Advanced Embedded Systems Final Project: Team Delta

Jiajia Liang
University of Virginia
jl9pg@virginia.edu

Zongdi Qiu
University of Virginia
zq9ms@virginia.edu

CJ Rogers
University of Virginia
cjr2cu@virginia.edu

Author Keywords
Embedded System, micro-controller sensors, multithreading, game design, rtos

INTRODUCTION

An embedded system is a computer system—a combination of a computer processor, computer memory, and input/output peripheral devices—that has a dedicated function within a larger mechanical or electrical system [3].With the development of information technology, embedded systems have been widely used in every corner of modern society. From the electronic products we use daily to space probes flying into space, we can all see the application scenarios of embedded systems It has a strong specificity, and its function is corresponding to the application. Due to the application-centric characteristics of embedded systems, users can tailor software and hardware as needed to reduce system power consumption and cost.

In our final project, we use tm4c123GH6MP micro-controller, an embedded system with Cortex-M4 as the core. We implement a real-time cube game, using an LCD, acceleration sensor, light sensor, and joystick. In addition, we experiment on ways to prevent deadlocks, reduce jitters, and efficient multithreading. As our final product, we create a well-functioning cube game, key features includes using accelerometer to control crosshair movement, adjusting LCD brightness according to environment lightning condition, and implementing a multi-level game.

SYSTEM DESCRIPTION

Shown in Figure 1, our real-time operating system has five tasks, namely *T1: Accelerometer input*, *T2: Light input*, *T3: consumer*, *T4: Restart*, and *T5: cube creation and movement*.

- T1 contains a background thread *Producer1*, which executes at 2Hz sampling rate. Producer1 reads raw data value from accelerometer periodically, and converts the raw data to crosshair location data. The data is sent to T3 Consumer thread for actual display.
- T2 reads raw data from light sensor with a background thread *Producer2* executing at $1/25$ Hz sampling rate. The sampling frequency is 50 times lower then producer1 sampling frequency.
- T3 is a foreground thread that runs continuously from the main function. It accepts data from T1 and draws the latest objects on the LCD.
- T4 implements the restart function. It includes a periodic thread *SW2Push* that executes whenever the SW2 button on the BSPMKII is pushed. SW2Push adds a foreground thread *Restart* to the system. Restart thread kills other foreground threads currently running, sleeps for 50 ms, and then restarts all foreground threads and resets their global variables.
- T5 implements the functionality of generating cubes, and cube movement functions. Each cube is a foreground thread *CubeThread*, and random number of CubeThreads are generated.

DESIGN AND IMPLEMENTATION

Key Components in Part I

Random Number Generator

Random number generation is implemented using linear feedback shift registers [1]. This implementation of generating random number is computational light-weighted, as it only requires one right-shift operation and a XOR operation. Therefore, it is very ideal for our micro-controller. We adopt the implementation from Maxim Integrated™design documentation [1], and we use mask value $0x7A5BC2E3$ for 31 degree mask, and $0xB4BCD35C$ for 32 degree mask. The generator generates random number ranging from 0 to 65535, and we perform simple module to get a number within the desired range. The random number generator is being used for generating number of cubes, and determining cube's moving direction.

Deadlock Prevention

We implement this by setting a flag for each block, with flag value 0 represents available block that is not occupied by other cube, while flag value 1 represents block is currently occupied and unavailable. In total, we result in $6x6$ flags, and we use a 2D array to implement it in the code.

Initially, all blocks are free (*CubeFree[i][j] = 0 for all i, j*). When a cube is generated and randomly assigned a block position i,j, we mark that block as occupied. As the object is moving, we first check whether its destination block *D* is occupied or not. If block D is available, we simply move the object to that block and update the flag for previous block and block D. However, if D is marked as occupied, we randomly find another neighbor block until we find a valid position to move the cube. The algorithm is shown in Figure 2.

Each cube will be in one of the two cases.

1. At least one neighbor is available. The cubeThread can find that available block eventually, if not interrupted or preempted.
2. In worst case, none of the four neighbor block is available, the thread will stunk in the loop until being interrupted or preempted.

