Moxie_Invader.ino

For the **Moxie_Invader.ino** file, I'll provide an in-depth explanation, detailing each line of code and its purpose, as well as how it interfaces with other components.

Header Inclusions and Initial Setup

```
#include "UltrasonicSensor.h"
#include "PIDController.h"
#include "MotorAndESC.h"
#include "Encoder.h"
```

These lines include the **header** files for the ultrasonic sensor, PID controller, motor and ESC, and encoder functionalities. By including these files, we make the classes and functions defined in them available to our main program, allowing us to create objects and call methods from these classes.

Defining Constants

```
const int MOTOR_PIN = 11; // The PWM pin connected to the motor
driver.
const int MAX_SIGNAL = 2000; // The maximum signal in microseconds for
the ESC.
const int MIN_SIGNAL_FOR_WORK = 1000; // The minimum signal in
microseconds for the ESC to operate.
const int ENCODER_PIN = 2; // The pin connected to the encoder.
const int CLOCK_SPEED = 1000000; // The clock speed for SPI
communication with the encoder, in Hz.
const int TRIGGER_PIN = 9; // The pin triggering the ultrasonic
sensor.
const int ECHO_PIN = 10; // The pin receiving the echo from the
ultrasonic sensor.
```

These lines define constants used throughout the program. They set up important parameters like pin numbers and signal timings, which are crucial for interfacing with the hardware components correctly. Using const int ensures these values cannot be accidentally modified elsewhere in the code, enhancing reliability.

Object Initialization

```
UltrasonicSensor sensor(TRIGGER_PIN, ECHO_PIN);
PIDController pid(0.5, 0.1, 0.2, 100);
MotorAndESC motorControl(MOTOR_PIN, MAX_SIGNAL, MIN_SIGNAL_FOR_WORK);
```

```
Encoder encoder(ENCODER_PIN, CLOCK_SPEED);
```

Here, we instantiate objects for each of our components. These objects will use the constructors defined in their respective classes to initialize their state with the parameters provided. This step is crucial for setting up the initial conditions under which our components will operate.

Setup Function

```
void setup() {
  Serial.begin(9600);
  encoder.Initialize();
}
```

- `Serial.begin(9600);`: Initializes serial communication at a baud rate of 9600. This enables the Arduino to send data back to the computer over the USB connection, useful for debugging and monitoring sensor values.
- `encoder.Initialize();`: Calls the initialization method for the encoder. This might include setting up SPI communication parameters, ensuring the encoder is ready to provide accurate readings.

Loop Function

```
void loop() {
  float distance = sensor.getDistance();
  Serial.print("Distance: ");
  Serial.println(distance);
  ...
}
```

The `loop()` function contains the code that runs repeatedly. It starts by obtaining the distance measured by the ultrasonic sensor and printing it to the serial monitor. This allows for real-time monitoring of sensor readings.

```
if (distance < 20) {
  pid.setTarget(0);
} else {
  pid.setTarget(100);
}
```

This conditional block adjusts the target for the PID controller based on the distance reading. If an obstacle is detected within 20 cm, it sets the target to 0, effectively stopping the motor. Otherwise, it sets a default target, which could represent a desired speed or position.

```
double currentPosition = encoder.GetPosi();
double controlSignal = pid.calculate(currentPosition);
motorControl.applyControlSignal(controlSignal);
```

These lines read the current position from the encoder and use it to calculate a control signal with the PID algorithm. This signal is then applied to the motor via the motor control object, adjusting the motor's speed or position accordingly.

```
delay(1000);
```

Finally, a delay is introduced to pace the loop execution. This delay can be adjusted based on the application's responsiveness requirements.

Interaction and Data Flow

The program starts by initializing communication protocols and setting initial conditions for each component. In the main loop, sensor readings are taken to inform control decisions. The PID controller computes a control signal based on the current state (as read by the encoder) and a set target. This signal is then used to adjust the motor's behavior, aiming to achieve or maintain the set target. This process repeats, creating a feedback loop that allows the system to respond dynamically to changes in sensor readings and adjust its operation accordingly.
This comprehensive explanation provides a foundation for understanding the **`Moxie_Invader.ino`** file's functionality. Each component's role and how they work together to achieve the desired behavior in your robotic system are outlined, offering a clear picture of the system's operation.

Encoder.h

Let's dissect the **`Encoder.h`** file, providing an in-depth explanation for each component and its role in the system.

