# EC535 Final Presentation: Meteor Dash

Samuel Gossett, Liam Hayes

https://github.com/LiamCHayes/EC535-project-2

## 1 Introduction

Our game, Meteor Dash, is an IMU-driven game that uses low-latency I$^2$C sensing, scheduling, and a custom kernel module to test your steadiness and reaction speed. By mapping IMU tilt directly to movement, the game adds a built-in challenge of hand stability. This game is reinforced by safety checks and kernel-level logic that keeps the system secure and uncheatable.

Our motivation came from two places. First, we wanted to elevate the game-play with motion controls to help create a more dynamic gaming experience. IMU-based inputs can create creative mechanics, like translating sharp adjustments into high-speed movement. It also allows sensitivity tuning for difficulty-scaling across levels. Second, this game grew out of our original plan to build a Reaction Wheel Inverted Pendulum. We had already developed IMU and magnetic-encoder drivers for closed-loop balance control, but a broken motor on delivery and long replacement time forced us to abandon this idea. To save time, we wanted to re-purpose our work. This pivot allowed us to continue exploring and learning about sensor integration and polling.

While not society-changing, the project raises interesting ideas. Motion controls reduce reliance on keyboard inputs, which can contribute to joint strain and carpal-tunnel risks. Motion controls also encourages and trains steadier hands since as difficulty increases, more stability is needed. For our game, our user space program and kernel module were on the BeagleBone Black which had two main hardware components that were used in the game: an IMU and an LCD screen. We calculated character control and position by creating a custom IMU library then polling the IMU data. By using a probability function for meteor spawn chance and sampling a uniform distribution within the LCD screen bounds for the spawn location, we created the obstacles of our game. These were both polled at 50ms. By using the kernel space for collision checking and frame buffer writing, we created the visualization and physics to our game. We added additional features, like difficulty scaling, scoring, high-scores, and additional visualizations as well.

## 2 Design Flow

Overall, we have a user space application running on the BeagleBone that reads inputs from the IMU and sends the control input to the kernel module. This user space application also computes the meteor spawn locations, difficulty levels, and keeps track of the leaderboard. It interfaces with the game logic kernel module through a device file. The kernel module runs the game logic and draws to the framebuffer, as well as alerting the user space application of a collision through an error code returned from the write function.

As far as hardware, we have an LCD screen and an IMU. The LCD is connected to the BeagleBone through the ribbon cables and the display cape provided. This LCD is displaying the game that the user interacts with in an 8-bit style design using the framebuffer, written to from the kernel module.

Our IMU is a 9-DOF SparkFun IMU that interfaces through $I^2C$. We wrote a separate library to easily interface with our IMU via $I^2C$. This library has functions to initialize the device and read from the device. Initialization involves setting the register bank, doing a soft reset of the internal parameters, waking up the chip and setting the PLL clock, and enabling the accelerometer and gyroscope. For the read function, we defined a struct datatype that contains the values available from the IMU to create an easy interface with the IMU registers. The read function simply reads from the registers, fills out the struct fields, and returns the struct.

Figure 1 shows the information flow through our system, from the IMU to the LCD screen and back. The user space program is running the loop, while the kernel module is just responding to the user space requests. Meteor position updates are on a separate timer in the kernel space.

Figure 2 shows the hardware setup, as well as which tasks are implemented in the user and kernel space. As mentioned before, the user space is running the game loop, and the kernel module is responding to the user space commands by drawing on the screen, spawning and updating meteors,

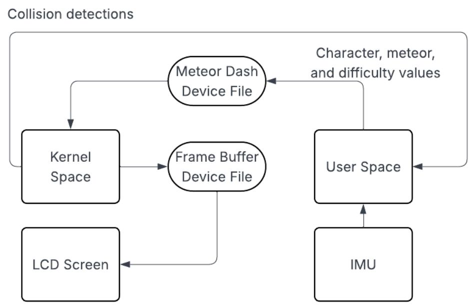

Figure 1: Dataflow of our system

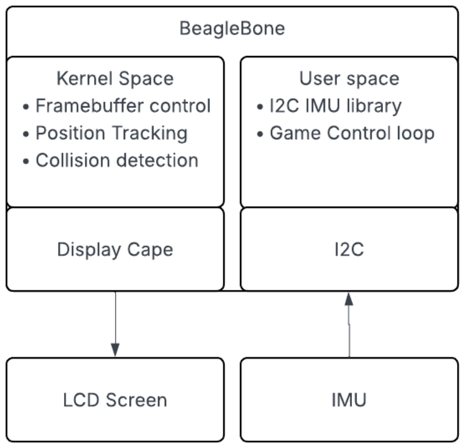

Figure 2: Hardware setup and kernel/user space tasks

checking for collisions, and changing difficulty parameters.

Liam: 50%

- Kernel space functions

Sam: 50%

- User space functions

# 3 Project Details

## 3.1 IMU Library

The IMU library interfaces with the IMU directly with $I^2C$. It consists of two functions, an initializer and a read. The initializer is important to configure the IMU for our needs. First, we set the slave address and current bank. This gives us access to the registers that are important to us. We also soft reset the entire device to clear any stored data and ensure a clean slate when we start reading from the IMU. We found that after these 3 steps, the code must sleep for a small amount of time to allow the settings to take effect. If not, this could lead to undefined behavior. The other step of initialization is to turn on the components that we need. If we had a power constrained device, this is where we would be picky with what we set. However, since our application is plugged into the wall, we can afford to turn everything on. This step powers on the device, sets the clock, and turns on both the accelerometer and gyroscope.