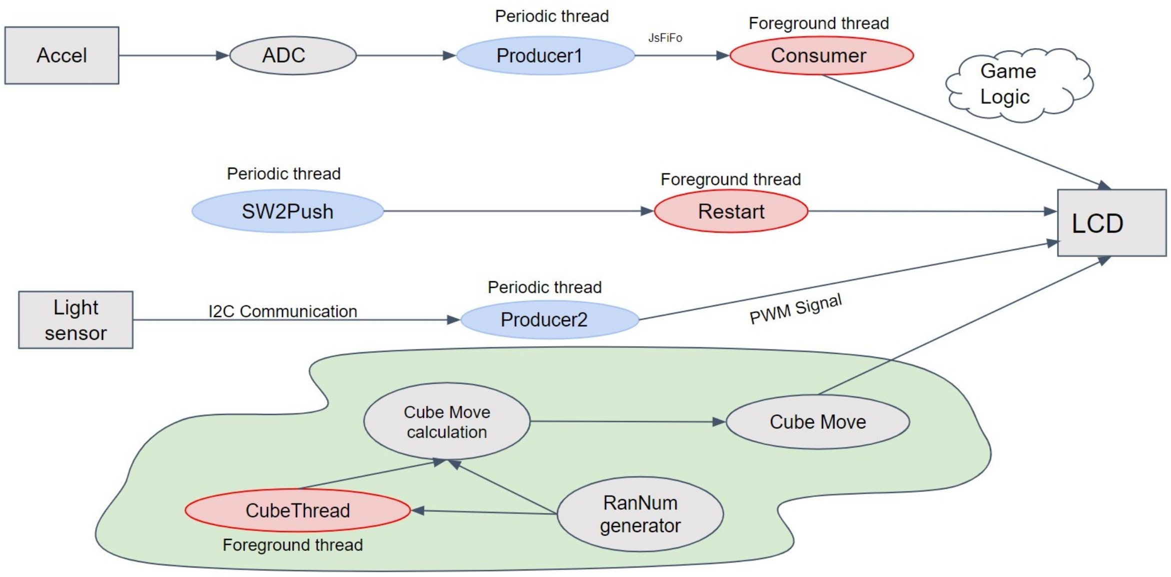

Figure 1. Data flow chart for Part 2 implementation. Background threads are shown in blue circle, and foreground threads are shown in red circle.

Since the maximum number of cube that can be generated at a given time is set to five, for case 2, at least one thread will be able to move and free its block. Therefore, no deadlock situation will occur. However, we do recognize that this is not an ideal implementation, as the thread might spend extra time to randomly explore a feasible move. This might increase data lost and jitters value for our system.

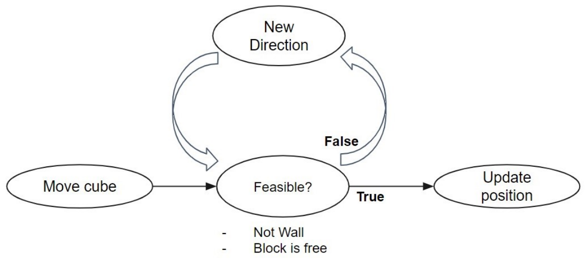

Figure 2. Flowchart for Deadlock Prevention. CubeThread randomly searches new direction until finds a feasible move.

Implementation of Accelerometer Input

Instead of using the joystick to control the crosshair in the final project, we use the accelerometer. The accelerometer uses a set of three analog signals to indicate the values for x, y, and z directions. The component uses pins PD0 for x, PD1 for y, and PD2 for z.

ADC Initialization

The main changes that come from switching the input from the joystick to the accelerometer are in the initialization of the ADC. First, the analog function for pins PD0, PD1, and PD2 must be enabled in register GPIO_PORTD_AMSEL_R. Then they must all be set to inputs in register GPIO_PORTD_DIR_R. In order for the MCU to know that it should look at the ADC for the input the alternate function must be enabled on all three pins in register GPIO_PORTD_AFSEL_R, as well as enabling the analog functionality on the pins in register GPIO_PORTD_DEN_R. The adc_init function from the joystick.c file stays the same since it was designed to be a generalized initialization function.

```
ADC0_ACTSS_R &= ~0x0004;      // 10)
ADC0_EMUX_R &= ~0x0F00;       // 11)
ADC0_SSMUX2_R = 0x0567;       // 12)
ADC0_SSCTL2_R = 0x0060;       // 13)
ADC0_IM_R &= ~0x0004;         // 14)
ADC0_ACTSS_R |= 0x0004;       // 15)
```

