Pusan National University
Computer Science and Engineering
Embedded System (059)

임베디드 시스템 (059 분반)

Term Project 보고서

주제: Monty Hall Dilemma Simulator

202155513 김경환

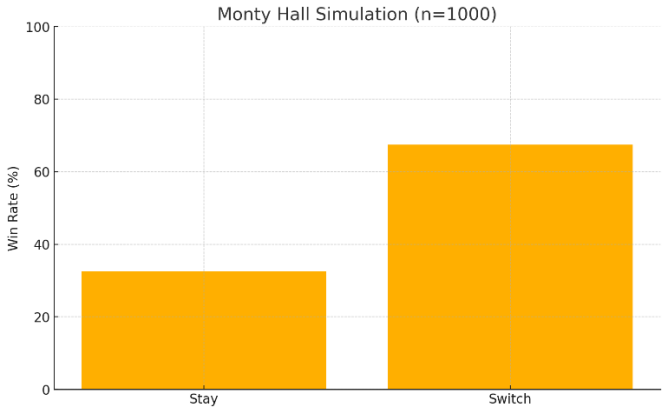
202170116 윤민석

목 차

1. 과제 목표

1.1. 과제 선정 이유 및 소개

몬티홀 딜레마(Monty Hall Dilemma)는 1960년대 미국의 유명 TV 퀴즈쇼 *Let's Make a Deal*의 사회자인 몬티 홀(Monty Hall)에서 유래한 확률 역설 문제이다. 참가자는 세 개의 문 중 하나를 골라 그 뒤에 숨겨진 상품(자동차)을 찾는데, 진행자가 나머지 문 하나를 열어 염소를 보여준 뒤 "선택을 바꾸시겠습니까?"라고 물을 때, 바꿨을 때와 바꾸지 않았을 때의 승률이 어떻게 달라지는지가 직관에 반하는 결과로 화제가 되었다. 이 딜레마는 단순한 게임 룰에도 불구하고 확률·통계에 대한 직관적 이해를 크게 뒤흔들어, 수많은 강의와 연구에서 다뤄지며 전 세계적으로 큰 관심을 받아왔다.



전략에 따른 승률 차이(시뮬레이션 1 000 회).
'Stay'(초기 선택 유지) 전략의 평균 승률은 **약 33 %**,
'Switch'(호스트 공개 후 변경) 전략은 **약 67 %**로 두 배 이상 높다.

최근 유튜브에서 한 크리에이터가 이 역설을 메타버스 환경에서 실험하는 영상을 시청하던 중, "직접 인터랙티브 시뮬레이터를 만들어 보면 이해가 더 쉽지 않을까?"라는 생각이 들었다. 특히 임베디드 시스템 수업에서 배운 STM32F429 보드와 uC/OS-III 실시간 운영체제를 활용하면, 조이스틱이나 버튼 같은 물리적 입력 장치와 직렬 터미널 출력, LED 표시 등을 통해 실제로 게임을 체험하며 확률의 변화를 직접 눈으로 확인하면서 이해도를 높일 수 있을 것이다. 이 과제는 이론적 개념으로만 접했던 몬티홀 딜레마를 실제 환경에서 구현하고 실험함으로써, 임베디드 소프트웨어 설계·디버깅 능력과 RTOS 태스크 간의 통신 및 동기화에 대한 이해를 동시에 높이기 위해 선정되었다.

본 프로젝트를 통해 몬티홀 딜레마의 확률적 특성을 체험적으로 이해하면서, **세마포어**·뮤텍스·메시지 큐 등 uC/OS-III의 핵심 기능들을 실제 하드웨어에서 구현·검증함으로써 임베디드 시스템 개발 역량을 종합적으로 강화하는 것을 목표로 한다.

1.2. 임베디드 시스템 개요

임베디드 시스템은 **특정 목적을 수행하도록 설계된 전용 컴퓨팅 시스템**으로, 하드웨어·소프트웨어가 하나의 제품 내부에 긴밀히 통합되어 동작한다. 스마트폰 카메라 모듈, 자동차 ECU, IoT 센서 노드 등에서 볼 수 있듯, **제한된 메모리·전력·실시간성**이라는 제약을 만족시키면서 신뢰성 있는 동작이 중요하다.
본 프로젝트는 **STM32F429ZI Nucleo**(Cortex-M4, 180 MHz, 2 MB Flash, 256 KB SRAM)를 기반으로, **조이스틱·푸시 버튼**(입력)과 **LED·ANSI UART**(출력)를 제어하는 *소형 인터랙티브 게임* 플랫폼을 구현함으로써 임베디드 시스템 설계 전 과정을 경험한다.
특히, 하드웨어 RNG·ADC·GPIO·USART 주요 주변장치를 직접 다루면서, 실시간 상호작용이 요구되는 게임 로직을 검증한다.

