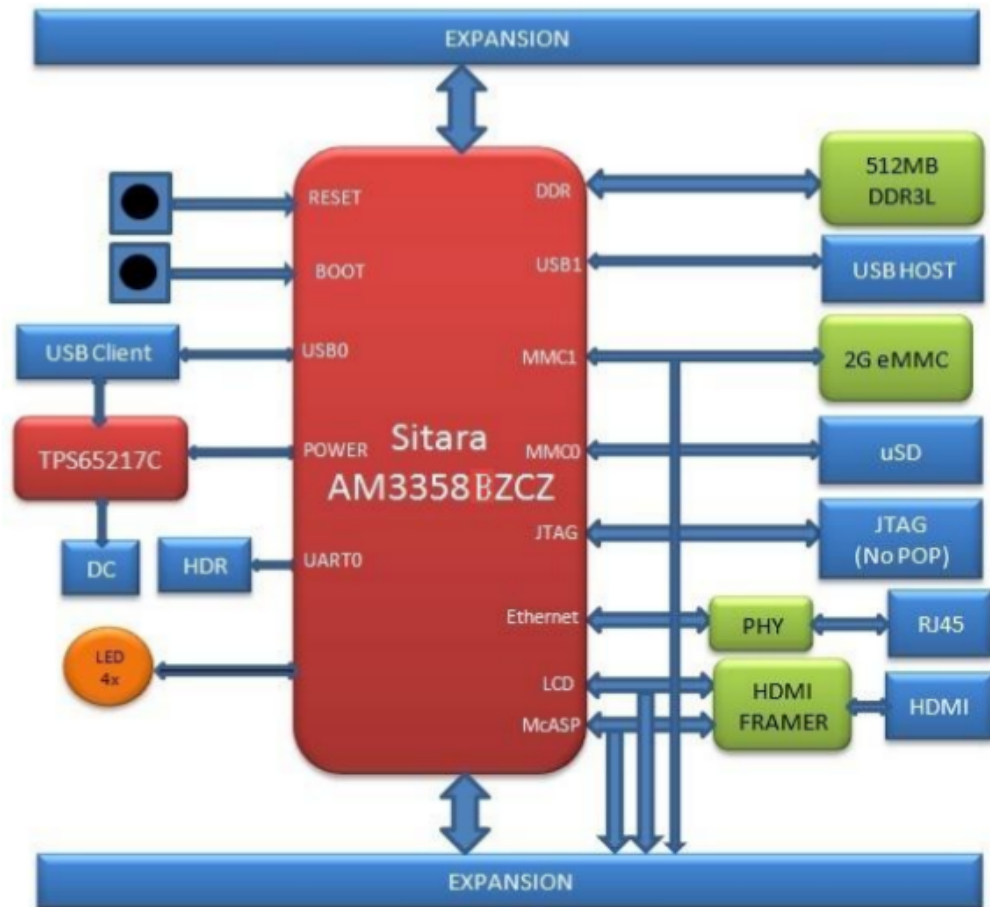
**Figure 4 Beagle Bone Rev C Board**

BeagleBone is a fan-less embedded computer which is equipped with a wide range of peripherals such as Universal Serial Bus (USB), Ethernet, Universal Asynchronous Receiver/Transmitter (UART), High-Definition Multimedia Interface (HDMI), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), audio codec, 2x 46 pin headers

and so on. The heart of the platform is a TI AM335x System on Chip (SoC) that has an ARM Cortex-ARM A-8 1GHz core processor, an integrated graphics processor Power VR Graphic Processing Unit (GPU), 512MB DDR3 of RAM, and an SD card in place of a hard disk drive. The platform is powered from an external power supply via a USB micro connector. It is recommended from the official Beagle Bone Black website that the board be powered from a good-quality power supply that gives at least 1A at 5V. This is because if the board is powered by a USB micro connector, there will not be enough current and, as a result, the board may reboot sometimes when it draws too much power.

## 4.2 Hardware Processor

The revision C board has moved to the Sitara AM3358BZCZ100 device.



**Figure 5 AM3358BZCZ100 Processor Diagram**

It has following features:

- Up to 1-GHz Sitara ARM Cortex-A8 32-Bit RISC Processor
- NEON SIMD Coprocessor
- 32KB of L1 Instruction and 32KB of Data Cache with Single-Error Detection (Parity)
- 256KB of L2 Cache with Error Correcting Code (ECC)
- 176KB of On-Chip Boot ROM
- 64KB of Dedicated RAM
- Emulation and Debug - JTAG
- Interrupt Controller (up to 128 Interrupt Requests)
- On-Chip Memory (Shared L3 RAM)
- 64KB of General-Purpose On-Chip Memory Controller (OCMC) RAM
- Accessible to all Masters
- Supports Retention for Fast Wakeup
- External Memory Interfaces (EMIF)
- mDDR(LPDDR), DDR2, DDR3, DDR3L Controller:
- mDDR: 200-MHz Clock (400-MHz Data Rate)
- DDR2: 266-MHz Clock (532-MHz Data Rate)
- DDR3: 400-MHz Clock (800-MHz Data Rate)
- DDR3L: 400-MHz Clock (800-MHz Data Rate)
- 16-Bit Data Bus
- 1GB of Total Addressable Space
- Supports One x16 or Two x8 Memory Device Configurations
- General-Purpose Memory Controller (GPMC)
- Flexible 8-Bit and 16-Bit Asynchronous Memory Interface with up to Seven Chip Selects (NAND, NOR, Muxed-NOR, SRAM)

