ChatGPT

Week 1 Plan (Aug 11–17, 2025) – NinJoy Junior Project

Day 1 (Mon Aug 11) – Environment Setup & Initial Planning

Person A (Hardware & ESP32 Firmware):
- **Hardware Inventory & Phone Board Evaluation:** Gather all hardware (Samsung Galaxy S5, Galaxy J3 2016 motherboard, two ESP32 modules). Inspect the Galaxy J3 board for any obvious damage. Attempt to power the J3 board (using a compatible battery or bench power) to see if it boots. If it's not easily usable, decide to set it aside for now (potentially salvage connectors or battery if needed, otherwise plan to discard).
- **Arduino IDE Installation:** Download and install the latest Arduino IDE. In Arduino, install ESP32 board support via the Boards Manager:
- Open **File > Preferences** and add Espressif's JSON URL for board definitions (e.g. `https://dl.espressif.com/dl/package_esp32_index.json`).
- Open **Tools > Board > Boards Manager**, search for "ESP32" and install the **esp32 by Espressif Systems** package (this provides ESP32 device support in Arduino IDE).
- (If on Windows, install USB-UART drivers such as CP210x or CH340 if the ESP32 dev boards require them).
- **ESP32 Connectivity Test:** Connect each ESP32 dev board via USB and select the appropriate port in Arduino IDE. For each board:
- Select the **"ESP32 Dev Module"** (or specific board model) as the target board and a suitable upload speed.
- Load a simple **Blink** sketch or a "Hello, world" serial print sketch. For example, use the built-in LED (usually on GPIO2) and blink it:

```
pinMode(2, OUTPUT);
void loop() {
  digitalWrite(2, HIGH);
  delay(500);
  digitalWrite(2, LOW);
  delay(500);
}
```

- Upload to each ESP32 to ensure they program correctly. Open the Serial Monitor at 115200 baud to confirm output (if using a print statement).
- **Checkpoint:** Both ESP32 modules are recognized, can be flashed, and run the test code (LED blinks or serial message appears). This confirms the dev environment and cables are working.
- **GitHub Repo Initialization (Firmware):** Create a GitHub repository (e.g. **ninjoy-firmware**). Structure it with folders for each microcontroller: perhaps `controller-esp32` and `console-esp32`. Add a basic README outlining the project and hardware. Use `git init` locally and push the initial repo to GitHub. (Person A and B will both have access).
- **Planning Coordination:** Discuss with Person B the overall system architecture (ESP32 controller -> ESP-

NOW -> ESP32 console -> BLE HID -> Android phone). Ensure both have a shared understanding of data flow. No coding of BLE or game logic yet – just outline responsibilities and interfaces (e.g., what data packet format the ESP32 will send for button presses). Write down this plan in the README or a shared document for reference.

Person B (Android Launcher & Game Dev Environment):
- **Android Studio Setup:** Install the latest **Android Studio** along with required SDKs. Make sure the Java JDK is properly configured (Android Studio bundles one, but verify).
- **Galaxy S5 Developer Setup:** Enable **Developer Options** on the Galaxy S5 (tap Build Number 7 times) and turn on **USB Debugging**. Connect the phone via USB and verify that the PC recognizes it: run `adb devices` in a terminal – the device should show up. (If on Windows, install the Samsung USB driver if needed so that ADB works).
- **Project Creation – Custom Launcher:** Create a new Android Studio project for the custom launcher app (call it "NinJoyLauncher"). Use an **Empty Activity** template. In the AndroidManifest of this app, set up the launcher activity with the appropriate intent filters:

```
<activity android:name=".LauncherActivity" android:exported="true"
          android:label="NinJoy Launcher">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.HOME" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This declares the app as a Home/Launcher replacement. Confirm the `LauncherActivity` launches by default when the app is opened.
- **Launcher UI Skeleton:** Implement a very basic UI for the launcher – for now, maybe a full-screen TextView that says "**NinJoy Home**" and three placeholder buttons (or list items) for "Game 1", "Game 2", "Game 3". Keep it simple (e.g., vertical column of buttons). Ensure the launcher activity is set to use a stable orientation (likely landscape, since games will be in landscape).
- **Game Projects Setup:** Create three separate Android Studio projects (or modules) for the games: e.g., **NinJoyGame1**, **NinJoyGame2**, **NinJoyGame3**. Each should be a basic app with its own unique application ID (e.g., `com.ninjoy.game1`, `com.ninjoy.game2`, etc.). For each game:
- Use an Empty Activity (e.g., `Game1Activity`) and set the content view to a simple layout (perhaps just a TextView like "Game 1 – Coming Soon" as a placeholder).
- Set the orientation to landscape in the manifest if desired (using `android:screenOrientation="landscape"` for the activity).
- (No actual game logic yet – just scaffolding to ensure the apps run and can be launched.)
- **GitHub Repo Initialization (Android):** Create a GitHub repository (or multiple, depending on preference) for the Android code. For simplicity, you can use one repo **ninjoy-android** and organize it into subfolders or Android Studio modules for the launcher and each game. Alternatively, create separate repos (ninjoy-launcher, ninjoy-game1, etc.). Initialize Git, commit the new project files (with proper `.gitignore` for Android), and push to GitHub.
- **Testing on Device:** Build and install the NinJoyLauncher app on the Galaxy S5 using Android Studio (via ADB). After installation, press the Home button on the S5 – you should be prompted to choose a default

launcher. Select the NinJoy Launcher and **set it as default**. You should see your simple launcher UI appear. (At this point it's just a static UI).
- Also deploy **Game1**, **Game2**, and **Game3** apps to the phone. You can run each from Android Studio or use `adb install <apk>` for each game's APK. Ensure they install successfully (you'll see their icons maybe in the app drawer, and they should just show a placeholder screen when opened).
- Test the launcher's placeholders: Currently, the buttons don't do anything. That's okay for Day 1 – the goal is just to have all projects set up on the device and ensure the custom home replaces the default one.
- **Checkpoint:** By end of Day 1, development environments are ready. Person A can program microcontrollers, and Person B can build & deploy Android apps to the phone. The custom launcher is running on the S5 (with placeholder UI), and game apps are installed (showing placeholder screens). Repos are created on GitHub with initial code pushed.

Day 2 (Tue Aug 12) – Controller Wiring & Software Skeletons

Person A (Hardware & ESP32 Firmware):
- **Controller Wiring Setup:** Start assembling the **detachable controller** hardware. Decide on the control scheme (e.g., a D-pad plus a couple of buttons for A/B, maybe Start/Select). Use tactile pushbuttons for each input. Plan which ESP32 GPIO pins to use for each button: for example, GPIO 32 for Up, 33 for Down, 25 for Left, 26 for Right, 4 for A, 5 for B (ensure these pins are available on your ESP32 module and are input-capable).
- Wire each button: connect one leg of the button to the chosen ESP32 GPIO, and the other leg to ground. Enable the ESP32's internal pull-ups to simplify wiring (so no need for external resistors). This means in code you will use `pinMode(pin, INPUT_PULLUP)` and consider the button *pressed* when the reading is LOW (grounded).
- If an analog input (like a joystick thumbstick) is part of the design, wire its potentiometer outputs to analog-capable pins (e.g., GPIO 34, 35 for X and Y). Ensure you share a common ground between the joystick and ESP32.
- Double-check connections with a multimeter or continuity tester (to avoid shorts). Label the wires or document which pin is which control for clarity.
- **ESP-NOW Communication – Initial Test:** Begin coding the **ESP32 controller firmware** (Person A's focus) to send input data to the console ESP32 via ESP-NOW (a low-latency Wi-Fi direct protocol):
- In the controller's Arduino sketch, include the WiFi and ESP-NOW libraries:

```
#include <WiFi.h>
#include <esp_now.h>
```

In `setup()`, initialize Wi-Fi in station mode with `WiFi.mode(WIFI_STA)` (ESP-NOW requires Wi-Fi to be on). Then call `esp_now_init()` and check it returns `ESP_OK`.
- Determine the **peer address** (MAC address) of the console ESP32. You can get this by running `WiFi.macAddress()` on the console ESP32 and printing it, or using `Serial.println(WiFi.macAddress())` in a simple sketch on that board. It will output a MAC like `24:6F:28:...`. Note this down.
- On the controller, add the console as a peer for ESP-NOW:

```
uint8_t consoleAddress[] = {0x24, 0x6F, 0x28, 0xAA, 0xBB, 0xCC}; // example MAC
esp_now_peer_info_t peerInfo = {};
memcpy(peerInfo.peer_addr, consoleAddress, 6);
peerInfo.channel = 0;  // default channel
peerInfo.encrypt = false;
if (esp_now_add_peer(&peerInfo) != ESP_OK) {
    Serial.println("Failed to add peer");
}
```

(Use the actual MAC of the console ESP32 and same channel for both devices; by default, if not connected to WiFi, both might use channel 1. Setting `channel=0` lets ESP-NOW use current WiFi channel.)
- Define a data structure for the controller state to send. For example:

```
struct ControllerState {
  uint8_t buttons;  // bit mask: bit0=Up,1=Down,2=Left,3=Right,4=A,5=B, etc.
  int16_t joyX;     // analog joystick X (e.g., -512 to 512 or 0-1023)
  int16_t joyY;     // analog joystick Y
} state;
```

Populate `state` each loop by reading the GPIOs (`digitalRead(pin)`) and analogs (`analogRead(pin)` if applicable). Convert the button readings into bits of the `buttons` byte.
- Send the state via ESP-NOW: use `esp_now_send(consoleAddress, (uint8_t*)&state, sizeof(state))`. You can do this at a fixed interval (e.g., 50-100 ms) or on change. Start with a simple periodic send (say every 100 ms inside `loop()` with a `delay(100)`).
- **Console ESP32 Receiver:** On the console ESP32, write a corresponding test sketch to receive ESP-NOW data:
- Initialize Wi-Fi (station mode) and ESP-NOW in setup similarly.
- Register a receive callback: `esp_now_register_recv_cb(onDataRecv);` where `onDataRecv` is a function you define to handle incoming data. For example:

```
ControllerState incoming;
void onDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  memcpy(&incoming, incomingData, sizeof(incoming));
  Serial.print("Got button bitmask: ");
  Serial.println(incoming.buttons, BIN);
}
```

- This will print the received button bitmask to Serial for debugging.
- **Test ESP-NOW:** Flash the receiver code to the console ESP32 and the sender code to the controller ESP32. Open two serial monitors (one for each ESP32). When the controller is powered, you should see messages arriving on the console's serial (e.g., the button bitmask changing as you press buttons). Press each button on the controller and verify the console's log reflects the change (e.g., if you press the "Up" button, maybe you see bit0 go from 0 to 1 in the printed binary).
- If no data is received, troubleshoot: ensure both ESP32s are on the same Wi-Fi channel (you might call `WiFi.printDiag(Serial)` to check), and that the MAC in the sender code matches the receiver's station

MAC. Also verify you added the peer before sending.
- **Checkpoint:** By the end of Day 2, basic wireless communication is established: the controller ESP32 can send a data packet to the console ESP32 via ESP-NOW, and you can detect button presses in the console's logs. The hardware wiring for buttons is done and verified (use the serial output to ensure each button press changes the data). Commit this working ESP-NOW test code to the firmware repo.

Person B (Android Launcher & Game Apps):
- **Launcher Menu Implementation:** Expand the NinJoyLauncher app to have a functioning menu for game selection. For now, use simple UI components: e.g., three Buttons labeled "Game 1", "Game 2", "Game 3". Arrange them vertically or in a grid on the LauncherActivity layout. Give them onClick handlers that attempt to launch the respective game.
- Implement onClick for each game button. Use package names to launch:

```
Intent launchGame1 =
getPackageManager().getLaunchIntentForPackage("com.ninjoy.game1");
if (launchGame1 != null) {
    startActivity(launchGame1);
} else {
    Toast.makeText(this, "Game 1 not installed", Toast.LENGTH_SHORT).show();
}
```

Do similar for Game2 (package `com.ninjoy.game2`) and Game3. This way, clicking the button opens the game if it's installed.
- Test this on the phone: With the game stubs installed from Day 1, pressing the "Game 1" button in the launcher should launch the Game1 app. Use the phone's Back button to return to the launcher afterward. Confirm each of the three buttons successfully launches the correct game.
- **Game Stub Enhancement:** In each game app (Game1, Game2, Game3), set up basic scaffolding to prepare for controller input and game logic:
- **Game1:** For example, create a simple Activity that will represent a playable game. In its layout, maybe use a Canvas or SurfaceView for drawing. For Day 2, you can keep it simple: display a TextView that says "Game 1 – Running" and perhaps change text when certain keys are pressed.
- Override `onKeyDown` and `onKeyUp` in the Activity to capture input events. For now, just log them or update the TextView. For example, if `keyCode == KeyEvent.KEYCODE_DPAD_UP`, update the TextView to "Up pressed".
- Ensure the Activity is focusable for gamepad input: you might need to request focus on a view or use `android:focusable="true"` on the layout root. Many game controllers will send DPAD and button events as key events.
- **Game2 & Game3:** You can keep these minimal for now (perhaps just text placeholders), but set up the Activities similarly to listen for key events so that when you get to them, they're ready. For instance, Game2Activity could log "Button A pressed" if received, etc.
- **Tip:** If you have a USB or Bluetooth keyboard/gamepad available, you can test the key event handling in an emulator or on the phone by pairing that device to simulate the eventual controller. This can confirm your input handling works even before the custom ESP32 controller is ready.
- **ADB Setup & Efficiency:** To speed development, consider enabling **ADB over Wi-Fi** now that the phone has the launcher running (so you don't have to keep the phone tethered while testing controller range later):

- With the phone connected via USB initially, run `adb tcpip 5555` to put ADB in TCP/IP mode. Then find the phone's IP (Settings > About phone > Status, or via `adb shell ifconfig`). Run `adb connect <phone_ip>:5555`. This should allow deploying apps wirelessly.
- Test by disconnecting the USB and using Android Studio to run an app (the device should appear as an IP address in `adb devices`). This will be useful when Person A tests wireless range and you still want to debug the app.
- **GitHub Updates:** Commit and push the Day 2 changes: the launcher now has navigation logic, and each game app has initial input handling code. Make sure to commit all three game projects and the launcher. It's a good idea to tag a version or create a release in GitHub (e.g., "day2-basic-ui-working") for reference.
- **Checkpoint:** By end of Day 2, the custom launcher can launch each game stub, and each game stub is set up to handle input (currently just logging or UI feedback). The user can navigate between launcher and games using the phone's screen (touch or back button). The groundwork is laid for integrating controller input next. Person B can now see key codes from input events in logcat when pressing keys (preparing for when the actual controller is connected). ADB is configured for easier debugging.

Day 3 (Wed Aug 13) – ESP32 BLE HID & Controller Input Integration

Person A (Hardware & ESP32 Firmware):
- **BLE HID Controller Setup (Console ESP32):** Now focus on making the console ESP32 act as a **Bluetooth LE gamepad** that can control the Android phone. Using the Arduino IDE, set up the console ESP32's firmware to become a BLE HID device:
- Install a suitable Arduino library for BLE HID if available. For example, **ESP32-BLE-Gamepad** (by `lemmingDev`/T-vK) or **ESP32 BLE Keyboard** library can be used to simplify HID creation. You can install these via Arduino Library Manager or as a .zip. This will provide classes to send gamepad or keyboard reports easily.
- If using the library, initialize it in `setup()`. For example, with the ESP32-BLE-Gamepad library:

```
#include <BleGamepad.h>
BleGamepad bleGamepad("NinJoy Controller", "NinJoy", 100);
void setup() {
  bleGamepad.begin();
  // ... ESP-NOW init, etc.
}
```

This will make the ESP32 start advertising as a gamepad named "NinJoy Controller". The `100` is the battery level (just a placeholder for now).
- If not using a specialized library, use the ESP32 BLE library (NimBLE or Classic) to create a HID service manually. (Using the library is recommended for speed – it provides a predefined gamepad HID report descriptor).
- **Combine ESP-NOW Receiver with BLE:** Merge the ESP-NOW receiving code (from Day 2) with the BLE HID functionality on the console ESP32:
- Keep the `onDataRecv` callback as implemented, but now instead of just printing the data, map it to actual gamepad outputs. For example:

```
void onDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  ControllerState state;
  memcpy(&state, incomingData, sizeof(state));
  // Map D-pad bits to directions
  if (state.buttons & 0x01) bleGamepad.setHat(0);      // Up pressed (assuming
hat 0 = up)
  else if (state.buttons & 0x02) bleGamepad.setHat(4); // Down (hat value 4)
  // ... handle Left/Right similarly (hat 6 for left, 2 for right, etc.)
  // Map A/B buttons to gamepad buttons:
  if (state.buttons & 0x10) bleGamepad.press(BUTTON_1); else
bleGamepad.release(BUTTON_1); // A button
  if (state.buttons & 0x20) bleGamepad.press(BUTTON_2); else
bleGamepad.release(BUTTON_2); // B button
  // Map analog joystick (if used) to X/Y axes:
  int xVal = map(state.joyX, 0, 4095, -127,
127);  // example mapping ADC 0-4095 to -127 to 127
  int yVal = map(state.joyY, 0, 4095, -127, 127);
  bleGamepad.setAxes(xVal, yVal, 0, 0, 0, 0);  // assuming a 2-axis joystick,
other axes 0
}
```

The exact implementation will depend on the library's API. The key idea is: on receiving controller input, update the BLE gamepad state. Use *hat switch* for D-pad if available (hat = 8-direction POV). Use `.press()` / `.release()` for buttons, and `.setAxes()` for analog sticks/triggers.
- In `loop()`, call `bleGamepad.task()` if required by the library (some libraries need a periodic task call, others handle internally). If using NimBLE, you might not need this.
- **Pairing with Android Phone:** On the Galaxy S5, turn on Bluetooth and look for the "NinJoy Controller" device advertised by the console ESP32. Pair/connect to it as you would a normal gamepad. Android should recognize it as an input device (it may show up as a gamepad or keyboard). **Testing the BLE connection:**
- Once connected, the `bleGamepad.isConnected()` in your ESP32 code should return true. Now simulate some input: manually call `bleGamepad.press(BUTTON_1)` in code or press a wired button on the controller to send a message that triggers a press. If successful, the phone should register an input. For example, if mapped to a keyboard key, open a notepad app to see characters, or if mapped as gamepad, use a gamepad tester app or even check logcat.
- At this stage, do a simple test: map one button to a keyboard key (for instance, have the ESP32 send a SPACE or ENTER on button press using a keyboard HID library) to verify input arrives. For example, if using BLEKeyboard library, you might do: `BleKeyboard bleKeyboard; bleKeyboard.print("A");`. Because games are not fully ready yet, a quick way is to see any reaction on the phone (text entry, etc.) to confirm BLE HID is working.
- **Integration Test (basic):** Now test the end-to-end path for **one button**: Press a physical controller button -> ESP32 controller sends ESP-NOW -> ESP32 console receives and emits BLE HID -> Android phone receives it. You can log `onKeyDown` in Game1's Activity (from Person B's setup) to see if the press is detected. For example, press the "A" button on the controller and watch Android Studio logcat: it should show something like "keyCode 96 (BUTTON_A) down" in the logs of the game if all is working.
- Likely you will need to adjust the HID mapping to match Android's expected codes. Many BLE gamepad libraries map BUTTON_1 to Android's KeyEvent for A (typically keyCode 96 or 97). If the mapping is off, you

may see a different keyCode; adjust accordingly or use the library's defaults.
- **Troubleshooting:** If the phone doesn't receive input:
- Make sure the BLE device is actually connected (Android Settings > Bluetooth should show it as "Connected"). If not, you might need to press a button or call `bleGamepad.begin()` *after* the phone initiates connection (some libraries wait for a client to connect).
- Check that your ESP32 console isn't stuck or resetting (open serial monitor for any crashes). Providing power via a stable source (USB) is fine for now; the S5's USB port can't act as host for the ESP32's power easily, so continue using an external USB power for the console ESP32 during tests.
- Verify that the ESP-NOW reception is continuous and not interfering with BLE. ESP32 can handle Wi-Fi (ESP-NOW) and BLE concurrently, but heavy Wi-Fi traffic can sometimes cause BLE lag. Our usage is light, so it should be fine.
- **GitHub Commit:** Commit the new console ESP32 firmware code that includes BLE HID support, and the updated controller code if any changes were made. Include a README update describing how to pair the BLE controller to the phone for testers.
- **Checkpoint:** By end of Day 3, the console ESP32 should function as a BLE gamepad recognized by Android, and at least one or two buttons from the controller can control something on the phone (e.g., moving through a menu or triggering a log message in a game). The wireless controller pipeline (ESP-NOW + BLE) is proven out on a small scale. Person A and B should coordinate to verify that the Android side sees the input – e.g., have Person B's app log the key events and confirm matching button presses.

Person B (Android Launcher & Game Integration):
- **Integrate Controller Input with Launcher:** Modify the NinJoyLauncher so that it can be operated with the gamepad (BLE controller) instead of touch, for a console-like experience:
- In LauncherActivity, use focus and selection for the game list. If using Buttons for games, you can use the D-pad events from the controller to move focus: e.g., overriding `onKeyDown` to detect KeyEvent.KEYCODE_DPAD_UP/DPAD_DOWN and move the focus or selection index among the game buttons. Highlight the selected button (you can use `requestFocus()` on the target button or change its background).
- Detect the "A" or "Start" button (e.g., KeyEvent.KEYCODE_BUTTON_A or KEYCODE_ENTER) to launch the highlighted game. For example:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
      // move selection down
      return true;
  } else if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
      // move selection up
      return true;
  } else if (keyCode == KeyEvent.KEYCODE_BUTTON_A || keyCode ==
KeyEvent.KEYCODE_ENTER) {
      // launch selected game
      return true;
  }
  return super.onKeyDown(keyCode, event);
}
```