1.3. RTOS(uC/OS-III) 개요

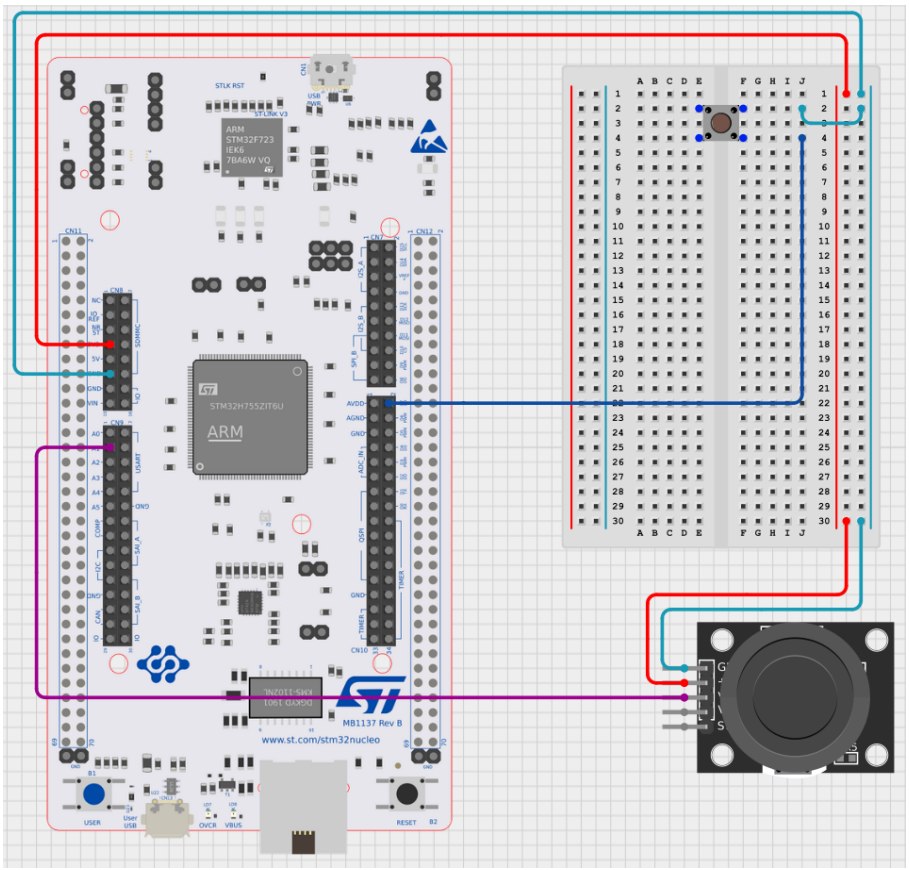
전통적인 **Bare-metal 루프**는 코드 구조가 단순하지만, **복수 작업이 동시 요구**될 때 우선순위 관리·블로킹 처리·시간 결정성이 떨어진다. 이를 해결하기 위해 **RTOS(Real-Time Operating System)** 가 사용된다.
uC/OS-III는 µ-kernel 구조의 상업용 RTOS로서

1. **프리엠티브 우선순위 스케줄링**(0=최상위)
2. **태스크 간 IPC**: 세마포어, 메시지 큐, 이벤트 플래그
3. **런타임 Stk Usage 체크 / 시간지연 API**
   등을 제공해, **예측 가능한 실시간 응답**을 보장한다.

본 프로젝트는 4개의 주요 태스크를 **서로 다른 우선순위**로 배치하고, **바이너리 세마포어 8종**으로 이벤트를 전달한다. 이를 통해 *입력 지연을 10 ms 미만으로 억제*하고, *LED/ANSI UART 표시를 게임 결과와 동기화*해 RTOS의 효과를 확인하였다.

1.4. 프로젝트 개요

- **과제 목표**: STM32F429 보드와 uC/OS-III 기반 RTOS 환경에서 몬티홀 딜레마를 물리적으로 체험할 수 있는 인터랙티브 시뮬레이터를 구현
- **입력 디바이스:** 조이스틱 (ADC: PC0) ←/→ 커서 이동, 버튼 (GPIO PF13) ↓ 선택/확인
- **출력 디바이스:** UART 터미널 (USART3, ANSI 화면 제어), 물리 LED (녹색/빨강)
- **모니터링 태스크:** USARTTask를 통해 현재 단계·통계 정보를 TTY 출력
- **회로도:**



1.5. Task 간 상호작용

본 시스템은 총 5개의 주요 Task가 uC/OS-III 세마포어를 통해 아래와 같이 협력한다.

