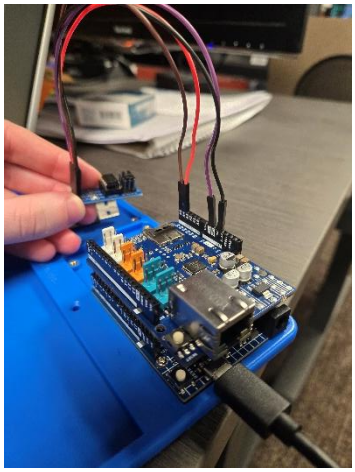
Die Ignition Project



This is the Die Ignition project that I helped with as an intern at Pridgeon and Clay. In this project we needed a reliable method to automatically detect and record when tooling or a press die is loaded into a machine, along with the identity of the die set being used. We had to find a way to capture this information onto an Arduino Uno R4 Minima.

Our solution was to embed an EEPROM (Electrically Erasable Programmable Read-Only Memory) chip into each die via the die protection cord (the wires is the die protection cord in this example). The EEPROM would store critical usage and historical data directly on the die like the ID or each die, accessible whenever the die is loaded into a press. An Arduino or similar microcontroller would serve as the interface, reading and updating EEPROM data as needed. In this project we were able to get dieID working and read from the PLC.

In this project, we used Modbus for network connectivity, and we have ethernet on port 502. This project was implemented using the Arduino IDE with three key libraries:

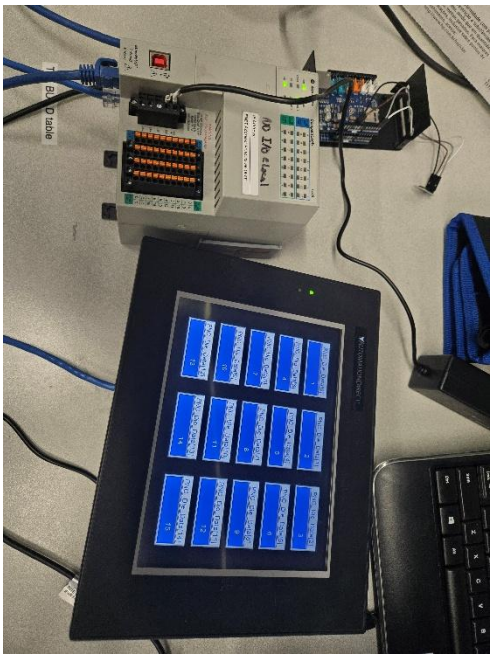
- **`Adafruit_EEPROM_I2C.h`** for I²C communication with the EEPROM chip.
- **Ethernet.h** chosen for its support for the Ethernet shield.
- **ArduinoModbus** Handles Communication between Arduino and PLC.

I learned how all the registers work in Modbus. The registers are organized as follows:

- **01**: Read Coil
- **02**: Read Discrete Input
- **03**: Read Holding Registers
- **04**: Read Input Registers
- **05**: Write Single Coil
- **06**: Write Single Register

How my project works:

I set up a CompactLogix PLC as the Modbus master and an Arduino as the Modbus slave over Ethernet. The Arduino acts as the Modbus server, providing the dieID value split across two 16-bit registers since the CompactLogix PLC uses 32-bit DINTs. This approach ensures the PLC can correctly read the full 32-bit value without exceeding the size limits of a single Modbus register. In this implementation, we communicate to the Programmable Logic Controller (PLC) over ArduinoModbus. This code turns the Arduino into a Modbus TCP slave for a PLC master to read and control. At startup, it initializes Ethernet, Modbus TCP, and EEPROM, then loads a unique die ID. PLC can send commands via Modbus to reset counts or start a new die, which clears or regenerates values accordingly. All data is stored in EEPROM and synced with Modbus so the PLC always sees the latest information.