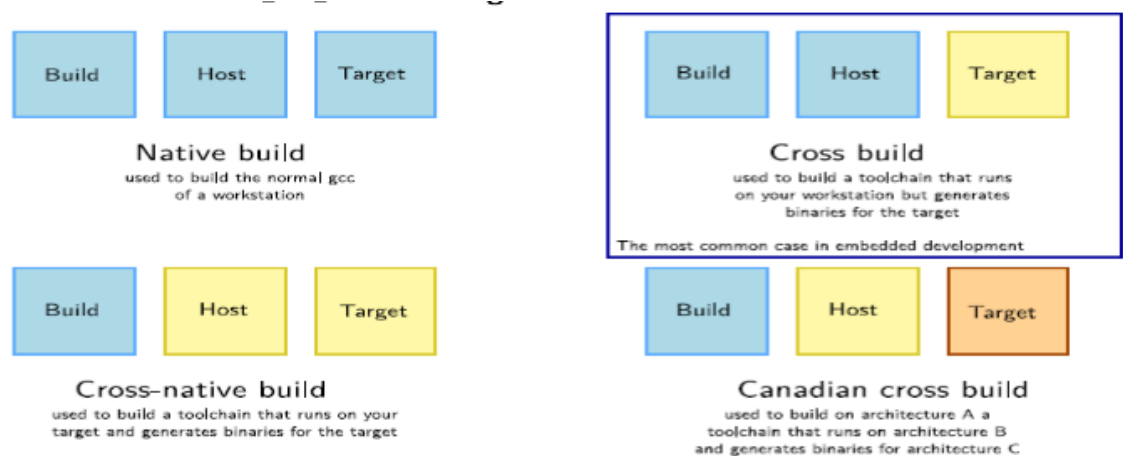


# Software Requirements & Specification

Our Project have two parts. 1<sup>st</sup> was Compilation of Linux Kernel for Beagle Bone Board and 2<sup>nd</sup> was Developing the Device Driver For Beagle Bone Board to control GPIO. As we were doing compiling the Kernel and Developing the Device Driver on different machine, so we used the cross compilation tool chain, Development enviornment.

Below Figure describes the embedded linux system architecture which requires a host for building an linux based embedded system. In embedded linux system development, development host is the workplace where the required files are cross-compiled for the targeted embedded system. The software components required for embedded system development are;

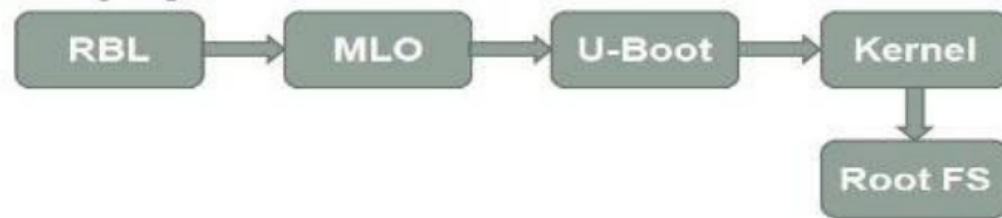
**5.1 Cross-compilation Toolchain:** A cross-compiler is capable of generating executable file for a different platform than the host. Toolchain provides cross-compilation support for several architectures. Choosing right toolchain plays a crucial role in building a kernel. Below mentioned **Fig.** demonstrates types of cross-compilations. However, using the latest version of a toolchain is not always preferable as, some of the latest versions do not build the kernel properly [3]. The cross-compilation toolchain used for this project is: “**gcc-linaro-5.4.1-2017.01-x86\_64\_arm-linux-gnueabi**”



**Figure 6 Cross-compilation Type**

**5.2 Bootloader:** It is the first section of code that executes after the system is powered ON or reset on any platform. The bootloader places the OS of a computer into memory. There are various bootloader available for specific architectures. The bootloader used in this project is u-boot as BeagleBone Black SoC is based on arm architecture. Booting sequence of BeagleBone Black needs to be understood thoroughly in order to develop an

embedded linux system on BeagleBone Black. **Fig.** shows the BeagleBone Black booting sequence.



**Figure 7 BeagleBone Black Booting Sequence**

**5.2.1 RBL (ROM Bootloader):** When the system is first booted, the CPU invokes the reset vector to start the code at a known location in ROM. This code performs minimal clocks, memories and peripheral configurations. Moreover, it searches the booting devices for a valid booting image and loads the MLO into SRAM and executes it.