Preprocessor Directives

```
#ifndef Encoder_h
#define Encoder_h
```

This code snippet ensures that the **Encoder.h** file is included only once in the compilation process, preventing multiple definitions errors. It checks if **Encoder_h** is not defined; if true, it defines **Encoder_h** and includes the contents of the file.

```
#include "Arduino.h"
```

Includes the main Arduino library, providing access to its standard functions and types. This is necessary for using Arduino-specific functionalities like digital and analog I/O operations, and for the **Encoder** class to interact with the hardware.

Encoder Class Definition

```
class Encoder{
```

Begins the definition of the **Encoder** class. This class is designed to interface with an encoder hardware component, handling its initialization, data reading, and processing.

Public Section

```
Encoder(unsigned int pin, unsigned int ClockSpeed);
```

Constructor declaration for the **Encoder** class. It initializes a new instance with a specific pin for chip selection and a clock speed for SPI communication. These parameters are essential for setting up and communicating with the encoder hardware.

```
float GetPosi();
```

Member function declaration that will return the current position or rotation count of the encoder as a floating-point number. This allows other parts of the program to obtain the encoder's position.

```
void UpdatePosi(uint16_t command);
```

Function to send a command to the encoder and update its position reading. The command parameter allows for different operations or queries to be made to the encoder.

```
void ResetPosi();
```

Resets the encoder's position count to zero. This is useful for recalibrating the encoder's position in relation to a known reference point.

```
void Initialize();
```

Initializes SPI communication or other necessary protocols for the encoder. This method must be called once, typically in the Arduino **`setup()`** function, to prepare the encoder for operation.

```
void interpretResponse(uint16_t response);
```

A method to process or interpret the response from the encoder. This could involve converting raw data into a usable format, such as the position count.

Private Section

```
int _pin; // Private variable for the chip select pin.
int _ClockSpeed; // Private variable for SPI communication clock
speed.
```

Private member variables **`_pin`** and **`_ClockSpeed`** store the chip select pin and clock speed for SPI communication, respectively. These are set during object construction and are used internally by the class to communicate with the encoder.

Preprocessor Endif

```
#endif
```

Ends the conditional inclusion initiated at the beginning of the file. This marks the end of the **`Encoder.h`** file content, ensuring everything within the **`#ifndef`** and **`#endif`** is included only once.

Interaction and Data Flow

The **`Encoder`** class serves as a wrapper around the hardware encoder, abstracting its operations through public methods. It encapsulates the details of initializing the encoder, reading its position, and processing its data. By including **`Encoder.h`** and creating an **`Encoder`** object, other parts of the program can interact with the encoder without needing to manage its low-level details directly. The class's separation of public and private members enforces encapsulation, ensuring that the encoder's internal state can only be modified through its public interface, promoting safer and more reliable code.

Encoder.cpp

Let's break down the **`Encoder.cpp`** source file, detailing each line and how it contributes to the encoder's functionality within your robot's system.

Including Necessary Libraries

```
#include "Encoder.h"
```

This line includes the **`Encoder.h`** header file, linking the **`Encoder`** class definition with its implementation.

```
#include "Arduino.h"
```

Includes the Arduino base library, providing access to fundamental Arduino functions and types necessary for interacting with hardware.

```
#include "SPI.h"
```

Includes the SPI (Serial Peripheral Interface) library for Arduino, enabling SPI communication, which is crucial for interacting with the encoder hardware that communicates over SPI.

Global Variable

```
static bool initialize(false);
```

Defines a **`static`** boolean variable named **`initialize`**, initialized to **`false`**. This variable is used to track whether the SPI system has been initialized, ensuring that **`SPI.begin()`** is called only once, regardless of how many instances of **`Encoder`** are created.

Constructor

```
Encoder::Encoder(unsigned int pin, unsigned int ClockSpeed) {
```

Defines the constructor for the **`Encoder`** class, taking two parameters: a pin number for the chip select **`(CS)`** pin and the clock speed for SPI communication.

```
pinMode(pin, OUTPUT);
```

Sets the mode of the chip select pin to **`OUTPUT`**. This is necessary for controlling the encoder via SPI, as the CS pin signals the encoder when to listen for commands.

```
digitalWrite(pin, HIGH);
```

Initially sets the chip select pin to **HIGH**, indicating that the encoder is not currently being communicated with (SPI devices are often active-low).

```
_pin = pin;
_ClockSpeed = ClockSpeed;
```

Assigns the values of the arguments passed to the constructor to the private member variables **_pin** and **_ClockSpeed**. This stores the chip select pin number and the clock speed for later use in SPI communication.