1) AppTask_INPUT (입력 처리)

- 조이스틱·버튼 이벤트 감지 시
  · PHASE_SELECT 단계
  - Sem_DisplaySelectDone, Sem_UserSelectDone 포스트
    · PHASE_REVEAL 단계
  - Sem_DisplaySwitchDone, Sem_SwitchSelectDone 포스트
    · PHASE_RESULT 단계
  - Sem_NextRoundDisp, Sem_NextRoundLogic 포스트

2) AppTask_GameLogic (게임 로직)

- Sem_UserSelectDone 대기 → 사용자 선택 처리 → Sem_DisplaySwitchDone 포스트
- Sem_SwitchSelectDone 대기 → 최종 선택·승패 판정 → Sem_ResultReady 포스트
- Sem_NextRoundLogic 대기 → 다음 라운드 재시작

3) AppTask_GAME (화면 렌더링)

- Sem_DisplaySelectDone 대기 → 선택 UI 갱신 (RenderScreen)
- Sem_DisplaySwitchDone 대기 → 교체 UI 갱신
- Sem_ResultReady 대기 → 결과 UI 갱신 → Sem_LedShow 포스트
- Sem_NextRoundDisp 대기 → 다음 라운드 화면 준비

4) AppTask_LED (LED 표시)

- Sem_LedShow 대기 → 승리 시 녹색 LED, 패배 시 적색 LED 2초 점등

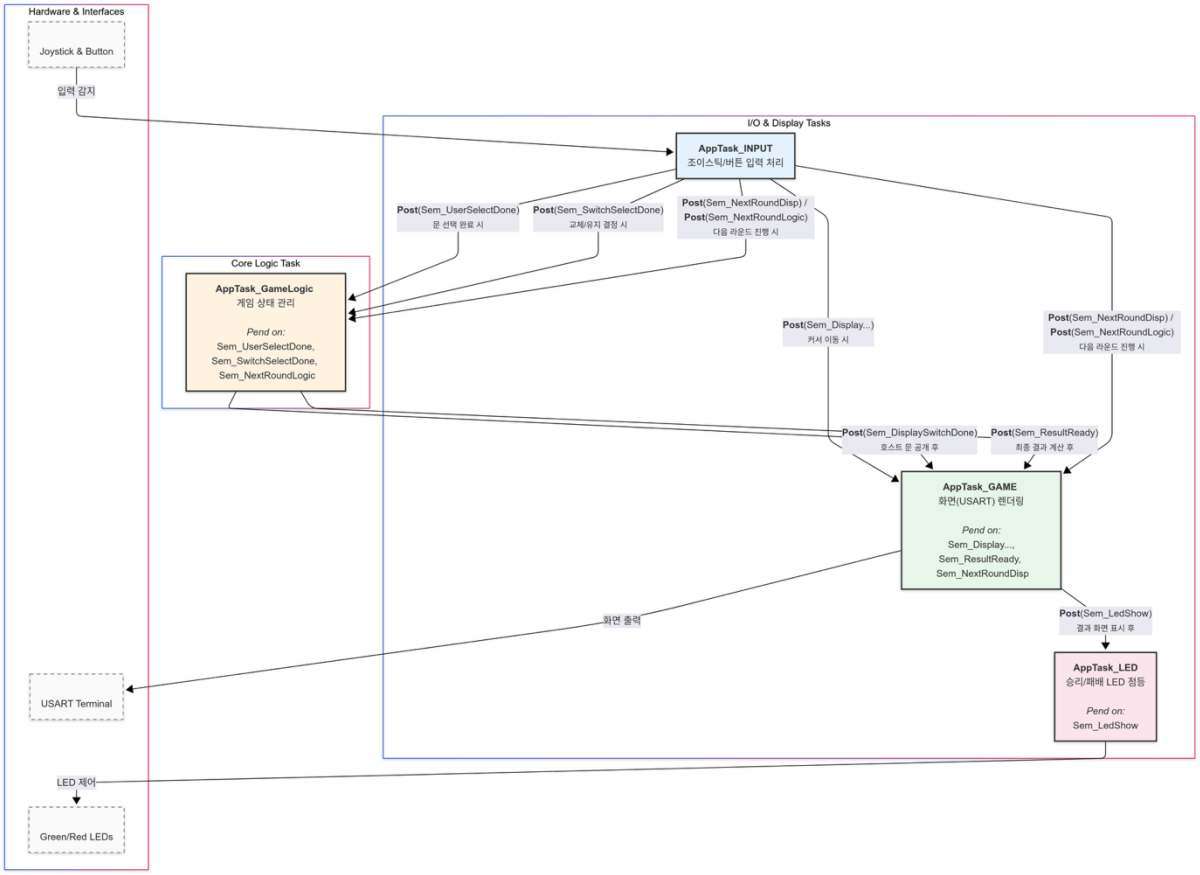
5) USARTTask (모니터링)

