INTRODUCTION

Introducing a railway gate control system utilizing image recognition technology represents a significant advancement in railway safety and efficiency. By harnessing the power of computer vision, this innovative system aims to automate the process of opening and closing railway barriers based on the detection of approaching trains.

Traditionally, railway gate control has relied on predetermined schedules or manual operation by gatekeepers. However, these methods are prone to human error and may not adequately respond to dynamic changes in train schedules or unexpected events.

The integration of image recognition technology revolutionizes this process by enabling real-time detection of trains using cameras installed at railway crossings. Through sophisticated algorithms, the system analyzes live video feeds to identify the presence of trains with precision and accuracy.

Upon detecting an approaching train, the system initiates the appropriate actions to ensure the safety of both railway users and pedestrians. This includes timely closing of barriers to prevent vehicles and pedestrians from crossing the railway tracks, as well as reopening them once the train has safely passed.

By automating the gate control process with image recognition, several benefits are realized. Firstly, it enhances safety by reducing the risk of accidents at railway crossings, as the system can swiftly respond to train movements. Secondly, it improves efficiency by optimizing the flow of traffic and minimizing disruptions caused by manual gate operation. Additionally, it offers scalability, allowing for seamless integration with existing railway infrastructure.

In summary, the railway gate control system using image recognition represents a forward-thinking solution to enhance safety and efficiency in railway operations. By leveraging cutting-edge technology, it sets a new standard for railway crossing management, ensuring smoother and safer journeys for all.

Here are some benefits of a railway gate control system using image recognition, presented in points:

**Enhanced Safety**: Real-time detection of trains prevents accidents at crossings.

**Improved Efficiency**: Automation reduces delays and optimizes traffic flow.

**Cost Savings**: Fewer accidents and manual operations lead to long-term savings.

**Accurate Detection**: Advanced algorithms minimize false detections.

**Scalability**: Easily integrates with existing infrastructure, adaptable to different locations.

**User Experience**: Smooth operations enhance satisfaction for commuters and travelers.

**Adaptability**: Works reliably in various environmental conditions.

Objective:

The primary objective of the railway gate control system using image recognition is to enhance safety and efficiency at railway crossings through automation. Specifically, the project aims to:

**1**.**Improve Safety**: Prevent accidents at railway crossings by automatically detecting approaching trains and closing barriers in real-time to prevent vehicles and pedestrians from crossing the tracks.

**2**.**Optimize Efficiency**: Streamline the gate control process to minimize delays and optimize traffic flow, reducing the risk of congestion and improving overall railway operations.

**3**.**Reduce Human Error**: Minimize the potential for human error associated with manual gate operation, ensuring consistent and reliable safety measures at railway crossings.

**4**.**Utilize Advanced Technology**: Harness the power of image recognition technology to accurately detect trains, enabling swift and accurate response to train movements.

**5**.**Enhance User Experience**: Provide a smoother and safer experience for commuters and travelers by reducing waiting times and ensuring reliable operation of railway crossings.

**6**.**Enable Scalability**: Design a system that can be easily integrated with existing railway infrastructure and adapted to different geographical locations, allowing for widespread implementation and scalability.

**7**. **Promote Cost Savings**: Achieve long-term cost savings for railway operators by reducing accidents, minimizing repair and maintenance expenses, and optimizing operational efficiency.

Abstract:

This project proposes the development of a railway gate control system utilizing image recognition technology to enhance safety and efficiency at railway crossings. The system aims to automate the process of opening and closing barriers in response to approaching trains, thereby reducing the risk of accidents and optimizing traffic flow. By leveraging advanced image recognition algorithms, the system can accurately detect trains in real-time, enabling swift and reliable barrier operation. The objectives include improving safety by preventing accidents, optimizing efficiency by reducing delays, minimizing human error, utilizing cutting-edge technology, enhancing user experience, enabling scalability, and promoting cost savings. The implementation of this system has the potential to revolutionize railway crossing management, ensuring smoother and safer journeys for commuters and travelers.

Problem Statement:

Railway crossings represent critical points of potential danger, where the intersection of train tracks and roadways poses significant safety challenges. Traditional methods of gate control rely on manual operation or fixed schedules, leading to inefficiencies, delays, and safety risks. Human error, unpredictable train schedules, and varying traffic conditions further compound these challenges, increasing the likelihood of accidents and disruptions.

The lack of real-time detection systems means that gate closure often occurs too early, causing unnecessary traffic congestion, or too late, risking collisions. Additionally, manual gate operation introduces the possibility of errors or oversight, jeopardizing the safety of both motorists and pedestrians.

In light of these issues, the need for an automated railway gate control system using image recognition technology becomes evident. This system seeks to address the shortcomings of existing methods by providing timely and accurate detection of approaching trains, enabling precise control of gate operations. By automating this process, the system aims to enhance safety, optimize traffic flow, reduce human error, and improve overall railway crossing management.

In summary, the problem we are solving revolves around the inefficiencies, safety risks, and operational challenges associated with traditional railway gate control methods. The proposed solution aims to revolutionize this process by leveraging advanced technology to ensure smoother, safer, and more efficient railway operations.

ARDUINO UNO :

The Arduino UNO is a standard board of Arduino. Here UNO means 'one' in Italian. It was named as UNO to label the first release of Arduino Software. It was also the first USB board released by Arduino. It is considered as the powerful board used in various projects. Arduino.cc developed the Arduino UNO board. Arduino UNO is based on an ATmega328P microcontroller. It is easy to use compared to other boards, such as the Arduino Mega board, etc. The board consists of digital and analog Input/Output pins (I/O), shields, and other circuits. The Arduino UNO includes 6 analog pin inputs, 14 digital pins, a USB connector, a power jack, and an ICSP (In-Circuit Serial Programming) header. It is programmed based on IDE, which stands for Integrated Development Environment. It can run on both online and offline platforms.

