EC535 Final Project Project - Beaglebone Home Security System

Abigail Skerker and Cristian Palencia

Introduction

We developed a Home Security System using a Linux Kernel Module for our final project. The home security system consisted of a Beaglebone Black, a motion sensor to detect movement, a buzzer to notify a user when the sensor is tripped, and a push button to enable and disable the system. Our final circuit diagram is shown below in Figure 1.

As a part of our project, we incorporated complexity through the use of multiple interrupts, multi-threading, a multimodal system, PWM, and GPIO control, all of which will be discussed in this report. All of these components are implemented through a Kernel Module in C. Our project design flow outlining the operation of our system is included in Appendix D.

Primary failures as part of this project included being unable to implement wireless communication using XBee, as well as not being able to leverage the PWM Linux library.

Links to our Github Repository and Project Video are in Appendices E and F respectively.

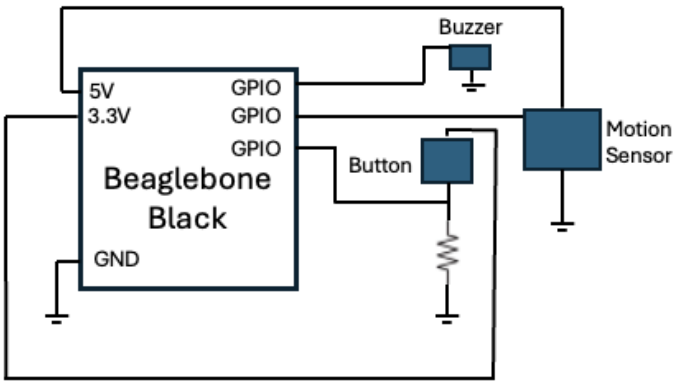


Figure 1: Final Circuit Diagram

Discussion of Successes

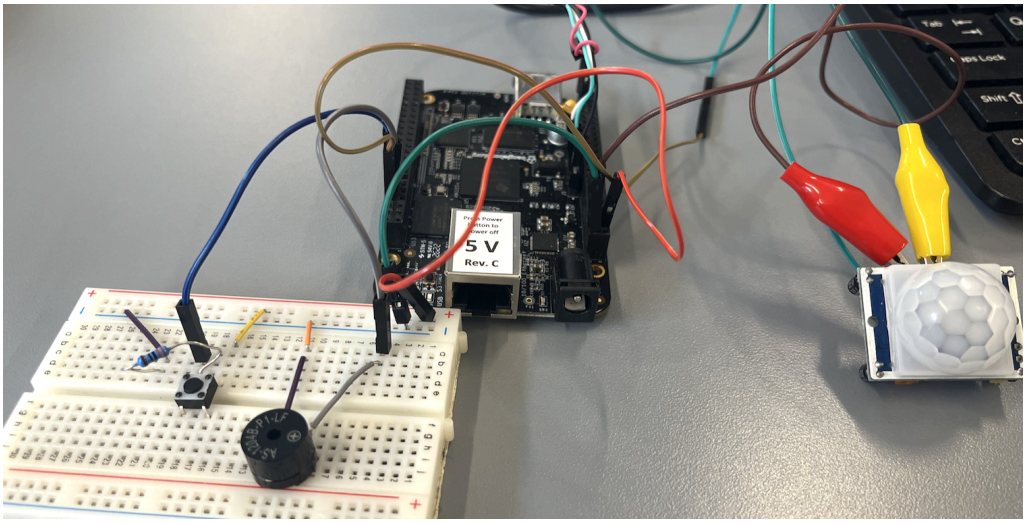


Figure 2: Final Circuit

Multimodal System

We designed our project to have an “on” and an “off” mode to toggle between utilizing a push button. When the kernel module is initially loaded onto the Beaglebone, it is on. This means that if the motion sensor is tripped, the buzzer will go off. However, the user also has the option to disable the alarm system through the press of a push button, and then can re-enable it by re-pressing the push button later. The push button was implemented through an interrupt request function, defined in more detail in the interrupt request section, and is shown in figure 3 below. Whenever the button is pressed, the mode of the system is changed to the opposite of the current mode.

```
static irqreturn_t button_handler(int irq, void *dev_id) {
    mode = !mode;

    return IRQ_HANDLED;
}
```