- 각 Task의 send_string() 호출 시 TTY로 상태·통계 정보 출력

※ 세마포어(OS_SEM): Task 간 이벤트(입력, 화면 갱신, 결과 표시, 다음 라운드) 전달

※ 크리티컬 섹션(OS_CRITICAL_ENTER/EXIT): 전역 변수(gamePhase, cursor 등)의 일관된 읽기·쓰기를 보장

1.6. 각 Task 간의 관계도 및 Flow Chart



1.7. 사용한 디바이스 소개

1.7.1. STM32F429ZI Nucleo 보드

USB 케이블로 PC에 연결하여 전원 공급 및 디버깅/시리얼 통신(USB↔USART3 브리지) 역할

1.7.2. 브레드보드

모든 모듈(조이스틱, 푸시 버튼)을 장착하고 점퍼선으로 STM32F429ZI와 연결

1.7.3. 조이스틱 모듈

5핀 (GND, VCC, VRx, VRy, SW) 중 VRx 핀을 PC0(ADC1_IN10)에 연결

GND→브레드보드 GND 레일, VCC→3.3V 레일

1.7.4. 푸시 버튼

브레드보드 상단에 설치, 한쪽 다리를 3.3V 레일, 다른 쪽 다리를 PF13(GPIO)으로 연결

내부 풀업 사용(GPIO PuPd_UP)으로 버튼 눌림(LOW 엣지) 감지

2. 과제 수행 과정

2.1. 전역 상태 및 커널 오브젝트 설정

게임의 진행 단계(gamePhase), 사용자 선택(userChoice), 상금 위치(prizeDoor), 최종 승패 여부(gameWin) 등은 모두 volatile 전역 변수로 선언하였다. 이들 변수는 여러 Task 에서 공유되므로 크리티컬 섹션(OS_CRITICAL_ENTER/EXIT)으로 보호하며, Task 간 이벤트 전달은 uC/OS-III의 바이너리 세마포어를 사용한다.

```
typedef enum {
    PHASE_SELECT,   // 1단계: 문 선택 대기
    PHASE_REVEAL,   // 2단계: 호스트 문 공개 및 교체 선택 대기
    PHASE_RESULT    // 3단계: 최종 결과 표시
} GamePhase_t;

static volatile GamePhase_t gamePhase;
static volatile uint8_t    prizeDoor;
static volatile uint8_t    userChoice;
static volatile uint8_t    hostChoice;
static volatile bool       switchChoice;
static volatile bool       gameWin;
static volatile uint8_t finalDoorChoice = 0;   /* RESULT 단계에서 ▲ 표시용 */
```

2.2. 태스크 생성 및 우선순위 할당

게임 입력 처리, 로직 수행, 화면 렌더링, LED 제어를 위한 네 가지 Task는 AppTaskCreate() 함수에서 OSTaskCreate()로 생성되며, 각기 다른 우선순위로 설정된다. 우선순위 값은 숫자가 작을수록 스케줄러 상 더 높은 우선순위를 의미하며, 입력 처리(Task_INPUT)는 즉각적인 반응이 필요하므로 가장 높은 우선순위(0)를, LED 표시(Task_LED)는 게임 화면 갱신 이후 동작하면 되므로 가장 낮은 우선순위(5)를 부여하였다.

2.3. AppTask_INPUT 구현

AppTask_INPUT는 반복 루프에서 조이스틱과 버튼 입력을 감지하여, 현재 게임 단계에 맞춰 커서 위치(cursorDoor/cursorSwitch)를 갱신하고, 화면 갱신이나 다음 단계 진행을 알리는 세마포어를 포스트한다. 조이스틱이 좌우로 움직이면 Joystick_ReadDir() 호출 후 PHASE_SELECT 단계에서는 문 선택 커서를, PHASE_REVEAL 단계에서는 Stay/Switch 커서를 토글하고 해당 화면 갱신 세마포어(Sem_DisplaySelectDone 또는 Sem_DisplaySwitchDone)를 깬다. 버튼(PF13)이 눌렸을 때는 PHASE_SELECT 단계에서 userChoice를 저장하고 선택 완료 세마포어를, PHASE_REVEAL 단계에서는 switchChoice를 저장하고 교체 완료 세마포어를, PHASE_RESULT 단계에서는 다음 라운드 진입 세마포어를 각각 포스트한다.

2.4. AppTask_GameLogic 구현

AppTask_GameLogic는 한 라운드를 다음과 같은 순서로 처리한다. 먼저 화면 갱신 세마포어를 리셋한 후, 크리티컬 섹션에서 cursorDoor와 cursorSwitch를 초기화하고, 하드웨어 RNG(RNG_GetRandom32())로 prizeDoor를 무작위 결정하여 gamePhase를 PHASE_SELECT로 설정한다.

```
CPU_SR_ALLOC();
OS_CRITICAL_ENTER();
cursorDoor = 1;
cursorSwitch = 0;
prizeDoor = (RNG_GetRandom32() % 3) + 1;
gamePhase = PHASE_SELECT;
userChoice = 0;
switchChoice = false;
doors[1] = doors[2] = doors[3] = DOOR_CLOSED;
strcpy(footer, "←/→ to move, BTN select");
OS_CRITICAL_EXIT();
```