The components of Arduino UNO board are shown below:

Let's discuss each component in detail.

1. ATmega328 Microcontroller- It is a single chip Microcontroller of the ATmel family. The processor code inside it is of 8-bit. It combines Memory (SRAM, EEPROM, and Flash), Analog to Digital Converter, SPI serial ports, I/O lines, registers, timer, external and internal interrupts, and oscillator.

2. ICSP pin - The In-Circuit Serial Programming pin allows the user to program using the firmware of the Arduino board.

3. Power LED Indicator- The ON status of LED shows the power is activated. When the power is OFF, the LED will not light up.

4. Digital I/O pins- The digital pins have the value HIGH or LOW. The pins numbered from D0 to D13 are digital pins.

5. TX and RX LED's- The successful flow of data is represented by the lighting of these LED's.

6. AREF- The Analog Reference (AREF) pin is used to feed a reference voltage to the Arduino UNO board from the external power supply.

7. Reset button- It is used to add a Reset button to the connection.

8. USB- It allows the board to connect to the computer. It is essential for the programming of the Arduino UNO board.

9. Crystal Oscillator- The Crystal oscillator has a frequency of 16MHz, which makes the Arduino UNO a powerful board.

10. Voltage Regulator- The voltage regulator converts the input voltage to 5V.

11. GND- Ground pins. The ground pin acts as a pin with zero voltage.

12. Vin- It is the input voltage. 13. Analog Pins- The pins numbered from A0 to A5 are analog pins. The function of Analog pins is to read the analog sensor used in the connection. It can also act as GPIO (General Purpose Input Output) pins.

SERVO MOTOR:

A servomotor (or servo motor) is a simple electric motor, controlled with the help of servomechanism. If the motor as a controlled device, associated with servomechanism is DC motor, then it is commonly known as a DC Servo Motor. If AC operates the controlled motor, it is known as a AC Servo Motor. A servomotor is a linear actuator or rotary actuator that allows for precise control of linear or angular position, acceleration, and velocity. It consists of a motor coupled to a sensor for position feedback. It also requires a relatively sophisticated controller, often a dedicated module designed specifically for use with servomotors.

Servo Motor Theory

16. There are some special types of applications of an electric motor where the rotation of the motor is required for just a certain angle. For these applications, we require some special types of motor with some special arrangement which makes the motor rotate a certain angle for a given electrical input (signal). For this purpose, servo motor comes into the picture.

17. The servo motor is usually a simple DC motor controlled for specific angular rotation with the help of additional servomechanism (a typical closed-loop feedback control system). Nowadays, servo systems are used widely in industrial applications.

18. Servo motor applications are also commonly seen in remote-controlled toy cars for controlling the direction of motion, and it is also very widely used as the motor which moves the tray of a CD or DVD player. Besides these, there are hundreds of servo motor applications we see in our daily life.

19. The main reason behind using a servo is that it provides angular precision, i.e. it will only rotate as much we want and then stop and wait for the next signal to take further action. The servo motor is unlike a standard electric motor which starts turning as when we apply power to it, and the rotation continues until we switch off the power. We cannot control the rotational progress of electrical motor, but we can only control the speed of rotation and can turn it ON and OFF. Small servo motors are included many beginner Arduino starter kits, as they are easy to operate as part of a small electronics projects.

BUZZER:

An audio signaling device like a beeper or buzzer may be electromechanical or piezoelectric or mechanical type. The main function of this is to convert the signal from audio to sound. Generally, it is powered through DC voltage and used in timers, alarm devices, printers, alarms, computers, etc. Based on the various designs, it can generate different sounds like alarm, music, bell & siren.

The pin configuration of the buzzer is shown below. It includes two pins namely positive and negative. The positive terminal of this is represented with the ‘+’ symbol or a longer terminal. This terminal is powered through 6Volts whereas the negative terminal is represented with the ‘-‘symbol or short terminal and it is connected to the GND terminal.

These buzzers were invented by manufacturers of Japanese & fixed into a broad range of devices during the period of 1970s – 1980s. So, this development primarily came due to cooperative efforts through the manufacturing companies of Japanese. In the year 1951, they recognized the Application Research Committee of Barium Titanate that allows the corporations to be cooperative competitively & bring about numerous piezoelectric creations.

CODE OF ARDUINO UNO:

#include <Servo.h>

#include <Wire.h>

#include <Firmata.h>

#define I2C\_WRITE B00000000

#define I2C\_READ B00001000

#define I2C\_READ\_CONTINUOUSLY B00010000

#define I2C\_STOP\_READING B00011000

#define I2C\_READ\_WRITE\_MODE\_MASK B00011000

#define I2C\_10BIT\_ADDRESS\_MODE\_MASK B00100000

#define I2C\_END\_TX\_MASK B01000000

#define I2C\_STOP\_TX 1

#define I2C\_RESTART\_TX 0

#define I2C\_MAX\_QUERIES 8

#define I2C\_REGISTER\_NOT\_SPECIFIED -1

// the minimum interval for sampling analog input

#define MINIMUM\_SAMPLING\_INTERVAL 1

/\*==============================================================================

\* GLOBAL VARIABLES

\*============================================================================\*/

#ifdef FIRMATA\_SERIAL\_FEATURE

SerialFirmata serialFeature;

#endif

/\* analog inputs \*/

int analogInputsToReport = 0; // bitwise array to store pin reporting

/\* digital input ports \*/

byte reportPINs[TOTAL\_PORTS]; // 1 = report this port, 0 = silence

byte previousPINs[TOTAL\_PORTS]; // previous 8 bits sent

/\* pins configuration \*/

byte portConfigInputs[TOTAL\_PORTS]; // each bit: 1 = pin in INPUT, 0 = anything else

