```c
#include <stdint.h>

#include <stdbool.h>


// Constants for motor control

#define MAX_PWM_OUTPUT 255

#define MAX_SPEED 100.0 // Maximum speed in units per second

#define MAX_ACCELERATION 5.0 // Maximum acceleration in units per second^2

#define MAX_DECELERATION 5.0 // Maximum deceleration in units per second^2

#define ENCODER_TICKS_PER_REV 1000 // Example encoder ticks per revolution

#define TIMESTAMP_INTERVAL_MS 10 // Sample interval for speed calculation


// PID Control Structure

typedef struct {

   float Kp;

   float Ki;

   float Kd;

   float set_point;

   float last_error;

   float integral;

} PIDControl;


// Function Prototypes

void PID_init(PIDControl *pid, float Kp, float Ki, float Kd);

float PID_calculate(PIDControl *pid, float current);

void control_motor_x(float output);
```

```
void control_motor_y(float output);
void control_motor_z(float output);
float read_encoder_x();
float read_encoder_y();
float read_encoder_z();
void update_speed();
void apply_acceleration_control(float* current_speed, float target_speed);

// Speed control function prototypes
void set_pwm_x(int32_t output);
void set_pwm_y(int32_t output);
void set_pwm_z(int32_t output);

// Motor and Encoder Variables
volatile int32_t encoder_x_count = 0;
volatile int32_t encoder_y_count = 0;
volatile int32_t encoder_z_count = 0;

// PID Instances for Position and Speed
PIDControl pid_position_x, pid_speed_x;
PIDControl pid_position_y, pid_speed_y;
PIDControl pid_position_z, pid_speed_z;

// Variables for speed tracking
float speed_x = 0.0;
```

```
float speed_y = 0.0;

float speed_z = 0.0;

float last_position_x = 0.0;

float last_position_y = 0.0;

float last_position_z = 0.0;


// Target speeds

float target_speed_x = 0.0;

float target_speed_y = 0.0;

float target_speed_z = 0.0;


void PID_init(PIDControl *pid, float Kp, float Ki, float Kd) {

    pid->Kp = Kp;

    pid->Ki = Ki;

    pid->Kd = Kd;

    pid->set_point = 0.0;

    pid->last_error = 0.0;

    pid->integral = 0.0;

}


float PID_calculate(PIDControl *pid, float current) {

    float error = pid->set_point - current;

    pid->integral += error;

    float derivative = error - pid->last_error;
```

```
    float output = (pid->Kp * error) + (pid->Ki * pid->integral) + (pid->Kd * derivative);
    pid->last_error = error;

    return output;
}

// This function calculates current speed based on encoder changes over time
void update_speed() {
    static uint32_t last_time = 0; // timestamp from the last speed update
    uint32_t current_time = get_time(); // function to get current timestamp in ms
    if (current_time - last_time >= TIMESTAMP_INTERVAL_MS) {
        float delta_time = (current_time - last_time) / 1000.0; // Convert time to seconds
        speed_x = (encoder_x_count - last_position_x) / ENCODER_TICKS_PER_REV / delta_time;
        speed_y = (encoder_y_count - last_position_y) / ENCODER_TICKS_PER_REV / delta_time;
        speed_z = (encoder_z_count - last_position_z) / ENCODER_TICKS_PER_REV / delta_time;

        last_position_x = encoder_x_count;
        last_position_y = encoder_y_count;
        last_position_z = encoder_z_count;

        last_time = current_time;
    }
}

// Function to apply acceleration control
```

```
void apply_acceleration_control(float* current_speed, float target_speed) {
    if (target_speed > *current_speed) {
        // Accelerating
        float potential_speed = *current_speed + MAX_ACCELERATION * (TIMESTAMP_INTERVAL_MS /
1000.0);
        *current_speed = (potential_speed < target_speed) ? potential_speed : target_speed;
    } else {
        // Decelerating
        float potential_speed = *current_speed - MAX_DECELERATION * (TIMESTAMP_INTERVAL_MS /
1000.0);
