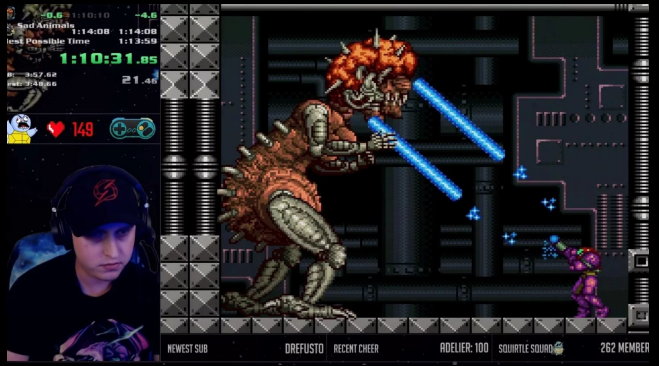
ESE 323 Project Presentation

By Chris Nielsen

Defining the project

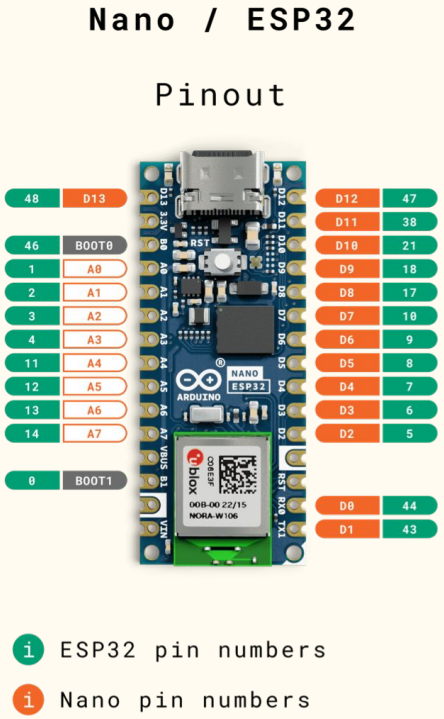
In many competitive and showcase scenarios the legitimacy of player control/inputs as well as other input based performance is brought into question. My original goal was to create a bridge between the game layer and a discrete/serial based reporting system. This would require a microcontroller to decode and decipher the signals sent by old school controllers, and then both display in real time and report to a computer what is going on.



MicroController Selected

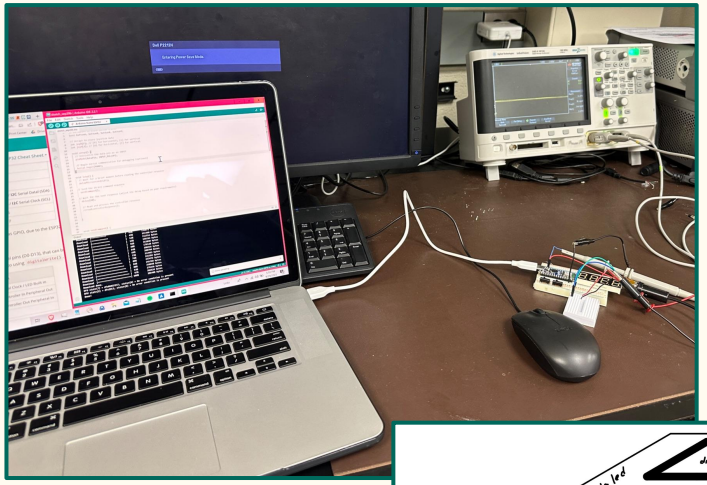
I chose the ESP32 as recommendation from Professor Westerfeld.

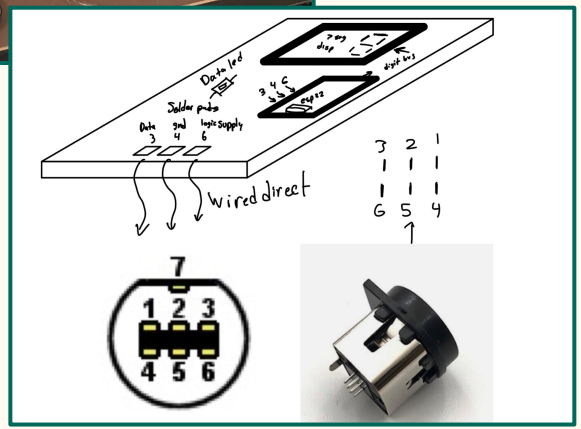
Arduino’s easy to use system programmer as well as interface made debugging and development easy.



Development (Hell?)

Development and research brought to my attention that the emulation and control of controllers made by game companies was not a reasonable task, and my device would have to be parasitic in its operations, only reading inputs when sensed across the bus.