Initialize Method

```
void Encoder::Initialize() {
```

Defines the **Initialize** method, responsible for setting up SPI communication.

```
if(!initialize) {
```

Checks if SPI has not already been initialized by examining the **initialize** variable.

```
initialize = true;
SPI.begin();
```

If SPI was not already initialized, sets **initialize** to **true** and calls **SPI.begin()** to start SPI communication. This is crucial for ensuring SPI is ready to communicate with the encoder.

UpdatePosi Method

```
void Encoder::UpdatePosi(uint16_t command){
```

Defines the method for updating the encoder's position by sending a command via SPI.
```
digitalWrite(_pin, LOW);
```

Activates the encoder by setting its chip select pin to **LOW**, indicating the start of an SPI communication session.

```
SPI.beginTransaction(SPISettings(_ClockSpeed, MSBFIRST, SPI_MODE1));
```

Begins an SPI transaction with settings specified by **_ClockSpeed**, bit order (MSB first), and SPI mode (mode 1). These settings must match the encoder's requirements for correct communication.

```
uint16_t response = SPI.transfer16(command);
```

Sends the 16-bit command to the encoder and stores the received 16-bit response. The **transfer16** function is used for sending and receiving 16-bit data over SPI.

```
digitalWrite(_pin, HIGH);
```

Deactivates the encoder by setting its chip select pin to **HIGH**, signaling the end of the SPI communication session.

```
SPI.endTransaction();
```

Ends the SPI transaction, ensuring the SPI bus is released for other devices that might use it.

```
interpretResponse(response);
```

Calls the **interpretResponse** method, passing the received response for processing or interpretation.

interpretResponse Method

```
void Encoder::interpretResponse(uint16_t response){
```

Defines the method for interpreting the encoder's response. This might involve converting raw data into a more usable format.

```
posi = static_cast<float>(response);
```

Casts the response to a **float** and assigns it to **posi**, the member variable holding the encoder's position. This conversion is necessary if the raw data from the encoder needs to be in a specific format or unit.

Summary

This file encapsulates the functionality required to communicate with an encoder via SPI, providing methods for initialization, sending commands, and processing responses. Each function contributes to the overall ability to read and manage the encoder's position,

crucial for tasks like monitoring the robot's movements or controlling motors with precision. The use of SPI communication underscores the need for careful management of hardware resources, ensuring that interactions with the encoder are both efficient and accurate.

MotorAndESC.h

Let's dive into the **`MotorAndESC.h`** header file to understand its structure and purpose, ensuring that each line is clearly explained.

Preprocessor Directives

```
#ifndef MotorAndESC_h
#define MotorAndESC_h
```

These lines ensure that the header file is included only once during compilation. If **`MotorAndESC_h`** isn't defined yet, it defines it, preventing multiple inclusions which could lead to redefinition errors.

Including Libraries

```
#include "Arduino.h"
```

Includes the Arduino base library for access to its core functions and types, essential for interacting with Arduino hardware and performing tasks like digital and analog I/O.

```
#include "Servo.h"
```

Includes the Servo library, allowing the code to control servo motors easily. This library provides high-level functions to control the position of the servo motor.

Class Definition

```
class MotorAndESC{
```

Begins the definition of the **`MotorAndESC`** class. This class abstracts the control of motors using an Electronic Speed Controller (ESC) through the Arduino.

Public Section

```
public:
```

This section contains all members of the **`MotorAndESC`** class that can be accessed from outside the class.

Constructor

```
MotorAndESC(unsigned int pin, unsigned int maxSignal, unsigned int
minSignalForWork);
```

Defines the constructor for the class. It takes three parameters: a pin number for controlling the ESC, the maximum signal value (in microseconds) the ESC can handle, and the minimum signal value (in microseconds) required to make the motor work. This setup allows for flexible control over various ESCs and motors.

Method Declaration

```
void applyControlSignal(double controlSignal);
```

Declares a method for applying a control signal to the motor. The method takes a double value, which is intended to be a calculated control signal (likely from a PID controller) that adjusts the speed or position of the motor.

Private Section

```
private:
```

This section declares private member variables. These are only accessible from within the class's own methods, encapsulating the class's internal state.

Member Variables

```
int _pin;
```

Stores the pin number associated with the ESC signal input. This is where the control signal will be sent.

```
unsigned int _maxSignal;
unsigned int _minSignalForWork;
```

Stores the maximum signal value and the minimum working signal value for the ESC, both in microseconds. These are used to ensure that signals sent to the ESC are within its operational range, preventing damage to the motor or ESC and ensuring proper operation.