Now that the IMU is initialized, we can read from the buffers that contain the sensor values. The read function does this all in one read, since these buffers occupy a continuous set of memory. The IMU library also defines a struct datatype that allows for clean and readable code when in the game loop. This struct has fields for x, y, and z values for both the accelerometer and gyroscope. This is what gets returned from the read function.

## 3.2 User Space Program

The user space program manages the IMU data processing, character tracking, meteor generation, difficulty progression, and score tracking, all within a 50 ms polling loop.

For the IMU data and character tracking, before entering the loop, we initialize our custom IMU library. Each cycle, we read using the IMU library's read function to obtain the latest gyroscope and accelerometer data (we only use gyroscope-x data). The character's absolute position is computed by converting the the x-axis tilt angle (rads), scaling it by a constant, then adding it to the previous character position. This updated position is eventually written to the device file via write() for the kernel module to use.

For meteor spawning, a probability function determines whether a meteor actually spawns. If one spawns, it polls samples a uniform distribution within the LCD screen's bounds for the spawn location. The spawn probability function was refined using play-testing. If no meteor spawns, we write -1; otherwise the x location is written in the same write() as the character position to the device file for kernel-side tracking.

For the difficulty progression and score tracking, the difficulty scales with survival time and current score. A global difficulty scalar increases the player sensitivity, the meteor fall-rate, and meteor spawn probability. The player gains points equal to the current difficulty level each time step. The user space program also maintains a leaderboard stored in a text file. It notifies the player if the high-score is reached. It reads, compares the current score and the high-score, and, if greater, overwrites the high-score file.

For the communication to the kernel space and game termination, every 50ms, the program writes both the meteor spawn location (or-1 if no spawn) and the character position. If the kernel module detects an overlap between the character and a meteor, it will return a termination signal of -2. If received, the user-space calculates the final score, compares it to the high-score, if greater, sets it as the new high-score, then prompts the player to restart or exit.

## 3.3 Kernel Module

The kernel module handles drawing the graphics and managing the life cycle of the meteors and characters. It uses a character device file that implements the init, exit, open, release, and write functions. There is also a timer that handles the meteor updates.

In the init and exit functions, this is where we create and free the memories for global variables that need memory allocated for them. This includes three structs for drawing rectangles (one for the actual drawing, one for temporary meteor positions, and one for temporary character positions) and a timer struct. These functions also initialize a sleeping mutex, framebuffer info for drawing to the framebuffer, and the device file. To avoid bugs, we also check if the list of meteors is empty and if it is not, then we free that memory as well. On the open, we start the game. This involves starting the timer and drawing the character. The release stops the timer and frees the character memory.

The timer started in the open function controls the meteor updates. The timer goes off every 100ms. In the handler, it loops through the array of active meteors, redraws each meteor a few pixels down, and checks if the meteor has gone off screen yet. If the meteor has gone off screen, it gets deleted from the list and the rest of the meteors get moved down in the list. This allows for the indexing of the meteors to be based on the meteor count.

Finally, The write function is the main driver of the program. It gets sent two integers representing the character location and the meteor spawn location every 50ms. The two integer interface is very simple. Both the character location and the meteor spawn location are based on only the x coordinate; the character is always drawn at the bottom of the screen and the meteors are always spawned at the top of the screen. When these get drawn, the y values are statically read from a global variable. Since we are getting sent data every 50ms, the character gets redrawn every 50 ms independently from the 100ms update rate of the meteor redrawing. However, we don't want to spawn a meteor every 50ms, which would make the game impossible. Instead, if the second integer representing the meteor spawn location is -1, then no meteor is spawned and only the character gets redrawn. Another task in the write function is to check for collisions. To do this, we loop through the list of meteors and check if the location of the character is anywhere inside the location of the meteor. This is happening directly after the character redraw on every call to the write function.

This interface can also be used to increase the difficulty based on commands from the user space. If the first integer is -1, this means increase the difficulty level. The second integer now represents the falling rate for the meteors, which gets set to a global variable that the timer handler uses to redraw the meteor. This will also reset the meteor color (in a separate global variable used by the timer handler) to a random color to indicate a level change.

Mutexes are needed to ensure there are no race conditions on both the frame buffer and the global variables keeping track of the meteors, as both of these are accessed by the timer handler and the write functions every 100ms and 50 ms, respectively. Originally, we protected these data with two mutexes, one for the frame buffer and the other for the meteor data.

Unfortunately this was causing deadlocks and we had to change our strategy. We decided to protect both the frame buffer and the global variables with a single mutex. This was a simple solution that can impact performance, but after trying it we saw that it was not noticeable for our application.

### 3.4 Implementation Details

Everything was implemented in C with only standard libraries and cross-compiled on the lab computers. We used the stock image provided for lab 5 with no modifications other than disabling HDMI and enabling the frame buffer device tree overlays to get the LCD to work.

## 4 Summary

Overall, we implemented a game that communicates between kernel and user space using control inputs from an IMU over $I^2C$. We learned about sensor integration, polling, kernel and user space interactions, and framebuffer interactions. The largest hurdles were understanding the IMU datasheet and debugging mutexes. We learned how to define interfaces for communication and smooth operation between different applications running on the embedded system. Remaining challenges include using other degrees of freedom of the IMU to expand the game, adding power-ups, and adding better graphics through the framebuffer.