“SPACE SHOOTING” GAME ON AK EMBEDDED BASE KIT REPORT

1. Introduction

“Space Shooting” is an AK Embedded Base Kit using state machine and event-driven programming to operate. The report will document the game play, active objects, and how each active object function.

2. Hardware

AK Embedded Base Kit integrated with 1.3” LCD display screen, 1 Buzzer speaker, 1 power and data transfer USB-C port, and 4 switch buttons.

The Kit also integrated with RS485, NRF24L01+, and Flash up to 32MB.

3. How “Space Shooting” is played

The player objective is to navigate a spaceship through an endless void space with asteroids, enemy space ship and occasionally the player have to avoid terrain as their space ship is fly near a planet. The basic input for the space ship is Up and Down button to control the Up and Down flight path of the ship. The Mode button is to fire a missile from the ship. The score is calculated for the number of asteroids destroyed, enemy ships destroyed and the terrain avoided.

The “Space Shooting” start with 3 options: “START”, “HIGH SCORE”, and “EXIT”. “START” option is to start the main game sequence. “HIGH SCORE” is to view top 3 highest score achieved, but the score is automatically erased when the game is reset. “EXIT” option is to exit the game, display a screen saver. The screen saver is also display after a period of time when there are no inputs.

a. Game active objects

Active Object	Name	Description
Player Ship	`myShip`	Main object, move with inputs, fire a missile one at a time
Enemy Ship	`myEnemyShip`	Automated ship actions, from move to fire
Asteroids	`myAsteroid`	Fly toward player ship, able to destroy player ship
Terrain	`v_terrain`	Auto-generated to resemble mountains and valleys
Explosion	`myExplosion`	Animation when a missile collided with an active object
Missiles	`myMissile` `myEnemyMissile`	Player missile fly with a fixed speed Enemy missile depends on player ship fly speed

b. Inputs

Player ship can be control with “UP” button to move up. “DOWN” button to move down, and “MODE” button to fire a missile. There can be only one player missile in flight at a time, therefore, pressing “MODE” button when a missile is already in flight has no effect.

In the terrain stage, player ship is automated descend with 1 pixel per tick, stop at the bottom of the screen.

c. Points calculation

Each time an asteroid is destroyed, it will send 10 points via a message to player’s ship. Destroy an enemy ship will gain 100 points. Each node of the terrain the player ship passed through gain 5 points.

Every time player gained 200 points, player's ship will increase it fly speed by one, making the game harder, and the fly speed maxed out at 8.

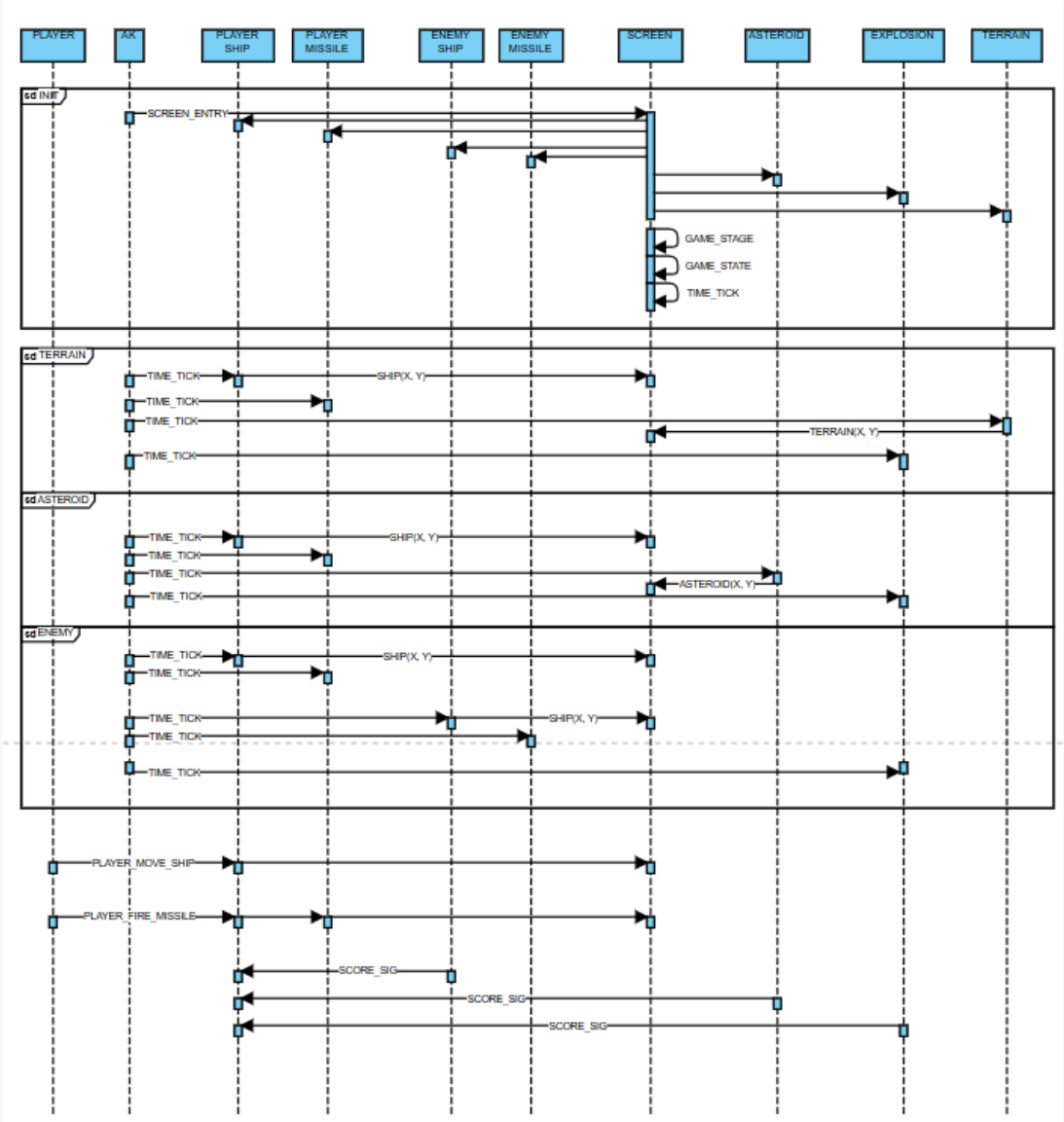
The game is over when player's ship is collided with an asteroid, enemy's missile, or a terrain node. All active objects will be reset and points accumulated will be store in high-score board.