Implement logic to track which game is selected (e.g., an index 0-2 for the three games). This way the user can navigate the launcher with the controller solely.
- Test on the device: With the BLE controller connected, try pressing the D-pad on the controller. The focus should move through the game options. Press the A button – the corresponding game should launch. This will confirm that Person A's BLE input is reaching the Android UI.
- **Game1 Prototype Development:** Now that input can come from the controller, start building a simple game in **Game1** to utilize it. Aim for a minimal but functional prototype: for example, a simple 2D character movement demo.
- Use either a Canvas draw loop or a simple ImageView that you move around. For an intermediate developer, using a Canvas in a SurfaceView with a separate thread for game loop might be educational, but given time, even updating a view's `x,y` position in response to input events is sufficient.
- For Game1, assume it's something like an "endless runner" or a platformer demo: make a rectangle or sprite on screen. On DPAD left/right, move the sprite horizontally; on A button, make it "jump" (move up then down). You can simulate gravity by incrementing/decrementing a y-coordinate. Keep it very simple (the goal is just to confirm game logic can respond to controller).
- Handle input in Game1Activity's `onKeyDown` (and maybe `onKeyUp` for stopping movement). Use the same KeyEvent codes from the BLE controller. For example:

```
if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) { player.vx = -5; }  // move left
if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) { player.vx = 5; }  // move right
if (keyCode == KeyEvent.KEYCODE_BUTTON_A) { player.jump(); }   // jump on A
```

(Where `player` could be an object representing the character with properties vx for velocity, etc.)
- Use a simple game loop: you can post a Runnable to a Handler or use `View.postDelayed` to update the character's position (x += vx, y += vy, etc.) and invalidate the view to redraw. This avoids heavy framework knowledge and keeps things accessible to an intermediate dev.
- Draw the frame: either override `onDraw(Canvas)` in a custom View or update an ImageView's `setX()`/`setY()`. For example, if using a Canvas, draw a circle or square at the player's coordinates.
- **Checkpoint for Game1:** You should be able to **move a shape on screen using the physical controller** now. Test on the device: run Game1, pair the controller, and use the D-pad and A button. Confirm the shape moves and responds. If something isn't working (e.g., no response to input), debug by checking if `onKeyDown` is getting called (use `Log.d` or Toast). Possibly the view might not have focus; ensure you call `setFocusable(true)` and `requestFocus()` on your game view. You might also need to enable gamepad focus: in your Activity's layout, add `android:focusableInTouchMode="true"` to the root layout so it can capture keys even if no focused widget.
- **Game2 and Game3 Planning:** Outline simple ideas for the other two games (they will be tackled more in depth in later weeks, but set the stage now):
- Game2 could be something like a Pong or Brick Breaker clone (where the controller moves a paddle).
- Game3 could be a menu-based or turn-based game (less real-time, maybe a trivia or a simple puzzle that uses select/start buttons).
Write these ideas down in a planning document or the README so that in future weeks you have a direction. For now, you don't need to implement them beyond their current stub, but ensure their projects are compiling and ready.
- **Testing with Person A:** Coordinate an integration test: Person A and Person B together should verify the **full loop**: controller button -> ESP-NOW -> ESP32 console -> BLE -> Android launcher/game. For example, have Person A press various buttons while Person B watches the phone: navigate the launcher with the

controller and launch Game1, then move the character in Game1 with the controller. If any part fails, identify where: - If input isn't moving in Game1, check if Android is receiving it (logcat) – if not, perhaps BLE pairing or HID mapping is off (Person A would adjust). - If Android sees it but game logic not responding, Person B fixes the input handling. - Work together to fine-tune (this might involve small code tweaks on both sides). This real-time test is a big milestone – essentially a **primitive console experience** is now running.
- **GitHub Commits:** Person B pushes all changes: the improved launcher (now gamepad-navigable) and the Game1 prototype code. It's wise to tag this commit as something like "day3-controller-working". Possibly attach a screenshot of the game or launcher in the repo/wiki to show progress.
- **Checkpoint:** By end of Day 3, the project has a working wireless controller interfacing with the phone. The custom launcher can be controlled via the ESP32 gamepad, and one game (Game1) is minimally playable with the controller. Both team members have coordinated to get the system working together. This is a major functionality milestone for Week 1.

Day 4 (Thu Aug 14) – Refinement & Feature Expansion

Person A (Hardware & Firmware):
- **Optimize Controller Communication:** Refine the ESP-NOW packet structure and sending logic:
- Instead of sending constantly every 100ms regardless of changes, update the controller code to send **only on state change** or at a higher rate for smoother input. A good compromise: send at ~50 Hz but also immediately send an extra packet on any button edge (press/release) for responsiveness. This ensures quick reaction to presses but avoids flooding the channel.
- Compress the data if possible: since our state is small, it's fine. But for example, if only 1 byte of buttons and 2 bytes each for X/Y axes are needed, you can trim the struct to 5 bytes. (ESP-NOW can handle larger, ~250 bytes, but keeping it small reduces transmission time).
- Test that rapid button presses or holding a button results in repeated action in the game (for instance, holding the right button continuously moves the character in Game1 smoothly). If the movement is jittery, increase send frequency or ensure the game loop on Android is not the bottleneck.
- **BLE HID Tuning:** Check that the BLE HID reports are being sent properly:
- If using a library, many handle sending only on changes. But if not, ensure after setting axes or buttons, you call something like `bleGamepad.sendReport()` if required (some libraries do this internally).
- Verify that when multiple buttons are pressed, they all register. For example, hold down "Right" and tap "A" (move and jump). The controller should send both in its state, and the BLE should reflect both a DPAD direction and the A button. Android should register this as well (e.g., character moving and jumping).
- Calibrate analog inputs (if any): If your joystick readings are noisy or not centered, apply a small deadzone in code. E.g., if `abs(xVal) < 5` then treat as 0 to avoid drift.
- **Battery Level Report:** (Optional) The BLE gamepad library often can report battery level. If your controller will be battery-powered, you might later want to update this. For now, you can leave it at a fixed 100% or implement a dummy decrement to test the feature.
- **Hardware Power and Assembly:**
- Now that the electronics are working on breadboard, think about powering and packaging them. For the **controller ESP32**, plan to use a portable power source. If you have a small Li-Ion battery (3.7V), check if your ESP32 dev board has a built-in LiPo charging circuit (some do, especially if they have a JST battery connector). If yes, you can use it; if not, use a battery holder with a regulated 3.3V output or even a USB power bank for now.
- For the **console ESP32**, since it will be with the phone, you might power it via a USB cable from the phone's charger or a power bank. Another idea: the Galaxy S5 supports USB OTG but giving power from phone to