**5.2.2 MLO (Memory Loader):** This part of code sets up the pin multiplexing, initializes clocks and memory and loads the u-boot image to the DDR memory and executes it.

**5.2.3 U-boot:** This part of code performs some additional platform initialization, sets the boot arguments and passes control to the kernel image. We used the uboot version “**u-boot-2017.03**”

**5.3 Kernel:** It decompresses the kernel into SDRAM, sets up peripherals and mounts the linux file-system (ext-4 for BeagleBone Black) we used the kernel “**Linux Kernel V4.4**”.

**5.4 Root File-system:** The root file-system contains many system-specific configuration files. Possible examples include a kernel that is specific to the system, a specific hostname, etc. This means that the root file-system isn't always shareable between networked systems. Keeping it small on servers in networked systems minimizes the amount of lost space for areas of un-shareable files. It also allows workstations with smaller local hard drives. The root file-systems for BeagleBone Black are available on the internet. We used the “**debian-8.7-minimal-armhf-2017-03-02**”.

## **5.5 Steps for Setting up Embedded Linux for BeagleBone Black:**

### **5.5.1 Compiling Linux For Beagle Bone Board:**

We compiled the Linux Kernel for Board using below commands with cross chain compiler:

```
#make ARCH=arm bb.org_defconfig V=1
```

**#make menuconfig**

*//Check processor type. It is x86*

**#make ARCH=arm bb.org\_defconfig V=1**

*//Configure for Beaglebone*

**#make ARCH=arm CROSS\_COMPILE=\${CC} menuconfig**

*//Check processor type. It is ARM*

**#make ARCH=arm CROSS\_COMPILE=\${CC} zImage 2>&1 |  
tee ../32\_make\_zImage.log**

*//Make kernel. Add -j 6 to speed up*

*//zImage is compressed version of the Linux kernel image that is self-extracting.*

*// zImage is the actual binary image of the compiled kernel. It's what the boot loader will load and attempt to execute*

**#make ARCH=arm CROSS\_COMPILE=\${CC} modules 2>&1 |  
tee ../33\_make\_modules.log**

*//Make modules and firmware. Add -j 6 to speed up*

*//Make modules and firmware.*

*//The make modules command will compile the modules, leaving the compiled binaries in the build directory.*

**#make ARCH=arm CROSS\_COMPILE=\${CC} dtbs 2>&1 | tee ../34\_make\_dtbs.log**

*//Make device tree blobs*

*/"Device Tree Blob" (DTB) file is loaded into memory by U-Boot, and a pointer to it is passed to the kernel. This DTB file describes the system's hardware layout to the Linux kernel, allowing for platform-specific code to be moved out of the kernel sources and replaced with generic code that can parse the DTB and configure the system as required.*

**#make ARCH=arm CROSS\_COMPILE=\${CC} modules\_install -  
INSTALL\_MOD\_PATH=./my\_modules 2>&1 | tee ../35\_modules\_install.log**

//make modules\_install will make sure that there are compiled binaries (and compile the modules, if not) and install the binaries into kernel's my\_modules directory.

```
#make ARCH=arm CROSS_COMPILE=${CC} firmware_install |  
tee ../36_firmware_install.log
```

//make firmware\_install sends the firmware to my\_modules directory.

### 5.5.2 Building uBoot For Beagle Bone Board:

We Build the u-boot for Embedded Linux for Arm Processor using below commands with cross chain compiler:

```
# patch -p1 < 0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch  
# patch -p1 < 0002-U-Boot-BeagleBone-Cape-Manager.patch  
    //patch - apply a different file to an original.  
    //uEnv : A uEnv.txt boot configuration text file is written that can be used to pass  
parameters to linux kernel. It sets the boot parameters for your board.  
# make ubootversion  
    // To find out version  
#make distclean V=1  
    //Clean up everything  
    //V=1 option controls the degree and type of verbosity that you will be exposed to  
during the make process.  
#make ARCH=arm CROSS_COMPILE=${CC} am335x_boneblack_defconfig V=1  
    //Set ARCH=arm for cross-compile  
    //And configure it for BeagleboneBlack  
    //All configuration required for BeagleboneBlack are specified here.  
#make ARCH=arm CROSS_COMPILE=${CC} 2>&1 | tee ../22_make_uboot.log  
    //Make all
```

### 5.5.3 Porting Linux on SD-Card:

We Build the u-boot for Embedded Linux for Arm Processor using below commands with cross chain compiler:

```
#sudo dd if=/dev/zero of=${SDCARD} bs=1M count=10  
    //dd command is used to copy a file,converting and formatting according to the  
operands.  
    //if=FILE :read from the file instead of stdin  
    //of=FILE :write to FILE instead of stdout  
    //bs=BYTES:read and write up to BYTES mensioned at a time  
    //count=N:copy only N input blocks  
#sudo parted -s $SDCARD mklabel msdos
```

//mklabel creates a new disc label of type label-type.the new disk label will have no partitions.