4. "Space Shooting" game design

Event-driven architecture in software is driven by external events, such as user actions or system triggers. Each event comprises an event header, specifying the destination, and an event body, detailing the required actions. Task handlers in this architecture receive messages, which can be either pure signals or signals with data, to perform tasks. The "Space Shooting" system exemplifies this setup, incorporating a timer service that dispatches signals periodically (`'PERIODIC'`) or just once (`'ONE_SHOT'`) to task handlers.

A state machine is an algorithm that operates in one of several defined states. It consists of state variables, which represent its current state, and commands, which change its state. A state is a condition resulting from a specific relationship between inputs and outputs, while a command is an input that transitions the state machine from one state to another.

a. UML sequence



TIME_TICK is the most importance event in game. This event generated by the timer from the system timer at the rate of 10 times per second, or 100ms per generation, which is needed to smooth the animation of the display. With every TIME_TICK, the screen call for all active objects in the game.

The sequence diagram for the "Space Shooting" game illustrates the interactions among key components such as the player ship, missiles, enemy ship, screen, asteroid, explosion, and terrain. The game begins with an initialization phase (INIT), where SCREEN_ENTRY sets up the game parameters (GAME_STAGE, GAME_STATE, TIME_TICK) and initialize all active objects. These parameters are essential for managing the game's progress and ensuring that all subsequent actions are synchronized.

Following initialization, various game elements are continuously updated by the TIME_TICK signal, which acts as the game's synchronization mechanism. Terrain (TERRAIN), asteroids (ASTEROID), and enemy ships (ENEMY) all have their positions updated regularly to reflect dynamic changes in the game environment. Player actions such as moving the ship (PLAYER_MOVE_SHIP) and firing missiles (PLAYER_FIRE_MISSILE) are processed in real-time, updating the respective positions and interactions with other game objects.

Scoring mechanisms are highlighted through SCORE_SIG signals, which are generated when specific events occur, like hitting an enemy or asteroid. The overall flow of events ensures a coordinated gameplay experience, with synchronization achieved through TIME_TICK signals. This approach maintains the game's dynamic nature and responsiveness, ensuring that state transitions and actions are processed smoothly to create an engaging and challenging game.

In event-driven programming***, a task handler*** is an independent unit assigned a set of tasks. When the scheduler receives a signal to perform a task from a waiting list, it calls the relevant task handler to process the message. Task handlers can be linked to specific events and execute one or multiple tasks when those events occur. They provide synchronization by ensuring that related functions are processed sequentially, with other tasks waiting until the current task is completed. This orderly processing prevents conflicts. The listing 1 list all the task handler in the application.