/\* timer variables \*/

unsigned long currentMillis; // store the current value from millis()

unsigned long previousMillis; // for comparison with currentMillis

unsigned int samplingInterval = 19; // how often to run the main loop (in ms)

/\* i2c data \*/

struct i2c\_device\_info {

byte addr;

int reg;

byte bytes;

byte stopTX;

};

/\* for i2c read continuous more \*/

i2c\_device\_info query[I2C\_MAX\_QUERIES];

byte i2cRxData[64];

boolean isI2CEnabled = false;

signed char queryIndex = -1;

// default delay time between i2c read request and Wire.requestFrom()

unsigned int i2cReadDelayTime = 0;

Servo servos[MAX\_SERVOS];

byte servoPinMap[TOTAL\_PINS];

byte detachedServos[MAX\_SERVOS];

byte detachedServoCount = 0;

byte servoCount = 0;

boolean isResetting = false;

// Forward declare a few functions to avoid compiler errors with older versions

// of the Arduino IDE.

void setPinModeCallback(byte, int);

void reportAnalogCallback(byte analogPin, int value);

void sysexCallback(byte, byte, byte\*);

/\* utility functions \*/

void wireWrite(byte data)

{

#if ARDUINO >= 100

Wire.write((byte)data);

#else

Wire.send(data);

#endif

}

byte wireRead(void)

{

#if ARDUINO >= 100

return Wire.read();

#else

return Wire.receive();

#endif

}

/\*==============================================================================

\* FUNCTIONS

\*============================================================================\*/

void attachServo(byte pin, int minPulse, int maxPulse)

{

if (servoCount < MAX\_SERVOS) {

// reuse indexes of detached servos until all have been reallocated

if (detachedServoCount > 0) {

servoPinMap[pin] = detachedServos[detachedServoCount - 1];

if (detachedServoCount > 0) detachedServoCount--;

} else {

servoPinMap[pin] = servoCount;

servoCount++;

}

if (minPulse > 0 && maxPulse > 0) {

servos[servoPinMap[pin]].attach(PIN\_TO\_DIGITAL(pin), minPulse, maxPulse);

} else {

servos[servoPinMap[pin]].attach(PIN\_TO\_DIGITAL(pin));

}

} else {

Firmata.sendString("Max servos attached");

}

}

void detachServo(byte pin)

{

servos[servoPinMap[pin]].detach();

// if we're detaching the last servo, decrement the count

// otherwise store the index of the detached servo

if (servoPinMap[pin] == servoCount && servoCount > 0) {

servoCount--;

} else if (servoCount > 0) {

// keep track of detached servos because we want to reuse their indexes

// before incrementing the count of attached servos

detachedServoCount++;

detachedServos[detachedServoCount - 1] = servoPinMap[pin];

}

servoPinMap[pin] = 255;

}

void enableI2CPins()

{

byte i;

// is there a faster way to do this? would probaby require importing

// Arduino.h to get SCL and SDA pins

for (i = 0; i < TOTAL\_PINS; i++) {

if (IS\_PIN\_I2C(i)) {

// mark pins as i2c so they are ignore in non i2c data requests

setPinModeCallback(i, PIN\_MODE\_I2C);

}

}

isI2CEnabled = true;

Wire.begin();

}

/\* disable the i2c pins so they can be used for other functions \*/

void disableI2CPins() {

isI2CEnabled = false;

// disable read continuous mode for all devices

queryIndex = -1;

}

void readAndReportData(byte address, int theRegister, byte numBytes, byte stopTX) {

// allow I2C requests that don't require a register read

// for example, some devices using an interrupt pin to signify new data available

// do not always require the register read so upon interrupt you call Wire.requestFrom()

if (theRegister != I2C\_REGISTER\_NOT\_SPECIFIED) {

Wire.beginTransmission(address);

wireWrite((byte)theRegister);

Wire.endTransmission(stopTX); // default = true

// do not set a value of 0

if (i2cReadDelayTime > 0) {

// delay is necessary for some devices such as WiiNunchuck

delayMicroseconds(i2cReadDelayTime);

}

} else {

theRegister = 0; // fill the register with a dummy value

}

Wire.requestFrom(address, numBytes); // all bytes are returned in requestFrom

// check to be sure correct number of bytes were returned by slave

if (numBytes < Wire.available()) {

Firmata.sendString("I2C: Too many bytes received");

} else if (numBytes > Wire.available()) {

Firmata.sendString("I2C: Too few bytes received");

numBytes = Wire.available();

}

i2cRxData[0] = address;

i2cRxData[1] = theRegister;

for (int i = 0; i < numBytes && Wire.available(); i++) {

i2cRxData[2 + i] = wireRead();

}

// send slave address, register and received bytes

Firmata.sendSysex(SYSEX\_I2C\_REPLY, numBytes + 2, i2cRxData);

}

void outputPort(byte portNumber, byte portValue, byte forceSend)

{

// pins not configured as INPUT are cleared to zeros

portValue = portValue & portConfigInputs[portNumber];

// only send if the value is different than previously sent

if (forceSend || previousPINs[portNumber] != portValue) {

Firmata.sendDigitalPort(portNumber, portValue);

previousPINs[portNumber] = portValue;

}

}

/\* -----------------------------------------------------------------------------

\* check all the active digital inputs for change of state, then add any events

\* to the Serial output queue using Serial.print() \*/

void checkDigitalInputs(void)

{

/\* Using non-looping code allows constants to be given to readPort().

