# Microprocessor Application

## Final Report

21900783
Sanghwa Han
21900628
Sechang Jang

**2024/6/23**

# Table of Contents

# Problem Description

## Motivation

The inspiration for this project came to us while playing ping pong. However, our vision extended beyond simply implementing a ping-pong game.

## Increasing game market

Video games have been a source of joy and entertainment for decades, captivating people of all ages. From the early days of simple arcade games to the complex and immersive experiences available today, gaming has consistently provided a fun and engaging way for people to spend their time. The evolution of video games reflects the advancements in technology and changing consumer interests, showing how this form of entertainment has grown and adapted over the years.

The global games market is expected to generate revenue of $455.30 billion in 2024[1]. This large amount highlights the immense popularity and importance of the gaming industry worldwide. As technology continues to improve, more people are becoming interested in gaming, leading to even more growth in the market. The increasing revenue indicates that gaming is not just a pastime but a significant part of global entertainment culture.

## Problem playing with integrated LED

However, Making nRF52840 DK as a game-playing device has an issue.

The LED has only one color, making it difficult to distinguish objects while playing the game. This limitation significantly impacts the gaming experience, as players often rely on different colors to identify various game elements, such as characters, obstacles, and items. The lack of color differentiation can lead to confusion, mistakes, and a less engaging experience. In more complex games where color-coded signals or indicators are crucial, this single-color LED becomes a substantial drawback.

Additionally, the LED brightness is intense, which can cause discomfort during extended gaming sessions. High brightness levels, while initially striking, can lead to eye strain and fatigue over time. For gamers who enjoy long sessions, this can become a serious issue, affecting both their comfort and performance. Prolonged exposure to excessive brightness can also contribute to more severe visual problems, such as headaches or blurred vision, which further detracts from the overall gaming experience.

Moreover, the LED matrix screen is too small to support a variety of games effectively. The limited screen size restricts the display of detailed graphics and complex game interfaces, which are essential for many modern games. This size constraint makes it challenging to include intricate visuals and detailed user interfaces, thereby limiting the types of games that can be effectively played on the device. For games that require detailed graphics or a larger field of view, a small LED matrix screen can severely compromise the gameplay experience.

The small display size can also make it difficult for players to see important game details, leading to a less immersive and enjoyable gaming experience.

# Proposed method (How to implement)

We have embarked on an ambitious project to develop a versatile gamepad using the nRF52840 Development Kit (DK) that is capable of operating across a variety of gaming platforms. Our goal is to create a Bluetooth Low Energy (BLE) gamepad that can enhance and elevate the gaming experience for users, offering compatibility with multiple gaming systems and providing seamless integration and high performance. By leveraging the advanced capabilities of the nRF52840 DK, we aim to design a gamepad that not only meets the diverse needs of gamers but also delivers a robust and reliable gaming experience across different platforms.

The BLE gamepad setup is designed to enhance user interaction and gameplay with multiple components. LED 1 indicates joystick actions, providing visual feedback for intuitive control. LED 2 shows the Bluetooth connection status, ensuring the gamepad is connected to the server. The gamepad includes four buttons: Buttons 1, 2, and 3 start different games, while Button 4 terminates the current game, allowing seamless switching and quick stops. The joystick enables precise movement control, and the LED matrix displays the movement direction, improving the overall gaming experience.

## Background: Used Device

### nRF52840 DK

The nRF52840 DK[2] is a versatile single-board development kit designed for the nRF52840 SoC. It supports a wide range of applications including Bluetooth Low Energy, Bluetooth mesh, NFC, Thread, Zigbee, 802.15.4, ANT, and proprietary 2.4 GHz protocols. This development kit is recommended for Amazon Sidewalk and supports development on the nRF52811 SoC as well.

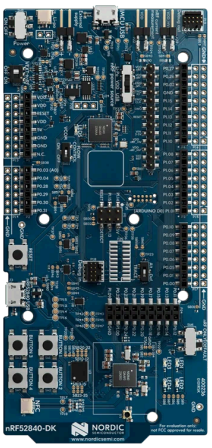
The nRF52840 DK is particularly useful for Matter over Thread applications, where Thread is used for transport and Bluetooth LE is used for commissioning. Matter devices that utilize Thread require concurrent Bluetooth LE functionality to facilitate the addition of new devices to the network.

This development kit leverages all the features of the nRF52840 SoC. It includes an NFC antenna for quick use of the NFC-A tag peripheral, provides access to all GPIOs via edge connectors and headers, and features 4 buttons and 4 LEDs for easy input and output. Additionally, it has onboard external memory connected to the QSPI peripheral of the nRF52840 SoC.

Compatible with Arduino Uno Revision 3, the nRF52840 DK allows easy mounting of third-party shields. It includes an on-board SEGGER J-Link debugger for programming and debugging both the on-board SoC and external targets through the debug-out header. It also interfaces directly with the Power Profiler Kit II.

The development kit is typically powered via USB but supports a range of power sources from 1.7 to 5.0 V. It can also be powered by an external source, a CR2032 battery, or a Li-Po

battery for in-field testing. Current consumption can be measured using dedicated current measurement pins.




<Figure 1. nRF52840 dk>

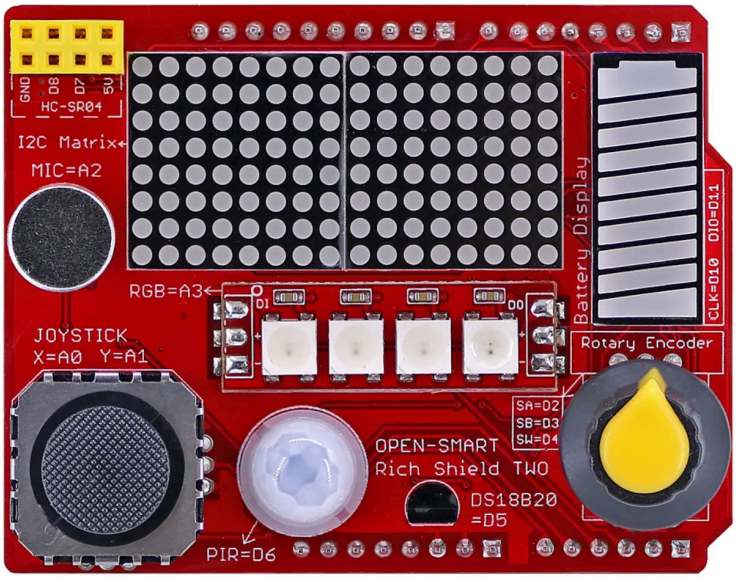
## Rich Shield TWO

This device is equipped with a variety of features to enhance its functionality[3]. It includes a 16×8 pixel I2C LED Matrix, capable of displaying characters, numbers, and basic graphics. A joystick is also provided for user input. Additionally, there is a 10-bar battery display with one blue bar, four green bars, three orange bars, and one red bar to indicate battery levels.

A rotary encoder with 360-degree rotation outputs 20 pulses per rotation and includes a button that outputs a high level when pressed. For temperature sensing, a DS18B20 digital temperature sensor is included, which can detect temperatures ranging from -25 to +100 degrees Celsius using just one signal line.

Movement detection is handled by a PIR sensor that outputs a high level for five seconds when movement is detected; otherwise, it outputs a low. Sound detection is managed by a sound sensor that outputs an analog value proportional to the loudness of the sound, dropping to zero in the absence of sound.

An RGB LED based on the WS2812, which features four individually controllable LEDs, allows for various color displays using a single I/O. There is also a female pin header for an HC-SR04 ultrasonic sensor, providing additional flexibility for distance measurement.




<Figure 2. Rich Shield TWO>

## Bluetooth-low Energy

Bluetooth Low Energy (BLE) is a wireless, low-power personal area network operating in the 2.4 GHz ISM band, designed for connecting devices over short distances[4]. Developed with Internet of Things (IoT) applications in mind, BLE is optimized for devices that are typically resource-constrained and require long battery life. To achieve this, BLE prioritizes low power consumption, entering sleep mode when not in use to conserve energy, rather than continuous data transfer.

Understanding BLE-equipped devices involves grasping the technology's architecture, particularly its asymmetrical nature. Devices can operate in either a central or peripheral role. For example, consider a smartphone and a smart band: the more complex smartphone acts as the central device, while the smart band, with its limited functionality, serves as the peripheral. Communication occurs only between a central and a peripheral device, meaning two central or two peripheral devices cannot directly communicate. However, many devices, like smartphones, can be configured to operate in both central and peripheral modes to overcome this limitation.

The BLE protocol stack is divided into the controller and host, similar to classic Bluetooth devices. Profiles and applications operate on top of the GAP (Generic Access Profile) and GATT (Generic Attribute Profile) layers. The physical layer (PHY) uses a 1-Mbps GFSK radio in the 2.4-GHz ISM band[5].

Key components of the stack include:

- **GAP**: Manages device states (Standby, Advertising, Scanning, Initiating, Connected) and controls the RF state.
- **HCI (Host Controller Interface)**: Facilitates communication between host and controller via a standardized interface (e.g., UART, SPI, USB).
- **L2CAP (Logical Link Control and Adaptation Protocol)**: Encapsulates data for end-to-end communication.

- **Security Manager**: Handles pairing, key distribution, and secure data exchange.
- **ATT (Attribute Protocol) and GATT (Generic Attribute Profile)**: Enable data exposure and communication between devices through defined sub-procedures.

The stack is primarily provided as object code, with selective enabling of features at build time to optimize memory usage.

# Background: Development

## SDK 2.5.0

The nRF Connect SDK[6] is a comprehensive software development kit designed for creating low-power wireless applications with Nordic Semiconductor's nRF52, nRF53, nRF70, and nRF91 Series devices. It supports a wide range of communication protocols, including cellular IoT (LTE-M and NB-IoT), Bluetooth® Low Energy, Thread, Zigbee, Wi-Fi®, and Bluetooth mesh.

## Zephyr RTOS

The Zephyr OS[7] is built around a small-footprint kernel, making it ideal for resource-constrained and embedded systems. It is versatile enough to be used in simple devices like environmental sensors and LED wearables, as well as in more complex applications such as embedded controllers, smart watches, and IoT wireless solutions.

Within the Zephyr OS, a subsystem refers to a distinct part of the operating system responsible for specific functions or services. These subsystems can include components like networking, file systems, device driver classes, power management, and communication protocols. Each subsystem is designed to be modular, allowing it to be configured, customized, and extended to meet the unique needs of various embedded applications.