I used RSLinx Classic to see what the PLC was sending over the network. I went to communications -> RSWho to see my device that was connected by ethernet. The ip address for the PLC is 192.168.1.1 and the Arduino has 192.168.1.23 so they are on the same subnet When the connection succeeds then you should be able to see the dieID change on the HMI connected to the PLC by ethernet in the rear end of the PLC. If you disconnect the wires going to the EEPROM chip the value on the HMI will go to 0 indicating that the die is disconnected. The dieID is saved onto the eeprom chip when you connect it again.

I spend some time researching how would I use Ethernet/IP to accomplish this same task. The issue was that CompactLogix doesn't natively talk raw TCP or UDP unless you use the Socket Interface with MSG instructions, which requires Studio 5000 to configure. Without Studio 5000, you can't set up the MSG commands for socket communication, so the PLC can't open a TCP connection to the Arduino. We confirmed that Rockwell's ENET-AT002-EN-P

publication documents exactly how to configure this. Marc Kramer was able to help me with this by showing data exchange between Raspberry PI and an Allen Bradley PLC. In the end though we used Modbus for communication because there was no library that could handle ethernet/ip for Arduino and doing ethernet over IP without a library was also very hard and involved a lot of memory management techniques that I could not wrap my head around fully.

On the HMI it is possible a negative number could show, that is not really a negative number. Any dieID that goes above the value 2,147,483,647 would appear negative on the HMI but the number is still a valid dieID it is just bigger then that number. In our setup, the PLC communicates with the Arduino over Modbus TCP, where PNC_Die_Data[0] represents the client side (read by the PLC for display on the HMI) and PNC_Die_Data[1] represents the server side (data sent from the PLC to the Arduino and this is set up but never actually used). Each EEPROM chip stores a unique 32-bit dieID at address 0. This value is read using memcpy(buffer, &value, 4), which copies four bytes from EEPROM memory into a variable (value), ensuring every chip has its own identifier. It then increments this value by one to create a new, unique ID and writes this updated value back to the EEPROM. By always storing the latest ID in memory, the system guarantees that the next time the function runs—even after a power cycle—it will start from the last used ID, preventing duplicates. It is possible a token system could work better for uniqueness Although I ran out of time to implement this fully, me and Doug came up with this solution.

The DieID is transmitted across two 16-bit Modbus registers—40001 and 40002—using low-word-first and high-byte-first ordering. Writing values to the registers and coils relies on Modbus Function Codes 16 and 05, respectively. For testing, ModScan32 was configured as a remote Modbus server with the IP address 192.168.1.23 to verify the data sent from the Arduino. It is important to note that the PLC may require up to an hour after power-up before entering Run mode and accepting commands. These two registers are then combined inside the PLC to show the full ID.

(Keep in mind it takes the PLC around an hour to boot into run mode so your code will not work until initialization is complete after every time you shut off the PLC).

```
// -------------------- EEPROM Helpers --------------------
bool write32BitsToEEPROM(uint16_t address, uint32_t value) {
  memcpy(buffer, (void *)&value, 4);
  return i2ceeprom.write(address, buffer, 4);
}

uint32_t read32BitsFromEEPROM(uint16_t address) {
  uint32_t value;
  i2ceeprom.read(address, buffer, 4);
  memcpy((void *)&value, buffer, 4);
  return value;
}

// -------------------- Modbus Helpers --------------------
void write32ToModbus(int reg, unsigned long value) {
  modbusTCPServer.holdingRegisterWrite(reg,   (value >> 16) & 0xFFFF); // High word
  modbusTCPServer.holdingRegisterWrite(reg+1, value & 0xFFFF);         // Low word
}

unsigned long read32FromModbus(int reg) {
  unsigned long high = modbusTCPServer.holdingRegisterRead(reg);
  unsigned long low  = modbusTCPServer.holdingRegisterRead(reg+1);
  return (high << 16) | low;
}
```