Basic Program Flow

For the arduino, i have it setup for a maximum of 4 different controller types can be sensed and used. Here, i have hardcoded handling the gamecube outputted controller data packet structure. It works by looking first for the command sent by the game console and then delaying until it sees the 64 bit controller state response. Then once all bits are received it parses them and can be use to communicate everything over serial, and also output which button has been pressed to the 7-Segment display using the multi usage ports on the arduino.

```
Users > chrisnielsen > 🌀 .cpp > …
// Define the pin connected to the data line (Pin 2 - Red)
const int dataPin = 2;

void setup() {
  // Initialize the data pin as an OUTPUT
  pinMode(dataPin, OUTPUT);

  // Begin serial communication for debugging (optional)
  Serial.begin(9600);
}

void loop() {
  // Wait for a brief moment before sending the command
  delayMicroseconds(15);

  // Send the 24-bit command sequence
  sendCommand();

  // Wait for the controller response (adjust the delay based on your requirements)
  delay(10);
}

void sendCommand() {
  // Define the 24-bit command sequence
  byte command[3] = {0b01000000, 0b00000011, 0b00000010};

  // Loop through each byte of the command
  for (int byteIndex = 0; byteIndex < 3; byteIndex++) {
    byte currentByte = command[byteIndex];

    // Loop through each bit of the byte
    for (int bitIndex = 7; bitIndex >= 0; bitIndex--) {
      // Determine whether to set the data line high or low based on the bit
      bool isBitSet = (currentByte >> bitIndex) & 0x01;

      if (isBitSet) {
        digitalWrite(dataPin, HIGH);
      } else {
        digitalWrite(dataPin, LOW);
      }

      // Wait for the specified timing (adjust as needed)
      delayMicroseconds(5);

      // Set the data line high
      digitalWrite(dataPin, HIGH);

      // Wait for the specified timing (adjust as needed)
      delayMicroseconds(5);
    }
  }
}

// Define the pin connected to the data line (Pin 2 - Red)
const int dataPin = 2;

// Variables to store button states
bool buttonA, buttonB, buttonX, buttonY;

// Arrays to store joystick data
int joyX[2]; // [0] for horizontal, [1] for vertical
int joyY[2]; // [0] for horizontal, [1] for vertical

void setup() {
  // Begin Serial communication at 115200 baud rate
  Serial.begin(115200);

  // Initialize the data pin as an OUTPUT
  pinMode(dataPin, OUTPUT);
}

void loop() {
  // Wait for a brief moment before sending the command
  delayMicroseconds(15);

  // Send the 24-bit command sequence
  sendCommand();

  // Wait for the controller response (15ms after sending the command)
  delay(15);

  // Read and process the controller response
  readControllerResponse();

  // Wait for the next response (adjust the delay based on your requirements)
  delay(10);
}

void sendCommand() {
  // Define the 24-bit command sequence
  byte command[3] = {0b01000000, 0b00000011, 0b00000010};

  // Loop through each byte of the command
  for (int byteIndex = 0; byteIndex < 3; byteIndex++) {
    Serial.write(command[byteIndex]);
  }
}

void readControllerResponse() {
  // Store the received data
  unsigned long receivedData = 0;

  // Read the 64 bits of data
  for (int i = 0; i < 8; i++) {
    while (!Serial.available()) {} // Wait until data is available
    receivedData <<= 8;
    receivedData |= Serial.read();
  }

  // Extract button states
  buttonA = (receivedData >> 35) & 0x01;
  buttonB = (receivedData >> 34) & 0x01;
  buttonX = (receivedData >> 33) & 0x01;
  buttonY = (receivedData >> 32) & 0x01;

  // Extract joystick data
  joyX[0] = (receivedData >> 48) & 0xFF;
  joyX[1] = (receivedData >> 40) & 0xFF;
  joyY[0] = (receivedData >> 32) & 0xFF;
  joyY[1] = (receivedData >> 24) & 0xFF;

  // Debugging: Print the extracted data
  Serial.print("A: "); Serial.println(buttonA);
  Serial.print("B: "); Serial.println(buttonB);
  Serial.print("X: "); Serial.println(buttonX);
  Serial.print("Y: "); Serial.println(buttonY);
  Serial.print("JoyX[0]: "); Serial.println(joyX[0]);
  Serial.print("JoyX[1]: "); Serial.println(joyX[1]);
  Serial.print("JoyY[0]: "); Serial.println(joyY[0]);
  Serial.print("JoyY[1]: "); Serial.println(joyY[1]);
}
```

With an official controller attached, there is an interval of about 6.7ms between successive updates. Each update looks to last a little over 450us. The sequence starts with a 24-bit command from the console:

0100 0000 0000 0011 0000 0010

The last two bits of this sequence appear to be the 'rumble' control. The pattern is normally '10', but seems to change to '01' when the motor is running. After a rumble the pattern sometimes remains '00' for a while, but eventually reverts to '10'. It's not yet clear whether these are a simple enable and drive signal, or if some initialisation sequence is required as with the N64 controller.

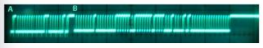
This sequence is probably a command word from the console. Although I've not seen the other bits change in this word, I'm sure they have some function codes (for example, there may be commands to reset the controller, or query what kind of hardware is attached to the console).