## Used Programming Language

C - Client
Python - Server

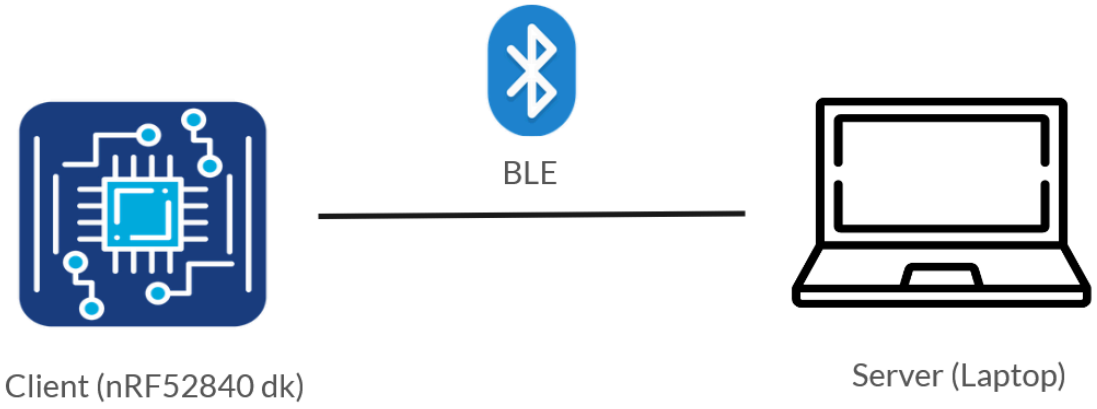
# Design - System

## System Architecture

We adopted a server-client architecture for our system, where the server is a laptop and the clients are nRF52840 development kits.

Client-server architecture is a network design framework in which multiple computers, known as clients, connect to a central computer, referred to as the server, to request and receive services or resources. The server acts as the central hub, managing resources and services such as data storage, processing power, and applications while handling requests from multiple clients. Clients are devices or applications that send requests to the server for specific tasks or data and receive and utilize the responses or resources provided by the server. This architecture allows for centralized management, making it easier to maintain and update the system.

This architecture allows the laptop to serve as the central hub, managing and coordinating communications with multiple nRF52840 devices connected via Bluetooth. In this setup, the laptop (server) handles the game's main processing tasks, while the nRF52840 devices (clients) act as input devices, sending user commands back to the server. This configuration not only facilitates seamless connectivity and interaction but also enables the possibility of multiplayer gaming by allowing multiple nRF52840 devices to connect to the server simultaneously.




<Figure 3. Server - Client Architecture of current System>

## Nordic UART Service (NUS)

The Nordic UART Service (NUS) is a custom Bluetooth Low Energy (LE) GATT service provided by Nordic Semiconductor, designed to facilitate data transmission between a Bluetooth device and a UART interface. It is incorporated in the Bluetooth: Peripheral UART sample, the Thingy:91 Connectivity bridge (disabled by default), and the Matter door lock sample as an optional feature. The service uses a 128-bit vendor-specific UUID and includes two main characteristics: RX (for receiving data) and TX (for sending data via notifications). The NUS API provides functions for initializing the service, sending data, and retrieving the maximum data length for transmission. The service also defines several constants, enums

for send status, and callback structures for handling received and sent data events, as well as notification status changes. The API documentation includes detailed definitions and parameters for implementing these functionalities in applications.

# Design: Server-side (Central)

The server used Python to set the server. Python code sets up a Bluetooth Low Energy (BLE) connection to a device and reads data from it to control a subprocess that runs a game script. The code uses the `adafruit_ble` library to handle BLE communication and the `subprocess` and `threading` modules to manage the game processes.

## Server Actions

**Initialize BLE Radio**:

The BLE radio is initialized using the `BLERadio` class from the `adafruit_ble` library.

**Scan for BLE Devices**:

The script scans for BLE devices within a given RSSI range for a specified timeout period (20 minutes). If a device with the name "Jang" is found, the script connects to it and stops scanning.

**Start Game Subprocess**:

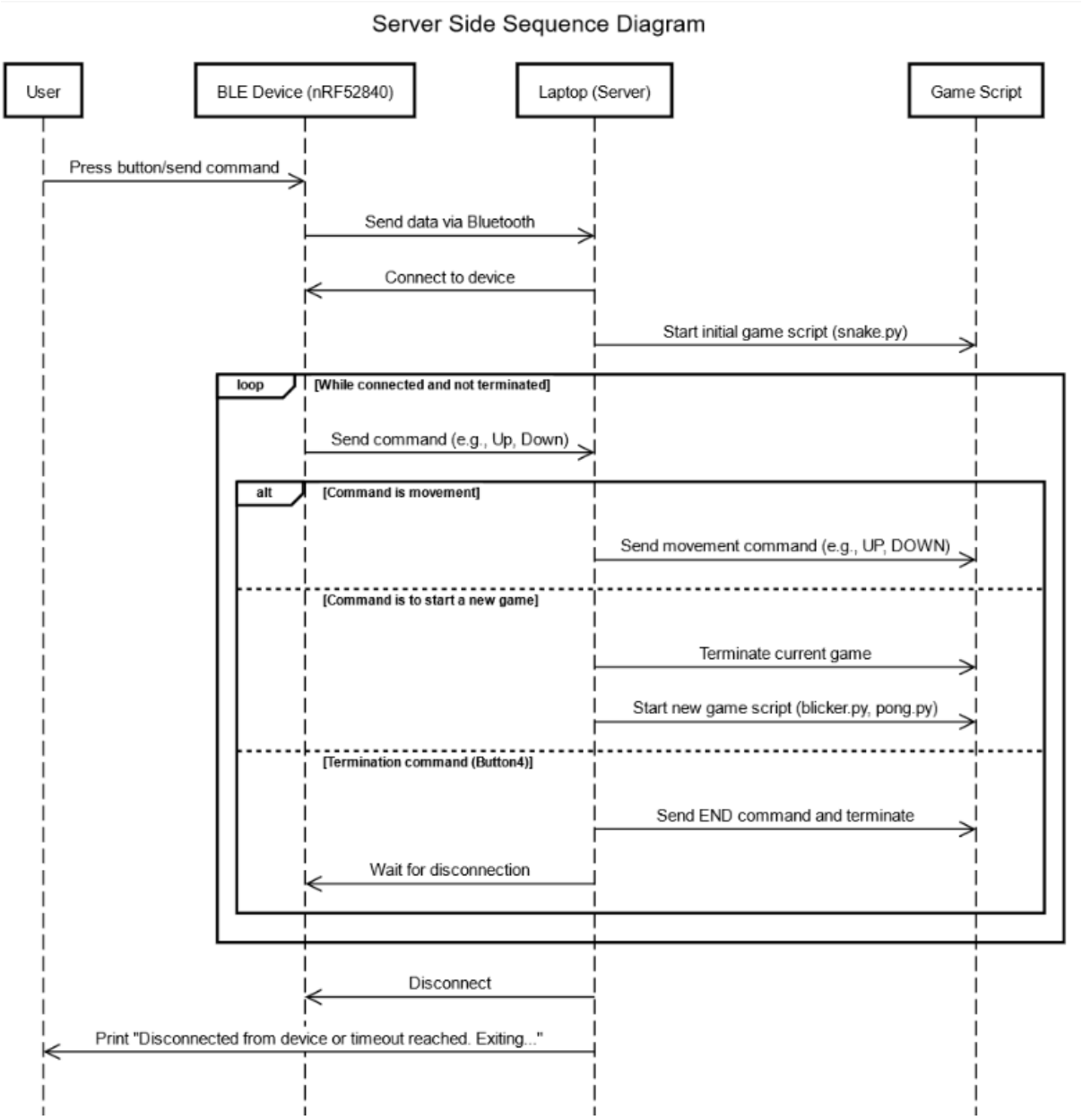
Once connected to the BLE device, a new thread is started to run the `run_subprocess` function, which starts an initial game (`snake.py`) and begins reading data from the BLE device.

**Read Data and Control Subprocess**:

The `read_from_device_and_send` function continuously reads data from the BLE device. Based on the received data (e.g., "Up", "Down", "Left", "Right", "Button1", etc.), the script sends commands to the running game subprocess or starts a new game script.

**Terminate Subprocess**:

If the "Button4" command is received, the current subprocess is terminated, and the script waits for the device to disconnect. The script then exits gracefully.


Server Side Sequence Diagram
User
BLE Device (nRF52840)
Laptop (Server)
Game Script
Press button/send command
Send data via Bluetooth
Connect to device
Start initial game script (snake.py)
loop
[While connected and not terminated]
Send command (e.g., Up, Down)
alt
[Command is movement]
Send movement command (e.g., UP, DOWN)
[Command is to start a new game]
Terminate current game
Start new game script (blicker.py, pong.py)
[Termination command (Button4)]
Send END command and terminate
Wait for disconnection
Disconnect
Print "Disconnected from device or timeout reached. Exiting..."

# Implementation: Server-side (Central)

## Code Analysis

1. Setup - Dependencies

```
1  from adafruit_ble import BLERadio
2  from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
3  from adafruit_ble.services.nordic import UARTService
4  import subprocess
5  import threading
6  import time
```

**`adafruit_ble`**: This library is used to handle BLE communication.
**`subprocess`**: This module is used to create and manage subprocesses.
**`threading`**: This module is used to run tasks concurrently.
**`time`**: This module is used for time-related functions.
**`current_process`**: A global variable to store the current running subprocess.
**`terminate_signal`**: A global variable used to signal the termination of the subprocess.

2. Function: `read_from_device_and_send`

```
def read_from_device_and_send(device):
    global current_process, terminate_signal

    while device and device.connected:
        # Read data from the Bluetooth device
        data = device[UARTService].read(20)  # Adjust buffer size as
needed

        if data:
            data = data.decode('utf-8').strip()
            print(f"Received data: {data}")

            # Send the appropriate commands to the subprocess
            if data.find("Button4") != -1:
                if current_process:
                    current_process.stdin.write(b"END\n")
                    current_process.stdin.flush()
                    terminate_signal = True
                    break
```

```
        elif data.find("Up") != -1:
            current_process.stdin.write(b"UP\n")
            current_process.stdin.flush()

        elif data.find("Down") != -1:
            current_process.stdin.write(b"DOWN\n")
            current_process.stdin.flush()

        elif data.find("Left") != -1:
            current_process.stdin.write(b"LEFT\n")
            current_process.stdin.flush()

        elif data.find("Right") != -1:
            current_process.stdin.write(b"RIGHT\n")
            current_process.stdin.flush()

        elif data.find("Button2") != -1:
            start_new_game("blicker.py")

        elif data.find("Button3") != -1:
            start_new_game("pong.py")

        elif data.find("Button1") != -1:
            start_new_game("snake.py")
```

This function reads data from the connected BLE device and sends commands to a running subprocess based on the received data.
It listens for specific strings like "Up", "Down", "Left", "Right", and various button presses, and then sends corresponding commands to the subprocess or starts a new game.