In this code snippet, the last few lines of the ADC initialization are shown. This contains some of the main differences between the joystick code and the accelerometer code. Instead of using Sample Sequencer 1, the accelerometer uses Sample Sequencer two, indicated by the abbreviation SS2. Line 10 serves to disable the sequencer while it is being set up. Line 11 will set the SS2 software trigger. Line 12 tells the ADC that the data from A5 (z), A6 (y), and A7 (x) are going to be the inputs to the sequencer. It sets the sequence for retrieving information as x, then y, then z. Line 13 is the settings for the ADC, which was not changed from the joystick except for changing SS1 to SS2. The last two lines then disable interrupts for SS2 and finally enable SS2.

The change in line 12 means that the `BSP_Accel_Input` function will look slightly different than its joystick counterpart. It will instead initiate SS2 and acknowledge the completion of SS2. Initially it was also going to retrieve three results from the SS2 FIFO but since our project only uses the x and y information, the z data does not need to be retrieved from the FIFO.

Scaling Movement
To actually make the change from the joystick to the accelerometer as the control mechanism the only other thing that had to be changed was the producer function. This needed to be updated to use the new `BSP_Accel_Input` function. Since the accelerometer has a wider range of motion the scaling factor for the control mechanism was altered slightly as well. Instead of shifting the raw data right by 9, the raw accelerometer data is instead shifted right by 4. This resultes in a more intuitive control scheme that was both manageable, while still making aspects of the game challenging.

Implementation of LCD Brightness Adjustment

Light Sensor
We use the light sensor to detect the light intensity of the environment and adjust the LCD screen brightness in real time. As shown in Figure 3, there is a light sensor (OPT3001) on the top of the BoosterPack we use. OPT3001 is an ambient light sensor manufactured by TI. Its measuring range is 0.01lux to 83klux. The spectral response of the sensor matches the verse response of the human eye. We don't need to use ADC to process the output signal when using this light sensor. The digital signal is output to the serial port through I^2C communication method.

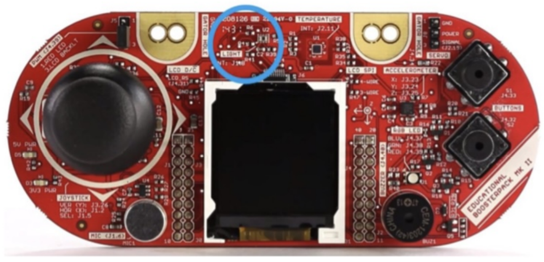

Figure 3. Light sensor on the BoosterPack

Implementation of I^2C
I^2C is a bus developed by Philips Semiconductors to connect integrated circuits (ICs). The Standard I2C bus may operate as a multi-master bus: Multiple ICS may be connected to the I^2C bus and each one may act as a master by initiating a data transfer[4]. To maximize hardware efficiency and simplify circuit design, a simple bi-directional 2-wire, serial data (SDA) and serial clock (SCL) bus for inter-IC control (IC) was developed[2].

In the BoosterPack we use, port J1.9 and port J 1.8 are connected to the SCL and SDA of the light sensor. Because J1.8 and J1.9 are connected with PA6 and PA7. We need to initialize PortA as the port that accepts I^2C signals.

```
IC2_Init();
GPIO_PORTA_PCTL_R =
(GPIO_PORTA_PCTL_R & 0xFF0FFFFF);
GPIO_PORTA_DIR_R &= ~0x20;
GPIO_PORTA_AFSEL_R &= ~0x20;
GPIO_PORTA_DEN_R |= 0x20;
```

Above is the code snippet that initializes the light sensor. First, we need to initialize the I2C Port. In this process, except for some regular IO initialization operations, we need to activate clock for Port A, set the I^2C1 register to master and set the clock frequency output by the master. There can be multiple slaves on an I^2C bus but only one master. Each slave on the I^2C bus needs a clock signal provided by the master.