ESP32 is tricky. Instead, you could embed the ESP32 in the phone's case and use a short USB cable to a power bank or to the phone's charger. For now, keep using your PC or a wall adapter to power it during development.
- Begin **physical assembly**: It's time to move from loose wires to something the user can hold. Mount the buttons and ESP32 for the controller into a temporary enclosure: e.g., cut holes in a plastic project box or even use a piece of cardboard for layout. Place the ESP32 in the center and buttons in ergonomic positions. Ensure all wiring is secure (solder if possible, or use dupont wires firmly attached).
- If comfortable, open the back of the Galaxy S5 (remove the battery cover). See if there is space to fit the ESP32 module in there. The S5 has a removable battery – *do not interfere with the battery contacts*, but there might be a gap beside it where a small ESP32 board could sit. (If not, plan B: attach the ESP32 to the outside of the case or design a 3D-printed dock that holds the phone and the ESP32 module together).
- Evaluate the Galaxy J3 board one last time for any useful components for assembly: for example, does it have button caps or a shell that could be used for the controller? If not, it will likely be fully set aside now.
- **Testing After Changes:** After making code optimizations and reassembling hardware, run through all functions again:
- Check that rapid inputs (like quickly tapping a button multiple times) register each time in the game.
- Walk away ~5-10 meters with the controller to test ESP-NOW range and BLE stability; have someone watch the game character or use a log to see if inputs still come through. (ESP-NOW typically has good range, BLE might limit to ~10m reliably).
- Test continuous play for an extended period (e.g., 10 minutes) to ensure no memory leaks or crashes on the ESP32 (watch the serial output for any errors or restarts).
- If any disconnect occurs (BLE lost connection perhaps if out of range), see if it reconnects automatically when back in range. If not, consider adding reconnection logic: e.g., if `bleGamepad.isConnected() == false`, call `bleGamepad.startAdvertising()` to allow reconnection.
- **Documentation:** Start a simple schematic diagram of your wiring for the record (it can be hand-drawn and scanned or created with Fritzing/KiCad later). At minimum, document in the README which pins are used for which buttons and the communication flow between devices (could be a text diagram or bullet points).
- **Commit:** Push the refined firmware code to GitHub, with notes on what was improved (e.g., "optimized input sending, added analog support, improved BLE reconnection"). Possibly include a photo of the assembled controller in the repo or project wiki for reference (not mandatory, but useful if sharing progress).

Person B (Android Apps & UX Refinement):
- **Game1 Improvements:** Continue building out Game1 now that basic movement works:
- Implement simple game mechanics to make it more engaging. For example, add a ground platform and gravity for the jumping logic (so the player can only jump when "on the ground"). You can hardcode a ground level (y = constant value). Reset the player to that level when falling back down.
- Add an objective or challenge: perhaps place an "coin" or target on the screen and if the player sprite touches it, increase a score. This could be as simple as randomly placing a small rectangle that the player can "collect".
- Display the score on screen (use a TextView overlay or draw text in Canvas).
- Ensure the game can be **paused or exited**: maybe map the controller's Start button or the phone's Back button to pause/exit. E.g., if Back is pressed in Game1, call `finish()` to return to launcher; or if Start is pressed, show a "Paused" text (you can handle start as KEYCODE_BUTTON_START if BLE provides it, or map one of your extra buttons to an in-game menu).
- **Performance check:** Since the S5 is an old device, make sure the simple game runs at a decent frame

rate. If using Canvas drawing in a loop, try to keep it ~30 FPS. If it's choppy, reduce any unnecessary computations or consider using `SurfaceView` with a separate thread to not hog the UI thread. But given the simplicity, it should be fine.
- **Game2 & Game3 Structure:** Begin minimal development on the other two games (so that all three aren't blank):
- **Game2:** Implement a very basic **Pong-like game**: draw a paddle (controlled by the player) and a ball that bounces. The player can move the paddle with left/right on the controller. For now, it can be single-player Pong where missing the ball just resets it. This can reuse some logic from Game1 (movement, drawing).
- Set up the game to start when any button is pressed and allow exiting with Back.
- This will not be fully polished, but at least you have a second game to switch to and test that launching different games works smoothly.
- **Game3:** Perhaps make this a simple **menu-based game or demo**. For example, a "settings" app or a simple quiz: use the controller to navigate a list or select an answer. This could be largely UI-based (to experiment with another style of input handling). For instance, arrow keys to move up/down a list of options, A to select, B to go back. It's okay if it's just a placeholder like a settings menu for NinJoy (where nothing actually saves).
- Alternatively, if not enough time, Game3 can remain a stub with just a message like "Game3 coming soon" but ensure it handles Back to exit.
- **Launcher Polish:** Now that you have more content, refine the launcher's appearance and behavior:
- Replace the placeholder buttons with a nicer UI. For example, use ImageButtons or a RecyclerView with custom icons for each game. You could create simple icon images (e.g., a "1", "2", "3" or some graphic) and place them in the drawable resources.
- Improve the focus highlight for game selection. If using Buttons, Android's default focus highlight might be okay. If using a custom view, you can change the background of the selected item to a bright frame, etc. Make sure this highlight is clearly visible on the screen from a distance (assuming the phone might be connected to a TV later or just for user clarity).
- Hide the system UI for a more immersive feel: enable **immersive fullscreen mode** for the launcher (and games). You can do this by calling `getWindow().getDecorView().setSystemUiVisibility(...)` with flags like `View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY | View.SYSTEM_UI_FLAG_FULLSCREEN | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION`. This will remove the status and navigation bars. (On the S5, it means the app takes the whole screen and you won't accidentally swipe home or see notifications).
- Test that after a reboot, the NinJoyLauncher still comes up automatically and remains default. If the system ever asks for default launcher again, make sure to select "Always". (Sometimes app updates can reset this, so keep an eye on it during development).
- **Testing:** Do a comprehensive test of the user flow now:
- **Cold boot test:** Turn off the S5 completely, then power it on. Verify it boots directly into NinJoyLauncher without user intervention. (If any popups appear, clear or handle them).
- Once launcher is up, ensure the controller auto-reconnects: since the ESP32 console starts advertising at boot, the phone should reconnect to it as a known BLE device. If it doesn't, you might need to manually connect via Bluetooth settings; note this and coordinate with Person A to possibly call `bleGamepad.begin()` later or ensure continuous advertising until connected.
- Navigate the launcher with the controller, start Game1, play a bit, then press the controller's "Exit" (Back or mapped button) to return to launcher. Then start Game2, test its basic functionality, exit, etc. Do this for all three games.
- Observe if any crashes happen or if any app fails to return to the launcher properly. Fix issues accordingly (e.g., maybe a game didn't handle the Back button, so implement `onBackPressed()` in that Activity to simply `finish()`).

- Check for memory leaks or performance issues by watching if the phone slows down or gets warm; given the simplicity, it should be fine.
- **Collaboration:** Share with Person A the results, especially about BLE reconnection at boot. If the controller didn't auto-connect, plan a strategy (like possibly launching Bluetooth settings via ADB or prompt user to connect at start – or simply document that the controller should be powered on *before* the phone boots to auto-connect). This might be improved in Week 2 if needed.
- **GitHub Commit:** Commit all changes for launcher and games. At this point, you have a somewhat functional system, so consider creating a **Week1 demo release** on GitHub – not necessarily for public use, but to mark the progress. This could include the launcher APK and game APKs for archival and the firmware code.
- **Checkpoint:** By end of Day 4, the software side is much more polished: multiple games have basic functionality, the launcher is user-friendly and works with the controller, and the overall user experience is closer to a real console. The team has tested the system thoroughly in various scenarios (booting, switching games, continuous play). Any major bugs discovered are addressed or noted for future fixes.