3. Function: `start_new_game`

```
def start_new_game(game_script):
    global current_process

    if current_process:
        current_process.stdin.write(b"END\n")
        current_process.stdin.flush()
        current_process.terminate()
        current_process.wait()

    current_process = subprocess.Popen(['python3', game_script], stdin=subprocess.PIPE,
                                       stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

This function starts a new game script by terminating any currently running subprocess and starting a new one.
It uses `subprocess.Popen` to start a new Python script, which allows for sending commands to the script via `stdin`.

4. Function: `run_subprocess`

```python
# Function to run the subprocess
def run_subprocess():
    global current_process, terminate_signal

    # Start the initial game
    start_new_game('snake.py')

    # Start thread to read from Bluetooth device and send to subprocess
    read_thread = threading.Thread(target=read_from_device_and_send, args=(device,))
    read_thread.start()

    while not terminate_signal:
        time.sleep(0.1)

    if current_process:
        current_process.terminate()
        current_process.wait()

    # Wait for the read thread to finish
    read_thread.join()
```

This function manages the lifecycle of the subprocess, starting with an initial game and handling the reading of commands from the BLE device in a separate thread.
It checks for the termination signal to properly shut down the subprocess and the reading thread.

5. Main Code

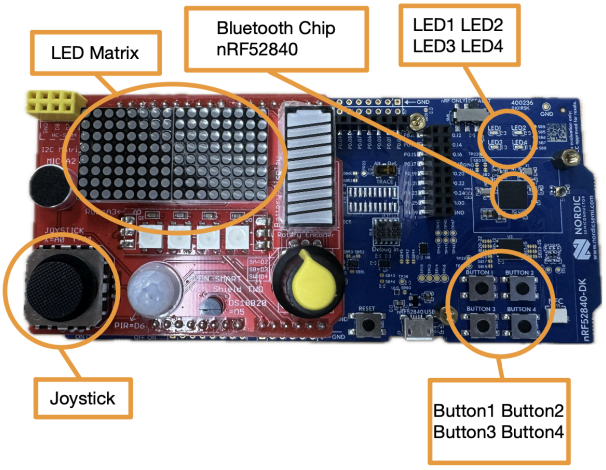
```python
# Initialize BLE radio
radio = BLERadio()

print("Scanning for devices...")
found = set()

# Scan for devices
for entry in radio.start_scan(timeout=1200, minimum_rssi=-80):
    addr = entry.address

    if addr not in found:
        print(entry.complete_name)
```

```
        # Replace with your device IDs
        if entry.complete_name in ["Jang"]:
            found.add(addr)
            device = radio.connect(entry)
            print("Device connected!")
            break

radio.stop_scan()

if device:
    # Start subprocess in a separate thread
    subprocess_thread = threading.Thread(target=run_subprocess)
    subprocess_thread.start()

    while device.connected:
        pass

    subprocess_thread.join()

print("Disconnected from device or timeout reached. Exiting...")
```

The BLE radio is initialized, and the script scans for BLE devices for up to 20 minutes (1200 seconds) with a minimum RSSI of -80 dBm.
If a device named "Jang" is found, it connects to the device and stops scanning.
**Device name could be vary, so you need to set the device name of your device.**
Once connected, it starts the `run_subprocess` function in a separate thread.
The main thread waits until the device disconnects, then joins the subprocess thread and exits.

# Design: Client-side (Peripheral)

## nRF52840 DK and Rich Shield TWO Usage




<Figure 4. nRF 52840 DK uses>

The BLE gamepad setup is designed with multiple components, each serving a specific function to enhance user interaction and gameplay experience. The first LED, LED 1, is dedicated to indicating the actions of the joystick. Whenever the joystick is moved, LED 1 provides visual feedback, ensuring the user is aware of the input being registered. This feature is crucial for maintaining intuitive control during gameplay.

Another vital component is LED 2, which indicates the status of the Bluetooth connection. This LED lights up to show when the gamepad is successfully connected to the server. This connectivity indicator is essential for troubleshooting and confirming that the gamepad is ready for use.

The gamepad is equipped with four buttons, each assigned a unique role. Button 1, Button 2, and Button 3 are used to start different games, labeled Game 1, Game 2, and Game 3, respectively. This setup allows users to switch between games seamlessly. Button 4, on the other hand, is used to terminate the currently running game. This feature provides a quick and efficient way to stop gameplay, which can be particularly useful in various scenarios, such as ending a game session or switching to another game.

The joystick is a central element of the gamepad, enabling users to move objects within the games. It provides precise control, making the gaming experience more engaging. To complement this, the LED matrix displays the direction of movement. This visual representation helps players understand the direction in which they are moving objects, enhancing the overall gameplay experience.

## Client Actions

**Initialization**:

It defines constants for LEDs, buttons, and UART buffer sizes. It defines structures for UART and BLE communication and initializes a semaphore (`ble_init_ok`) to signal when BLE is ready.

**UART Initialization**:

The `uart_init` function sets up the UART device, enables USB if needed, and configures UART callbacks. The `uart_cb` function handles different UART events like transmission done, reception ready, buffer requests, etc.

**BLE Initialization and Advertising**:

The `bt_enable` function initializes the Bluetooth stack. The `bt_nus_init` function sets up the Nordic UART Service for BLE communication. Advertising data (`ad` and `sd`) is set up to make the device discoverable and connectable.

**Connection Handling**:

The `connected` and `disconnected` callbacks handle BLE connection events, updating LED status to indicate connection status. The `bt_receive_cb` function processes received BLE data and forwards it to the UART.

**Button and LED Configuration**:

The `configure_gpio` function initializes buttons and LEDs. The `button_changed` function sends BLE messages when buttons are pressed.

**Main Loop**:

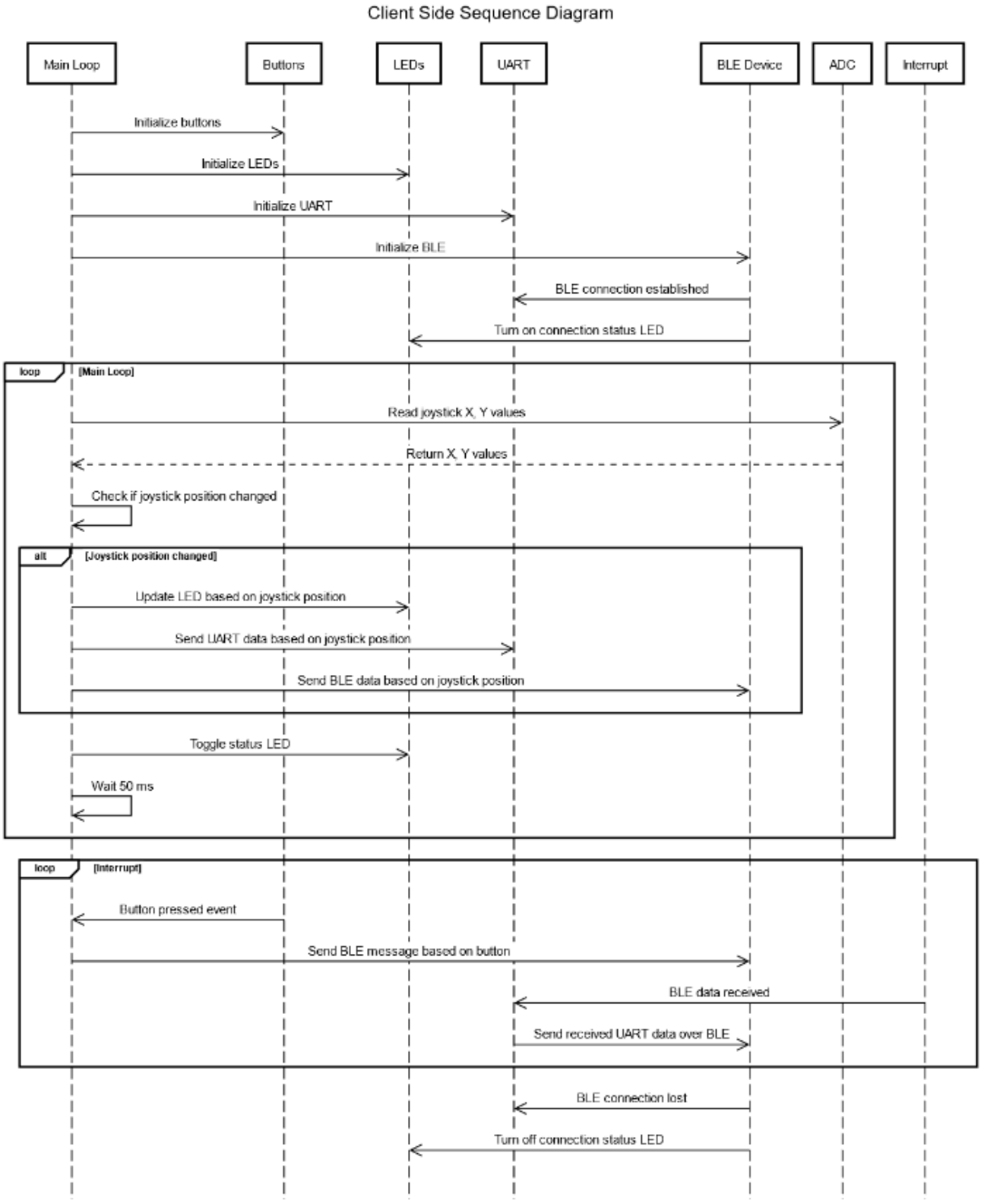
The `main` function sets up the GPIO, UART, and BLE, then enters a loop. In the loop, it reads ADC values (representing joystick positions) and sends corresponding BLE messages. LEDs indicate joystick direction, and a status LED blinks periodically.

**LED Control**:

The `isChange` function detects significant changes in joystick position. The ADC values are read and compared to predefined thresholds to determine joystick direction LEDs are turned on or off based on the joystick's position.

**BLE Write Thread**:

The `ble_write_thread` function waits for data in the UART RX FIFO and sends it over BLE. This thread runs concurrently with the main loop, ensuring continuous UART-BLE communication.


Client Side Sequence Diagram
Main Loop
Buttons
LEDs
UART
BLE Device
ADC
Interrupt
Initialize buttons
Initialize LEDs
Initialize UART
Initialize BLE
BLE connection established
Turn on connection status LED
loop
[Main Loop]
Read joystick X, Y values
Return X, Y values
Check if joystick position changed
alt
[Joystick position changed]
Update LED based on joystick position
Send UART data based on joystick position
Send BLE data based on joystick position
Toggle status LED
Wait 50 ms
loop
[Interrupt]
Button pressed event
Send BLE message based on button
BLE data received
Send received UART data over BLE
BLE connection lost
Turn off connection status LED

# Implementation: Client-side (Peripheral)

## Code Analysis

1. Setup - Dependencies