After the 24-bit command word, there is a short delay (about 15us, the data line being high) before the controller responds. After this period, the controller responds with a string of bits that contain the state of all the buttons along with joystick position data. From examination with an oscilloscope, the sequence appears to be as follows:

000, Start, Y, X, B, A, 1, L, R, Z,
DPadUp, DPadDown, DPadRight, DPadLeft,
JoyX (8-bit signed), JoyY (8-bit signed),
CX (8-bit signed), CY (8-bit signed),
Left (8-bit), Right (8-bit)

Finally, the data line returns high again. Above, the L/R buttons are the end-stops on the L/R shoulder buttons. Note that between A and L there is a bit that always appears to be high. Also, I haven't seen the three leading zeros change. The buttons (and extra bits) make up a 16-bit field, the analogue controls make up a further 48-bits. Therefore, the GC sends a 24-bit command to the controller, and the control responds with a total of 64-bits. On the oscilloscope, the data bits look to be 5us, which gives a total of (64+24+3)*5us = 455us. This is assuming that the 'scope time-base is accurate.

Here is an example of what a complete data string looks like on the 'scope. In this figure, point A is the start of the 24-bit command word sent by the console, and point B marks the start of the 64-bit response from the controller. The quality of the image is quite poor, but it's actually possible to see the individual data bits.



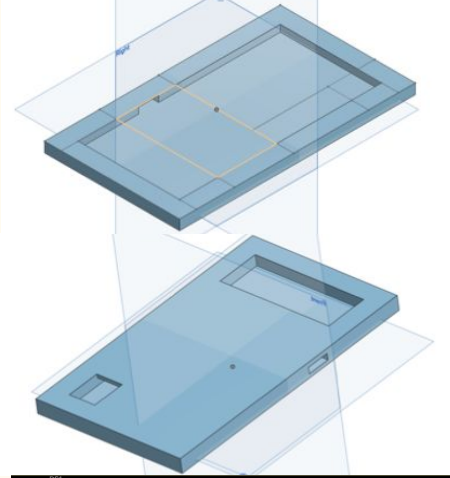
The bit-pattern looks the same regardless of which controller port is used, so I guess that the control word doesn't contain a port identifier. If the console is reset, the first rumble bit seems to go low initially and then returns high after a few seconds. The same thing happens when you power on the console initially. In general, the communication with controllers seems to happen almost immediately when the console is turned on, so it seems likely that there is some dedicated interface here, rather than something that needs to be set up programmatically by the CPU.

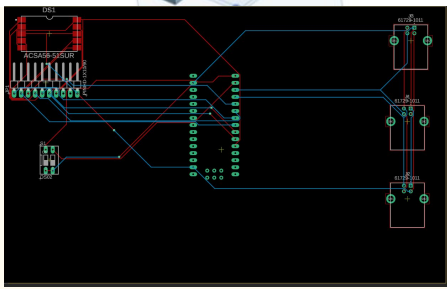
Finally, here is a close-up view of the individual data bits when transmitting binary 0100. This shows the presence of the short spikes in between every data bit, presumably where the transistor is switched off (internal bus tri-stated?) and the data line floats high again.

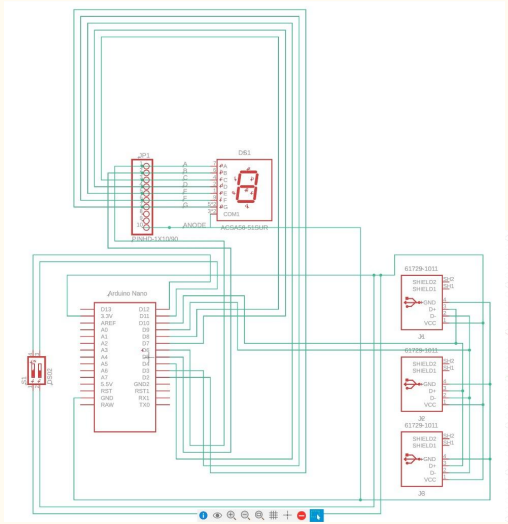
Final Design

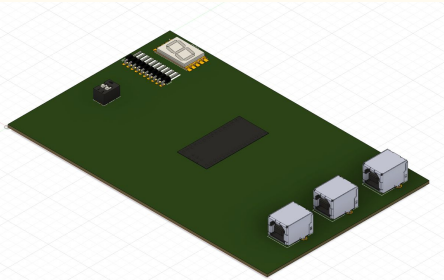
The final design is as follows:

A series of data lines from some USB compatible headers, which can be connected to any kind of game controller using soldering and wires after the fact, a dip switch able to tell the arduino which kind of controllers to process (if ever further developed for), as well as the top case and bottom case. This includes slots for the LED display, a programmer cable, and leaves room for any game controller port header.









Conclusion

This project had a lot of eureka moments, difficulties in development, hurdles in difficulty, expectation issues, etc.

Overall, I feel i know a lot more about the difficulties of taking a conceptual product on a breadboard, or even in a VHDL simulation environment, and producing it into an unchangeable and hard wired environment in which the system can run and live forever. I definitely will need the context of this project and this class to further my skill set in development/design of digital systems.