//this command won't technically destroy the data,but it will make it unusable.Label-type must be one of these supported disk labels: bsd,loop,gpt,mac,msdos,pc98,sun

**#sudo parted -s \$SDCARD unit MB mkpart primary ext4 -- 4 100%**

//-s :--script :never prompts for user intervention

//mkpart : part-type[fs-type] start end

//mkpart : primary [ext4] 4 100%

//part-type should be one of the 'primary', 'logical','extended'.

**#sudo parted -s \$SDCARD set 1 boot on**

//set :change the state of the flag on partition to state.state should be 'on' or 'off'

**#sudo parted -s \$SDCARD print**

//Make sure there is single partion starting from 4mb till end and has flag - 'boot'

//print: Display the partition table

**#sudo mkfs.ext4 -L rootfs -O ^metadata\_csum,^64bit \${SDCARD}1**

//mkfs:build a linux filesystem

//mkfs [options] [-t type] [fs-options] device [size]

//64bit :the number of blocks to be used for the filesystem.

#### 5.5.4 Mounting Rootfs on SD-Card:

We mounted prebuild rootfs for our Embedded Linux for Arm Processor using below commands with help cross chain compiler:

**#tar xJf debian-8.7-minimal-armhf-2017-03-02.tar.xz**

//xjf :x:extract j:bzip2 f:file

//tar command is used to extract the zip files.

**#sudo tar xfpv debian-8.7-minimal-armhf-2017-03-02/armhf-rootfs-debian-jessie.tar -C /media/sunbeam/rootfs/**

**#sudo chown root:root /media/sunbeam/rootfs/**

/chown changes the user and group ownership of each given file.here owner of the '/media/sunbeam/rootfs/' is changed to the root and also the group of it is also changed to the root.

**# sudo chmod 755 /media/sunbeam/rootfs/**

//chmod changes the file mode bits of each given file according to mode,which can be either symbolic or octal representation.

//755-user:rw,group:r-x,owner:r-x

here user and group both are 'root'

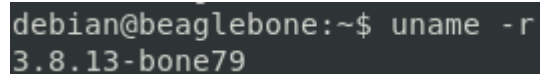
```
#sudo sh -c "echo '/dev/mmcblk0p1 / auto errors=remount-ro 0 1' >>  
/media/sunbeam/rootfs/etc/fstab"
```

#### 5.5.5. Setting up Bootloader in first partiton of disk:

we used folling commands

```
sudo dd if=./MLO of=${SDCARD} count=1 seek=1 bs=128k  
sudo dd if=./u-boot.img of=${SDCARD} count=2 seek=1 bs=384k
```

after above steps linux was setup on board:



```
debian@beaglebone:~$ uname -r  
3.8.13-bone79
```

Figure 8

BeagleBone

Version

#### 5.6 Communication Between HOST machine and Target Machine:

Communication between the host development system and the embedded Linux target is done via Secure Shell (SSH) protocol. In order to use the SSH protocol as a means of communication between the host and the target, an SSH daemon was installed in Beagle Bone Black, which makes the target become a server, whereas an SSH client was installed in the host development system, thus making the host a client.

We used the following to login into BBB

```
# ssh debain@<beaglebone-ip>
```

#### 5.7 Make File for compilation Linux Device Driver:

The modules directory contains kernel modules which BBB will install when it is booting. It is, undoubtedly, impossible to compile a Linux device driver without a Makefile file. Therefore, inside the Linux Modules and various subdirectories have Makefile files used by the Vim IDE during Linux device driver compilation process.



```
Target := GPIO_Driver
obj-m := ${Target}.o
KDIR :=/home/dhruvshn/BeagleBoneBlack/Linux_Porting/kernel/linux-4.4

all:
    make ARCH=arm CROSS_COMPILE=${CC} -C $(KDIR) M=`pwd` modules

install:
    scp ${Target}.ko debian@192.168.0.1:/home/debian/test_led

clean:
    make -C $(KDIR) M=`pwd` clean
```

**Figure 9 Makefile file for Linux device driver compilation**



# Source Code Explanation

## 6.1 Implementation of Per-device Structure:

The design of the per-device structure, is as per below, This section gives more details on how each of them was implemented by analyzing their source code. Below **Image** shows a snippet of code that was implemented for the per-device data structure.

```
typedef struct my_priv_struct
{
    struct cdev mycdev;
    int dev_num;
}my_priv_t;
```

**Figure 10 Per-device data structure for the device driver**

The very first pieces of code implemented for the GPIO device driver were the per-device data structure. This data structure is named my\_priv\_struct. In the project almost every name of variables, structures, and functions were named beginning with the prefix gpio\_. Each instance of the per-device data structure represents a GPIO pin. The first element of the structure is an instance of struct cdev structure. This is the internal structure of the Linux kernel that represents character devices. This structure is defined in <linux/cdev.h> header file. Second no is dev\_num which shows which device is selected by user.