```
#include <zephyr/types.h>
#include <zephyr/kernel.h>
#include <zephyr/drivers/uart.h>
#include <zephyr/usb/usb_device.h>

#include <zephyr/device.h>
#include <zephyr/devicetree.h>
#include <soc.h>

#include <zephyr/bluetooth/bluetooth.h>
#include <zephyr/bluetooth/uuid.h>
#include <zephyr/bluetooth/gatt.h>
#include <zephyr/bluetooth/hci.h>

#include <bluetooth/services/nus.h>

#include <dk_buttons_and_leds.h>

#include <zephyr/settings/settings.h>

#include <stdio.h>

#include <zephyr/logging/log.h>

//led
#include <zephyr/drivers/adc.h>
#include <zephyr/kernel.h>
#include <zephyr/sys/util.h>
#include <inttypes.h>
#include <stddef.h>
#include <stdint.h>
```

This section includes various headers required for the functionality of the application, such as Bluetooth, UART, USB, GPIO, ADC, and logging.

- Kernel Services:
  - `kernel.h`
- Peripherals:
  - `gpio.h`
  - `uart.h`

- Bluetooth APIs
    - `inbluetooth/bluetooth.h`
    - `bluetooth/gatt.h`
    - `bluetooth/hci.h`
    - `bluetooth/uuid.h`

- Nordic UART Service (NUS)
    - `bluetooth/services/nus.h`

- Buttons, LEDs, Joystick, and I2C Matrix

    - `dk_buttons_and_leds.h`
    - `drivers/adc.h`
    - `sys/util.h`
    - `settings/settings.h`

2. Device Name, button/LED Definition

```
#define DEVICE_NAME CONFIG_BT_DEVICE_NAME
#define DEVICE_NAME_LEN (sizeof(DEVICE_NAME) - 1)

#define RUN_STATUS_LED DK_LED1
#define CON_STATUS_LED DK_LED2

#define BUTTON1 DK_BTN1_MSK
#define BUTTON2 DK_BTN2_MSK
#define BUTTON3 DK_BTN3_MSK
#define BUTTON4 DK_BTN4_MSK
```

This section defines the device name and LED/button configurations.

3. UART Configuration

```c
#define UART_BUF_SIZE CONFIG_BT_NUS_UART_BUFFER_SIZE
#define UART_WAIT_FOR_BUF_DELAY K_MSEC(50)
#define UART_WAIT_FOR_RX CONFIG_BT_NUS_UART_RX_WAIT_TIME

static K_SEM_DEFINE(ble_init_ok, 0, 1);

static struct bt_conn *current_conn;
static struct bt_conn *auth_conn;

static const struct device *uart = DEVICE_DT_GET(DT_CHOSEN(nordic_nus_uart));
static struct k_work_delayable uart_work;

struct uart_data_t {
    void *fifo_reserved;
    uint8_t data[UART_BUF_SIZE];
    uint16_t len;
};

static K_FIFO_DEFINE(fifo_uart_tx_data);
static K_FIFO_DEFINE(fifo_uart_rx_data);

```

This section configures the UART, including buffer sizes, delays, and defines a semaphore for Bluetooth initialization. It also defines structures and FIFOs for UART data.

4. Bluetooth Advertising Data

```c
static const struct bt_data ad[] = {
    BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
    BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};

static const struct bt_data sd[] = {
    BT_DATA_BYTES(BT_DATA_UUID128_ALL, BT_UUID_NUS_VAL),
};
```

This section sets up the advertising and scan response data for the Bluetooth NUS service.

5. ADC Configuration for Joystick

```c
#define DT_SPEC_AND_COMMA(node_id, prop, idx) \
    ADC_DT_SPEC_GET_BY_IDX(node_id, idx),

/* Data of ADC io-channels specified in devicetree. */
static const struct adc_dt_spec adc_channels[] = {
    DT_FOREACH_PROP_ELEM(DT_PATH(zephyr_user), io_channels,
                 DT_SPEC_AND_COMMA)
};

// add for joystick
int32_t preX = 0 , perY = 0;
static const int ADC_MAX = 1023;
// static const int ADC_MIN = 0;
static const int AXIS_DEVIATION = ADC_MAX / 2;
int32_t nowX = 0;
int32_t nowY = 0;

bool isChange(void)
{
    if((nowX < (preX - 50)) || nowX > (preX+50)){
        preX = nowX;
        return true;
    }

    if((nowY < (perY - 50)) || nowY > (perY+50)){
        perY = nowY;
        return true;
    }
    return false;
}
```

6. UART Callback and Initialization

```c
static void uart_cb(const struct device *dev, struct uart_event *evt, void *user_data)
{
  ARG_UNUSED(dev);

  static size_t aborted_len;
  struct uart_data_t *buf;
  static uint8_t *aborted_buf;
  static bool disable_req;
```

```
switch (evt->type) {
case UART_TX_DONE:
    LOG_DBG("UART_TX_DONE");
    if ((evt->data.tx.len == 0) ||
        (!evt->data.tx.buf)) {
        return;
    }

    if (aborted_buf) {
        buf = CONTAINER_OF(aborted_buf, struct uart_data_t,
                   data);
        aborted_buf = NULL;
        aborted_len = 0;
    } else {
        buf = CONTAINER_OF(evt->data.tx.buf, struct uart_data_t,
                   data);
    }

    k_free(buf);

    buf = k_fifo_get(&fifo_uart_tx_data, K_NO_WAIT);
    if (!buf) {
        return;
    }

    if (uart_tx(uart, buf->data, buf->len, SYS_FOREVER_MS)) {
        LOG_WRN("Failed to send data over UART");
    }

    break;

case UART_RX_RDY:
    LOG_DBG("UART_RX_RDY");
    buf = CONTAINER_OF(evt->data.rx.buf, struct uart_data_t, data);
    buf->len += evt->data.rx.len;

    if (disable_req) {
        return;
    }

    if ((evt->data.rx.buf[buf->len - 1] == '\n') ||
        (evt->data.rx.buf[buf->len - 1] == '\r')) {
        disable_req = true;
        uart_rx_disable(uart);
    }

    break;

case UART_RX_DISABLED:
    LOG_DBG("UART_RX_DISABLED");
    disable_req = false;

    buf = k_malloc(sizeof(*buf));
    if (buf) {
        buf->len = 0;
```

```
        } else {
            LOG_WRN("Not able to allocate UART receive buffer");
            k_work_reschedule(&uart_work, UART_WAIT_FOR_BUF_DELAY);
            return;
        }

        uart_rx_enable(uart, buf->data, sizeof(buf->data),
                       UART_WAIT_FOR_RX);

        break;

    case UART_RX_BUF_REQUEST:
        LOG_DBG("UART_RX_BUF_REQUEST");
        buf = k_malloc(sizeof(*buf));
        if (buf) {
            buf->len = 0;
            uart_rx_buf_rsp(uart, buf->data, sizeof(buf->data));
        } else {
            LOG_WRN("Not able to allocate UART receive buffer");
        }

        break;

    case UART_RX_BUF_RELEASED:
        LOG_DBG("UART_RX_BUF_RELEASED");
        buf = CONTAINER_OF(evt->data.rx_buf.buf, struct uart_data_t,
                           data);

        if (buf->len > 0) {
            k_fifo_put(&fifo_uart_rx_data, buf);
        } else {
            k_free(buf);
        }

        break;

    case UART_TX_ABORTED:
        LOG_DBG("UART_TX_ABORTED");
        if (!aborted_buf) {
            aborted_buf = (uint8_t *)evt->data.tx.buf;
        }

        aborted_len += evt->data.tx.len;
        buf = CONTAINER_OF(aborted_buf, struct uart_data_t,
                           data);

        uart_tx(uart, &buf->data[aborted_len],
            buf->len - aborted_len, SYS_FOREVER_MS);

        break;

    default:
        break;
    }
}
```

```
static void uart_work_handler(struct k_work *item)
{
    struct uart_data_t *buf;

    buf = k_malloc(sizeof(*buf));
    if (buf) {
        buf->len = 0;
    } else {
        LOG_WRN("Not able to allocate UART receive buffer");
        k_work_reschedule(&uart_work, UART_WAIT_FOR_BUF_DELAY);
        return;
    }

    uart_rx_enable(uart, buf->data, sizeof(buf->data), UART_WAIT_FOR_RX);
}

static int uart_init(void)
{
    int err;
    int pos;
    struct uart_data_t *rx;
    struct uart_data_t *tx;

    if (!device_is_ready(uart)) {
        return -ENODEV;
    }

    if (IS_ENABLED(CONFIG_USB_DEVICE_STACK)) {
        err = usb_enable(NULL);
        if (err && (err != -EALREADY)) {
            LOG_ERR("Failed to enable USB");
            return err;
        }
    }

    rx = k_malloc(sizeof(*rx));
    if (rx) {
        rx->len = 0;
    } else {
        return -ENOMEM;
    }

    k_work_init_delayable(&uart_work, uart_work_handler);


    err = uart_callback_set(uart, uart_cb, NULL);
    if (err) {
        k_free(rx);
        LOG_ERR("Cannot initialize UART callback");
        return err;
    }

    if (IS_ENABLED(CONFIG_UART_LINE_CTRL)) {
        LOG_INF("Wait for DTR");
```

```
    while (true) {
        uint32_t dtr = 0;