Figure 3: Button Interrupt Request

Multiple Interrupt Requests

We have two GPIO pins as input as part of our home security system, one for reading in motion sensor data and the other for reading in button data. The interrupt request function for reading in sensor data is shown below in figure 4. The interrupt request function for reading in button data is shown above in figure 3.

```
static irqreturn_t sensor_handler(int irq, void *dev_id) {

    // aslong as we are in "read sensor" mode
    if(mode == 0 && ACTIVE_TIMERS == 0) {
        // turn buzzer on for 3 seconds
        gpio_set_value(BUZZER,0);

        // call timer to turn off buzzer in 3 seconds
        mod_timer(etx_timer, jiffies + msecs_to_jiffies( 5000 ));
        ACTIVE_TIMERS = 1;

        // Create a kernel thread
        thread_task = kthread_run(PWMtoggle, NULL, "my_thread");
        if (IS_ERR(thread_task)) {
            printk(KERN_ERR "Failed to create thread\n");
            return PTR_ERR(thread_task);
        }

    }

    return IRQ_HANDLED;
}
```

Figure 4: Sensor Interrupt Request

For implementing the interrupt request of the motion sensor, we initially incorrectly set the response to the interrupt request to be on the falling edge of the motion sensor. We quickly realized that this was wrong, as we had a delayed response in our system, and wanted our security system to respond as quickly as possible to an incoming “invader”. We therefore modified the interrupt request to respond to the rising edge of the motion sensor output.

Read Method

The following method is in charge of formatting return_buffer and sending it back to the user via the prepareOutput() function. The prepareOutput function formats the display of three variables indicating if the Security System is On or Off, indicating if the Alarm is On or Off, and indicating if the Sensor is On or Off.

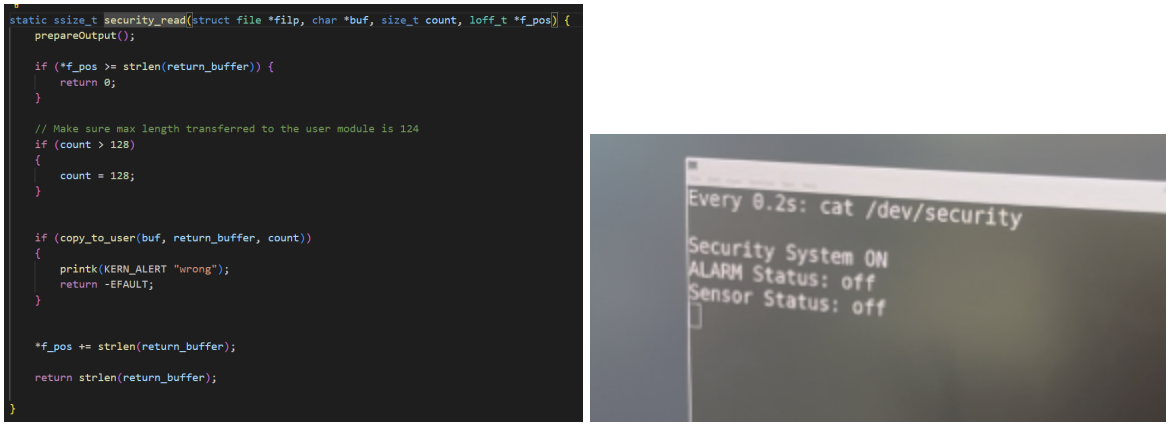


Figure 4: Read Method & Read Method Output

Final PWM Implementation

Discussed in further detail in our failure discussion, we originally struggled immensely with figuring out a PWM implementation with the beaglebone black using the pwm.h library. Thus, we instead implemented our own PWM method. To do this, we converted a GPIO pin to a custom “PWM” pin. Looking at Appendix A, we used the relevant specification sheet for the buzzer we were using to determine that the operating frequency for the buzzer was 2048 Hz, with a 50% duty cycle. We manually created a while loop that would be in charge of setting the specified GPIO pin on and off with the correct timing. This allowed us to give the buzzer a PWM wave.