When the GPIO kernel module is loaded by the kernel upon request, the initialization routine will be called. First of all, DESDGPIO\_init () invokes alloc\_chrdev\_region() to dynamically request a range of unused character device numbers. Variable first of data type dev\_t contains the allocated major number if the call is successful. The second and third arguments passed into alloc\_chrdev\_region() require the function to allocate NUM\_GPIO\_PINS number of minor devices starting from zero. The last argument is the device name which will appear in /proc/devices when the device driver is registered successfully. This argument is defined using a macro called DEVICE\_NAME. The output from cat /proc/devices command. When the BBB GPIO kernel module has been registered successfully, displaying the content of /proc/devices directory shows the major number allotted for the device which, as shown in figure, is 245 in the first column. After that, the initialization routine creates a class under the virtual filesystem which is mounted at /sys by using function class\_create(). Together with function class\_create(), the function device\_create() results in the generation of 2 uevents. Since there are, in total, the gpio\_init() loops 2 times. With each iteration, it requests the kernel to allocate memory dynamically for the per-device structure using kernel API function kmalloc(). It

should be noted that the cdev is embedded inside each per-device structure. When the kernel allocates memory for the per-device structure, cdev will also be allocated.

## 6.2 Implementation of Initialization and Exit Routines:

Image shows the implementation of the exit routine. When a kernel module is being unloaded from the system, the kernel calls this function to take care of that job. The exitfunction of the GPIO device driver is called DESDGPIO\_exit(). Basically, its job is to deallocate resources that were requested from the initialization routine. Firstly, it unregisters from the system a range of NUM\_GPIO\_PINS device numbers. To be specific, thevalue of NUM\_GPIO\_PINS is defined to be 2. There are 2 per-device structures representing each GPIO pin. These structures were allocated dynamically by the kernel. In order to de-allocate those, the kernel API function kfree() is used.

```
static void __exit DESDGPIO_exit(void){
    int i = devcnt;
    free_irq(irqNumber, NULL);
    for(i=i-1; i>=0; i--){
        cdev_del(&devices[i].mycdev);
    }
    for(i = 0; i < devcnt; i++){
        {
            devno = MKDEV(major,i);
            device_destroy(s_pDeviceClass, devno);
            printk(KERN_INFO "%s: device_destroy() delete device file.\n", THIS_MODULE->name);
        }
    }
    class_destroy(s_pDeviceClass);
    unregister_chrdev_region(first, devcnt);
    kfree(devices);
    gpio_set_value(gpioLED, 0); // Turn the LED off, indicates device was unloaded
    gpio_direction_output(gpioLED, 0);
    gpio_free(gpioLED); // Free the LED GPIO
    gpio_direction_output(gpioButton,0);
    gpio_free(gpioButton);
    printk(KERN_INFO "%s: exited from the GPIO Device Driver\n", THIS_MODULE->name);
}
```

**Figure 11 Implementation of exit routine**

As explained earlier, the exit routine should set the direction of all GPIO pins to output and their logic levels to low. For that reason, gpio\_direction\_output() is called. This function is defined in <linux/gpio.h> Linux header file. The resource for GPIO pins were requested by gpio\_request\_one() before, so gpio\_free() will take care of releasing those resources. Finally, all the information that populates the virtual filesystem mounted at /sys directory and all the uevents created by device\_create() are destroyed by class\_destroy() and device\_destroy(), respectively.

## 6.3 Implementation of the Entry Points for the GPIO Device Driver

a user space application opens a corresponding device file, the `gpio_open()` entry point will be invoked by the kernel. Image shows the implementation of the `gpio_open()` entry point.

```
static int gpio_open(struct inode *pinode, struct file *pfile)
{
    printk(KERN_INFO "%s: gpio_open() is called.\n", THIS_MODULE->name);
    pfile->private_data = container_of(pinode->i_cdev, my_priv_t, mycdev);
    return 0;
}
```

**Figure 12 Implementation of `gpio_open` entry point**

The implementation of the open entry point was first started by obtaining a minor number that corresponds to a GPIO pin number. Mapping between a minor number and a GPIO pin number was specified by design. The inode which is passed as an argument to `gpio_open()` holds the address of the cdev structure allocated during initialization. Since struct cdev structure is nested inside the my\_priv\_struct per-device structure as show in Image, the `gpio_dev` structure is the container of the struct cdev structure. In order to access data elements of the per-device structure, the address of it needs to be elicited. The open entry point uses the kernel helper function `container_of()` to do this. The address of `my_priv_struct` structure is needed not only in the open entry point but also in other entry points including release, read, and write. Hence, this address is assigned to the `private_data` data field that is part of the struct file structure, the second argument.

```
static int gpio_close(struct inode *pinode, struct file *pfile)
{
    printk(KERN_INFO "%s: gpio_close() is called.\n", THIS_MODULE->name);
    return 0;
}
```