        uart_line_ctrl_get(uart, UART_LINE_CTRL_DTR, &dtr);
        if (dtr) {
            break;
        }
        /* Give CPU resources to low priority threads. */
        k_sleep(K_MSEC(100));
    }
    LOG_INF("DTR set");
    err = uart_line_ctrl_set(uart, UART_LINE_CTRL_DCD, 1);
    if (err) {
        LOG_WRN("Failed to set DCD, ret code %d", err);
    }
    err = uart_line_ctrl_set(uart, UART_LINE_CTRL_DSR, 1);
    if (err) {
        LOG_WRN("Failed to set DSR, ret code %d", err);
    }
}

tx = k_malloc(sizeof(*tx));

if (tx) {
    pos = snprintf(tx->data, sizeof(tx->data),
              "Starting Nordic UART service example\r\n");

    if ((pos < 0) || (pos >= sizeof(tx->data))) {
        k_free(rx);
        k_free(tx);
        LOG_ERR("snprintf returned %d", pos);
        return -ENOMEM;
    }

    tx->len = pos;
} else {
    k_free(rx);
    return -ENOMEM;
}

err = uart_tx(uart, tx->data, tx->len, SYS_FOREVER_MS);
if (err) {
    k_free(rx);
    k_free(tx);
    LOG_ERR("Cannot display welcome message (err: %d)", err);
    return err;
}

err = uart_rx_enable(uart, rx->data, sizeof(rx->data), 50);
if (err) {
    LOG_ERR("Cannot enable uart reception (err: %d)", err);
    /* Free the rx buffer only because the tx buffer will be handled in the callback */
    k_free(rx);
}
```

```
    return err;
}
```

This section defines the UART callback function to handle various UART events and defines the UART work handler and the UART initialization function.

7. Bluetooth Connection Callbacks

```c
static void connected(struct bt_conn *conn, uint8_t err)
{
    char addr[BT_ADDR_LE_STR_LEN];

    if (err) {
        LOG_ERR("Connection failed (err %u)", err);
        return;
    }

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
    LOG_INF("Connected %s", addr);

    current_conn = bt_conn_ref(conn);

    dk_set_led_on(CON_STATUS_LED);
}

static void disconnected(struct bt_conn *conn, uint8_t reason)
{
    char addr[BT_ADDR_LE_STR_LEN];

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));

    LOG_INF("Disconnected: %s (reason %u)", addr, reason);

    if (auth_conn) {
        bt_conn_unref(auth_conn);
        auth_conn = NULL;
    }

    if (current_conn) {
        bt_conn_unref(current_conn);
        current_conn = NULL;
        dk_set_led_off(CON_STATUS_LED);
    }
}


BT_CONN_CB_DEFINE(conn_callbacks) = {
    .connected    = connected,
    .disconnected = disconnected
};
```

This section defines the Bluetooth connection and disconnection callbacks and registers the Bluetooth connection callbacks.

8. NUS Callbacks

```c
static void bt_receive_cb(struct bt_conn *conn, const uint8_t *const data,
              uint16_t len)
{
    int err;
    char addr[BT_ADDR_LE_STR_LEN] = {0};

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, ARRAY_SIZE(addr));

    LOG_INF("Received data from: %s", addr);

    for (uint16_t pos = 0; pos != len;) {
        struct uart_data_t *tx = k_malloc(sizeof(*tx));

        if (!tx) {
            LOG_WRN("Not able to allocate UART send data buffer");
            return;
        }

        /* Keep the last byte of TX buffer for potential LF char. */
        size_t tx_data_size = sizeof(tx->data) - 1;

        if ((len - pos) > tx_data_size) {
            tx->len = tx_data_size;
        } else {
            tx->len = (len - pos);
        }

        memcpy(tx->data, &data[pos], tx->len);

        pos += tx->len;

        /* Append the LF character when the CR character triggered
         * transmission from the peer.
         */
        if ((pos == len) && (data[len - 1] == '\r')) {
            tx->data[tx->len] = '\n';
            tx->len++;
        }

        err = uart_tx(uart, tx->data, tx->len, SYS_FOREVER_MS);
        if (err) {
            k_fifo_put(&fifo_uart_tx_data, tx);
        }
    }
}

static struct bt_nus_cb nus_cb = {
    .received = bt_receive_cb,
};
```

This section defines the NUS receive callback and initializes the NUS callback structure.

9. GPIO configuration and Callback

```c
void button_changed(uint32_t button_state, uint32_t has_changed)
{
    uint32_t buttons = button_state & has_changed;

    if (buttons & BUTTON1) {
        char buf[7] = "Button1\n";
        bt_nus_send(NULL, buf, sizeof(buf));
    }

    if (buttons & BUTTON2) {
        char buf[7] = "Button2\n";
        bt_nus_send(NULL, buf, sizeof(buf));
    }

    if (buttons & BUTTON3) {
        char buf[7] = "Button3\n";
        bt_nus_send(NULL, buf, sizeof(buf));
    }

    if (buttons & BUTTON4) {
        char buf[7] = "Button4\n";
        bt_nus_send(NULL, buf, sizeof(buf));
    }

}

static void configure_gpio(void)
{
    int err;

    err = dk_buttons_init(button_changed);
    if (err) {
        LOG_ERR("Cannot init buttons (err: %d)", err);
    }

    err = dk_leds_init();
    if (err) {
        LOG_ERR("Cannot init LEDs (err: %d)", err);
    }
}
```

This section sets up the necessary GPIO configuration for buttons and LEDs. It ensures that any button press will trigger the callback. When triggered by a button press, checks which button was pressed and sends a corresponding message over Bluetooth using the Nordic UART Service (NUS).

10. Main Function

```c
int blink_status = 0;
    int err = 0;

    configure_gpio();

    err = uart_init();
    if (err) {
        error();
    }

    err = bt_enable(NULL);
    if (err) {
        error();
    }

    LOG_INF("Bluetooth initialized");

    k_sem_give(&ble_init_ok);

    if (IS_ENABLED(CONFIG_SETTINGS)) {
        settings_load();
    }

    err = bt_nus_init(&nus_cb);
    if (err) {
        LOG_ERR("Failed to initialize UART service (err: %d)", err);
        return 0;
    }

    err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), sd,
                  ARRAY_SIZE(sd));
    if (err) {
        LOG_ERR("Advertising failed to start (err %d)", err);
        return 0;
    }

    //led
    uint32_t count = 0;
    uint16_t bufled;
    struct adc_sequence sequence = {
     .buffer = &bufled,
     /* buffer size in bytes, not number of samples */
     .buffer_size = sizeof(bufled),
    };

    /* Configure channels individually prior to sampling. */
    for (size_t i = 0U; i < ARRAY_SIZE(adc_channels); i++) {
     if (!adc_is_ready_dt(&adc_channels[i])) {
         printk("ADC controller device %s not ready\n", adc_channels[i].dev->name);
         return 0;
     }

     err = adc_channel_setup_dt(&adc_channels[i]);
     if (err < 0) {
         printk("Could not setup channel #%d (%d)\n", i, err);
         return 0;
     }
    }

    led_init();
    led_on_right();

    char right[5] = "Right"; //TODO: Need to change
    char left[4] = "Left";
    char down[4] = "Down";
    char up[2] = "Up";
```

```c
for (;;) {

        printk("ADC reading[%u]: ", count++);

        (void)adc_sequence_init_dt(&adc_channels[0], &sequence);
        err = adc_read(adc_channels[0].dev, &sequence);
        if (err < 0) {
         printk("Could not read (%d)\n", err);
         continue;
        }

        nowX = (int32_t)bufled;

        (void)adc_sequence_init_dt(&adc_channels[1], &sequence);
        err = adc_read(adc_channels[1].dev, &sequence);
        if (err < 0) {
         printk("Could not read (%d)\n", err);
         continue;
        }

        nowY = (int32_t)bufled;

        if (nowX >= 65500 || nowY >= 65500){
         k_sleep(K_MSEC(50));
         continue;
        }

        bool checkFlag = isChange();
        if(!checkFlag){
            k_sleep(K_MSEC(50));
         continue;
        } else {
         led_off_all();
        }

        if (nowX == ADC_MAX && nowY == ADC_MAX){
         led_on_center();
         printk("Center");
        } else if (nowX < AXIS_DEVIATION && nowY == ADC_MAX){
         led_on_left();
         bt_nus_send(NULL, left, sizeof(left));
        } else if (nowX > AXIS_DEVIATION && nowY == ADC_MAX){
         led_on_right();
         bt_nus_send(NULL, right, sizeof(right));
        } else if (nowY > AXIS_DEVIATION && nowX == ADC_MAX){
         led_on_up();
         bt_nus_send(NULL, up, sizeof(up));
        } else if (nowY < AXIS_DEVIATION && nowX == ADC_MAX){
         led_on_down();
         bt_nus_send(NULL, down, sizeof(down));
        }

        printk("\n");

        // k_sleep(K_MSEC(100));

        dk_set_led(RUN_STATUS_LED, (++blink_status) % 2);
        k_sleep(K_MSEC(50));
        // bt_nus_send(NULL, buf, 20);
    }
```

Firstly, It initializes required functions, such as UART and Nordic UART Service (NUS). Also, it sets up an ADC sequence, configures each ADC channel, checks if each is ready, and sets it up. If any channel isn't ready or fails to set up, it logs an error and exits. Also, it Initializes LEDs and turns on the right LED as an initial state.

Then, starts Bluetooth advertising with the specified parameters. If it fails, it logs an error and exits.

Continuously reads ADC values and processes. Reads the X and Y values from the ADC channels. Skips the current iteration if the read fails or if the values are too high. Uses the `isChange` function to check if there's a significant change in direction. If there's no change, it sleeps and continues.Depending on the values of `nowX` and `nowY`, it updates the LEDs and sends the corresponding direction message over Bluetooth. Toggles an LED to indicate the system is running and then sleeps for 50 milliseconds before the next iteration.

## Device Tree