Day 5 (Fri Aug 15) – Testing & Quality Assurance

Person A (Hardware & Firmware):
- **Robustness Testing:** Spend Day 5 stress-testing the hardware and firmware to ensure reliability:
- **Extended Play Test:** Use the controller to play Game1 or Game2 continuously for a longer session (e.g., 30+ minutes). Observe if the ESP32 modules remain stable (no resets or disconnects). Monitor the console ESP32 via serial (if possible, though it might be hard while playing; alternatively, add LED indicators for certain events as mentioned earlier).
- **Multiple Inputs:** Test pressing multiple buttons at once and unusual input patterns: e.g., hold down two directional buttons to simulate a diagonal (if both pressed, ensure your firmware handles it – maybe set hat to neutral if conflicting, or choose one). Press many buttons together (if your controller has enough) to see if any get missed.
- **ESP-NOW Packet Loss:** Though ESP-NOW is reliable for short distances, try blocking the signal briefly (e.g., cover the device with your hand or walk far) to see how the system behaves. When signal is restored, the controller should regain control quickly. If not, consider adding a reconnect or re-init routine: for example, if no packet received for >1 second, maybe have the console ESP32 print a warning or try re-adding the peer.
- **Power Interruption:** Simulate battery replacement or power cycle: turn off the controller ESP32 (if on battery, just switch it off or reset it). The console ESP32 should detect no input (maybe character stops moving) and when controller turns back on, the phone should reconnect BLE automatically and control resumes. If BLE does not reconnect, you might need to press the pairing button on the ESP32 (some libraries require you to re-begin advertising). Plan to address this either by instructing the user or coding an auto-reconnect (like calling `bleGamepad.begin()` again if disconnected for long).
- **Temperature & Battery:** If you have a thermal sensor or just by touch, ensure none of the ESP32s are overheating (they shouldn't in this use case, but good to check if enclosed). If using battery, check the battery level after the play test to estimate how many hours of use one charge might support. For example, if a 1000mAh battery dropped ~10% in 30 minutes, you have roughly 5 hours run time – note such estimates.
- **Firmware Code Quality:** Clean up the firmware code in preparation for sharing or expanding:
- Remove or comment out excessive debug `Serial.println` calls (or guard them with a debug flag) now that things are working, to reduce any slight performance hit and clutter. Keep some essential ones (or better, use an LED or small buzzer on certain events if desired for debugging without serial).
- Make the code more configurable: define constants for number of buttons, pins used, etc., at the top of

the sketch for easy changes. This way, if next week you decide to add a new button, it's straightforward.
- If not done, implement a simple **power-saving** measure for the controller: e.g., if no buttons pressed for a long time, maybe slow down ESP-NOW sending or dim an LED. (Deep sleep might be too aggressive since reconnecting BLE from deep sleep is complex; skip deep sleep for now unless battery life is a big issue in testing).
- Ensure both the controller and console firmware handle edge cases: for instance, what if the console ESP32 starts *before* the controller (on boot)? Ideally, the console just waits – ESP-NOW doesn't require an active connection, so that's fine. If the controller starts first, it will send packets that might not find a peer until the console is on; that's okay too (they'll be dropped). Basically, things are robust as is, but just note the startup order doesn't critically matter.
- **Version Control & Branching:** Since things are stable, consider making a **git branch** or tag for this stable state (e.g., branch "week1-stable-hardware"). This preserves the working version in case next changes introduce issues. Encourage Person B to do similarly.
- **GitHub Documentation:** Add more info to the firmware repository's README or Wiki:
- List the hardware components and where to buy/find them (e.g., "ESP32 DevKit V1 module", "generic tactile pushbuttons", etc.).
- Provide a short guide on pairing the controller to the phone, and what to do if it disconnects.
- If you drew a schematic or have photos from Day 4, post them here for reference.
- This will be useful if you share the project or if another team member needs to replicate the setup.
- **Stretch Goal – Sound/Vibration Feedback:** If you have spare time or want to experiment: the ESP32 console could drive some feedback hardware. For example, connect a small buzzer or LED to indicate when a button press is received or a game event happens (though without two-way communication from phone, the console ESP32 doesn't know game events, but it knows button presses). Alternatively, the phone could send feedback via BLE (advanced, requires BLE notifications from Android which might be out of scope for now). This is optional brainstorming for future features.
- **Checkpoint:** By end of Day 5, Person A ensures the controller and console hardware are **reliable and ready for demo**. All known bugs or issues on the firmware side are resolved or documented. The hardware is assembled in a semi-final form (maybe not pretty, but functional for demonstration). The code is cleaned up and well-documented.

Person B (Android & Software Quality):
- **Comprehensive App Testing:** Methodically test each app (launcher and 3 games) for stability and user experience:
- Go through **Launcher** functionality: scroll through game list with controller (ensure the selection wraps around or stops at ends as intended). If you press an invalid key (like a letter on a keyboard or some unassigned button), the launcher should ignore it gracefully (currently it will by default, which is fine).
- Launch **Game1**: play until you perhaps "collect" a few coins or reach some score. Test pausing (if implemented) and resuming. Press Back to exit mid-game – ensure it goes back to launcher without issues.
- Launch **Game2**: since this is a simple Pong demo, see that the ball bounces and the paddle moves via controller input. Test edge cases like missing the ball (does it reset?), or pressing A (if A isn't used, ensure it doesn't cause an error – it might just be ignored).
- Launch **Game3**: if it's menu-based, navigate its menu with the controller to ensure focus moves and selection works. If it's just a placeholder, just test that it opens and you can go back.
- If any app crashes, use Android Studio's logcat to find the stack trace and fix the bug. Common issues might be null pointer if something wasn't initialized, or maybe an arithmetic error in game logic. Fix and re-test.
- Test device rotation (if not locked): Ideally you locked orientation to landscape. Verify that – rotate the

phone, the app should stay in landscape (no unintended orientation changes that restart the activity).
- **UI/UX Enhancements:** Based on testing, make minor improvements:
- Maybe add a **startup splash screen** for games – a simple full-screen logo "NinJoy Game 1" that displays for 2 seconds on launch. This can be a Drawable or just a styled Activity that immediately transitions. It's not critical, but it's a nice polish if time permits.
- Fine-tune **visuals**: e.g., if text in the games is too small on the phone screen, increase font size since this will be viewed at arm's length or possibly on a TV. Use contrasting colors for any HUD (score, etc.) to be clearly visible.
- **Sound effects:** If possible, add one or two sound effects in Game1 and Game2 to enhance feedback (Android Studio can import WAV/OGG to `res/raw` and then use SoundPool or MediaPlayer). For example, a "jump" sound on A press in Game1, or a "blip" when the ball hits a paddle in Game2. Keep volume moderate. (Ensure the phone's media volume is up when testing.)
- **Vibration:** Utilize the phone's vibrator as simple haptic feedback on certain events (if appropriate). For instance, on Game1 jump or collision, call `vibrate()` using `Vibrator` service for a short buzz. This makes it feel more like a console controller (since controllers rumble). The Galaxy S5's vibration can act as a stand-in for now (later, maybe the controller could have a motor, but that's hardware not yet present).
- **App Storekeeping:**
- Set proper app names and icons for each game and the launcher (so that if someone looks at installed apps, they see "NinJoy Launcher", "NinJoy Game1" etc., with unique icons). Right now, they might all use the default Android icon. Create simple icon images (even just colored numbers or basic illustrations) and add to `res/mipmap`. Update the AndroidManifest `application label` and `icon` attributes for each app.
- Disable the launcher icons for games if you want them only accessible via the custom launcher (optional advanced step): You can add `<category android:name="android.intent.category.LAUNCHER" />` to the main activity of each game if you want them to appear in the Android UI. If you **remove** that category, the game won't show in the regular app drawer – meaning the only way to open it is via NinJoyLauncher. This could make the experience cleaner (so the user isn't tempted to launch games outside the console environment). If doing this, ensure your custom launcher is reliable, because you'd lose the ability to manually launch the games from the Android UI. For Week 1, you might leave them visible for easier debugging.
- **GitHub & Code Review:**
- Review your code for any hard-coded paths or debug leftovers (e.g., `Log.d` statements). It's fine to leave some logs for debugging input, but if spammy, remove or comment them.
- Write a short **README** for the Android repo explaining how to install the apps on a device, and listing the controls for each game (e.g., "D-Pad to move, A to jump" for Game1, etc.). This will be helpful for anyone testing the system.
- Push final Day 5 changes to GitHub. You might consider using GitHub issues to note any bugs you plan to fix (for example, "controller doesn't reconnect after phone reboots" if that was observed, or "Game2 ball sometimes gets stuck" – just as a way to track for next week).
- **Demo Preparation:** If you plan to demonstrate the project (maybe to a class or team), prepare a brief script or checklist so you don't forget to show key features. For instance: boot sequence, controller navigation, playing each game a bit, etc. This is more of a soft task but ensures you're ready to showcase progress.
- **Checkpoint:** By end of Day 5, the Android side is stable and user-friendly. All three game stubs function (with Game1 being the most fleshed out). The launcher and games are polished with basic sound/graphics. The system has been tested thoroughly, and any known issues are documented. Both team members should feel confident that the Week 1 prototype is demo-ready.