Servo Object

```
Servo esc;
```

Declares an instance of the **`Servo`** class named **`esc`**. Despite the name **`Servo`**, this object is used to control the ESC based on the Servo library's ability to generate precise pulse-width modulated (PWM) signals, which are required for ESC operation.

Endif for Preprocessor Directive

```
#endif
```

Ends the conditional inclusion started at the beginning. This ensures the file's content is processed only once, avoiding redefinition errors during compilation.

Summary

The **`MotorAndESC.h`** header file defines a class for controlling motors through an ESC with an Arduino, encapsulating details such as the PWM signal generation using the **`Servo`** library. It abstracts away the complexities of ESC signal requirements, allowing for straightforward motor control in the broader robotics project. The use of **`unsigned int`** for signal values emphasizes the non-negative nature of these parameters, aligning with the semantic correctness of their usage.

MotorAndESC.cpp

Let's dissect the **`MotorAndESC.cpp`** source file to understand its implementation details and how it interacts with other components. This source file is critical for managing the interaction between your Arduino project and the motors via an Electronic Speed Controller (ESC).

Including Necessary Libraries

```
#include "Arduino.h"
#include "MotorAndESC.h"
#include "Servo.h"
```

`Arduino.h`: Includes the main Arduino library, enabling access to standard Arduino functions and types.

`MotorAndESC.h`: Includes the header file for this source file, linking the declaration of the **`MotorAndESC`** class and its members.

`Servo.h`: Includes the Servo library, which is utilized to control the ESC through PWM signals, similar to how you would control a servo motor.

Constructor Definition

```
MotorAndESC::MotorAndESC(unsigned int pin, unsigned int maxSignal,
unsigned int minSignalForWork)
  : _pin(pin), _maxSignal(maxSignal),
_minSignalForWork(minSignalForWork) {
  pinMode(pin, OUTPUT);
  esc.attach(pin, maxSignal, maxSignal);
}
```

This constructor initializes a **`MotorAndESC`** object with specified pin numbers for the ESC control and signal range limits.

`_pin(pin), _maxSignal(maxSignal), _minSignalForWork(minSignalForWork)` initializes class member variables with values provided during object construction.

`pinMode(pin, OUTPUT);` configures the specified pin as an output, which is necessary for sending signals to the ESC.

`esc.attach(pin, maxSignal, maxSignal);` attaches the **`esc`** object (a Servo object) to the specified pin. The comment indicates that min/max parameters may not be needed unless your ESC requires specific pulse width boundaries. This setup is prepared for direct control without the min/max range, but you might adjust it based on your ESC specifications.

Applying Control Signal

```
void MotorAndESC::applyControlSignal(double controlSignal) {
    int pwmOutput = map(controlSignal, -1.0, 1.0, _minSignalForWork,
_maxSignal);
    esc.writeMicroseconds(pwmOutput);
}
```

Maps a control signal (from PID output, in this case ranging from -1.0 to 1.0) to the PWM range that the ESC understands **`(_minSignalForWork`** to **`_maxSignal)`**.

`esc.writeMicroseconds(pwmOutput);` sends the mapped signal to the ESC via the pin specified in the constructor, controlling the motor speed or position accordingly.

Commented-out Motor Method

The two versions of the **`Motor(int pwm)`** method are commented out because they have been superseded by the **`applyControlSignal`** method. The old methods show alternative ways of mapping PWM signals for motor control, potentially for reverse and forward control. However, in this revised design, **`applyControlSignal`** offers a more straightforward and flexible approach to applying PID control outputs directly.

Summary

`MotorAndESC.cpp` implements the **`MotorAndESC`** class's functionality, primarily focusing on initializing the ESC control setup and applying control signals derived from PID computations. The transition to using the **`applyControlSignal`** method for controlling the motor through the ESC demonstrates an evolution towards integrating PID control outputs more directly and efficiently. This approach simplifies the motor control logic, making it easier to adapt to different control scenarios and ensuring a seamless interface with the rest of the robotic system's components.

UltrasonicSensor.h

Exploring the **`UltrasonicSensor.h`** file provides insight into how ultrasonic sensors are integrated into your Arduino project for distance measurement. This header file outlines the structure and functionality of the **`UltrasonicSensor`** class.

File Guard

```
#ifndef ULTRASONICSENSOR_H
#define ULTRASONICSENSOR_H
```

These lines prevent the header file from being included multiple times, which could cause compilation errors. If **`ULTRASONICSENSOR_H`** is not defined yet, it gets defined, ensuring the rest of the file is processed only once.