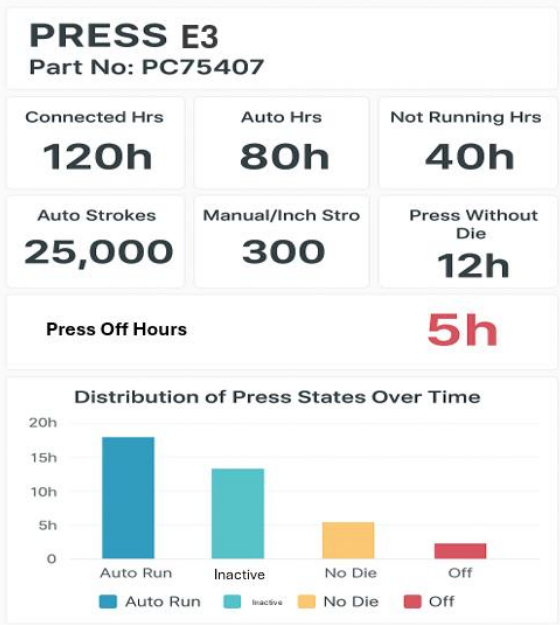
You can read about modbus here: [arduino-libraries/ArduinoModbus]

Follow this link to learn more about Ethernet/IP:

[EtherNet/IP Socket Interface Application Technique]

Wiring:

I connected my wires on the Arduino to the eeprom chip with **5v to VCC, GND to GND, A4 to SDA, and A5 to SCL.** The ethernet shield 2 does all the ethernet connections that I need so all you need to do after putting on the shield is to plug into ethernet and you should be good to go. Make sure to screw on the Arduino tightly to the case and make sure you are careful with bending the wires to much when putting the lid on. To avoid damage to the eeprom chip put the wires through the 4 holes on the back of the case, that worked for me. Special thanks to Cindy Martinez for making this diagram of how the data can be displayed at Pridgeon and Clay.