Day 6 (Sat Aug 16) – Buffer, Improvements & Documentation

Person A (Hardware & Firmware):
- **Final Hardware Touches:** Use this day to finish any remaining hardware tasks or improvements that were postponed:
- If any **additional buttons** or features were planned but not added (e.g., a "Select" button or a second analog stick for future use), you can wire them now while the controller is open. Even if the software doesn't use them yet, having them physically present could be useful. Be sure to update the firmware to include these new inputs in the ESP-NOW data structure (you might expand the buttons bitmask to a second byte if you go beyond 8 buttons). Test the reading of these new buttons via serial output to confirm they work (they can be integrated into games later).
- If the ESP32 controller board has a **built-in LED**, program it as a status indicator: for example, steady on when connected to the phone via BLE, blinking if not connected or if battery low (though you may not have battery monitoring yet). You can use the `bleGamepad.isConnected()` status: in your loop, if not connected, blink the LED every second; if connected, keep it on. This will help during demos to see connection status at a glance.
- Tidy up the controller's enclosure: secure the ESP32 module (with screws or hot glue or tape, as appropriate for a prototype), so it doesn't move around. Do the same for any loose wires. Close up the enclosure if possible, or tape it shut to simulate final form factor. The controller should be comfortably hand-held now.
- Mount the console ESP32: If you decided on a place for it (inside the phone's battery compartment or a case), secure it similarly. If it's inside the phone's back cover, ensure it doesn't short any phone components (use electric tape backing). Cut a small notch in the cover for any wires (like an antenna or USB cable) if needed. If mounting externally, perhaps attach it to the phone or a phone case with adhesive. **Note:** Ensure the ESP32's antenna (usually a little metal trace on the board) is not blocked by too much metal (the S5's back cover is plastic, which is good – but keep the antenna away from the phone's main board if possible to avoid interference).
- Power configuration: set up how the console ESP32 will be powered when the system is mobile. You might use a short USB cable from the phone's USB port to the ESP32's USB input. The Galaxy S5 in host mode might provide 5V out, but only if configured for OTG and with a special cable. If that's too complex, a simpler approach: use a slim USB power bank attached to the phone to power the ESP32. For this week, it's acceptable to have a second small battery for the ESP32 rather than drawing from the phone. Document whatever solution you choose, as it will affect user operation (e.g., "remember to charge both the phone and the ESP32's power bank").
- **Refine ESP-NOW Protocol (if needed):** If during tests you noticed any hiccups or inefficiencies, fine-tune them now:
- For example, if you found that the analog joystick was too sensitive or noisy, implement filtering (like averaging or increasing the deadzone).
- If the packet size grew due to new buttons, test that it's still transmitted quickly. (It should be – even 10-20 bytes is trivial for ESP-NOW).
- Make sure the **controller and console code are in sync** regarding the data structure. If you changed it (say, added a second byte for more buttons), update the struct definition on both sides accordingly and test again. Mismatched struct interpretations could cause weird bugs.
- **Collaboration for Next Steps:** Have a brief sync with Person B about any improvements on the Android side that might require hardware support. For example, if Person B wants to implement a second controller in future or a special input like tilt sensor, those would involve Person A's domain (ESP32 has an accelerometer? not in-built, but could attach one later). For now, likely out of scope, but just align on any

remaining tasks.
- **Documenting Hardware & Firmware:** Finalize documentation for the week:
- Ensure the **firmware GitHub README** has instructions to build and upload the code (which Arduino IDE board to select, any library dependencies like the BLE gamepad library).
- Include a pin-out table for the controller (e.g., "GPIO32 -> Up button, GPIO33 -> Down button, ...").
- Write down how to pair the BLE and any troubleshooting steps (e.g., "If controller won't connect, try forgetting the device on Android and re-pairing").
- If you made a schematic or wiring diagram, add it to the repo or as an image in the Wiki. A simple table or list may suffice if no diagram.
- **Future Planning:** Jot down ideas or requirements for Week 2 (not to do now, but to be ready): e.g., "Add multiplayer (second controller) support", "Improve game graphics or complexity", "Design a 3D printed case", etc. This isn't implementation, just planning ahead. Discuss these with Person B so both have a shared vision moving forward.
- **Checkpoint:** By end of Day 6, Person A's hardware is as complete as it will get in Week 1: the controller is fully built and portable, the console ESP32 is secured with the phone, and the firmware is polished and documented. Everything is ready for a final demo. Person A can step back knowing the physical side is solid for now.

Person B (Android & Final Touches):
- **Finalize App Details:** Today is about wrapping up and ensuring the software is presentation-ready:
- Double-check all **app names, icons, and version info**. Set the versionCode/versionName in the app build.gradle for each app to "1.0" and versionCode 1 (or some scheme) just to have it consistent. This is minor but shows professionalism.
- Remove any **unused permissions** from manifest. For example, if the template added `android.permission.INTERNET` but your games are offline, you can remove it. The launcher might not need any special permissions either. Keeping the app lean means fewer prompts (though on modern Android, these apps likely run on whatever OS the S5 has, maybe Android 6 with not as strict runtime permissions).
- Ensure the **launcher** is the only launcher on the device or at least default. If you haven't already, you might disable the stock TouchWiz launcher via Settings or ADB for demo (be careful: you have your custom one working well first!). This prevents the "choose launcher" dialog from popping up if something gets reset. You can do `adb shell pm disable <package_of_touchwiz>` as a one-time action (optional).
- **User Guide Creation:** Write a short **User Guide** that can be given to someone who might use this prototype. This can be in the project wiki or a PDF, but for now even a markdown file is fine. It should cover:
- How to turn on the console (e.g., "Power on the phone and ensure the console ESP32 module is powered. The launcher will start automatically.").
- How to turn on the controller (e.g., flip a switch or just connect battery, etc.) and how to pair if not already paired.
- Basic controls for each game.
- How to exit a game (usually Back button on controller or phone).
- Any quirks (e.g., "If controller disconnects, restart it and it should reconnect automatically within a few seconds.").
- This guide is useful for judges/teachers or even your future self when coming back to the project.
- **Dry Run Demo:** Do a full **end-to-end rehearsal** of demonstrating the project:
- Start with everything off. Then power on in the correct sequence and narrate (if you'll be presenting): phone boots to launcher, controller connects, etc.
- Show navigating the menu, launching each game briefly, demonstrating controls, then returning to menu.