이어서 Sem_DisplaySelectDone을 포스트해 화면 Task를 깨우고, Sem_UserSelectDone을 Pend하여 사용자의 선택을 대기한다. 사용자가 문을 고르면, 크리티컬 영역에서 userChoice와 prizeDoor를 비교해 호스트가 공개할 염소 문(hostChoice)을 결정하고, doors[hostChoice]를 DOOR_OPEN_GOAT으로 변경한 뒤 gamePhase를 PHASE_REVEAL로 전환하고 Sem_DisplaySwitchDone을 포스트한다. 이후 Sem_SwitchSelectDone 대기 후, switchChoice에 따라 최종 문(finalDoor)을 계산하고, 승패(gameWin)를 판정하여 통계를

누적한 뒤 doors 상태를 상금/실패 모드로 업데이트하고 gamePhase를 PHASE_RESULT로 바꾼다. 마지막으로 Sem_ResultReady를 포스트해 결과 화면을 갱신하고, Sem_NextRoundLogic을 Pend하여 다음 라운드 진입 신호를 대기한다.

2.5. AppTask_GAME 구현 (터미널 UI)

AppTask_GAME는 터미널 화면을 몬티홀의 세 단계(선택, 교체, 결과)에 맞추어 갱신하는 역할을 수행한다. 먼저 선택 단계에서는 크리티컬 섹션으로 cursorDoor와 gamePhase를 스냅샷한 뒤 RenderScreen(cursorDoor, 0)을 호출하여 초기 UI를 표시한다. 이후 Sem_DisplaySelectDone에 Pend하여 사용자가 커서를 이동할 때마다 깨어나 다시 스냅샷하고 화면을 갱신한다. 다음으로 교체 단계에 진입하면, 마찬가지로 cursorDoor와 cursorSwitch를 스냅샷한 뒤 RenderScreen(cursorDoor, cursorSwitch)을 호출해 Stay/Switch 선택 UI를 표시하고, Sem_DisplaySwitchDone을 Pend하며 반복 갱신한다. 마지막으로 결과 단계에서는 Sem_ResultReady를 Pend해 결과 준비를 대기한 뒤 RenderScreen(0, 0)으로 최종 문 상태와 통계를 출력하고, Sem_LedShow를 포스트해 LED Task를 실행한다. 이어서 Sem_NextRoundDisp를 Pend하여 사용자가 버튼을 눌러야 다음 라운드를 시작하도록 대기한다.

2.6. AppTask_LED 구현

AppTask_LED는 결과 단계에서 Sem_LedShow 세마포어를 Pend하여 게임 로직이 결과 준비 신호를 보낼 때까지 대기한다. 신호를 받으면 크리티컬 섹션으로 gameWin 값을 스냅샷한 뒤, 승리(gameWin == true) 시 녹색 LED, 패배 시 적색 LED를 물리적으로 점등한다. 약 2초간 유지한 후 모든 LED를 소등하여 결과 표시를 마무리한다. 이 Task는 결과 화면 렌더링과는 독립적으로 동작하며, LED 제어가 끝난 뒤 다시 대기 상태로 돌아간다.

2.7. 모니터링 Task (USART Task) 구현

모니터링 기능은 별도의 독립 Task 대신, 각 단계의 화면 갱신과 상태 변경 시 send_string() 호출을 통해 UART3로 메시지를 전송하는 방식으로 구현되었다. 게임의 진행 과정에서 RenderScreen()과 MakeStatsLine()이 ANSI 이스케이프 코드를 포함한 문자열을 생성하면, 이들을 send_string()이 한 문자씩 전송하여 PC 터미널(예: Tera Term)에 실시간으로 출력한다. 이를 통해 화면 렌더링뿐 아니라 승률, 라운드 수 등 통계도 함께 모니터링할 수 있다.

```
void send_string(const char *str) {
   while (*str) {
      while (USART_GetFlagStatus(Nucleo_COM1, USART_FLAG_TXE) == RESET);
      USART_SendData(Nucleo_COM1, *str++);
   }
}
```