Including Arduino Library

```
#include "Arduino.h"
```

Includes the main Arduino library, granting access to standard Arduino functions and types, essential for interacting with the hardware.

Class Definition

```
class UltrasonicSensor {
```

Begins the definition of the **`UltrasonicSensor`** class, encapsulating all functionality related to ultrasonic sensing.

Public Section

```
public:
    UltrasonicSensor(int triggerPin, int echoPin);
    float getDistance();
```

`UltrasonicSensor(int triggerPin, int echoPin);`: Constructor that initializes an ultrasonic sensor with specified trigger and echo pins. The trigger pin is used to send the ultrasonic pulse, and the echo pin reads the pulse's return signal.

`float getDistance();`: Method to measure and return the distance to the nearest obstacle in front of the sensor. It triggers an ultrasonic pulse and calculates the distance based on the time taken for the pulse to return.

Private Member Variables

```
private:
    int triggerPin;
    int echoPin;
    long duration;
    float distance;
```

`int triggerPin;` and **`int echoPin;`**: Store the pin numbers connected to the sensor's trigger and echo pins, respectively.

`long duration;`: Temporarily holds the time (in microseconds) it takes for the ultrasonic pulse to return to the sensor. This value is used to calculate the distance.

`float distance;`: Holds the calculated distance from the sensor to an obstacle, determined by the **`getDistance()`** method.

End of File Guard

```
#endif // ULTRASONICSENSOR_H
```

Marks the end of the conditional preprocessor directive. If the file has already been included, the code between **`#ifndef ULTRASONICSENSOR_H`** and **`#endif`** is skipped to avoid duplication.

Summary

The **`UltrasonicSensor.h`** file defines a class that encapsulates the functionality necessary for interacting with an ultrasonic sensor. It provides a clear interface for initializing the sensor with specific pins and for measuring distances. The separation of public methods and private variables ensures encapsulation and abstraction, key principles of object-oriented programming. This design allows easy integration and interaction with other parts of the robotics system, promoting modularity and code reuse.

UltrasonicSensor.cpp

Delving into the **`UltrasonicSensor.cpp`** file unveils the practical application of ultrasonic sensors in measuring distances through sound waves. This source file is a concrete implementation complementing the definitions in **`UltrasonicSensor.h`**.

Including Header Files

```
#include "UltrasonicSensor.h"
#include "Arduino.h"
```

These lines include the class definition from **`UltrasonicSensor.h`** and the core Arduino functionality from **`Arduino.h`,** respectively, allowing the use of Arduino-specific functions and access to the class definition.

Constructor Implementation

```
UltrasonicSensor::UltrasonicSensor(int triggerPin, int echoPin)
    : triggerPin(triggerPin), echoPin(echoPin) {
    pinMode(triggerPin, OUTPUT); // Set the trigger pin as an output
    pinMode(echoPin, INPUT); // Set the echo pin as an input
}
```

This is the constructor for the **`UltrasonicSensor`** class, initializing the object with specific trigger and echo pins. The use of an initializer list :
`triggerPin(triggerPin)`, **`echoPin(echoPin)`** directly assigns the passed pin numbers to the class's member variables.

`pinMode(triggerPin, OUTPUT);` configures the trigger pin as an output. This pin will emit ultrasonic pulses.

`pinMode(echoPin, INPUT);` sets the echo pin as an input. This pin listens for the return signal, or echo, from the ultrasonic pulse.

Distance Measurement

```
float UltrasonicSensor::getDistance() {
    digitalWrite(triggerPin, LOW); // Clear the trigger
    delayMicroseconds(2);
    digitalWrite(triggerPin, HIGH); // Sets the trigger on HIGH state
for 10 micro seconds
    delayMicroseconds(10);
    digitalWrite(triggerPin, LOW); // Resets the trigger to LOW
    duration = pulseIn(echoPin, HIGH); // Reads the echo pin, returns
```

```
the sound wave travel time in microseconds
    distance = duration * 0.034 / 2; // Calculating the distance
    return distance;
}
```

The sequence starting with **`digitalWrite(triggerPin, LOW);`** ensures a clean start before emitting a pulse by setting the trigger pin low briefly.

The pulse is emitted with **`digitalWrite(triggerPin, HIGH);`** for 10 microseconds, a standard duration for ultrasonic sensors to generate an effective signal.

`duration = pulseIn(echoPin, HIGH);` measures the time in microseconds between sending the pulse and receiving the echo, effectively capturing how long the sound wave travels before bouncing back.