- Time this process; ensure it can be done smoothly in a short time if needed. Also note if any part is slow (for example, does the BLE connection take a while on fresh boot? If yes, you might want to pre-connect before demo, or mention it while it's connecting).
- If possible, have someone else try to use the system without much guidance, and see if they find it intuitive. This can reveal if you forgot an important instruction or if something is not obvious (like maybe they didn't know which button is "A" if they are unlabeled – you could label the buttons on the physical controller with stickers).
- **Bug Fixes:** Any issues found during the dry run should be addressed: e.g., if the Back button on a game was not working reliably, fix that (maybe needed `android:enableHomeButton="true"` or handling onKeyUp instead of onKeyDown). If the launcher selection reset to top every time you come back (maybe you want it to remember last played game?), consider implementing that (e.g., store last index and set selection in `onResume()`). These are minor improvements that can make the experience smoother.
- **Prepare for Week 2:** Make a list of software features or improvements to tackle next. Possibly: integrating a scoring system online (if any), adding difficulty levels to games, improving graphics (maybe using a simple game framework or OpenGL ES if ambitious), etc. This is forward-looking and not to be implemented now. Share this with Person A especially if it involves new hardware (e.g., "Maybe use phone's accelerometer in a game" – Person A doesn't need to do anything for that, but it's something to note).
- **Repository Maintenance:** With the code pretty much final for the prototype, ensure all is checked in:
- Merge any branches (if you had experimental branches) back into `main` if they are stable.
- Tag the repository with **Week1_complete** or similar. This way you can always retrieve this version.
- If you have a continuous integration setup or just manual testing, maybe generate signed APKs for the launcher and games and attach to the release/tag for easy installation later.
- **Checkpoint:** By end of Day 6, Person B has the Android software side of NinJoy Jr. fully ready and documented. The user experience has been refined, and any last-minute bugs are squashed. The team is set to present or handoff the Week 1 deliverables confidently, with everything in GitHub and instructions available.

Day 7 (Sun Aug 17) – Review, Buffer, and Wrap-Up

Person A (Hardware & Firmware):
- **Review & Buffer Day:** Use this final day of Week 1 to tie up any loose ends and ensure everything is ready for next steps. This day serves as a buffer in case earlier tasks slipped. If Person A is on schedule, they will:
- Do a final review of the **hardware**: check all screws, tape, or mounts added on Day 6 are secure. Nothing should be overheating or under strain. If something looks fragile (e.g., a wire that could easily break off), address it now by reinforcing or planning a fix soon.
- Calibrate/center analog inputs one more time: perhaps in the firmware, print out the analog values with joystick at rest and adjust the center point if needed (e.g., subtract an offset). Tiny adjustments can improve control accuracy.
- Check the **firmware repositories** to ensure code is properly commented and the latest version is pushed. Double-check that any credentials or sensitive info (not likely in this project) are not in the repo.
- If behind schedule: use today to finish any critical hardware tasks not done (maybe soldering things that were left on a breadboard, etc.).
- **Future Prep:** If you foresee needing extra hardware for Week 2 (maybe a second ESP32 for a second controller, or some sensors or an SD card module for storing high scores, etc.), list them and possibly place orders now so they arrive in time.
- **Rest & Learn:** Take a short break from building and perhaps research briefly any topics that came up during Week 1 that you want to handle next week (for example, if BLE reconnection was tricky, read

Espressif forums for hints; or if thinking about 3D printing a case, look at CAD tutorials). No heavy work, just preparation.
- **Final Demo Participation:** Coordinate with Person B if a final end-of-week demo or review is happening today. Be on standby with hardware to show it working or to assist if any live issues appear (like a controller battery dying mid-demo – have a backup plan, e.g., keep USB power handy).

Person B (Software & Project Wrap-Up):
- **Week 1 Review:** This day is also a buffer for software tasks. Assuming all major issues are resolved, Person B will:
- Go through the **codebase** for each app and perform some housekeeping: format the code (Android Studio can auto-format with Ctrl+Alt+L on Windows/Cmd+Option+L on Mac), remove any leftover commented-out code that isn't needed, and ensure consistency in naming (for example, if you used `playerSprite` in one place and `player` in another, standardize it).
- Verify that all three games and the launcher have their **source code** committed. It's easy to forget to add new files (like that new icon image or a new Java class) to git – do a quick check of the GitHub repo vs. your local project to make sure nothing is missing.
- If time allows, write some basic **unit tests** or at least logic tests for any non-UI code. For example, if you made a function for collision detection in Game2, you could write a quick JUnit test for it. This isn't a high priority for a prototype, but demonstrating testing mindset is good practice.
- **App Backups:** Since this is an Android project, it can be useful to keep backups of the built APKs. Generate signed or unsigned APKs for the launcher and games as of Week 1 and save them (maybe in a `builds/` folder or as a GitHub release artifact). This way, if you need to quickly deploy the working version to a new device, you have them ready.
- **Project Documentation:** Finish any documentation needed for the end of Week 1:
- Combine Person A's and Person B's notes into a cohesive **Week 1 Report** (which might be exactly what this Q&A will become). This report should summarize what was accomplished each day by each person. It will help in writing any required progress reports or blog posts.
- Ensure that the main README of the project (or a separate docs file) clearly explains how to set up and run the entire system, as if for a new developer joining the project. This includes: environment setup (Arduino IDE, Android Studio, etc.), how to flash the firmware, how to install the apps, and how to operate the device. Essentially, someone else should be able to replicate the Week 1 result from the documentation and code provided.
- Perhaps create a small **FAQ** for issues encountered (e.g., "Q: Controller not connecting? A: Make sure the console ESP32 is powered and advertising (blue LED blinking). If needed, re-pair in Bluetooth settings.").
- **Rest & Plan:** If everything is truly done, take a breather and maybe do some light research or brainstorming for Week 2 features (similar to Person A's future prep). If planning to improve the game graphics, maybe research simple game engines or libraries (like libGDX or Unity – though switching might be too heavy; probably stick to Canvas for now). If planning online features, research how to use the phone's WiFi or Bluetooth for multi-player, etc. Just ideas for now.
- **Team Debrief:** It's good for Person A and B to have a wrap-up chat. Discuss what went well in Week 1 and what challenges came up. Update your project plan for Week 2 based on any new discoveries (for example, maybe you realized the phone's battery drains fast with screen on; in Week 2 you might plan power optimization or always-on charging).
- **Checkpoint:** By end of Day 7, Week 1 tasks are fully completed or any remaining items are noted for future. The project is in a stable state in version control. Both hardware and software are documented and the team is ready to present the prototype or move on to the next week's development. Week 1 concludes

with a functional “NinJoy Jr.” mini-console: an Android-driven game console with a custom wireless ESP32 controller, a custom launcher, and sample games – all set up in just seven days!