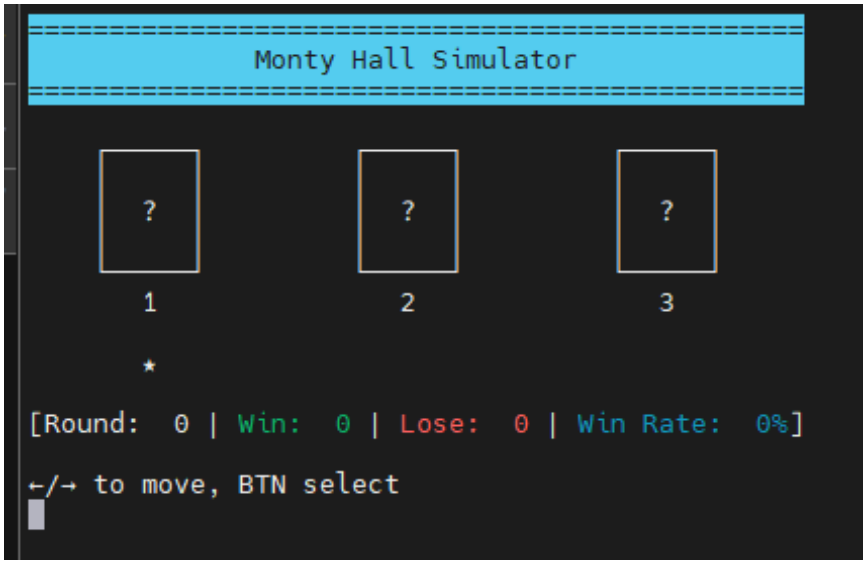
3. 과제 수행 결과

과제 수행 영상링크 :

- 기본적인 동작 및 로직 시연: https://drive.google.com/file/d/1LqxfFb7qlLFlOKKnbiCvtcCL4bLsm2zz/view?usp=sharing
- Stay (초기 선택 전략) 시뮬레이션 – 승률 40% (10 round): https://drive.google.com/file/d/1tG_Z3kOaYmiSMC2Hs8wObz8Qf3lJ6oCR/view?usp=sharing
- Switch (호스트 공개 후 변경) 시뮬레이션 – 승률 70% (10 round): https://drive.google.com/file/d/1JSx_sL-r_N6UWM8pBPWe9k0b-T1FMNFp/view?usp=sharing

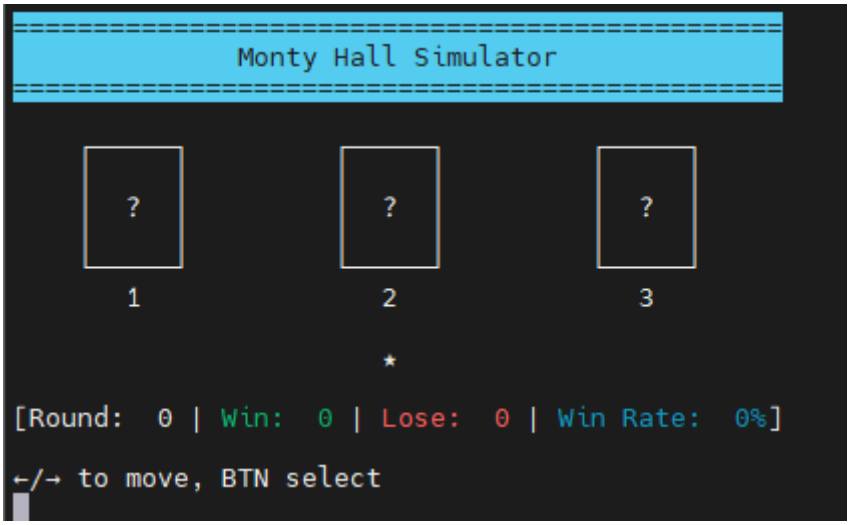
3.1. 라운드 초기화 및 RNG 동작 검증

라운드가 시작될 때마다 AppTask_GameLogic에서 cursorDoor와 cursorSwitch를 1, 0으로 초기화하고, 하드웨어 RNG를 통해 prizeDoor를 (RNG_GetRandom32() % 3) + 1 형태로 결정한다. 이 과정을 수십 차례 반복하여 로그를 확인한 결과, 1번부터 3번 문까지 균등한 확률로 선택되는 것을 확인하였다. 또한, 초기화 직후 모든 문 상태가 DOOR_CLOSED로 설정되고, 하단 안내 메시지가 "←/→ to move, BTN select"로 일관되게 출력되어 라운드 초기화가 올바르게 수행됨을 검증하였다.