Task ID	Task Level	Handler
SST_ASTEROID_TASK_ID	TASK_PRI_LEVEL_4	sst_asteroid_handler
SST_EXPLOSION_TASK_ID	TASK_PRI_LEVEL_4	sst_explosion_handler
SST_TERRAIN_TASK_ID	TASK_PRI_LEVEL_4	sst_terrain_handler
SST_PLAYER_SHIP_TASK_ID	TASK_PRI_LEVEL_4	sst_player_ship_handler
SST_PLAYER_MISSILE_TASK_ID	TASK_PRI_LEVEL_4	sst_player_missile_handler
SST_ENEMY_SHIP_TASK_ID	TASK_PRI_LEVEL_4	sst_enemy_ship_handler
SST_ENEMY_MISSILE_TASK_ID	TASK_PRI_LEVEL_4	sst_enemy_missile_handler
SST_GAMEPLAY_TASK_ID	TASK_PRI_LEVEL_4	sst_game_play_handler

Listing 1 Task handler

The task level enables prioritizing the processing of task messages in the system's queue. In the game, the task level of the game is 4, so all task will be processed in First-in-First-out order, whichever task come first will be handled first.

The listing 2 list all the signals that are linked to a specific task handler.

Object	Task ID	Signal
Asteroid	SST_ASTEROID_TASK_ID	SST_ASTEROID_INIT_SIG, SST_ASTEROID_SPAWN_SIG, SST_ASTEROID_FLIGHT_SIG, SST_ASTEROID_RESET_SIG
Explosion	SST_EXPLOSION_TASK_ID	SST_EXPLOSION_INIT_SIG, SST_EXPLPOSION_EXPLODE_SIG, SST_EXPLOSION_RESET_SIG

Terrain	SST_TERRAIN_TASK_ID	SST_TERRAIN_INIT_SIG, SST_TERRAIN_GENERATE_SIG, SST_TERRAIN_UPDATE_SIG, SST_TERRAIN_RESET_SIG
Player Ship	SST_PLAYER_SHIP_TASK_ID	SST_SHIP_INIT_SIG, SST_SHIP_FLIGHT_SIG, SST_SHIP_FIRE_SIG, SST_SHIP_MOVE_UP_SIG, SST_SHIP_MOVE_DOWN_SIG, SST_SHIP_RESET_SIG, SST_SCORE_UPDATE_SIG
Player Missile	SST_ENEMY_SHIP_TASK_ID	SST_MISSILE_INIT_SIG, SST_MISSILE_FIRE_SIG, SST_MISSILE_FLIGHT_SIG, SST_MISSILE_RESET_SIG
Enemy Ship	SST_ENEMY_SHIP_TASK_ID	SST_ENEMY_SHIP_INIT_SIG, SST_ENEMY_SHIP_TAKEOFF_SIG, SST_ENEMY_SHIP_MOVE_SIG, SST_ENEMY_SHIP_FLIGHT_SIG, SST_ENEMY_SHIP_FIRE_SIG, SST_ENEMY_SHIP_RESET_SIG,
Enemy Missile	SST_ENEMY_MISSILE_TASK_ID	SST_ENEMY_MISSILE_INIT_SIG, SST_ENEMY_MISSILE_FIRE_SIG, SST_ENEMY_MISSILE_FLIGHT_SIG, SST_ENEMY_MISSILE_RESET_SIG
Main Screen	SST_GAMEPLAY_TASK_ID	GAMEPLAY_TIME_TICK, GAME_EXIT

Listing 2 Signal of active objects

b. Asteroid

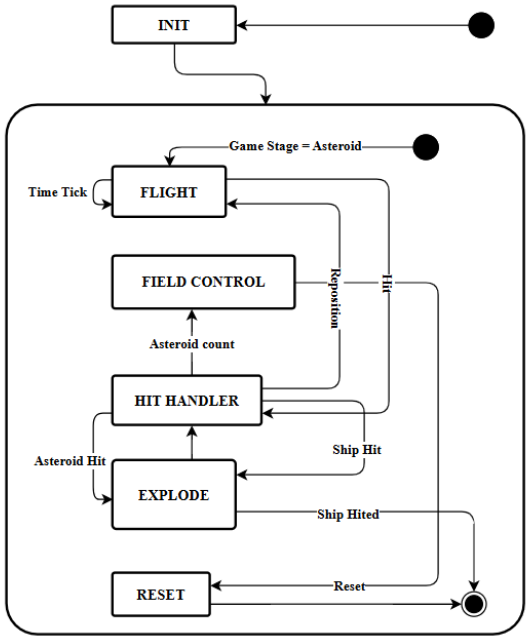


Figure 1 Asteroid state machine

Upon enter the initialization state, the Asteroid state machine transit to “FLIGHT” state. Its transit to itself with each “Time tick”. The Asteroid transit to “HIT HANDLER” state when a hit occurred. If it is a ship hit, transit to “Explode” and then to end state, else the state machine repositions an asteroid and transit back to “FLIGHT” state. If the Asteroid count traverse a threshold, it transits to “FIELD CONTROL” and reset the asteroid object.