**Figure 13 Implementation of `gpio_close` entry point**

When a device file is closed by a user space application, the kernel invokes the release entry point. The release entry point for the BBB GPIO device driver is called `gpio_close`. Two arguments are passed into the `gpio_close` entry point, namely inode and filp. The entry point does not make use of the filp pointer; only the inode is used to get the address of the per-device structure. It can, of course, elicit the per-device structure's address from the `private_data` filed inside the struct file struture as an alternative.

```

ssize_t gpio_read(struct file *pfile, char __user *pbuf, size_t size, loff_t *pfpes)
{
    my_priv_t *dev = (my_priv_t*)pfile->private_data;
    int max_bytes, bytes_to_read, nbytes;
    char status[100];
    int led_status = 0;
    max_bytes = 50- *pfpes;
    printk(KERN_INFO "%s: gpio_read() is called.\n", THIS_MODULE->name);
    bytes_to_read = size < max_bytes? size : max_bytes;
    printk(KERN_INFO "%s: bytes_to_read determined.\n", THIS_MODULE->name);
    led_status = LedOn?1:0;

    if(bytes_to_read == 0)
    {
        printk(KERN_INFO "%s: No data left on device.\n", THIS_MODULE->name);
        return 0;
    }
    if(dev->dev_num == 0)
    {
        sprintf(status,"led staus: %d, LED gpio: %u",led_status,gpioLED);
        printk(KERN_INFO "%s: status :%s\n", THIS_MODULE->name,status);
    }
    else
    {
        sprintf(status,"interrupt count: %u, Button gpio: %u",interrupt_count,gpioButton);
        printk(KERN_INFO "%s: status :%s\n", THIS_MODULE->name,status);
    }

    nbytes = bytes_to_read - copy_to_user(pbuf,status,bytes_to_read);
    *pfpes = *pfpes + nbytes;
    return 0;
}

```

**Figure 14 Implementation of gpio\_read entry point**

The last two entry points for the GPIO device driver are gpio\_read entry point and gpio\_write entry point. Listing 6 shows the implementation of the gpio\_read entry point. This entry point was the easiest and simplest part of the device driver implementation, so it will be explained briefly. The gpio\_read entry point first obtains a GPIO pin number from a device file by using the device struct member dev\_num. After that it gets the state of the GPIO pin with the help of the kernel function gpio\_get\_value(). This function returns either zero or one. Then the value returned from function will be converted into a character and saved to a temporary variable called byte. Finally this value will be copied to the userspace by using the kernel API function put\_user(). It should be noted that copying from kernel space to userspace cannot be done by using normal C library functions. Instead, this has to be done by using kernel API functions such as put\_user() or copy\_to\_user() functions.

Upon completion of copying, the gpio\_read entry point exits and returns the number of bytes copied to the userspace buffer pointed to by the buf pointer, the second argument passed into the gpio\_read() entry point.

```

ssize_t gpio_write(struct file *pfile, const char __user *pbuf, size_t size, loff_t *pfp)
{
    my_priv_t *dev = (my_priv_t*)pfile->private_data;
    int max_bytes, bytes_to_write, nbytes;
    char value[10];
    static unsigned int received_value;
    printk(KERN_INFO "%s: gpio_write() is called.\n", THIS_MODULE->name);
    max_bytes = 10 - *pfp;
    bytes_to_write = size < max_bytes ? size : max_bytes;
    if(bytes_to_write == 0)
    {
        printk(KERN_INFO "%s: No space left on device.\n", THIS_MODULE->name);
        return -ENOSPC;
    }
    nbytes = bytes_to_write - copy_from_user(value, pbuf, bytes_to_write);
    sscanf(value, "%u", &received_value);
    printk(KERN_INFO "%s: received GPIO value: %u", THIS_MODULE->name, received_value);
    if(dev->dev_num == 0)
    {
        ledOn = 0;
        gpio_direction_output(gpioLED, ledOn);
        gpio_free(gpioLED);
        gpioLED = received_value;
        gpio_request(gpioLED, "sysfs");
        gpio_direction_output(gpioLED, ledOn);
        printk(KERN_INFO "%s: GPIOOUT opened!!!", THIS_MODULE->name);
    }
    else
    {
        int result;
        printk(KERN_INFO "%s: GPIOIN opened!!!", THIS_MODULE->name);
        // gpio_direction_output(gpioButton, 0);
        free_irq(irqNumber, NULL);
    }
}

```

**Figure 15 Implementation of gpio\_write entry point**

Similarly, the write entry point performs the task of copying data between a kernel space and a userspace. The last implementation of entry points for the GPIO device driver was the gpio\_write() entry point. The struct file structure is passed as an argument into the gpio\_write entry point. The private\_data field contained in this struct file structure holds the address of the per-device structure which will be needed in this entry point. This is due to the fact that the address of the per-device structure was assigned to the private\_data field when the gpio\_open entry point was called. This assignment can be seen from the third to last line in Image . Passed as the third argument to the write entry point is a count which indicates the number of bytes a userspace application has written into the buffer which is pointed to by the buf pointer, the second argument. Based on this information, the gpio\_open entry point then reads count numbers of bytes from the buffer pointer to by buf pointer into its own temporary buffer in the kernel space called kbuf. Once the reading process has been completed and kbuf contains what has been read, the gpio\_open entry point will compare the value of the kbuf with a set of commands specified in the design phase .