\* The compiler will apply substantial optimizations if the inputs

\* to readPort() are compile-time constants. \*/

if (TOTAL\_PORTS > 0 && reportPINs[0]) outputPort(0, readPort(0, portConfigInputs[0]), false);

if (TOTAL\_PORTS > 1 && reportPINs[1]) outputPort(1, readPort(1, portConfigInputs[1]), false);

if (TOTAL\_PORTS > 2 && reportPINs[2]) outputPort(2, readPort(2, portConfigInputs[2]), false);

if (TOTAL\_PORTS > 3 && reportPINs[3]) outputPort(3, readPort(3, portConfigInputs[3]), false);

if (TOTAL\_PORTS > 4 && reportPINs[4]) outputPort(4, readPort(4, portConfigInputs[4]), false);

if (TOTAL\_PORTS > 5 && reportPINs[5]) outputPort(5, readPort(5, portConfigInputs[5]), false);

if (TOTAL\_PORTS > 6 && reportPINs[6]) outputPort(6, readPort(6, portConfigInputs[6]), false);

if (TOTAL\_PORTS > 7 && reportPINs[7]) outputPort(7, readPort(7, portConfigInputs[7]), false);

if (TOTAL\_PORTS > 8 && reportPINs[8]) outputPort(8, readPort(8, portConfigInputs[8]), false);

if (TOTAL\_PORTS > 9 && reportPINs[9]) outputPort(9, readPort(9, portConfigInputs[9]), false);

if (TOTAL\_PORTS > 10 && reportPINs[10]) outputPort(10, readPort(10, portConfigInputs[10]), false);

if (TOTAL\_PORTS > 11 && reportPINs[11]) outputPort(11, readPort(11, portConfigInputs[11]), false);

if (TOTAL\_PORTS > 12 && reportPINs[12]) outputPort(12, readPort(12, portConfigInputs[12]), false);

if (TOTAL\_PORTS > 13 && reportPINs[13]) outputPort(13, readPort(13, portConfigInputs[13]), false);

if (TOTAL\_PORTS > 14 && reportPINs[14]) outputPort(14, readPort(14, portConfigInputs[14]), false);

if (TOTAL\_PORTS > 15 && reportPINs[15]) outputPort(15, readPort(15, portConfigInputs[15]), false);

}

// -----------------------------------------------------------------------------

/\* sets the pin mode to the correct state and sets the relevant bits in the

\* two bit-arrays that track Digital I/O and PWM status

\*/

void setPinModeCallback(byte pin, int mode)

{

if (Firmata.getPinMode(pin) == PIN\_MODE\_IGNORE)

return;

if (Firmata.getPinMode(pin) == PIN\_MODE\_I2C && isI2CEnabled && mode != PIN\_MODE\_I2C) {

// disable i2c so pins can be used for other functions

// the following if statements should reconfigure the pins properly

disableI2CPins();

}

if (IS\_PIN\_DIGITAL(pin) && mode != PIN\_MODE\_SERVO) {

if (servoPinMap[pin] < MAX\_SERVOS && servos[servoPinMap[pin]].attached()) {

detachServo(pin);

}

}

if (IS\_PIN\_ANALOG(pin)) {

reportAnalogCallback(PIN\_TO\_ANALOG(pin), mode == PIN\_MODE\_ANALOG ? 1 : 0); // turn on/off reporting

}

if (IS\_PIN\_DIGITAL(pin)) {

if (mode == INPUT || mode == PIN\_MODE\_PULLUP) {

portConfigInputs[pin / 8] |= (1 << (pin & 7));

} else {

portConfigInputs[pin / 8] &= ~(1 << (pin & 7));

}

}

Firmata.setPinState(pin, 0);

switch (mode) {

case PIN\_MODE\_ANALOG:

if (IS\_PIN\_ANALOG(pin)) {

if (IS\_PIN\_DIGITAL(pin)) {

pinMode(PIN\_TO\_DIGITAL(pin), INPUT); // disable output driver

#if ARDUINO <= 100

// deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6

digitalWrite(PIN\_TO\_DIGITAL(pin), LOW); // disable internal pull-ups

#endif

}

Firmata.setPinMode(pin, PIN\_MODE\_ANALOG);

}

break;

case INPUT:

if (IS\_PIN\_DIGITAL(pin)) {

pinMode(PIN\_TO\_DIGITAL(pin), INPUT); // disable output driver

#if ARDUINO <= 100

// deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6

digitalWrite(PIN\_TO\_DIGITAL(pin), LOW); // disable internal pull-ups

#endif

Firmata.setPinMode(pin, INPUT);

}

break;

case PIN\_MODE\_PULLUP:

if (IS\_PIN\_DIGITAL(pin)) {

pinMode(PIN\_TO\_DIGITAL(pin), INPUT\_PULLUP);

Firmata.setPinMode(pin, PIN\_MODE\_PULLUP);

Firmata.setPinState(pin, 1);

}

break;

case OUTPUT:

if (IS\_PIN\_DIGITAL(pin)) {

if (Firmata.getPinMode(pin) == PIN\_MODE\_PWM) {

// Disable PWM if pin mode was previously set to PWM.

digitalWrite(PIN\_TO\_DIGITAL(pin), LOW);

}

pinMode(PIN\_TO\_DIGITAL(pin), OUTPUT);

Firmata.setPinMode(pin, OUTPUT);

}

break;

case PIN\_MODE\_PWM:

if (IS\_PIN\_PWM(pin)) {

pinMode(PIN\_TO\_PWM(pin), OUTPUT);

analogWrite(PIN\_TO\_PWM(pin), 0);

Firmata.setPinMode(pin, PIN\_MODE\_PWM);