Multithreading Usage

In order to implement the PWM function, we added multithreading to our implementation, something we did not initially intend to do. Without multithreading, the beaglebone would get stuck in the while loop and use all of the system’s resources within the interrupt request. To solve this problem we used the kthread.h library to run the while loop as a thread so that all our other code could still operate outside of the while loop. To implement this, we set up a 5 second timer (the desired duration for the buzzer to be active after being triggered), followed by creating a thread for the PWM process. When the timer ends, it terminates the thread, and thus terminates the PWM signal. In conjunction with the multithreading and PWM method we were able to implement our own unique PWM functionality that permitted the buzzer to ring for a determined amount of time (5 seconds) whenever an intruder was detected. This code is shown in Appendix C.

Wireless Communication with XBee

While we were not able to get XBee working as a part of our system (as discussed in the failure section), we were able to get the two XBee modules configured and communicating using API mode once we received a new module utilizing the XCTU software. In order to do this, we connected a Serial cable to each XBee module in the setup shown in Figure 10. Through the XCTU software, we modified the firmware to be under the DigiMesh protocol (XBee DigiMesh 2.4), and configured one module in API mode, and the other in AT mode, following the linked guide. Successful communication is demonstrated in Appendix B. Following the image in the Appendix, we observe that we send a message from XBee "A" (the one in API mode) over to XBee "B" (AT mode). We send a message saying "Hello XBee_B" and verify that XBee A correctly transmitted the message. Finally, we notice that the message is correctly received by XBee B.

Discussion of Failures

Wireless Communication with XBee

We originally aimed to communicate wirelessly with two XBee Zigbee modules as part of our project. Our intention was to connect one XBee module, the sender, to a sensor. This XBee module would be configured in API mode, and connected to the motion sensor. The second XBee module, the receiver, would be configured in AT mode, and connected to the beaglebone along with the main circuit. The XBee in AT mode would then, via serial communication, send over packets that would be interpreted as a high or low signal from the motion sensor. Our code would then process the information accordingly. However, one of our XBee modules stopped communicating with the configuration software (XCTU) and we were unable to get it to connect to it to configure it. We attempted to order a new XBee module but it did not arrive in time. The circuit diagram for our original implementation proposal is outlined below in Figure 5.

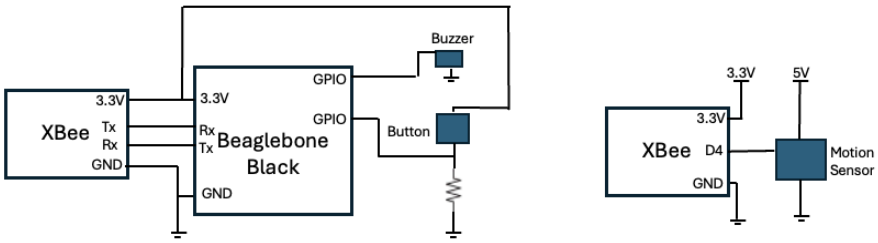


Figure 5: Wireless Circuit Diagram

We had some code additionally planned out, that involved reading the data from the wireless module and then sending the data via serial communication to the beaglebone for further processing. The beaglebone would essentially have been in charge of always listening to the serial line and respond accordingly as it received either a low or high signal from the wireless motion sensor.

Original PWM Implementation

We required a method of getting the PWM pins on the Beaglebone Black to function in order to drive the buzzer whenever the motion sensor was tripped. Initially, we tried implementing this via the built-in kernel library for PWM. Our initial attempts with the library were unsuccessful mainly for two reasons. For one, there was a lack of example's online using PWM with a kernel module on Beaglebone black. For this reason we struggled with finding out the right way to use the functions within the pwm.h library. Secondly, there were no pins on the Beaglebone that were configured as PWM pins. We attempted to remedy this both by configuring pins on the Beaglebone and by switching over to a Raspberry Pi device. However, we were missing firmware on our image to configure PWM pins on the Beaglebone. Despite some success with the Raspberry Pi we still struggled with some differences between the two development boards and decided to return to the Beaglebone and implement our own PWM method. Our successful operation was discussed previously in the successes section.

Conclusion

As part of this project, we successfully deployed a kernel module on Beaglebone Black to control a home security system with a motion sensor to detect movement, a buzzer to indicate when the sensor was tripped, and a button to turn the system on and off. While we ran into limitations with wireless components, we were able to implement extra other components including PWM and multithreading into our implementation.