```
&adc {
        #address-cells = <1>;
        #size-cells = <0>;

        channel@1 {
        reg = <1>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
        zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
        zephyr,input-positive = <NRF_SAADC_AIN1>; /* P0.03 */
        zephyr,resolution = <10>;
        // zephyr,differential;
        };

        channel@2 {
        reg = <2>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
        zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
        zephyr,input-positive = <NRF_SAADC_AIN2>; /* P0.04 */
        zephyr,resolution = <10>;
        // zephyr,differential;
        };
};
```

The DeviceTree overlay file provided is designed for a Zephyr-based project to configure various hardware peripherals, specifically ADC channels and an I2C device. At the root level, the file defines a `zephyr,user` node that specifies the use of two ADC channels.

These channels are detailed under the `&adc` node, where each channel is configured with specific properties: channel 1 and channel 2 use an internal reference voltage, a gain setting of `ADC_GAIN_1`, a default acquisition time, and a resolution of 10 bits. The positive inputs for these channels are connected to pins P0.03 and P0.04, respectively.

Additionally, the file configures an I2C bus under the `&i2c0` node, setting its clock frequency to the standard I2C rate. A device with the address 0x70, compatible with the Holtek HT16K33 driver, is defined on this bus. This device node includes a potential configuration for an interrupt line, although it is currently commented out, and a sub-node for key scanning functionality.

Furthermore, the overlay sets up a chosen node to define UART0 as the default interface for the Nordic UART Service (NUS), facilitating communication over Bluetooth. This comprehensive configuration ensures that the Zephyr operating system can properly initialize and manage these peripherals, enabling the application to utilize analog inputs and I2C communication effectively.

## prj.conf

```
#
# Copyright (c) 2021 Nordic Semiconductor
#
# SPDX-License-Identifier: LicenseRef-Nordic-5-Clause
#

# Enable the UART driver
CONFIG_UART_ASYNC_API=y
CONFIG_NRFX_UARTE0=y
CONFIG_SERIAL=y

CONFIG_HEAP_MEM_POOL_SIZE=2048

CONFIG_BT=y
CONFIG_BT_PERIPHERAL=y
CONFIG_BT_DEVICE_NAME="Jang" #TODO: Edit according to your device
CONFIG_BT_DEVICE_APPEARANCE=833
CONFIG_BT_MAX_CONN=1
CONFIG_BT_MAX_PAIRED=1

# Enable the NUS service
CONFIG_BT_NUS=y

# Enable bonding
```

```
CONFIG_BT_SETTINGS=y
CONFIG_FLASH=y
CONFIG_FLASH_PAGE_LAYOUT=y
CONFIG_FLASH_MAP=y
CONFIG_NVS=y
CONFIG_SETTINGS=y

# Enable DK LED and Buttons library
CONFIG_DK_LIBRARY=y

# Drivers and peripherals
# CONFIG_I2C=n
CONFIG_WATCHDOG=n
CONFIG_SPI=n
CONFIG_GPIO=n

CONFIG_ADC=y
# CONFIG_ADC_ASYNC=y

# CONFIG_LOG=y
CONFIG_I2C=y
CONFIG_LED=y
CONFIG_KSCAN=y
CONFIG_KSCAN_INIT_PRIORITY=95
CONFIG_HT16K33_KEYSCAN=y

# Power management

# Interrupts
CONFIG_DYNAMIC_INTERRUPTS=n
CONFIG_IRQ_OFFLOAD=n

# Memory protection
CONFIG_THREAD_STACK_INFO=n
CONFIG_THREAD_CUSTOM_DATA=n
CONFIG_FPU=n

# Boot
CONFIG_BOOT_BANNER=n
CONFIG_BOOT_DELAY=0

# Console
CONFIG_CONSOLE=n
```

```
CONFIG_UART_CONSOLE=n
CONFIG_STDOUT_CONSOLE=n
CONFIG_PRINTK=n
CONFIG_EARLY_CONSOLE=n

CONFIG_SIZE_OPTIMIZATIONS=y

CONFIG_ARM_MPU=n

CONFIG_BT_RX_STACK_SIZE=1024
CONFIG_BT_HCI_TX_STACK_SIZE_WITH_PROMPT=y
CONFIG_BT_HCI_TX_STACK_SIZE=640
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE=512
CONFIG_MPSL_WORK_STACK_SIZE=256
CONFIG_MAIN_STACK_SIZE=864
CONFIG_IDLE_STACK_SIZE=128
CONFIG_ISR_STACK_SIZE=1024
CONFIG_BT_NUS_THREAD_STACK_SIZE=512

# Disable features not needed
CONFIG_TIMESLICING=n
CONFIG_MINIMAL_LIBC_MALLOC=n
# CONFIG_LOG=n
CONFIG_ASSERT=n

# Disable Bluetooth features not needed
CONFIG_BT_DEBUG_NONE=y
CONFIG_BT_ASSERT=n
CONFIG_BT_DATA_LEN_UPDATE=n
CONFIG_BT_PHY_UPDATE=n
CONFIG_BT_GATT_CACHING=n
CONFIG_BT_GATT_SERVICE_CHANGED=n
CONFIG_BT_GAP_PERIPHERAL_PREF_PARAMS=n
CONFIG_BT_SETTINGS_CCC_LAZY_LOADING=y
CONFIG_BT_HCI_VS_EXT=n