}

break;

case PIN\_MODE\_SERVO:

if (IS\_PIN\_DIGITAL(pin)) {

Firmata.setPinMode(pin, PIN\_MODE\_SERVO);

if (servoPinMap[pin] == 255 || !servos[servoPinMap[pin]].attached()) {

// pass -1 for min and max pulse values to use default values set

// by Servo library

attachServo(pin, -1, -1);

}

}

break;

case PIN\_MODE\_I2C:

if (IS\_PIN\_I2C(pin)) {

// mark the pin as i2c

// the user must call I2C\_CONFIG to enable I2C for a device

Firmata.setPinMode(pin, PIN\_MODE\_I2C);

}

break;

case PIN\_MODE\_SERIAL:

#ifdef FIRMATA\_SERIAL\_FEATURE

serialFeature.handlePinMode(pin, PIN\_MODE\_SERIAL);

#endif

break;

default:

Firmata.sendString("Unknown pin mode"); // TODO: put error msgs in EEPROM

}

// TODO: save status to EEPROM here, if changed

}

/\*

\* Sets the value of an individual pin. Useful if you want to set a pin value but

\* are not tracking the digital port state.

\* Can only be used on pins configured as OUTPUT.

\* Cannot be used to enable pull-ups on Digital INPUT pins.

\*/

void setPinValueCallback(byte pin, int value)

{

if (pin < TOTAL\_PINS && IS\_PIN\_DIGITAL(pin)) {

if (Firmata.getPinMode(pin) == OUTPUT) {

Firmata.setPinState(pin, value);

digitalWrite(PIN\_TO\_DIGITAL(pin), value);

}

}

}

void analogWriteCallback(byte pin, int value)

{

if (pin < TOTAL\_PINS) {

switch (Firmata.getPinMode(pin)) {

case PIN\_MODE\_SERVO:

if (IS\_PIN\_DIGITAL(pin))

servos[servoPinMap[pin]].write(value);

Firmata.setPinState(pin, value);

break;

case PIN\_MODE\_PWM:

if (IS\_PIN\_PWM(pin))

analogWrite(PIN\_TO\_PWM(pin), value);

Firmata.setPinState(pin, value);

break;

}

}

}

void digitalWriteCallback(byte port, int value)

{

byte pin, lastPin, pinValue, mask = 1, pinWriteMask = 0;

if (port < TOTAL\_PORTS) {

// create a mask of the pins on this port that are writable.

lastPin = port \* 8 + 8;

if (lastPin > TOTAL\_PINS) lastPin = TOTAL\_PINS;

for (pin = port \* 8; pin < lastPin; pin++) {

// do not disturb non-digital pins (eg, Rx & Tx)

if (IS\_PIN\_DIGITAL(pin)) {

// do not touch pins in PWM, ANALOG, SERVO or other modes

if (Firmata.getPinMode(pin) == OUTPUT || Firmata.getPinMode(pin) == INPUT) {

pinValue = ((byte)value & mask) ? 1 : 0;

if (Firmata.getPinMode(pin) == OUTPUT) {

pinWriteMask |= mask;

} else if (Firmata.getPinMode(pin) == INPUT && pinValue == 1 && Firmata.getPinState(pin) != 1) {

// only handle INPUT here for backwards compatibility

#if ARDUINO > 100

pinMode(pin, INPUT\_PULLUP);

#else

// only write to the INPUT pin to enable pullups if Arduino v1.0.0 or earlier

pinWriteMask |= mask;

#endif

}

Firmata.setPinState(pin, pinValue);

}

}

mask = mask << 1;

}

writePort(port, (byte)value, pinWriteMask);

}

}

// -----------------------------------------------------------------------------

/\* sets bits in a bit array (int) to toggle the reporting of the analogIns

\*/

//void FirmataClass::setAnalogPinReporting(byte pin, byte state) {

//}

void reportAnalogCallback(byte analogPin, int value)

{

if (analogPin < TOTAL\_ANALOG\_PINS) {

if (value == 0) {

analogInputsToReport = analogInputsToReport & ~ (1 << analogPin);

} else {

analogInputsToReport = analogInputsToReport | (1 << analogPin);

// prevent during system reset or all analog pin values will be reported

// which may report noise for unconnected analog pins

if (!isResetting) {

// Send pin value immediately. This is helpful when connected via

// ethernet, wi-fi or bluetooth so pin states can be known upon

// reconnecting.

Firmata.sendAnalog(analogPin, analogRead(analogPin));

}

}

}

// TODO: save status to EEPROM here, if changed

}

void reportDigitalCallback(byte port, int value)

{

if (port < TOTAL\_PORTS) {

reportPINs[port] = (byte)value;

// Send port value immediately. This is helpful when connected via

// ethernet, wi-fi or bluetooth so pin states can be known upon

// reconnecting.

if (value) outputPort(port, readPort(port, portConfigInputs[port]), true);

}

// do not disable analog reporting on these 8 pins, to allow some

// pins used for digital, others analog. Instead, allow both types

// of reporting to be enabled, but check if the pin is configured

// as analog when sampling the analog inputs. Likewise, while

// scanning digital pins, portConfigInputs will mask off values from any

// pins configured as analog

}

/\*==============================================================================

\* SYSEX-BASED commands

\*============================================================================\*/

void sysexCallback(byte command, byte argc, byte \*argv)

{

byte mode;

byte stopTX;

byte slaveAddress;

byte data;

int slaveRegister;

unsigned int delayTime;

switch (command) {

case I2C\_REQUEST:

mode = argv[1] & I2C\_READ\_WRITE\_MODE\_MASK;

if (argv[1] & I2C\_10BIT\_ADDRESS\_MODE\_MASK) {

Firmata.sendString("10-bit addressing not supported");

return;

}

else {

slaveAddress = argv[0];

}

// need to invert the logic here since 0 will be default for client

// libraries that have not updated to add support for restart tx

if (argv[1] & I2C\_END\_TX\_MASK) {

stopTX = I2C\_RESTART\_TX;