Variable	Modbus Holding Register	Modscan Address Range
DieID	0–1	40001–40002

This is my code for your own viewing that makes a DieID and stores it into eeprom and sends it over modbus. We have a ethernet server, ethernet client and modbus client and server so the Arduino can be a modbus server and send data to the plc and the plc can read the data it gets from the Arduino from the eeprom chip connected to the dies at Pridgeon and Clay.

```
#include <Ethernet.h>
#include <ArduinoModbus.h>
#include "Adafruit_EEPROM_I2C.h"

// -------------------- Ethernet and Modbus --------------------
EthernetServer server(502);             // Server for Modbus TCP Server mode
EthernetClient ethClient;               // For Modbus TCP Client mode
ModbusTCPClient modbusClient(ethClient); // Modbus Client (master)
ModbusTCPServer modbusTCPServer;        // Modbus Server (slave)
```

```
// -------------------- EEPROM --------------------
Adafruit_EEPROM_I2C i2ceeprom;
#define EEPROM_ADDR 0x50  // typical address for 24LC256

// -------------------- Network Config --------------------
byte mac[] = { 0x00, 0xB1, 0x1A, 0x44, 0x9B, 0x8C };
byte ip[] = { 192, 168, 1, 23 };
IPAddress plcIP(192, 168, 1, 1);  // PLC IP for client mode

// -------------------- Variables --------------------
bool lastEEPROMConnected = false;
uint8_t buffer[4];
unsigned long lastEEPROMCheck = 0;
bool eepromConnected = false;

const int DIE_ID_ADDR = 0;   // EEPROM Address
const int DIE_REG     = 0;    // Modbus Register
unsigned long dieID = 0;

// -------------------- EEPROM Helpers --------------------
bool write32BitsToEEPROM(uint16_t address, uint32_t value) {
  memcpy(buffer, (void *)&value, 4);
  return i2ceeprom.write(address, buffer, 4);
}

uint32_t read32BitsFromEEPROM(uint16_t address) {
  uint32_t value;
  i2ceeprom.read(address, buffer, 4);
  memcpy((void *)&value, buffer, 4);
  return value;
}

// -------------------- Modbus Helpers --------------------
void write32ToModbus(int reg, unsigned long value) {
  modbusTCPServer.holdingRegisterWrite(reg,   (value >> 16) & 0xFFFF); // High
word
  modbusTCPServer.holdingRegisterWrite(reg+1, value & 0xFFFF);         // Low
word
}

unsigned long read32FromModbus(int reg) {
  unsigned long high = modbusTCPServer.holdingRegisterRead(reg);
  unsigned long low  = modbusTCPServer.holdingRegisterRead(reg+1);
  return (high << 16) | low;
```

```
}

// -------------------- Setup --------------------
void setup() {
  Serial.begin(9600);
  delay(2000);

  Ethernet.begin(mac, ip);
  delay(100);

  server.begin(); // Start TCP server for Modbus server mode
  modbusTCPServer.begin(1); // Start Modbus TCP Server
  modbusTCPServer.configureHoldingRegisters(0, 100);

  if (i2ceeprom.begin(EEPROM_ADDR)) {
    dieID = generateUniqueDieID();
  } else {
    dieID = 0;
  }

  write32ToModbus(DIE_REG, dieID);
}

// -------------------- Main Loop --------------------
void loop() {
  delay(5000);
  checkeeprom();

  // ----- Modbus TCP Server Mode -----
  EthernetClient client = server.available();
  if (client) {
    modbusTCPServer.accept(client);
    while (client.connected()) {
      modbusTCPServer.poll();
      checkeeprom();
    }
  } else {
    Serial.println("No Modbus client connected");
  }

  // ----- Modbus TCP Client Mode (Optional) -----
  // Example: Write dieID to PLC holding register 10
  if (modbusClient.connected()) {
    modbusClient.holdingRegisterWrite(10, dieID & 0xFFFF); // send low word
  }
```

```
}

// -------------------- EEPROM Check --------------------
void checkeeprom() {
  if (millis() - lastEEPROMCheck > 2000) {
    lastEEPROMCheck = millis();
    bool currentState = i2ceeprom.begin(EEPROM_ADDR);

    if (currentState != eepromConnected) {
      eepromConnected = currentState;
      if (eepromConnected) {
        dieID = read32BitsFromEEPROM(DIE_ID_ADDR);
      } else {
        dieID = 0;
      }
      write32ToModbus(DIE_REG, dieID);
    }
  }
}
/*
Reads the most recently used ID value from the EEPROM, which is non-volatile
memory
that keeps its data even after the Arduino loses power. It then increments this
value by one to create a new,
unique ID and writes this updated value back to the EEPROM. By always storing the
latest ID in memory,
the system guarantees that the next time the function runs—even after a power
cycle—it will start from the last used ID, preventing duplicates.
May want a more token based way to do this.
*/
unsigned long generateUniqueDieID() {
  unsigned long lastID = read32BitsFromEEPROM(DIE_ID_ADDR);
  unsigned long newID = lastID + 1;  // Increment to get a new unique ID

  write32BitsToEEPROM(DIE_ID_ADDR, newID);  // Store new ID
  return newID;
}
```

Amazon ordering parts:

We used a KKSB Project Case to contain the Arduino on the floor. NOYITO-AT24C256-EEPROM chips that were soldered onto the board and the ethernet shield 2 purchased from Amazon (The links to all these items are down below in this document).