# Disable Bluetooth controller features not needed
CONFIG_BT_CTLR_PRIVACY=n
CONFIG_BT_CTLR_PHY_2M=n

# Reduce Bluetooth buffers
CONFIG_BT_BUF_EVT_DISCARDABLE_COUNT=1
CONFIG_BT_BUF_EVT_DISCARDABLE_SIZE=43
```

```
CONFIG_BT_BUF_EVT_RX_COUNT=2

CONFIG_BT_CONN_TX_MAX=2
CONFIG_BT_L2CAP_TX_BUF_COUNT=2
CONFIG_BT_CTLR_RX_BUFFERS=1
CONFIG_BT_BUF_ACL_TX_COUNT=3
CONFIG_BT_BUF_ACL_TX_SIZE=27
```

It enables the UART driver with asynchronous API support and allocates a 2048-byte heap memory pool. Bluetooth functionalities are configured, including enabling the Bluetooth stack, setting the device as a peripheral, and specifying a device name. The Nordic UART Service (NUS) is activated for Bluetooth communication, and bonding settings are enabled to support device pairing. The configuration also includes enabling the ADC for analog-to-digital conversions, as well as the I2C interface for communication with peripheral devices like the HT16K33 LED driver and key scanner. Several Bluetooth-specific parameters are adjusted to optimize performance and memory usage, such as reducing Bluetooth buffer sizes and disabling unnecessary features. Additionally, certain console and logging features are disabled to save resources, and stack sizes for various threads are configured to ensure the efficient operation of the system.

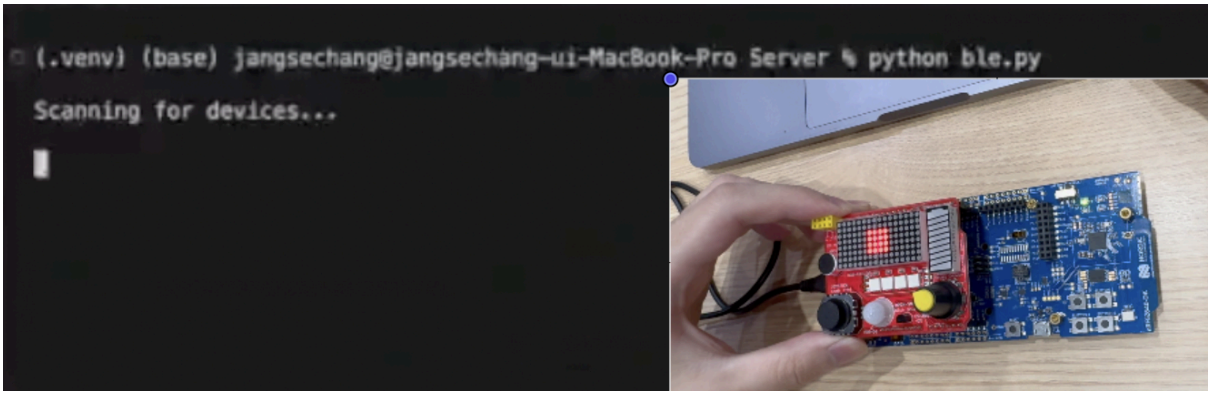
# Results

https://youtu.be/TSpOgR_slu8(Full game play)

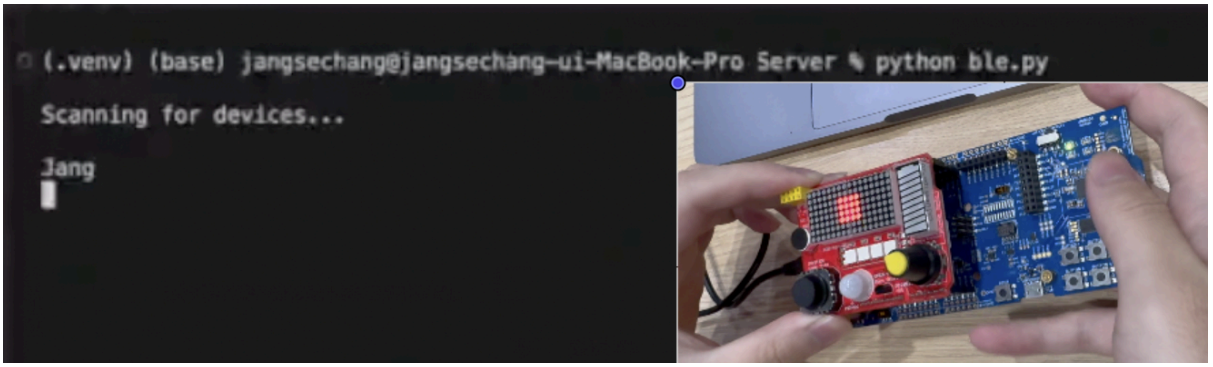
## Bluetooth Connection

https://youtu.be/hXV3-fMyu50

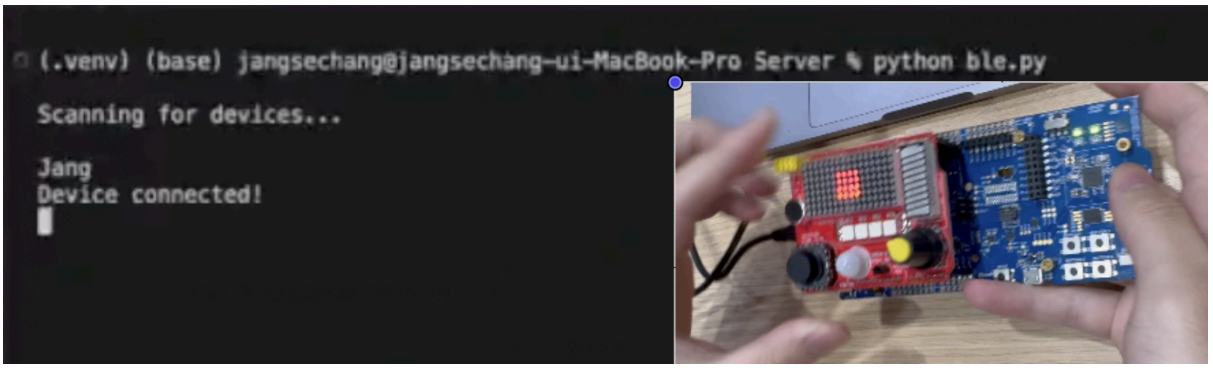
1. Scanning Device




2. Find Device(Device name : Jang)

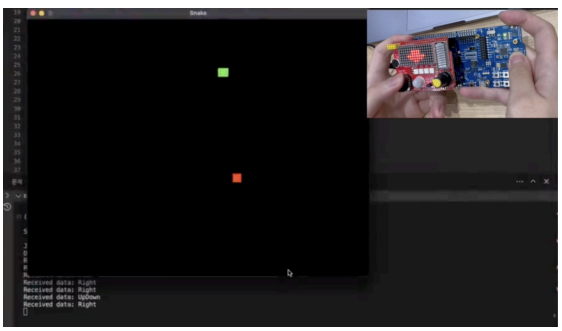
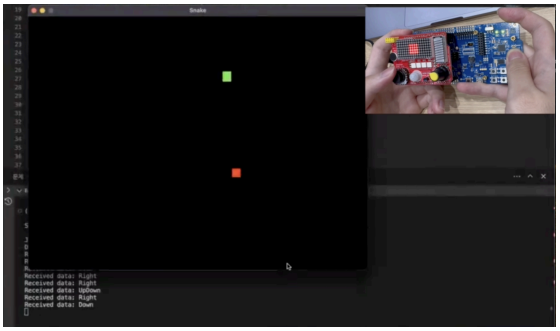



3. Device connect



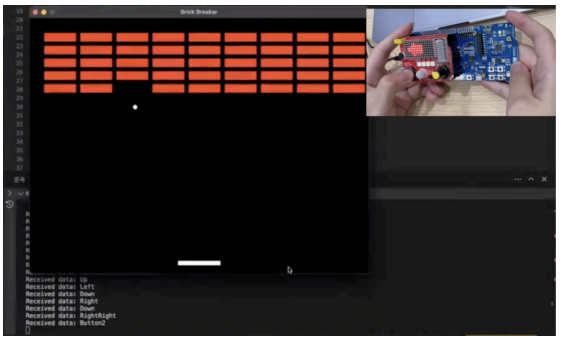
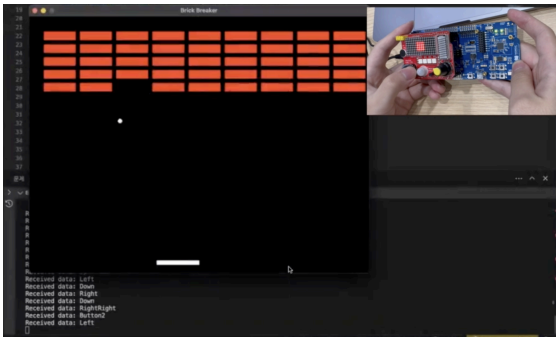
# Game Control

https://youtu.be/17VQqYvpkr4 (Game1)

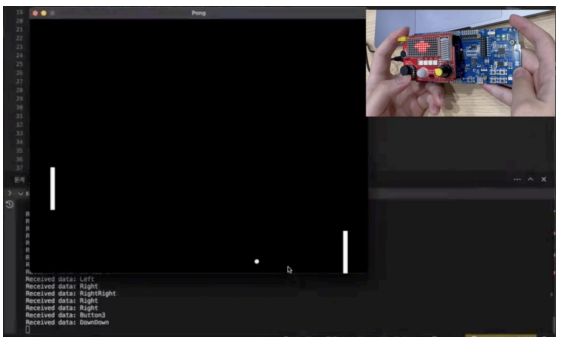
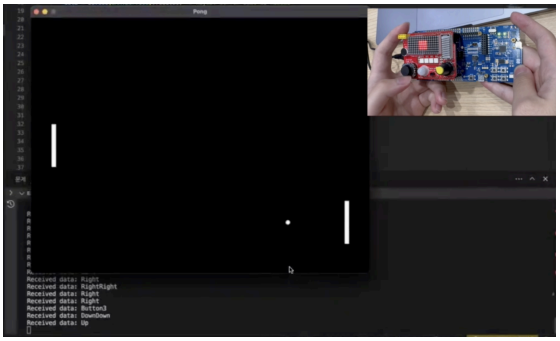



(In the console, you can see the direction)

https://youtu.be/LJATyglQOHk (Game2)

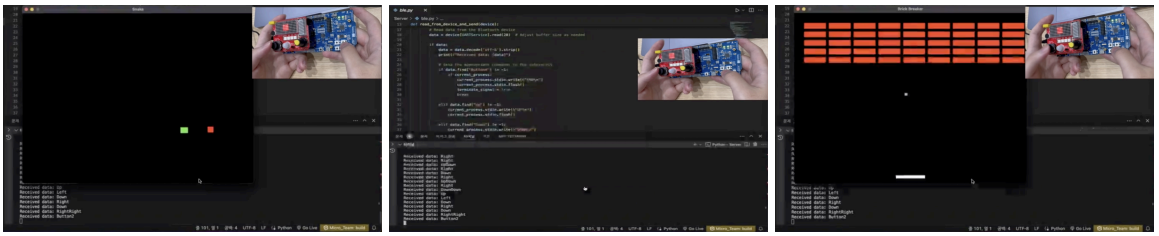



https://youtu.be/HIMJG0C0rP8 (Game3)

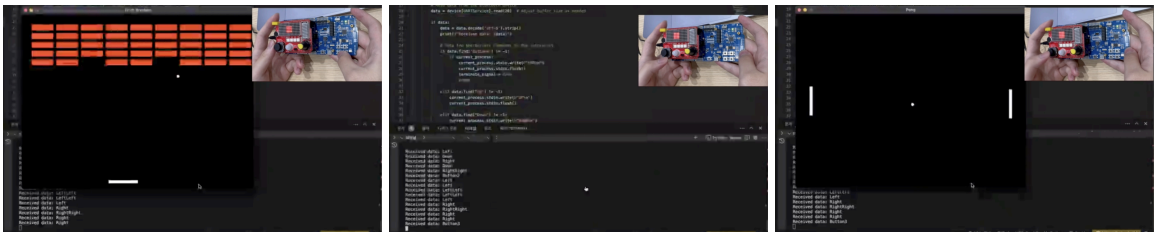



# Game Change

https://youtu.be/j3uJ-aVI5AM (Button2 - Game2)

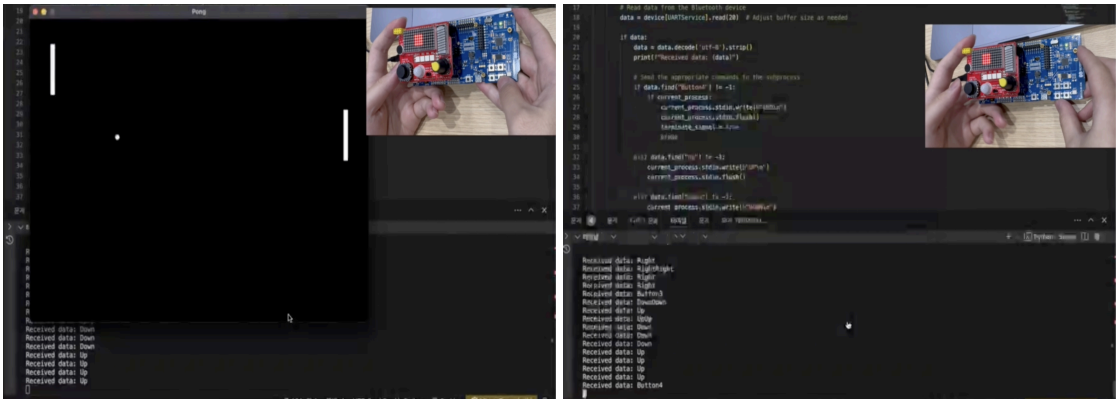


https://youtu.be/NJyc4rmjqtA (Button3 - Game3)



# Game Terminate

https://youtu.be/W_EHR-U7Vco (Button4 - Terminate)



# Discussion

## Usefulness of Product

### Wireless Game Pad

Wireless gamepads offer numerous benefits that significantly enhance the gaming experience compared to their wired counterparts. They provide freedom of movement by eliminating the constraints of cables, allowing gamers to play from various positions without worrying about tangling or distance limitations.

Wireless gamepads contribute to a cleaner gaming setup by reducing clutter. Without cables, gaming environments appear more organized and visually appealing, promoting a streamlined aesthetic. This aspect is complemented by the convenience of easy setup and use—there's no need to connect or manage cables, simplifying both initial setup and storage.

### Easy of Making Game

It is often challenging to develop a game that runs exclusively on a specific embedded system due to hardware limitations and the need for specialized programming. However, by focusing on creating the gamepad itself, we can strategically divide the game development process into manageable parts. The gamepad serves as the user interface and input device, while the game logic and rendering are handled by a separate server or computing system.

This approach not only simplifies the complexity of game development on the embedded system but also allows for leveraging more powerful computing resources for handling game mechanics, AI, and graphics rendering. By decoupling the gamepad from the intensive computational tasks, developers can focus on optimizing user interaction and ensuring seamless connectivity and responsiveness between the gamepad and the server. This division of labor enhances the overall flexibility and scalability of game development, enabling the creation of sophisticated and engaging gaming experiences across various platforms and devices.

### User Friendly UI/UX

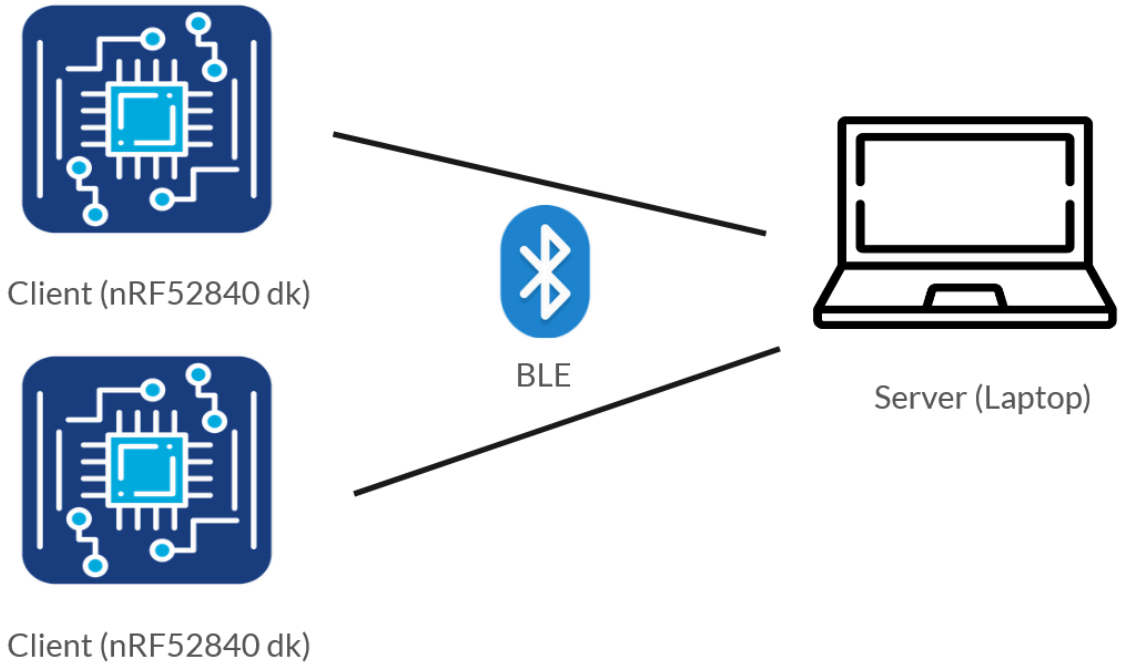
The nRF52840 DK, functioning as a game-playing device with its integrated LED, encounters several limitations. Primarily, the LED's single-color capability poses challenges in distinguishing objects and enhancing visual clarity during gameplay. Moreover, the LED's brightness, though initially eye-catching, can be excessively intense, potentially causing discomfort during prolonged gaming sessions and leading to eye strain and fatigue over time. Additionally, the LED matrix screen's size is inadequate for effectively supporting a diverse range of games, limiting the immersive experience for users.

To address these shortcomings, utilizing a laptop as the gameplay screen offers significant advantages in terms of user interface (UI) and user experience (UX). Laptops typically provide larger and higher-resolution displays compared to the nRF52840 DK's LED matrix, thereby enhancing visual detail and gameplay immersion. The larger screen real estate allows for more elaborate game designs, better graphic rendering, and improved interaction elements, contributing to a more user-friendly and engaging gaming experience. By leveraging the laptop's capabilities, developers can optimize game graphics, implement complex game mechanics, and deliver seamless gameplay across various genres and platforms.

# Possible Bigger Application

## Multiplayer Game

By adopting a server-client architecture, we enabled the connection of multiple nRF52840 devices to a central server, such as a laptop, allowing for multiplayer gaming experiences. This setup facilitates seamless communication and coordination between multiple gamepads, enhancing the potential for interactive and competitive gameplay among multiple players.




<Figure 5. Client-server Architecture with multiple clients>

## Wide Range of Connected Devices

We could make wireless gamepads compatible with a wide range of devices including PCs, gaming consoles, and mobile devices. Wireless gamepads that are compatible with a wide range of devices including PCs, gaming consoles, and mobile devices offer significant utility and convenience to gamers.

This compatibility ensures that users can enjoy their favorite games across different platforms without needing separate controllers for each device. For instance, gamers can seamlessly transition from playing on a PC to a gaming console or mobile device, using the same wireless gamepad. This versatility eliminates the need to invest in multiple controllers, thereby reducing costs and clutter in the gaming setup. Moreover, it enhances accessibility, allowing gamers to choose their preferred gaming platform without restrictions imposed by controller compatibility.