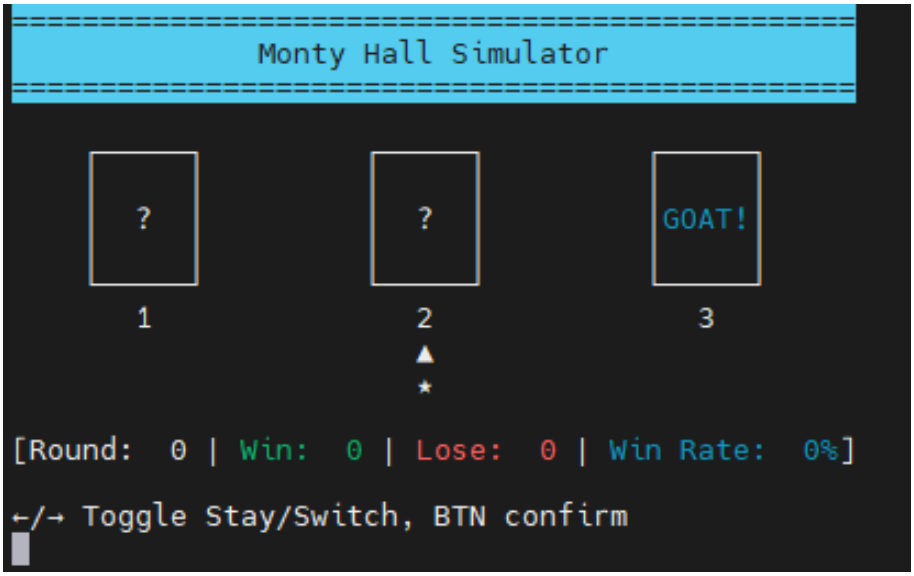
3.2. 선택 단계(PHASE_SELECT) 동작 확인

PHASE_SELECT 단계에서 조이스틱 좌/우 이동에 따라 cursorDoor가 1→2→3→1 순으로 순환하는지 확인하였다. 터미널 상에는 별표(★) 커서가 움직이는 문 아래에 정확히 표시되었으며, 버튼을 눌러 선택했을 때 userChoice가 해당 번호로 저장되고 Sem_UserSelectDone이 포스트되어 다음 단계로 정상 진입함을 검증했다. 여러 사용자가 동일한 동작을 반복해도 일관된 화면 갱신과 세마포어 트리거가 이루어졌다.



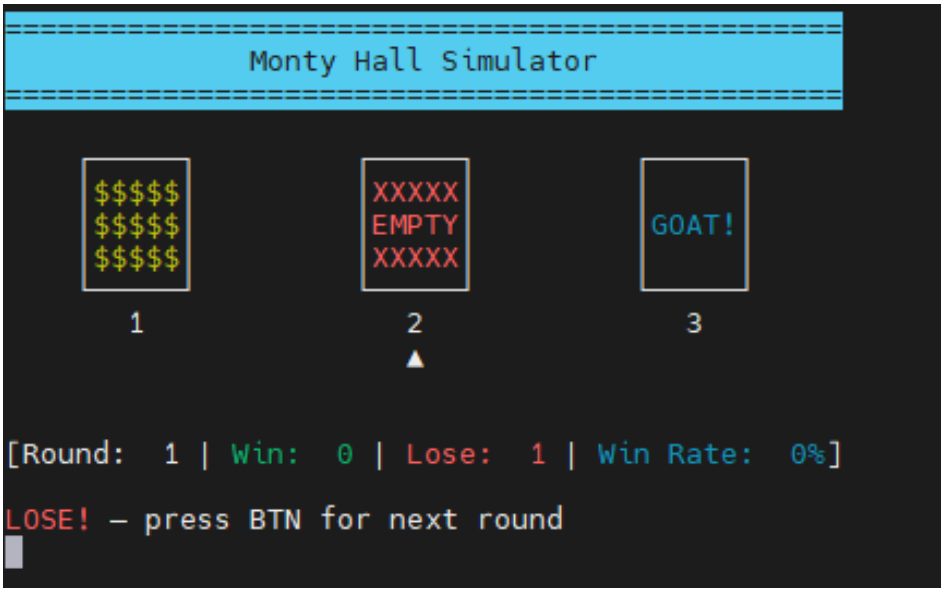
3.3. 교체 단계(PHASE_REVEAL) 동작 확인

첫 선택 후 PHASE_REVEAL로 전환되면 호스트가 염소 문을 하나 열고, 사용자는 "유지" 또는 "교체" 중 하나를 선택할 수 있다. 이때 조이스틱을 조작하면 cursorSwitch가 토글되며, 터미널에는 "Stay" 영역과 "Switch" 영역 아래에 별표가 정확히 위치하였다. 버튼 입력 시 switchChoice 값이 올바르게 저장되고 Sem_SwitchSelectDone이 포스트되어 최종 선택 로직이 실행되었다. 호스트 공개 로직과 사용자 교체 신호 간의 동기화가 모두 정상적으로 동작함을 확인했다.