The calculation **`distance = duration * 0.034 / 2;`** determines the distance to an object. The speed of sound is approximately **`340 m/s`** or **`0.034 cm/µs`**. The round trip for the sound wave (to the object and back) means the actual distance to the object is half the computed value.

return distance; outputs the calculated distance, making it accessible to other parts of your program.

Data Flow and Interaction

The **`UltrasonicSensor`** class encapsulates the functionality necessary for distance measurement using ultrasonic waves. Through the constructor, it initializes hardware pins with specific roles (trigger and echo) and sets them appropriately for input and output operations.

The **`getDistance`** method implements a sequence to emit an ultrasonic pulse and listen for its echo, calculating the distance based on the time taken for the echo to return. This method demonstrates a direct interaction with the hardware through digital pin manipulation and timing control to achieve its functionality.

This architecture allows the **`UltrasonicSensor`** class to be easily integrated into broader applications, providing a clear interface **`(getDistance())`** for obtaining distance measurements. The encapsulation of ultrasonic sensing in a class abstracts away the complexities of direct hardware manipulation, promoting code reuse and modularity.

PID.h

The **PID.h** header file encapsulates the declaration of the **PIDController** class, which is a fundamental component for implementing PID (Proportional-Integral-Derivative) control. This file lays out the blueprint for how a PID controller behaves and interacts with other parts of your program. Let's dissect this file to understand its structure and purpose.

File Description and Pragma Once

```
// Information from:
https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID
```

This comment provides a reference to an external resource for readers who want to learn more about PID control theory. It's a good practice to include such references for complex algorithms.

```
#ifndef PIDCONTROLLER_H
#define PIDCONTROLLER_H
```

These lines are known as "include guards". They prevent the file from being included multiple times in the same compilation unit, which could lead to redefinition errors. **PIDCONTROLLER_H** is a unique identifier that ensures the content of this file is only included once.

Class Declaration

```
class PIDController {
```

Begins the declaration of the **PIDController** class. This class encapsulates the data and methods necessary for implementing a PID control algorithm.

Public Section

```
public:
    PIDController(double kp, double ki, double kd, double target);
    double calculate(double measured_value);
    void setTarget(double target);
    void reset();
```

`The public:` specifier denotes that the following members can be accessed from outside the class.

`PIDController(double kp, double ki, double kd, double target);` is the constructor, initializing the PID controller with specific gains **`(kp, ki, kd)`** and a target value.

`double calculate(double measured_value);` defines a method for calculating the control signal based on a measured value and the internal state of the PID controller.

`void setTarget(double target);` allows for dynamically changing the target value the PID controller aims to achieve.

`void reset();` resets the internal state of the PID controller, useful for reinitializing the control process.

Private Section

```
private:
    double kp, ki, kd; // Proportional, Integral, and Derivative gains
    double target; // Target value
    double integral; // Integral term
    double prev_error; // Previous error
};
```

`The private:` specifier restricts access to the following members from outside the class. This encapsulation ensures the internal state of the PID controller can only be modified through its public interface.

`double kp, ki, kd;` stores the proportional, integral, and derivative gains, respectively. These parameters define how aggressively the PID controller responds to the error.

`double target;` holds the desired value the controller aims to reach.

`double integral;` accumulates the error over time, allowing the integral term to correct steady-state errors.

`double prev_error;` stores the error from the previous calculation step, necessary for computing the derivative term.

End of Class and Include Guard

```
#endif // PIDCONTROLLER_H
```

Marks the end of the include guard. This ensures that the content within the guards is only processed once per compilation unit, avoiding duplicate declarations.

Data Flow and Interaction

The **`PIDController`** class provides a modular component for implementing PID control in various applications, from robotics to temperature control. By adjusting the **`kp, ki, kd`** gains, and **`target`**, users can tailor the controller's response to their specific needs.

The separation of interface (**`public`** methods) from implementation (**`private`** members) follows best practices in object-oriented design, promoting encapsulation and modularity. This design allows other parts of the program to interact with the PID controller without needing to understand or interfere with its internal workings.

This header file lays the foundation for the actual PID control logic implemented in the corresponding **`.cpp`** file, establishing a clear contract between the PID controller's capabilities and the expectations of its users.

PID.cpp

The **`PID.cpp`** source file implements the functionality declared in **`PID.h`** for a PID (Proportional-Integral-Derivative) controller. This file is crucial for controlling systems by adjusting their output based on the error between a desired setpoint and the current value. Let's dissect it to understand how it operates and integrates with other components.