}

else {

stopTX = I2C\_STOP\_TX; // default

}

switch (mode) {

case I2C\_WRITE:

Wire.beginTransmission(slaveAddress);

for (byte i = 2; i < argc; i += 2) {

data = argv[i] + (argv[i + 1] << 7);

wireWrite(data);

}

Wire.endTransmission();

delayMicroseconds(70);

break;

case I2C\_READ:

if (argc == 6) {

// a slave register is specified

slaveRegister = argv[2] + (argv[3] << 7);

data = argv[4] + (argv[5] << 7); // bytes to read

}

else {

// a slave register is NOT specified

slaveRegister = I2C\_REGISTER\_NOT\_SPECIFIED;

data = argv[2] + (argv[3] << 7); // bytes to read

}

readAndReportData(slaveAddress, (int)slaveRegister, data, stopTX);

break;

case I2C\_READ\_CONTINUOUSLY:

if ((queryIndex + 1) >= I2C\_MAX\_QUERIES) {

// too many queries, just ignore

Firmata.sendString("too many queries");

break;

}

if (argc == 6) {

// a slave register is specified

slaveRegister = argv[2] + (argv[3] << 7);

data = argv[4] + (argv[5] << 7); // bytes to read

}

else {

// a slave register is NOT specified

slaveRegister = (int)I2C\_REGISTER\_NOT\_SPECIFIED;

data = argv[2] + (argv[3] << 7); // bytes to read

}

queryIndex++;

query[queryIndex].addr = slaveAddress;

query[queryIndex].reg = slaveRegister;

query[queryIndex].bytes = data;

query[queryIndex].stopTX = stopTX;

break;

case I2C\_STOP\_READING:

byte queryIndexToSkip;

// if read continuous mode is enabled for only 1 i2c device, disable

// read continuous reporting for that device

if (queryIndex <= 0) {

queryIndex = -1;

} else {

queryIndexToSkip = 0;

// if read continuous mode is enabled for multiple devices,

// determine which device to stop reading and remove it's data from

// the array, shifiting other array data to fill the space

for (byte i = 0; i < queryIndex + 1; i++) {

if (query[i].addr == slaveAddress) {

queryIndexToSkip = i;

break;

}

}

for (byte i = queryIndexToSkip; i < queryIndex + 1; i++) {

if (i < I2C\_MAX\_QUERIES) {

query[i].addr = query[i + 1].addr;

query[i].reg = query[i + 1].reg;

query[i].bytes = query[i + 1].bytes;

query[i].stopTX = query[i + 1].stopTX;

}

}

queryIndex--;

}

break;

default:

break;

}

break;

case I2C\_CONFIG:

delayTime = (argv[0] + (argv[1] << 7));

if (argc > 1 && delayTime > 0) {

i2cReadDelayTime = delayTime;

}

if (!isI2CEnabled) {

enableI2CPins();

}

break;

case SERVO\_CONFIG:

if (argc > 4) {

// these vars are here for clarity, they'll optimized away by the compiler

byte pin = argv[0];

int minPulse = argv[1] + (argv[2] << 7);

int maxPulse = argv[3] + (argv[4] << 7);

if (IS\_PIN\_DIGITAL(pin)) {

if (servoPinMap[pin] < MAX\_SERVOS && servos[servoPinMap[pin]].attached()) {

detachServo(pin);

}

attachServo(pin, minPulse, maxPulse);

setPinModeCallback(pin, PIN\_MODE\_SERVO);

}

}

break;

case SAMPLING\_INTERVAL:

if (argc > 1) {

samplingInterval = argv[0] + (argv[1] << 7);

if (samplingInterval < MINIMUM\_SAMPLING\_INTERVAL) {

samplingInterval = MINIMUM\_SAMPLING\_INTERVAL;

}

} else {

//Firmata.sendString("Not enough data");

}

break;

case EXTENDED\_ANALOG:

if (argc > 1) {

int val = argv[1];

if (argc > 2) val |= (argv[2] << 7);

if (argc > 3) val |= (argv[3] << 14);

analogWriteCallback(argv[0], val);

}

break;

case CAPABILITY\_QUERY:

Firmata.write(START\_SYSEX);

Firmata.write(CAPABILITY\_RESPONSE);

for (byte pin = 0; pin < TOTAL\_PINS; pin++) {

if (IS\_PIN\_DIGITAL(pin)) {

Firmata.write((byte)INPUT);

Firmata.write(1);

Firmata.write((byte)PIN\_MODE\_PULLUP);

Firmata.write(1);

Firmata.write((byte)OUTPUT);

Firmata.write(1);

}

if (IS\_PIN\_ANALOG(pin)) {

Firmata.write(PIN\_MODE\_ANALOG);

Firmata.write(10); // 10 = 10-bit resolution

}

if (IS\_PIN\_PWM(pin)) {

Firmata.write(PIN\_MODE\_PWM);

Firmata.write(DEFAULT\_PWM\_RESOLUTION);

}

if (IS\_PIN\_DIGITAL(pin)) {

Firmata.write(PIN\_MODE\_SERVO);

Firmata.write(14);

}

if (IS\_PIN\_I2C(pin)) {

Firmata.write(PIN\_MODE\_I2C);

Firmata.write(1); // TODO: could assign a number to map to SCL or SDA

}

#ifdef FIRMATA\_SERIAL\_FEATURE

serialFeature.handleCapability(pin);

#endif

Firmata.write(127);

}

Firmata.write(END\_SYSEX);

break;

case PIN\_STATE\_QUERY:

if (argc > 0) {

byte pin = argv[0];

Firmata.write(START\_SYSEX);