Ethernet Shield 2:

https://www.amazon.com/Ethernet-Development-ARDUINO-ETHERNET-SHIELD/dp/B011PCMMM0/ref=sr_1_3?crid=1IOTZNDLC43TP&dib=eyJ2IjoiMSJ9.uPl6pZAfoSJcRE2QAZX0NR8WuDMtvWcyezrXT-p-XFUNvCF0tXSghNmFRSd_Sym06SqYu36QNCs38w4Npt1O_3DZJBMD287FUrKdvR4xWFY6kxkL1ZYECmgb8wkZ1POADVG4Ws-PPC0wozF8I1cvJB4N2cMcDxKg6RbhsKrpagYnfzFmHCytNZZ7ZnCPUFeFynsB4XhKU3jnj2EYj-jOIA-BiJrM_UcRbJRiNU76hpJagaRQbNxAqzsnry5pCo8oj2G0ox5EEpO4AyUT1xPMatNSDkIarf7LwyU3stuk5Fg.WglkI_a59-xkqvPhacxaWkZNiqbpC5VsrYppluro26E&dib_tag=se&keywords=arduino+r4+minina+with+ethernet+shield&qid=1755627447&s=electronics&sprefix=arduino+r4+minina+with+ethernet+sheild%2Celectronics%2C230&sr=1-3

eeprom chip:

https://www.amazon.com/NOYITO-AT24C256-EEPROM-Memory-Interface/dp/B07GMCRPSP/ref=pd_sbs_d_sccl_2_3/136-7939186-7623155?pd_rd_w=B8wS7&content-id=amzn1.sym.2cd14f8d-eb5c-4042-b934-4a05eafd2874&pf_rd_p=2cd14f8d-eb5c-4042-b934-4a05eafd2874&pf_rd_r=ETEQ1DRDBPGBZWN3ENMW&pd_rd_wg=cBNue&pd_rd_r=47284a37-5f8a-4750-8e60-102ab35d46d3&pd_rd_i=B07GMCRPSP&psc=1

Arduino Case:

Amazon.com: KKSB Project Case for Arduino UNO Rev3 and Mega Rev3 - Steel SBC Enclosure - Space for Arduino Ethernet Shield : Electronics

Arduino uno R4 Minima:

https://www.amazon.com/Arduino-UNO-Minima-ABX00080-Connector/dp/B0C78K4CD4/ref=sr_1_3?crid=KOUBLTDS821P&dib=eyJ2IjoiMSJ9.Fl6ZZ0qt9tf13E2dOgJbh0sG-87JTWlGCqWrjRmCuyxPSgloS2aqCFz8cFjgmeaUGMpyHIlcBCzlyhzoJtZOgAfPI7mP66cxplqVkqMzDcWhZ2Si-sNN1LO8DZl9L2xYW3yMY9kawq5RaaSEIPRgTTkZ7WW08h2nBjnoLMNif1U48TrlElPHXgnGvixZHgdyWYo9BtMqQ2GbrbZVhsNvUOBi50zAIA25NZzv2Xv1ZHY6SX99FvtHU4GreXen_1CB6qJHIWKx9RV2rVmat0hjBp0qg24sMj10xl6MXgNH4Z0.PN8sd47K2-rO74yM9VdKnHeyVV4df-04eS0Rc7zA9vg&dib_tag=se&keywords=arduino+uno+r4+minima&qid=1756929115&s=electronics&sprefix=arduino+uno+r4+minima%2Celectronics%2C203&sr=1-3

I really enjoyed my time at Pridgeon and Clay and I learned a lot about IT and networking and trouble shooting techniques that I will use in my future tech career. I loved the

people that worked at Pridgeon and Clay and how open everyone was to let me learn and figure out problems for myself. Thank you Pridgeon and Clay.

ShortCut Version:

Here are very simple steps to get everything up and running.

1. Connect wires 3.3v->VCC, GND->GND, A4->SDA(Serial Data Line), and A5->SCL(Serial Clock Line).
2. Put the wires through the four holes in the back of the case.
3. Make sure WP, A0, A1, A2 is set to GND (moving the plastic cover over 1 pin should do the trick).
4. Plug in ethernet shield to PLC in front port and USB cable to computer to power up Arduino.
5. Copy my code into Arduino IDE and wait an hour and press upload.