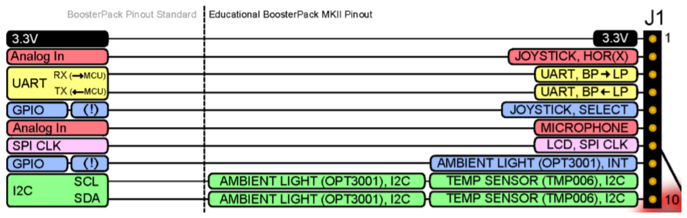
BoosterPack Pinout Standard
Educational BoosterPack MKII Pinout
J1
3.3V
Analog In
UART
RX (→MCU)
TX (←MCU)
GPIO
(!)
Analog In
SPI CLK
GPIO
(!)
I2C
SCL
SDA
AMBIENT LIGHT (OPT3001), I2C
AMBIENT LIGHT (OPT3001), I2C
3.3V
JOYSTICK, HOR(X)
UART, BP→LP
UART, BP←LP
JOYSTICK, SELECT
MICROPHONE
LCD, SPI CLK
AMBIENT LIGHT (OPT3001), INT
TEMP SENSOR (TMP006), I2C
TEMP SENSOR (TMP006), I2C
1
10

Figure 4. J1 Pin Descriptions on the BoosterPack.

After the initialization of Port A, we can receive data from the light sensor. There is a slave address in front of each data signal, so the master can judge the source of the signal by the slave address in the front part of the signal. The light sensor generates a digital signal, so we no longer need an ADC to process the generated signal. TM4c123 can directly obtain light intensity value.

Implementation of PWM
Under normal circumstances, if we want to adjust the brightness of the LED or LCD, we will not do so by adjusting the voltage. Because unstable voltage is likely to damage electronic devices. Using PWM (Pulse Width Modulation) is a more feasible solution. PWM is a digital signal generated by the processor, so it has a stronger anti-interference ability than an analog signal.

On our board, ports PF3, PB3, and PC4 can be used to generate PWM signals. Because these three ports support PWM mode, we only need to initialize these ports. After setting them to PWM mode and setting cycle, we can get the PWM signal from these ports. Combined with the previous section, we can use the light intensity obtained by the light sensor to adjust the cycle of output PWM signal. Due to the characteristics of PWM, we don't need to worry about extreme light intensity causing damage to the circuit.

Multi-level game design

In order to add more difficulty to the game, a Multi-Level system was designed. This system features a sequence of 5 different levels of increasing difficulty. The first level starts out as normal with a random number of cubes being created and having the user move the crosshair around the screen in order to capture them. Once they get a score of 10, they move to the second level. This second level creates a wall at the bottom of the screen that the player cannot touch or else they will lose a life. Level 3 has two walls, Level 4 has 3 walls, and Level 5 has a wall on each side of the screen. The progression of levels can be seen in Figure 5.
The calculation for determining whether the crosshair has intersected with any of the walls occurs in the `UpdatePosition`

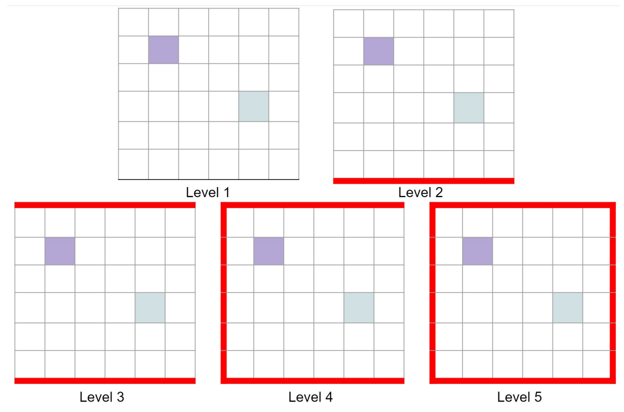

Figure 5. Level Progression

function. Based on the current level it will determine whether the player has hit one of the walls and will remove a life when they hit it. There is a cooldown timer to allow the player to adjust, otherwise the player would lose all of their lives before they even knew why.

RESULTS AND DISCUSSION

Light Sensor

To show light sensor is correctly implemented, we show the results for different environment lighting conditions. In the first condition, we let the light sensor detect the intensity of the ambient light to obtain a corresponding LCD screen brightness according to the light intensity. In the second test, we covered the light sensor with our thumbs, so that the light sensor gets very weak light. The result of test is shown in Figure 6.