Firmata.write(PIN\_STATE\_RESPONSE);

Firmata.write(pin);

if (pin < TOTAL\_PINS) {

Firmata.write(Firmata.getPinMode(pin));

Firmata.write((byte)Firmata.getPinState(pin) & 0x7F);

if (Firmata.getPinState(pin) & 0xFF80) Firmata.write((byte)(Firmata.getPinState(pin) >> 7) & 0x7F);

if (Firmata.getPinState(pin) & 0xC000) Firmata.write((byte)(Firmata.getPinState(pin) >> 14) & 0x7F);

}

Firmata.write(END\_SYSEX);

}

break;

case ANALOG\_MAPPING\_QUERY:

Firmata.write(START\_SYSEX);

Firmata.write(ANALOG\_MAPPING\_RESPONSE);

for (byte pin = 0; pin < TOTAL\_PINS; pin++) {

Firmata.write(IS\_PIN\_ANALOG(pin) ? PIN\_TO\_ANALOG(pin) : 127);

}

Firmata.write(END\_SYSEX);

break;

case SERIAL\_MESSAGE:

#ifdef FIRMATA\_SERIAL\_FEATURE

serialFeature.handleSysex(command, argc, argv);

#endif

break;

}

}

/\*==============================================================================

\* SETUP()

\*============================================================================\*/

void systemResetCallback()

{

isResetting = true;

// initialize a defalt state

// TODO: option to load config from EEPROM instead of default

#ifdef FIRMATA\_SERIAL\_FEATURE

serialFeature.reset();

#endif

if (isI2CEnabled) {

disableI2CPins();

}

for (byte i = 0; i < TOTAL\_PORTS; i++) {

reportPINs[i] = false; // by default, reporting off

portConfigInputs[i] = 0; // until activated

previousPINs[i] = 0;

}

for (byte i = 0; i < TOTAL\_PINS; i++) {

// pins with analog capability default to analog input

// otherwise, pins default to digital output

if (IS\_PIN\_ANALOG(i)) {

// turns off pullup, configures everything

setPinModeCallback(i, PIN\_MODE\_ANALOG);

} else if (IS\_PIN\_DIGITAL(i)) {

// sets the output to 0, configures portConfigInputs

setPinModeCallback(i, OUTPUT);

}

servoPinMap[i] = 255;

}

// by default, do not report any analog inputs

analogInputsToReport = 0;

detachedServoCount = 0;

servoCount = 0;

/\* send digital inputs to set the initial state on the host computer,

\* since once in the loop(), this firmware will only send on change \*/

/\*

TODO: this can never execute, since no pins default to digital input

but it will be needed when/if we support EEPROM stored config

for (byte i=0; i < TOTAL\_PORTS; i++) {

outputPort(i, readPort(i, portConfigInputs[i]), true);

}

\*/

isResetting = false;

}

void setup()

{

Firmata.setFirmwareVersion(FIRMATA\_FIRMWARE\_MAJOR\_VERSION, FIRMATA\_FIRMWARE\_MINOR\_VERSION);

Firmata.attach(ANALOG\_MESSAGE, analogWriteCallback);

Firmata.attach(DIGITAL\_MESSAGE, digitalWriteCallback);

Firmata.attach(REPORT\_ANALOG, reportAnalogCallback);

Firmata.attach(REPORT\_DIGITAL, reportDigitalCallback);

Firmata.attach(SET\_PIN\_MODE, setPinModeCallback);

Firmata.attach(SET\_DIGITAL\_PIN\_VALUE, setPinValueCallback);

Firmata.attach(START\_SYSEX, sysexCallback);

Firmata.attach(SYSTEM\_RESET, systemResetCallback);

// to use a port other than Serial, such as Serial1 on an Arduino Leonardo or Mega,

// Call begin(baud) on the alternate serial port and pass it to Firmata to begin like this:

// Serial1.begin(57600);

// Firmata.begin(Serial1);

// However do not do this if you are using SERIAL\_MESSAGE

Firmata.begin(57600);

while (!Serial) {

; // wait for serial port to connect. Needed for ATmega32u4-based boards and Arduino 101

}

systemResetCallback(); // reset to default config

}

/\*==============================================================================

\* LOOP()

\*============================================================================\*/

void loop()

{

byte pin, analogPin;

/\* DIGITALREAD - as fast as possible, check for changes and output them to the

\* FTDI buffer using Serial.print() \*/

checkDigitalInputs();

/\* STREAMREAD - processing incoming messagse as soon as possible, while still

\* checking digital inputs. \*/

while (Firmata.available())

Firmata.processInput();

// TODO - ensure that Stream buffer doesn't go over 60 bytes

currentMillis = millis();

if (currentMillis - previousMillis > samplingInterval) {

previousMillis += samplingInterval;

/\* ANALOGREAD - do all analogReads() at the configured sampling interval \*/

for (pin = 0; pin < TOTAL\_PINS; pin++) {

if (IS\_PIN\_ANALOG(pin) && Firmata.getPinMode(pin) == PIN\_MODE\_ANALOG) {

analogPin = PIN\_TO\_ANALOG(pin);

if (analogInputsToReport & (1 << analogPin)) {

Firmata.sendAnalog(analogPin, analogRead(analogPin));

}

}

}

// report i2c data for all device with read continuous mode enabled

if (queryIndex > -1) {

for (byte i = 0; i < queryIndex + 1; i++) {

readAndReportData(query[i].addr, query[i].reg, query[i].bytes, query[i].stopTX);

}

}

}

#ifdef FIRMATA\_SERIAL\_FEATURE

serialFeature.update();