Upon receiving this command, the write entry point will set the `irq_perm` data field in the per device structure to false indicating that the request for interrupt is not permitted. Any command received by the `gpio_write` entry point other than the ones specified in table 2 will be ignored, and `EINVAL` error code will be returned. Finally, the entry point will update the current position of the `f_pos` pointer by count positions and return the number of bytes received.

#### 6.4 Implementation of Interrupt Handling Functions:

```
static irqreturn_t gpio_irq_handler(unsigned int irq, void *dev_id){
    printk(KERN_INFO "%s: interrupt received!\n", THIS_MODULE->name);
    ledOn = !ledOn;
    gpio_set_value(gpioLED, ledOn);           // Set the physical LED accordingly

    interrupt_count++;
    return IRQ_HANDLED;                       // Announce that the IRQ has been handled correctly
}
```

**Figure 16 Implementation of Interrupt Handling Functions**

Regarding the interrupt handling function `irq_handler()`, this function will be called when an interrupt occurs. Due to the nature of interrupt handling, this function was implemented in such a way that it is fast and it does not contain any blocking functions. The interrupt handler toggle the `GPIOWRITE` value an interrupt occurred. By trial and error, it was determined that a time duration of 200 milliseconds is good enough to avoid voltage glitches caused by contact bouncing. Since an interrupt handler can be interrupted by another interrupt, The `irq_handler` interrupt handle is not of practical use since this feature is still experimental. It simply logs the string “interrupt received!” into the kernel logging system.

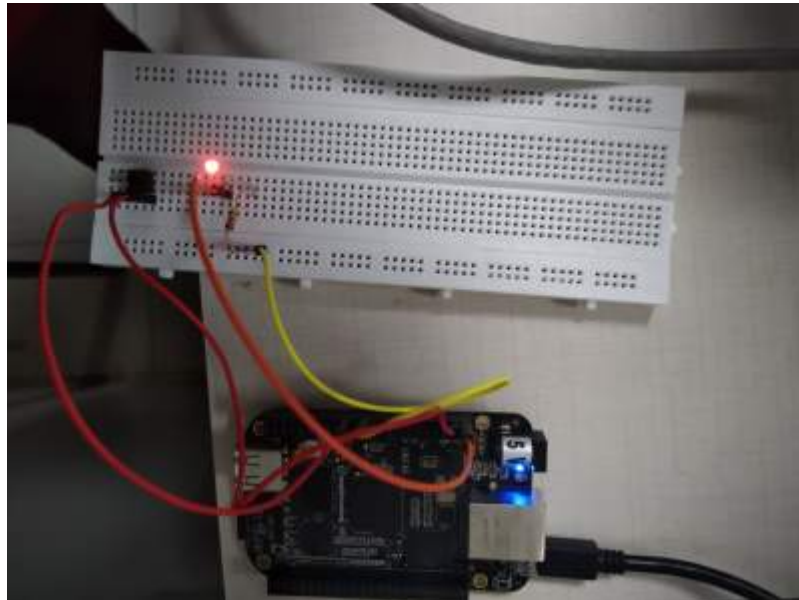
# Testing

The first test case was carried out to verify the output functionality offered by the GPIO device driver. An application program called `output_test` was run in the userspace to test the driver, and it has the following syntax: `./user_GPIO_wr_prog <Input GPIO no.> <Output GPIO no.>` where `<Input GPIO no.>` refers to the Input GPIO pin number that is passed as an argument to the program for setting a GPIO pin as Interrupted Input. `<Output GPIO no.>` refers to the Output GPIO pin number that is passed as an argument to the program for setting a GPIO pin as Output whose value will be toggled at every interrupt of GPIOIN.

A simple circuit was constructed on a breadboard for testing the output functionality.  $200\Omega$  resistors was connected in series with the 1 RED LED. The output from GPIO pin was wired to the anode terminal of an LED. The cathode terminal of the LED was connected in series with a  $200\Omega$  resistor which in turn was connected to a GND pin from the breadboard. Below is the output of the application program when invoked from the command line with arguments to select GPIOPIN.

```
debian@arm:~$ sudo test led/user/user GPIO wr prog.out 49 7
debian@arm:~$ sudo test led/user/user GPIO rd prog.out
status of GPIO OUT is: led staus: 0, LED gpio: 49
status of GPIO IN is: interrupt count: 2, Button gpio: 7
```

**Figure 17 User Application**



**Figure 18 Project Setup**

On every Interrupt at GPIOIN interrupt counter will increase and It will be shown something like this in kernel logs ;

```
[ 860.483752] GPIO_Driver: interrupt received!  
[ 860.679275] GPIO_Driver: interrupt received!  
[ 861.241578] GPIO_Driver: interrupt received!  
[ 862.366641] GPIO_Driver: interrupt received!  
[ 862.500866] GPIO_Driver: interrupt received!  
[ 862.511711] GPIO_Driver: interrupt received!  
[ 863.156182] GPIO_Driver: interrupt received!  
[ 863.303445] GPIO_Driver: interrupt received!  
[ 863.831551] GPIO_Driver: interrupt received!  
[ 864.561241] GPIO_Driver: interrupt received!  
[ 864.667914] GPIO_Driver: interrupt received!
```