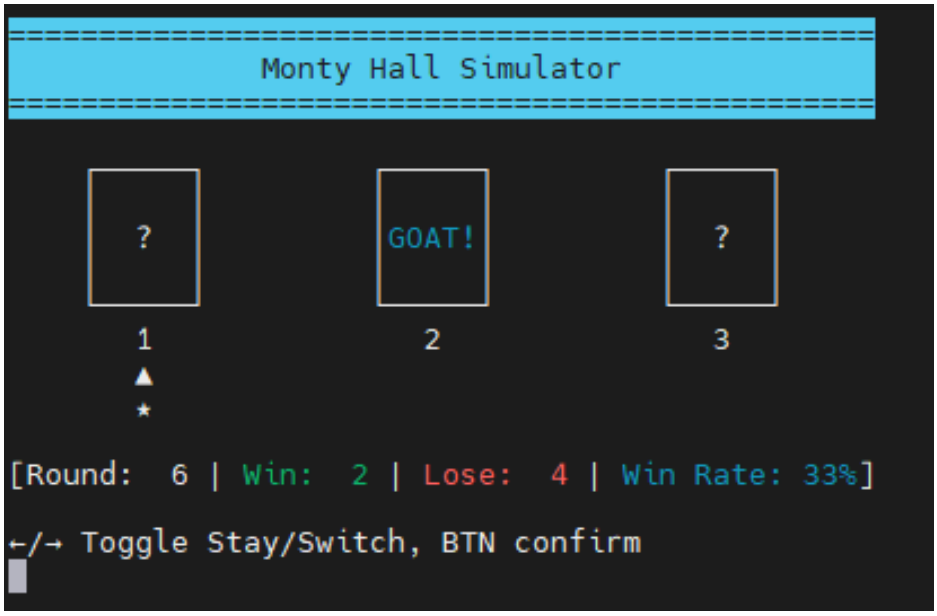
3.4. 결과 단계(PHASE_RESULT) 및 LED 표시 확인

최종 선택이 결정되면 PHASE_RESULT로 넘어가 상금 문은 노란색 $ 아트로, 실패 문은 붉은색 XXXXX/EMPTY 아트로 터미널에 출력되었다. 동시에 Sem_LedShow가 포스트되어 AppTask_LED가 2초간 녹색(GREEN) 또는 적색(RED) LED를 점등하였다. 실제 보드 상에서 LED가 정확히 2초간 유지된 뒤 소등되었으며, 터미널과 물리적 LED 표시가 일치함을 검증하였다.



3.5. 통계 결과 및 승률 분석

각 라운드마다 g_roundCount, g_winCount, g_loseCount가 누적되어, MakeStatsLine()을 통해 [Round: n | Win: w | Lose: l | Win Rate: x%] 형태로 표시되었다. 반복 실험 결과, "교체" 전략을 사용했을 때 약 66% 수준의 승률이, "유지" 전략을 사용했을 때 약 33% 수준의 승률이 나와 이론적 기대치(2/3 vs. 1/3)와 유사함을 확인하였다. 이 통계는 실험이 진행됨에 따라 실시간으로 갱신되어 사용자에게 직관적인 피드백을 제공하였다.



4. 조원 역할 분담

김경환: GameLogic / LED Task 구현, 회로 제작

윤민석: Game(UI) / INPUT Task 구현, 시뮬레이터 실험

5. 결론 및 기대 효과

이번 프로젝트를 통해 uC/OS-III 기반으로 여러 개의 태스크를 생성하고 세마포어와 크리티컬 섹션을 이용해 상호 동기화하는 방법을 익혔다. 특히, 조이스틱 입력을 처리하는 AppTask_INPUT, 난수를 생성하고 게임 로직을 관리하는 AppTask_GameLogic, 터미널 UI를 갱신하는 AppTask_GAME, 그리고 결과를 물리적 LED로 표시하는 AppTask_LED를 순차적으로 구현해 보며 RTOS 환경에서 태스크 간 메시지 통신 흐름을 체험했다. 하드웨어 RNG를 통해 상금 문을 무작위로 결정하고, 선택·교체·결과 각 단계에서 세마포어가 정확히 동작함을 로그와 LED 점등을 통해 확인할 수 있었다.

또한, 터미널 상의 ANSI 이스케이프 코드를 활용해 ASCII 아트 기반 UI를 구현하고, MakeStatsLine()으로 승률을 실시간으로 출력하며 간단한 통계 기능을 추가했다. 반복 실험 결과 "교체" 전략이 약 66%의 승률을, "유지" 전략이 약 33%의 승률을 보이며 이론값과 거의 일치함을 관찰함으로써 시뮬레이터의 정확도를 검증할 수 있었다. 이를 통해 하드웨어 제어뿐 아니라 데이터 기반 결과 검증의 중요성도 체감할 수 있었다.

앞으로는 이 기본 구조를 바탕으로 터치 센서나 무선 통신 모듈을 연동해 입력 방식을 다변화하거나, 웹 인터페이스를 통해 원격으로 결과를 모니터링하는 기능을 추가해 보고 싶다. 이번 과제를 통해 익힌 RTOS 설계·개발 과정은 차후 더 복잡한 프로젝트에 큰 밑바탕이 될 것으로 기대한다.

6. 참고 자료

https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf

https://www.st.com/en/microcontrollers-microprocessors/stm32f429zi.html

https://wiki.st.com/stm32mpu/wiki/Hardware_random_overview

https://en.wikipedia.org/wiki/Monty_Hall_problem