        *current_speed = (potential_speed > target_speed) ? potential_speed : target_speed;
    }
}

void control_motor_x(float output) {
    // Limit output to the range of PWM
    if (output > MAX_PWM_OUTPUT) {
        output = MAX_PWM_OUTPUT;
    } else if (output < -MAX_PWM_OUTPUT) {
        output = -MAX_PWM_OUTPUT;
    }
    set_pwm_x(output);
}

void control_motor_y(float output) {
    // Limit output to the range of PWM
```

```
    if (output > MAX_PWM_OUTPUT) {
        output = MAX_PWM_OUTPUT;
    } else if (output < -MAX_PWM_OUTPUT) {
        output = -MAX_PWM_OUTPUT;
    }
    set_pwm_y(output);
}

void control_motor_z(float output) {
    // Limit output to the range of PWM
    if (output > MAX_PWM_OUTPUT) {
        output = MAX_PWM_OUTPUT;
    } else if (output < -MAX_PWM_OUTPUT) {
        output = -MAX_PWM_OUTPUT;
    }
    set_pwm_z(output);
}

float read_encoder_x() {
    return (float)encoder_x_count; // Assuming encoder counts directly to position
}

float read_encoder_y() {
    return (float)encoder_y_count; // Assuming encoder counts directly to position
}
```

```c
float read_encoder_z() {
    return (float)encoder_z_count; // Assuming encoder counts directly to position
}

// Sample function to simulate setting PWM for motor drivers
void set_pwm_x(int32_t output) {
    // Implementation to set PWM for motor X
}

void set_pwm_y(int32_t output) {
    // Implementation to set PWM for motor Y
}

void set_pwm_z(int32_t output) {
    // Implementation to set PWM for motor Z
}

// Main function
int main(void) {
    // Initialize PID controllers for Position and Speed for X, Y, Z
    PID_init(&pid_position_x, 1.0f, 0.01f, 0.1f);
    PID_init(&pid_speed_x, 0.5f, 0.01f, 0.1f);

    PID_init(&pid_position_y, 1.0f, 0.01f, 0.1f);
```

```c
PID_init(&pid_speed_y, 0.5f, 0.01f, 0.1f);

PID_init(&pid_position_z, 1.0f, 0.01f, 0.1f);
PID_init(&pid_speed_z, 0.5f, 0.01f, 0.1f);

while (1) {
    // Update speed every few milliseconds
    update_speed();

    float current_position_x = read_encoder_x();
    float current_position_y = read_encoder_y();
    float current_position_z = read_encoder_z();

    // Fetch or set desired target positions & speeds here
    pid_position_x.set_point = /* desired X position */;
    target_speed_x = /* desired X speed */; // Target speed for X
    pid_speed_x.set_point = target_speed_x; // Set target speed for PID control

    pid_position_y.set_point = /* desired Y position */;
    target_speed_y = /* desired Y speed */; // Target speed for Y
    pid_speed_y.set_point = target_speed_y; // Set target speed for PID control

    pid_position_z.set_point = /* desired Z position */;
    target_speed_z = /* desired Z speed */; // Target speed for Z
    pid_speed_z.set_point = target_speed_z; // Set target speed for PID control
```

```
        // Calculate PID outputs for position and speed
        float pid_output_x_position = PID_calculate(&pid_position_x, current_position_x);
        float pid_output_y_position = PID_calculate(&pid_position_y, current_position_y);
        float pid_output_z_position = PID_calculate(&pid_position_z, current_position_z);

        // Apply acceleration control to current speeds
        apply_acceleration_control(&speed_x, target_speed_x);
        apply_acceleration_control(&speed_y, target_speed_y);
        apply_acceleration_control(&speed_z, target_speed_z);

        // Control motors based on PID outputs and current speeds
        control_motor_x(pid_output_x_position + PID_calculate(&pid_speed_x, speed_x));
        control_motor_y(pid_output_y_position + PID_calculate(&pid_speed_y, speed_y));
        control_motor_z(pid_output_z_position + PID_calculate(&pid_speed_z, speed_z));
    }
}
```