c. *Terrain*

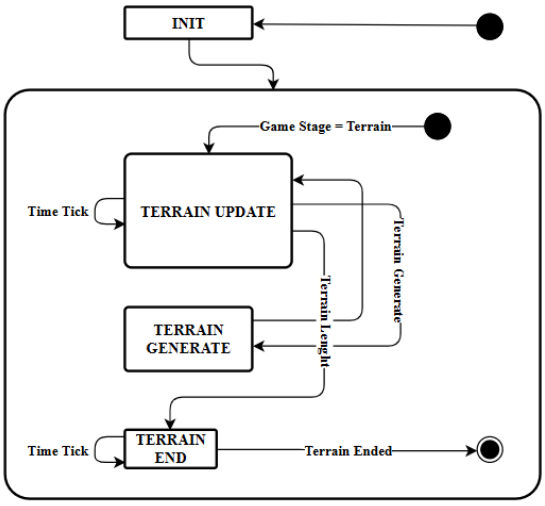


Figure 2 Terrain state machine

Similar to asteroid, after the initialization state, the Terrain state machine remain in Terrain Update state. The Terrain generate transit the state machine to Terrain generate state. When the terrain crosses a terrain max length, it transits in to terrain end state and then to end state.

d. *Player ship and enemy ship*

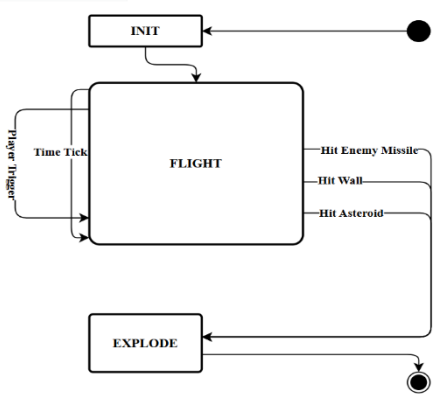


Figure 3.1 Player State Machine

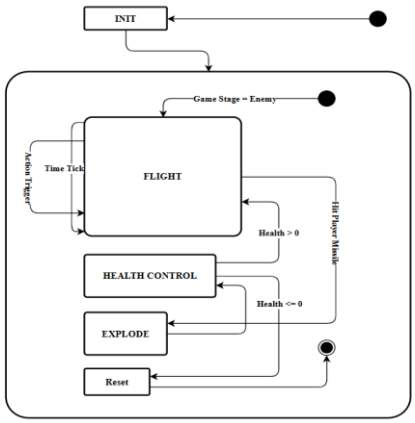


Figure 3.2 Enemy State Machine

While the player and enemy state machines in the game share similarities, they have distinct characteristics. The player state machine typically remains in the "FLIGHT" state, processing player inputs to perform various actions. If the player is hit by an enemy missile, an asteroid, or crashes into terrain, the state machine transitions to the "EXPLODE" state and then to the end state, resetting all other active objects.

In contrast, the enemy state machine only becomes active when the game reaches the "Enemy" stage. Like the player, the enemy spends most of its time in the "FLIGHT" state, performing actions automatically. When hit by a player's missile, it transitions to the "EXPLODE" state and then to the "HEALTH CONTROL" state. If the enemy's health is greater than 0, it returns to the "FLIGHT" state; otherwise, it resets the enemy active object and ends.

e. Player missile and enemy missile

Given the high similarity between the player missile and enemy missile state machines, detailing both would be redundant. The player missile state machine sufficiently illustrates the core functionality and behavior that apply to both entities. By focusing on one, the report remains concise and clear, while still effectively conveying the essential mechanics that govern missile behavior in the game.

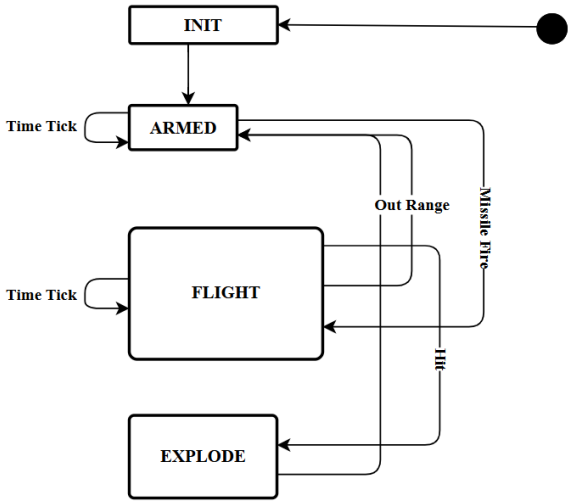


Figure 4 Missile State Machine

The Missile state machine does not have and end state due to the dependence nature of it in player ship object. When player ship is crash or destroyed, the whole game reset, both player missile and enemy missile is reset alongside its.