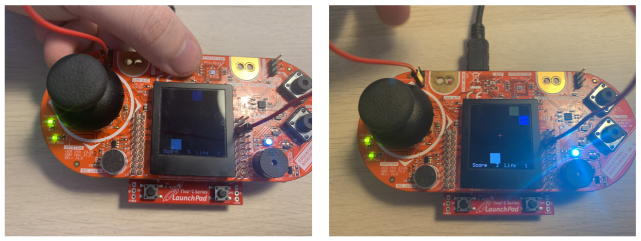

Figure 6. Use light sensor to adjust LCD screen brightness

In our system, the light sensor is added to the system in the form of a periodic thread. During our development, When light sensor is added to the system, this new periodic thread will bring large jitters to our crosshair movement. Every time the light sensor transmits data, the crosshair will be stuck. To solve this problem, we made two changes. Firstly, we reduced the frequency of light sensor data transmission. Setting a fixed data transmission frequency light sensor will no longer interfere with the operation of accelerometer or joystick. Secondly, we tried a lot of modifications to I^2C communication. Finally found that the system waits a long time for the conversion of the initialization Port of the light sensor to complete. After trying to remove the conversion waiting time, the system freeze problem is significantly improved.

Accelerometer

We experimented with different scaling values to convert the raw data to actual crosshair position. Raw acceelerometer data is scaled by right-shifting the raw data by a scaling value. We found a scaling value greater than 5 results in an overly-sensitive tilted experience (i.e a small tilted angle results in a large and fast crosshair movement). Oppositely, a scaling value lower than 2 results in a very insensitive accelerometer (i.e to move the crosshair to our destination, we need to tilt the broad by a large degree, otherwise, crosshair is moving slow). As a result, we choose to right shift the raw data by 4 to create a manageable yet challenging game.

Game Levels

As the player gets to a higher level, the difficulty of the game increases as well. At level 5, players can only move the crosshair in the middle region, restricted from touching the borders. Since our accelerometer input is being scaled, the mapping between the tilted angle and crosshair movement is very intuitive and easy to learn. However, we recognize that there may be a learning process for players to master the control by tilting the board. In addition, individuals might have difference sensitivity and mapping between the tilted angle and object movement. This can be addressed in the future work.

LESSONS LEARNED

The main lessons learned from this project are as follows:

1. How to implement a fast and efficient random number generator using an LSFR.
2. How to develop a Multi-threaded RTOS that avoids deadlocks.
3. How to use randomness as a solution to collisions.
4. The effects of a non-optimal implementation on Jitter and data loss.
5. How to use I^2C to interface with a Light Sensor.
6. Using a PWM signal to adjust the brightness of a LED and LCD.
7. Similarity between software implementation of Joystick vs. Accelerometer. How to read values from Joystick and Acceleromer, and how to map raw data into crosshair position data.
8. Cooldowns are important for punishment mechanics in video games.

There were also many general skills that the team was able to gain during the course of this project that are related to

1. Coordinating teams that work in different time zones
2. Source Control Management (Working with Git, handling Merge conflicts and Branch Management)
3. Task management and coordination

TEAM RESPONSIBILITY

Although we are working remotely and have different time zones, we managed to communicate and collaborate well. Tasks are divided evenly and all members are fully engaged and have contributed to the final product.

Jiajia Liang

- Adding semaphores to avoid deadlock
- Add lives and score
- Integrate accelerometer input into game
- Add Game Levels

Zongdi Qiu

- Drawing cubes on LCD
- Cube movement
- Light Sensor integration

CJ Rogers

- Creation of cube threads
- Modify cube movement for individual cube threads
- Method for detecting collisions
- ADC configuration for accelerometer

CONCLUSION

We successfully designed and implemented a fun cube game, which the player can interact with by tilting the board and changing environment brightness. Throughout the project, all members fully engaged in group discussion, game design, and actual implementation. Overall, we put a lot of effort into making this game well-functioning, and we gained more in-depth knowledge about thread scheduling and embedded systems.

REFERENCES

[1] 2010. Pseudo Random Number Generation Using Linear Feedback Shift Registers. (2010). https://www.maximintegrated.com/cn/app-notes/index.mvp/id/4400

[2] Anders Andersson and Daniel Van Berkom. 2003. Multi-chip addressing for the I2C bus. (Sept. 30 2003). US Patent 6,629,172.

[3] Michael Barr. 2007. Embedded systems glossary. *Neutrino Technical Library* (2007).

[4] Ryan Fukuhara, Leonard Day, Huy H Luong, Robert Rasmussen, and Savio N Chau. 2004. I2C bus protocol controller with fault tolerance. (April 27 2004). US Patent 6,728,908.