#endif

}

MAIN PROGRAM:

import random

import cv2

from time import sleep

from pyfirmata import Arduino, SERVO, OUTPUT

import threading

from ultralytics import YOLO

import torch

port = 'COM6' # Arduino Port Pin

pin = 10 # Signal Pin for Servo

buzzer\_pin = 11 # Signal Pin for Buzzer

board = Arduino(port)

board.digital[pin].mode = SERVO

board.digital[buzzer\_pin].mode = OUTPUT # Set buzzer pin as output

# Open the class list file

with open("utils/coco.txt", "r") as my\_file:

class\_list = my\_file.read().split("\n")

# Generate random colors for class list

detection\_colors = [(random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)) for \_ in range(len(class\_list))]

# Load a pretrained YOLOv8n model

model = YOLO("weights/yolov8n.pt", "v8")

# Capture video from webcam

url='http://192.168.29.40:8080/video'

vid = cv2.VideoCapture(0)

def rotateservo(pin, angle):

board.digital[pin].write(angle)

def buzzer\_control(state):

board.digital[buzzer\_pin].write(state)

def servo\_control(stop\_event):

servo\_position = 0 # Initial servo position

while not stop\_event.is\_set():

if is\_person\_detected and servo\_position != 90:

buzzer\_control(1) # Turn on the buzzer

for i in range(servo\_position, 90):

rotateservo(pin, i)

sleep(0.005) # Adjust the delay to control speed (faster)

servo\_position = i

elif not is\_person\_detected and servo\_position != 0:

buzzer\_control(0) # Turn off the buzzer

for i in range(servo\_position, 0, -1):

rotateservo(pin, i)

sleep(0.005) # Adjust the delay to control speed (faster)

servo\_position = i

sleep(0.1) # Additional delay after each complete movement

is\_person\_detected = False

def object\_detection(stop\_event):

global is\_person\_detected

while not stop\_event.is\_set():

ret, frame = vid.read()

if not ret:

print("Can't receive frame (stream end?). Exiting ...")

break

detect\_params = model.predict(source=[frame], conf=0.5, save=False)

person\_detected = False

for box in detect\_params[0].boxes:

clsID\_cpu = box.cls.cpu()

clsID = clsID\_cpu.numpy()[0]

if clsID == 2: # Check if detected class is person

person\_detected = True

xyxy = box.xyxy.cpu().numpy()

pt1 = (int(xyxy[0][0]), int(xyxy[0][1]))

pt2 = (int(xyxy[0][2]), int(xyxy[0][3]))

cv2.rectangle(frame, pt1, pt2, (0, 255, 0), 2)

font = cv2.FONT\_HERSHEY\_SIMPLEX

cv2.putText(frame, class\_list[int(clsID)], pt1, font, 1, (255, 255, 255), 2, cv2.LINE\_AA)

break

is\_person\_detected = person\_detected

cv2.imshow("ObjectDetection", frame)

key = cv2.waitKey(1)

if key == ord('q'):

stop\_event.set() # Set the event to stop the threads

break

# Create stop events for the threads

stop\_event\_servo = threading.Event()

stop\_event\_detection = threading.Event()

# Start the threads

thread\_servo = threading.Thread(target=servo\_control, args=(stop\_event\_servo,))

thread\_detection = threading.Thread(target=object\_detection, args=(stop\_event\_detection,))

thread\_servo.start()

thread\_detection.start()

# Wait for the threads to finish or until 'q' is pressed

while True:

if not thread\_servo.is\_alive() or not thread\_detection.is\_alive():

break

key = cv2.waitKey(1)

if key == ord('q'):

print("Stop event set by user.")

stop\_event\_servo.set()

stop\_event\_detection.set()

break

# Release video capture

vid.release()

cv2.destroyAllWindows()

Future Scope:

**1.Integration with Smart Transportation Systems**: The project can be expanded to integrate with broader smart transportation initiatives, such as intelligent traffic management systems, to further optimize traffic flow and enhance overall transportation efficiency.

**2. Enhanced Sensor Integration**: Incorporating additional sensors, such as radar or LiDAR, can improve the system's detection capabilities, especially in adverse weather conditions or challenging environments.

**3. Predictive Analytics**:Implementing predictive analytics algorithms can anticipate train movements based on historical data, allowing for proactive gate control adjustments and better coordination with train schedules.

**4. Remote Monitoring and Control**: Enabling remote monitoring and control capabilities can provide railway operators with real-time insights into crossing status and allow for remote adjustments or overrides when necessary.

**5.Collaboration with Vehicle-to-Infrastructure (V2I) Communication**: Integration with V2I communication systems can facilitate communication between vehicles and railway infrastructure, enhancing safety and coordination at railway crossings.

**6.Expansion to Unmanned Railway Crossings**: Adapting the system for use at unmanned railway crossings can help mitigate risks at remote or less frequently used crossings, improving safety in areas with limited human supervision.

**7.AI-driven Optimization**: Leveraging artificial intelligence techniques, such as machine learning, can enable the system to continuously learn and optimize its performance based on real-world data and feedback.

**8.Deployment in Urban Environments:** Expanding the deployment of the system to urban areas with high traffic volumes can help alleviate congestion and improve safety at busy railway crossings.

**9.Global Implementation:** Scaling the project for global implementation can address railway safety challenges on a larger scale, benefiting communities worldwide and promoting standardized safety measures.

**10.Partnerships with Industry Stakeholders:** Collaborating with railway operators, technology providers, and regulatory agencies can foster innovation, support regulatory compliance, and ensure the sustainability and scalability of the project.

Conclusion:

In summary, the railway gate control system using image recognition technology offers a significant advancement in enhancing safety and efficiency at railway crossings. By automating train detection and barrier operation, it minimizes the risk of accidents and optimizes traffic flow. Future enhancements and collaborations hold promise for further improving railway safety and transportation efficiency on a global scale.