5. Display in “Space Shooting” Game

The "Space Shooting" game is designed for a 128x64 pixel black and white LCD screen, focusing on clarity and functionality. The main gameplay area features simplified, pixelated graphics for the player ship, enemies, asteroids, and terrain, balancing frame rates with clear visuals.

The game uses bitmaps—grid-based images defining each pixel's color and position—to precisely control graphics, ensuring detailed and recognizable shapes despite the small screen resolution. Animation is achieved by sequencing multiple images stored in a variable called "action_image," creating smooth animations for player and enemy movements, explosions, and other actions. This method enhances visual engagement and overall gameplay.

Listings 4 to 7 show how each object is displayed on the screen.

Listing 4 Asteroid display

```
void sst_asteroid_draw()
{
    for (uint8_t i = 0; i < NUM_ASTEROIDS; i++)
    {
        // If asteroid is not visible, do nothing and return
        if (myAsteroid[i].visible != WHITE)
        {
            return;
        }
        const uint8_t *asteroidBitmaps[] = {nullptr, sst_bitmap_asteroid_1,
                                            sst_bitmap_asteroid_2,
```

```
                                            sst_bitmap_asteroid_3};

        if (myAsteroid[i].action_image >= 1 && myAsteroid[i].action_image <= 3)
        {
            view_render.drawBitmap(myAsteroid[i].x, myAsteroid[i].y,
                                   asteroidBitmaps[myAsteroid[i].action_image],
                                   SIZE_BITMAP_ASTEROIDS_X,
                                   SIZE_BITMAP_ASTEROIDS_Y, WHITE);
        }
    }
}
```

Listing 5 Explosion display

```
void sst_explosion_draw()
{
    if (myExplosion.visible != WHITE)
    {
        return;
    }
    const uint8_t *explosionBitmaps[] = {nullptr, sst_bitmap_explosion_1,
                                         sst_bitmap_explosion_2,
                                         sst_bitmap_explosion_3};

    if (myExplosion.action_image >= 1 && myExplosion.action_image <= 3)
    {
        view_render.drawBitmap(   myExplosion.x,
                                  myExplosion.y,
                                  explosionBitmaps[myExplosion.action_image],
                                  SIZE_BITMAP_EXPLOSION_2_X,
                                  SIZE_BITMAP_EXPLOSION_2_Y, WHITE);
    }
}
```

Since the display logic for the player ship and the enemy ship is fundamentally similar, presenting both would be redundant. The player ship display code adequately demonstrates the rendering techniques and principles that apply to both types of ships. By focusing on the player ship, the report remains concise and avoids unnecessary repetition, while still providing a comprehensive understanding of the display mechanics used for both ship types in the game.

Listing 6 Player Ship

```
void sst_player_ship_draw()
{
    // If ship is not visible, do nothing and return
    if (myShip.ship.visible != WHITE)
    {
        return;
    }
    const uint8_t *playerShipBitmaps[] = {nullptr, sst_bitmap_space_ship_1,
                                          sst_bitmap_space_ship_2,
                                          sst_bitmap_space_ship_3};

    if (myShip.ship.action_image >= 1 && myShip.ship.action_image <= 3)
    {
        view_render.drawBitmap(myShip.ship.x,
                               myShip.ship.y,
                               playerShipBitmaps[myShip.ship.action_image],
```

```
                                    SIZE_BITMAP_SHIP_X, SIZE_BITMAP_SHIP_Y, WHITE);
    }
}
```

Listing 7 Player Missile

```
void sst_player_missile_draw()
{
    if (myMissile.visible != WHITE)
    {
        return;
    }
    view_render.drawBitmap(myMissile.x,
                           myMissile.y,
                           sst_bitmap_missile,
                           SIZE_BITMAP_MISSILE_X,
                           SIZE_BITMAP_MISSILE_Y,
                           WHITE);
}
```

The pointer is used to store all the bitmap for an object and only pass their address as an argument. The `view_render.drawBitmap()` take objects x and y position, the bitmap of the object, size of the bitmap, and color (`BLACK` and `WHITE`) and draw the object on the LCD screen at x and y position.