References

Configure an XBee module in API mode. (n.d.). Www.digi.com. Retrieved 2024, from https://www.digi.com/resources/documentation/digidocs/90001526/tasks/t_configure_third_xbee.htm?tocpath=Set%20up%20%20your%20XBee%20devices%7CUse%20API%20mode%20to%20talk%20to%20XBee%20modules%7C_____1

How to properly configure the PWM on debian gnu/linux 10 (buster). (2022, January 31). BeagleBoard. https://forum.beagleboard.org/t/how-to-properly-configure-the-pwm-on-debian-gnu-linux-10-buster/31409/5

Low Level Serial API — The Linux Kernel documentation. (n.d.). Www.kernel.org. Retrieved 2024, from https://www.kernel.org/doc/html/v5.6/driver-api/serial/driver.html

Mata, D. (2019, October 7). *In this tutorial, I am going to show you how you can send data wirelessly between an Arduino to a…*. Medium. https://medium.com/@ing.david.mata/in-this-tutorial-i-am-going-to-show-you-how-you-can-send-data-wirelessly-between-an-arduino-to-a-cb18cac1c984

Security Monitor. (n.d.). Www.digi.com. Retrieved 2024, from https://www.digi.com/resources/examples-guides/security-monitor

Appendix

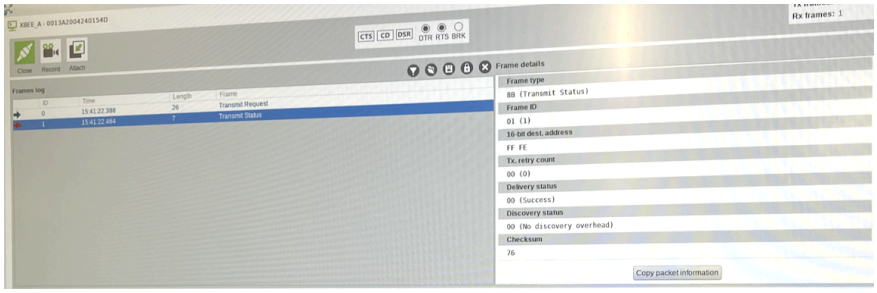
Appendix A: Buzzer Specifications - Relevant Specification from the Datasheet for the buzzer

1. SPECIFICATION **AS-1204B-LF**

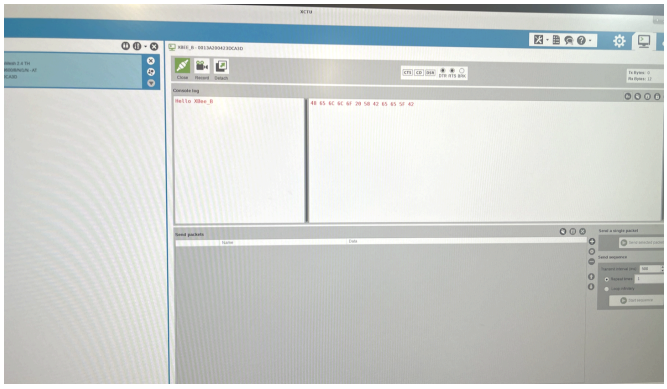
ITEM		UNITS	SPECIFICATIONS	CONDITIONS
01	Rated Voltage	V	3.5	
02	Operating Voltage	V	3 ~ 5	
03	Consumption Current	mA Max.	35	Standard state, standard drive circuit. Square wave 1/2 duty.
04	Sound pressure level (Distance at 10cm)	dB(A) Min.	85	Rated voltage, rated frequency.
05	Direct Current Resistance	Ω	42±6.3	
06	Rated Frequency	Hz	2048	

Appendix B: XBee Communication using API Mode Demonstration

Sent Message from XBee_A in API Mode:



Received message from XBee_B in AT Mode:

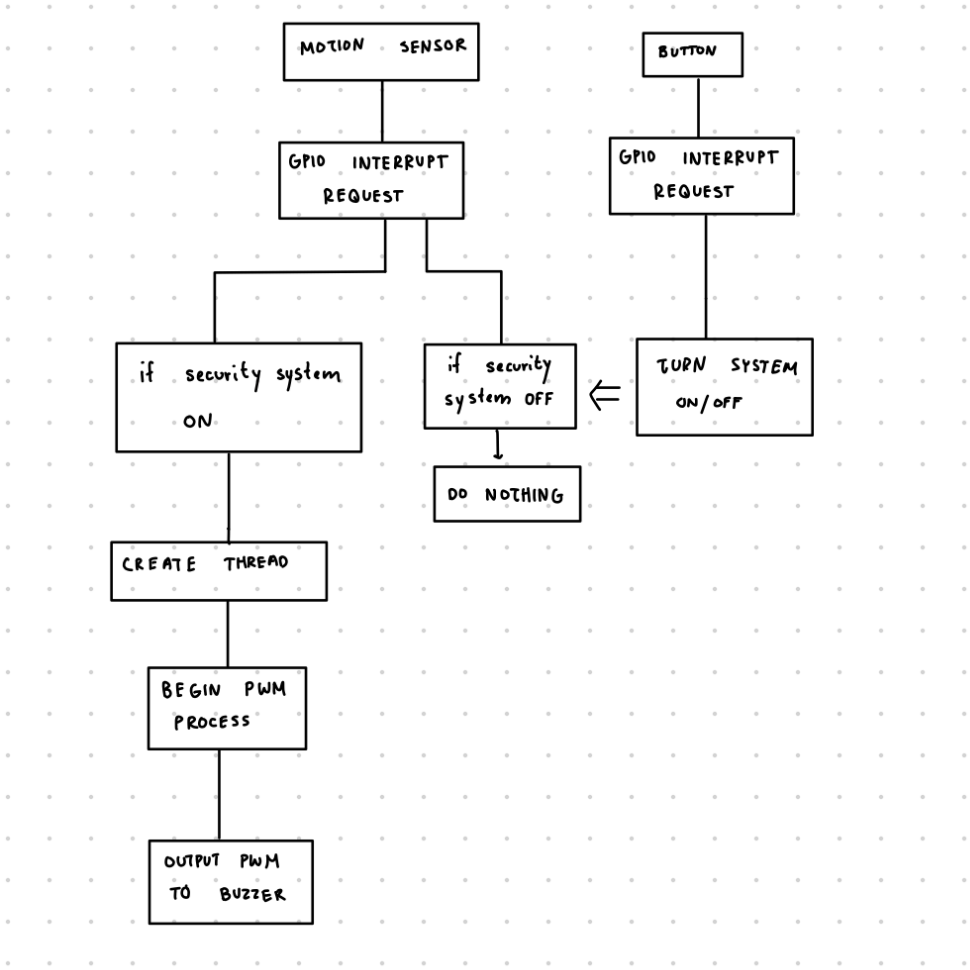


Appendix C: Multithreading and PWM code

```c
static int PWMtoggle(void *data) {
    unsigned int period_us = 1000000 / PWM_FREQUENCY; // Period in microseconds
    unsigned int half_period_us = period_us / 2; // Half period in microseconds

    while(!kthread_should_stop()) {
        // Set the GPIO pin high for the high time
        gpio_set_value(BUZZER, 1);
        udelay(half_period_us);

        // Set the GPIO pin low for the low time
        gpio_set_value(BUZZER, 0);
        udelay(half_period_us);
    }

    return 0;
}
```

```c
static irqreturn_t sensor_handler(int irq, void *dev_id) {

    // aslong as we are in "read sensor" mode
    if(mode == 0 && ACTIVE_TIMERS == 0) {
        // turn buzzer on for 3 seconds
        gpio_set_value(BUZZER,0);

        // call timer to turn off buzzer in 3 seconds
        mod_timer(etx_timer, jiffies + msecs_to_jiffies( 5000 ));
        ACTIVE_TIMERS = 1;

        // Create a kernel thread
        thread_task = kthread_run(PWMtoggle, NULL, "my_thread");
        if (IS_ERR(thread_task)) {
            printk(KERN_ERR "Failed to create thread\n");
            return PTR_ERR(thread_task);
        }

    }

    return IRQ_HANDLED;
}
```

Appendix D: Project Design Flow



Appendix E: Link to Git Repository

https://github.com/cpalencica/home-security

Appendix F: Link to Project Video

https://youtu.be/EEVpLaNmyOU