Include Directives

```
#include "PIDController.h"
```

This line includes the header file where the **`PIDController`** class is declared, ensuring this source file has access to the class definition and can implement its methods.

```
//#include <iostream> //(#include <iostream> is removed; not needed
for Arduino)
```

Initially intended for debugging or console output, this line is commented out because the Arduino environment does not support **`iostream`**. It's a remnant from a development phase where logging or output to a standard console might have been useful.

Constructor

```
PIDController::PIDController(double kp, double ki, double kd, double
target)
    : kp(kp), ki(ki), kd(kd), target(target), integral(0),
prev_error(0) {}
```

This is the constructor of the **`PIDController`** class, initializing the PID gains **`(kp, ki, kd)`**, the target value, and setting the initial values of the integral and previous error to zero. The use of an initializer list **`(: kp(kp), ki(ki), ...)`** is an efficient way to assign values to the class's member variables.

calculate Method

```
double PIDController::calculate(double measured_value) {
    double error = target - measured_value; // Calculate the error
between target and measured value
    integral += error; // Update the integral term
    double derivative = error - prev_error; // Calculate the
derivative term
    prev_error = error; // Update the previous error
    double control_signal = kp * error + ki * integral + kd *
derivative; // Calculate the control signal using PID formula
    return control_signal; // Return the control signal
}
```

Implements the PID control algorithm. It calculates the control signal based on the difference between the target setpoint and the measured value **`(error)`**, the sum of past errors **`(integral)`**, and the change in error **`(derivative)`**. This method demonstrates the core of PID control: adjusting the output **`(control_signal)`** to minimize the error.

setTarget Method

```
void PIDController::setTarget(double newTarget) {
    target = newTarget; // Update the target with the new value
}
```

Allows external modification of the target setpoint, providing flexibility to dynamically change the desired output of the controlled system.

reset Method

```
void PIDController::reset() {
    integral = 0;
    prev_error = 0;
}
```

Resets the integral and previous error to zero. This is useful for reinitializing the controller under new conditions or after a significant change in the setpoint, ensuring past errors do not undesirably influence future outputs.

Data Flow and Interaction

The PID controller's **`calculate`** function integrates error computation, integral update, derivative calculation, and control signal generation into a cohesive unit, providing a mechanism to achieve stable control over a system.

The **`setTarget`** and **`reset`** functions offer control over the controller's state, enabling adaptability to changing conditions.

This source file, in conjunction with its header, offers a modular PID control solution that can be easily integrated into broader systems, such as robotics or process control, demonstrating encapsulation and separation of concerns in software design.

PIDMainSimulationFunction(DONOTUSEONARDUINO).cpp

The **`PIDMainSimulationFunction.cpp`** is a standalone file designed for testing the functionality of the PID control system in a simulated environment. It's separate from the main Arduino codebase and is not intended for deployment in the final embedded system. Let's break down the file to understand its structure and purpose.

Include Directives

```
#include "PIDController.h"
```

This line includes the **`PIDController`** class definition, making the PID control functionalities accessible for simulation purposes.

```
#include <iostream>
```

Includes the C++ standard library's input/output stream header, allowing the program to perform console input and output operations. This is useful for displaying the simulation results to the screen.

Main Function

```
int main() {
```

The entry point of the program. In C++, main is the starting point of execution for any standalone program.

PID Controller Initialization

```
   PIDController controller(0.5, 0.1, 0.2, 100); // Example gains and
target
```

Creates an instance of the **`PIDController`** class named **`controller`**, initializing it with specific gains **`(kp=0.5, ki=0.1, kd=0.2)`** and a target setpoint of **`100`**. These values are examples and should be adjusted based on the system being controlled and the desired response characteristics.

Simulation Loop

```
   for (int i = 0; i < 10; ++i) {
        double measured_value = 90 + i * 5; // Simulated measured
value
        double control_signal = controller.calculate(measured_value);
// Calculate control signal
        std::cout << "Control Signal: " << control_signal <<
std::endl; // Output control signal
    }
```

This loop simulates ten iterations of control signal calculation, where **`measured_value`** represents a series of simulated sensor readings (starting from **`90`** and increasing by **`5`** in

each iteration). For each iteration, it calculates the control signal using the PID controller's **`calculate`** method based on the difference between the target and the current **`measured_value`**.

Outputs the calculated control signal to the console using **`std::cout`**, providing a straightforward way to observe the controller's behavior in response to changing inputs.
Program Exit

```
    return 0;
```

Signals the successful completion of the program. In C++ programs, returning **`0`** from the **`main`** function indicates that the program executed successfully without errors.