## Sophisticated Joystick Control Algorithm - Game Genre Expansion

Now it is working as a discrete joystick action. However, if we could elaborate an algorithm that can control objects sophisticatedly, we can provide a better user experience. Also, we can make this gamepad suitable for FPS.

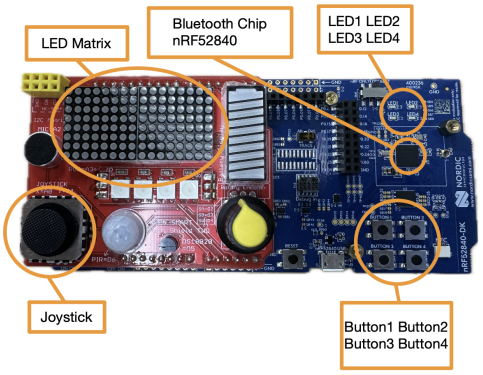
By refining the control algorithm, the gamepad can offer precise and responsive movement, aiming, and interaction within games. This improvement is crucial for FPS games where accuracy and speed of control directly impact gameplay and player performance. Implementing such advanced algorithms not only boosts gameplay realism but also expands the gamepad's versatility to support a broader range of game genres beyond traditional joystick actions. This enhancement enables gamers to enjoy immersive gaming experiences across different game types, thus making the wireless gamepad a versatile and preferred choice for diverse gaming needs.

# Conclusion

In the project, We initially thought of implementing a simple gamepad using LED Matrix. However, LED Matrix has

- LED has only one color
- LED brightness is intense
- LED matrix screen is too small

these shortcomings, We proposed a more advanced type of project.
In the project, we use




- LED 1: Indicate Action of Joystick
- LED2: Indicate Bluetooth Connection
- Buttons: Control Game Process
    - Button1 : Game1
    - Button2 : Game2
    - Button3 : Game3
    - Button4 : Terminate the game
- Joystick: Move Object in Game
- LED Matrix: Indicate Direction of Movement

And for the implementation of the project, we focused on

- System Architecture(Server-Client architecture)
    - Server : Laptop
    - Client : nRF52840 development kits
- Design and Implementation of Server part(Central)
- Design and Implementation of Client part(Peripheral)
- Design and Implementation of Game1,2,3
    - Game1 : Snake game
    - Game2 : Blicker game
    - Game3 : Ping Pong game

The advantage of the product are

- Wireless Game Pad
- Easy of Making Game
- User Friendly UI/UX

and, the expanded projects that can be presented are

- Multiplayer Game
- Wide Range of Connected Devices
- Sophisticated Joystick Control Algorithm

The advantages of our product extend beyond the basic functionality of a wireless gamepad. Unlike other projects that rely solely on the LED Matrix of the nRF52840 board, our design emphasizes scalability and flexibility. This approach opens up a range of possibilities for future developments and applications.

# Individual Contribution (각 조원의 역할)

## Sanghwa Han (21900783)

First, our team focused on connecting the computer to the nrf52840 board. It was my first time doing it, so I aimed to do it at the same time without division of roles.

And after successfully connecting the computer and the nrf52840 board, he focused on designing the server-client system architecture. The possibility of expansion, such as multiplayer and player-to-player confrontation, was considered, and this design was very important, and the server-client system architecture was successful with this in mind.

Finally, from this point on, the partner focused on the implementation of stable Bluetooth communication between the server and client, and the LED, button, joystick and I2C Matrix connections in the nrf52840 board. And I focused on building servers using thread and implementing Game1, 2, 3 source code.

## Sechang Jang (21900628)

Firstly, my primary focus was on meticulously designing the system architecture to ensure that our wireless gamepad was integrated within a client-server framework. This involved conceptualizing and implementing the structural layout and communication protocols necessary for optimal functionality across various gaming platforms.

Secondly, I dedicated considerable effort to establishing a robust Bluetooth connection between our gamepad (acting as the client) and the laptop (functioning as the server). This task was pivotal in enhancing the overall performance and user experience of our gamepad.

Additionally, I played a key role in configuring essential hardware components such as LEDs, buttons, a joystick, and the I2C Matrix. This involved setting up and calibrating these components to ensure they operated seamlessly with our gamepad system. By configuring these hardware elements, I contributed to creating a user-friendly and responsive interface that enhanced the gameplay experience.

# Discussion (조원 각각 작성)

## Sanghwa Han (21900783)

While working on the project, I studied passively mainly through PPT and class-related books, but this project was more meaningful because it seemed to have been carried out by reading official documents, performing tests directly, and experiencing many challenges and failures.

If I have a chance later, I would like to attach many sensors together to manage and implement interrupts, and implement stable mesh communication through Bluetooth.

It is a busy schedule while working on the project as a team, but it was very meaningful that we studied each other while reading official documents and I hope that we can work together through other projects next time.

## Sechang Jang (21900628)

During the project, I had a hard time finding good sources that explained how to connect Bluetooth to the laptop. Learning about Bluetooth Low Energy (BLE) was interesting and fun. It was cool to see how BLE works and understand all the details about how devices communicate wirelessly.

If I had more time, I would have focused on making the nRF52840 DK board work even better. This means tweaking its settings to make it have better-optimized architecture and code and perform more efficiently, so our wireless gamepad could be even better.

Working with my teammates was great. We all brought different skills to the project, and together we solved problems and came up with new ideas. I enjoyed collaborating with them and look forward to working together again on future projects.

# Reference

[1] https://www.statista.com/outlook/amo/media/games/worldwide
[2] https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK/GetStarted - nRF52840 DK
[3] https://www.arduinolearning.com/hardware/a-new-shield-for-beginners-rich-shield-two.php
[4] https://www.avsystem.com/blog/linkyfi/bluetooth-low-energy-ble/
[5] https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.00.00.22/exports/docs/blestack/html/ble-stack/index.html
[6] https://docs.nordicsemi.com/bundle/ncs-2.5.0/page/nrf/index.html
[7] https://docs.zephyrproject.org/latest/introduction/index.html

# Appendix

## Manual

### Server

**Requirement:**

- Python 3.x

**Dependency:**

- Pygame: *pip install pygame*
- adafruit-circuitpython-ble: *pip install adafruit-circuitpython-ble*

- If you need to install in a virtual environment, then:

    - *mkdir project-name && cd project-name*
    - *python3 -m venv .venv*
    - *source .venv/bin/activate*
    - *pip3 install adafruit-circuitpython-ble*

**Execution:**

- *python ble.py*