**Figure 19 Interrupt Notification**

On changing Interrupt pin it will reflect in Kernel logs, through /dev/GPIO\* and it is working successfully.

```
[ 1574.715330] GPIO_Driver:received GPIO value: 7  
[ 1574.719900] GPIO_Driver: GPIOIN opened!!!  
[ 1574.735511] GPIO_TEST: New IRQ: 34  
[ 1574.744309] GPIO_Driver: gpio close() is called.
```

**Figure 20 GPIO Pin Change**

On reading the GPIO Device status through ./user\_GPIO\_rd\_prog.out it is showing successfully.

```
debian@arm:~$ sudo test_led/user/user_GPIO_rd_prog.out  
status of GPIO OUT is: led staus: 0, LED gpio: 49  
status of GPIO IN is: interrupt count: 2, Button gpio: 7
```

**Figure 21 Read operation**



## **Future Scope**

This project is an experimental demonstration for the embedded linux device driver development. It, not giving the full-fledged functionality, has some serious scope of expanding the same project. Furthermore, not just adding the functionality to GPIOs, but also the functionality of UART, I2C, SPI, USB and Wifi can be added to make fully equipped embedded linux based system of BeagleBone Black. As discussed in the introduction section, most of the technology leaded devices are working on the embedded linux based platform and thus optimizing the resource usage in order to improve the performance of the devices will shortly be an extreme need of the industry, resulting into plenty of opportunities in the field of embedded systems. Moreover, with solid research work going on in the field of high performance computer architecture, building light yet, powerful embedded software has become an area of research interest.

## Conclusion

A complete GPIO character device driver for Beagle Bone Black was designed and implemented from scratch. It was also tested in the form of black-box testing during the testing phase of the project. The results of the testing phase show that the device driver does what it is supposed to function. By and large, the objective of the final year project was met. Regarding the output functionality feature, the direction of all GPIO pins on BBB could be set to output. These GPIO pins can also be set to a high logic level or a low logic level. The input functionality was verified to be working normally. When all the GPIO pins were grounded, the logic level values read from them were all zero. Likewise, the logic level values were read high or one when all the GPIO pins were connected to a 3.3V power supply. Finally the interrupt handling feature was tested in the interrupt handling test case. One or more processes were able to share an interrupt on a GPIO pin, and an external signal could invoke an interrupt on that GPIO pin. However, there still remain several limitations. First of all, interrupt handling offered by the device driver is rather an experimental feature. When a hardware interrupt signal arrives, the kernel will simply call the interrupt handler and display a message to its logging system. Therefore, it is not practically useful when a userspace application needs interrupt from a GPIO pin. In addition to that, the time duration for contact debouncing of a switch was determined based on trial and error. A time duration of 200 milliseconds was effective to avoid the contact bouncing problem in the case of this project. Each button or switch possesses its own characteristics; the bouncing time duration of each type of switch is different. Therefore, the debouncing solution implemented in the device driver will not be effective if another type of switch is used. Secondly, GPIO character device driver is deprecated with a newer version of the Linux kernel by the `gpio-sysfs` driver. This implies that for a newer Linux kernel version a GPIO device driver could be implemented so that the driver exposes interfaces for userspace to use in the virtual filesystem mounted at `/sys` directory.

## References

1. D.Dhivakar and L.K.Indumathi. "Common Kernel development for Heterogeneous Linux Platforms".
2. International Journal of Engineering Research & Technology (IJERT) ISSN: 2278-0181.
3. Bootlin. "Linux Kernel and Driver Development". Bootlin: 2018: 1-49.
4. Greg Kroah-Hartman. "Linux Kernel in a Nutshell". Edition 1. Shroff Publishers: Delhi; 2006: 5-121.
5. Lars Wirzenius, Joanna Oja, Stephen Stafford and Alex Weeks. "The Linux System Administrator's Guide" [online]. 2004 [cited 2004 Jan 14]. Available from: URL: <http://tldp.org/LDP/sag/html/root-fs.html>.
6. "Platform Devices and Drivers" [online]. Available from: URL: <https://www.kernel.org/doc/Documentation/driver-model/platform.txt>.
7. Robert Nelson. "BeagleBone Black" [online]. 2018 [cited 2018 Oct 12]. Available from: URL: <https://www.digikey.com/eewiki/display/linuxonarm/BeagleBone+Black>.
8. Arun M Kumar. "Building for BeagleBone" [online]. 2014 [cited 2014 Jun 23]. Available from: URL: [https://elinux.org/Building\\_for\\_BeagleBone](https://elinux.org/Building_for_BeagleBone).
9. Gerald Coley. "BeagleBone Black System Reference Manual". Texas Instruments: 2013: 1-10.