Application for Test Simulations

This simulation file is invaluable during the development phase, allowing developers to test and tweak the PID controller's parameters without the need for deploying code to the physical hardware. It helps in understanding how the control signal responds to changes in measured values, facilitating the tuning of PID parameters **`(kp, ki, kd)`** to achieve desired control behavior.

Integration with Project

While this file is not meant for inclusion in the final embedded application running on Arduino, it plays a critical role in the initial testing and parameter tuning of the PID control logic. Developers can run this simulation on their development machines to observe the behavior of the PID algorithm under various conditions, aiding in the optimization process before implementing the control logic on the actual hardware.

Let's break down how each component interacts within your robotics project. The core of the system revolves around managing motor control through PID algorithms, utilizing encoder feedback for precision, and employing ultrasonic sensors for environmental

awareness. Here's a comprehensive visual explanation of the interaction between the various files:

High-Level Overview:

- **Moxie_Invader.ino**: This is the main Arduino sketch that orchestrates the entire operation. It initializes all hardware components, including the motors (via ESC), encoder, and ultrasonic sensors. It periodically reads sensor data, processes it through the PID controller, and adjusts the motor speed accordingly.
- **Encoder.h/cpp**: These files define and implement the `Encoder` class, responsible for reading the encoder attached to the motors. This feedback is crucial for the PID controller to adjust the motor's speed and direction accurately.
- **MotorAndESC.h/cpp**: The `MotorAndESC` class defined in these files controls the motor's speed and direction using an Electronic Speed Controller (ESC). It receives a control signal from the PID controller and applies it to the motor.
- **UltrasonicSensor.h/cpp**: These files contain the `UltrasonicSensor` class for measuring distances using ultrasonic waves. This sensory information can be used to navigate or avoid obstacles.
- **PID.h/cpp**: The PID controller's implementation resides in these files. It calculates the appropriate control signal to achieve the desired setpoint based on the feedback from the encoder.
- **PIDMainSimulationFunction.cpp**: This standalone file is used for simulating the PID controller's behavior on a computer without the need for physical hardware. It's useful for testing and tuning the PID parameters.

Detailed Data Flow and Interactions:

- **Initialization (Setup in Moxie_Invader.ino)**:
  - Initializes serial communication for debugging purposes.
  - Instantiates the `Encoder`, `MotorAndESC`, `UltrasonicSensor`, and `PIDController` objects with appropriate parameters (pins, speed limits, PID gains, etc.).
  - Calls initialization methods where necessary (e.g., `encoder.Initialize()`).
- **Main Loop (Moxie_Invader.ino)**:
  - Reads distance from the `UltrasonicSensor` object to detect obstacles.
  - Gets the current position or speed from the `Encoder` object.
  - Passes this feedback to the `PIDController`, which calculates a control signal based on the difference between the desired setpoint and the feedback.
  - Applies the PID control signal to the motor through the `MotorAndESC` object.
  - Repeats this process, continuously adjusting the motor's behavior based on sensor feedback and the PID algorithm's output.
- **Feedback and Control (PID.h/cpp and Encoder.h/cpp)**:
  - The PID controller uses feedback from the encoder to calculate errors (the difference between the target setpoint and the current value).

- It then computes a control signal to minimize this error, adjusting the motor's operation to reach and maintain the setpoint.

- **Motor Control (MotorAndESC.h/cpp)**:
  - Receives the control signal from the PID controller.
  - Translates this signal into a PWM signal that is sent to the ESC, which controls the motor's speed and direction.
- **Environment Sensing (UltrasonicSensor.h/cpp)**:
  - Continuously measures the distance to nearby objects, providing data that can be used to avoid collisions or navigate through the environment.
- **Simulation (PIDMainSimulationFunction.cpp)**:
  - Although not part of the embedded system, this file allows for testing the PID algorithm in isolation, simulating different feedback values and observing the control signal output.

Visualization:

Consider a flowchart where **`Moxie_Invader.ino`** sits at the top, initiating and controlling the flow of operations. Arrows lead from it to the **`Encoder`**, **`MotorAndESC`**, and **`UltrasonicSensor`** components, indicating the setup and continuous polling of data. Another significant pathway leads from **`Moxie_Invader.ino`** to **`PID.cpp`**, symbolizing the control loop where feedback from **`Encoder`** influences PID calculations, which in turn dictates the behavior of **`MotorAndESC`**. This creates a closed loop, central to the robot's responsive and controlled movement.

This detailed explanation and visualization should provide a clear understanding of how each component of your robotics project interacts and the overall system architecture.