**Corrected Source Files**

**adcs controller program-**

#include "adcs\_controller.h"

#include <cmath>

#include <cstring>

#include <algorithm>

ADCSController::ADCSController(float kp\_detumble, float kp\_point, float kd\_point)

    : kp\_detumble\_(kp\_detumble), kp\_point\_(kp\_point), kd\_point\_(kd\_point),

      max\_torque\_(0.1f), first\_mag\_reading\_(true) {

    // FIXED: Now using array of pointers correctly

    for (int i = 0; i < 3; ++i) {

        pid\_controllers\_[i] = new PIDController(kp\_point, 0.01f, kd\_point);

        pid\_controllers\_[i]->setLimits(-max\_torque\_, max\_torque\_);

    }

    // FIXED: Using proper array with std::memset

    std::memset(previous\_mag\_, 0, sizeof(previous\_mag\_));

}

ADCSController::~ADCSController() {

    for (int i = 0; i < 3; ++i) {

        delete pid\_controllers\_[i];

    }

}

void ADCSController::computeControl(const float\* gyro\_rates, const float\* magnetometer,

                                  float sun\_angle, float\* wheel\_torques,

                                  float\* magnetorquer, int control\_mode) {

    // FIXED: Using pointers for memset

    std::memset(wheel\_torques, 0, 3 \* sizeof(float));

    std::memset(magnetorquer, 0, 3 \* sizeof(float));

    switch (control\_mode) {

        case 0: // Safe Mode

            safeMode(wheel\_torques, magnetorquer);

            break;

        case 1: // Detumble Mode

            detumbleControl(gyro\_rates, magnetometer, magnetorquer);

            break;

        case 2: // Pointing Mode

            pointingControl(gyro\_rates, wheel\_torques);

            break;

        case 3: // Science Mode

            scienceControl(gyro\_rates, wheel\_torques);

            break;

        default:

            safeMode(wheel\_torques, magnetorquer);

            break;

    }

}

void ADCSController::detumbleControl(const float\* gyro\_rates, const float\* magnetometer,

                                   float\* magnetorquer) {

    if (first\_mag\_reading\_) {

        // FIXED: Using std::memcpy with proper size

        std::memcpy(previous\_mag\_, magnetometer, 3 \* sizeof(float));

        first\_mag\_reading\_ = false;

        return;

    }

    float dt = 0.1f;  // 10 Hz control rate

    for (int i = 0; i < 3; ++i) {

        // Compute magnetic field derivative

        float b\_dot = (magnetometer[i] - previous\_mag\_[i]) / dt;

        // B-dot control law

        magnetorquer[i] = -kp\_detumble\_ \* b\_dot;

        // FIXED: Using array indexing properly

        magnetorquer[i] = std::max(-1.0f, std::min(1.0f, magnetorquer[i]));

        previous\_mag\_[i] = magnetometer[i];

    }

}

void ADCSController::pointingControl(const float\* gyro\_rates, float\* wheel\_torques) {

    float dt = 0.1f;  // 10 Hz control rate

    for (int i = 0; i < 3; ++i) {

        // FIXED: Using array indexing properly

        wheel\_torques[i] = pid\_controllers\_[i]->compute(0.0f, gyro\_rates[i], dt);

    }

}

void ADCSController::scienceControl(const float\* gyro\_rates, float\* wheel\_torques) {

    // Use more precise control for science operations

    float dt = 0.1f;

    for (int i = 0; i < 3; ++i) {

        // Tighter control with same target (zero rates)

        float torque = pid\_controllers\_[i]->compute(0.0f, gyro\_rates[i], dt);

        // Scale down for more precise control

        wheel\_torques[i] = torque \* 0.5f;

        // FIXED: Using array indexing properly

        wheel\_torques[i] = std::max(-0.05f, std::min(0.05f, wheel\_torques[i]));

    }

}

void ADCSController::safeMode(float\* wheel\_torques, float\* magnetorquer) {

    // FIXED: Using pointers for memset

    std::memset(wheel\_torques, 0, 3 \* sizeof(float));

    std::memset(magnetorquer, 0, 3 \* sizeof(float));

}

void ADCSController::setGains(float kp\_detumble, float kp\_point, float kd\_point) {

    kp\_detumble\_ = kp\_detumble;

    kp\_point\_ = kp\_point;

    kd\_point\_ = kd\_point;

    // Update PID controllers

    for (int i = 0; i < 3; ++i) {

        delete pid\_controllers\_[i];

        pid\_controllers\_[i] = new PIDController(kp\_point, 0.01f, kd\_point);

        pid\_controllers\_[i]->setLimits(-max\_torque\_, max\_torque\_);

    }

}

void ADCSController::setMaxTorque(float max\_torque) {

    max\_torque\_ = max\_torque;

    for (int i = 0; i < 3; ++i) {

        pid\_controllers\_[i]->setLimits(-max\_torque\_, max\_torque\_);

    }

}

**microcontroller.h program :-**

#include "microcontroller.h"

#include "adcs\_controller.h"

#include <iostream>

#include <cmath>

#include <cstring>

#include <algorithm>

Microcontroller::Microcontroller()

    : control\_mode\_(0), fault\_flags\_(0), control\_cycles\_(0), uptime\_seconds\_(0),

      fault\_threshold\_(0.5f), max\_torque\_(0.1f), fault\_count\_(0), consecutive\_faults\_(0) {

    adcs\_controller\_ = new ADCSController(0.1f, 0.05f, 0.01f);

    std::memset(gyro\_rates\_, 0, sizeof(gyro\_rates\_));

    std::memset(magnetometer\_, 0, sizeof(magnetometer\_));

    std::memset(wheel\_torques\_, 0, sizeof(wheel\_torques\_));

    std::memset(magnetorquer\_, 0, sizeof(magnetorquer\_));

    sun\_angle\_ = 0.0f;

    std::cout << "Microcontroller initialized - Safe Mode" << std::endl;

}

void Microcontroller::processSensorData(const SensorData& sensor\_data) {

    if (!sensor\_data.valid) {

        consecutive\_faults\_++;

        if (consecutive\_faults\_ > 5) {

            control\_mode\_ = 0;

        }

        // FIXED: reset actuators when sensor data invalid

        std::memset(wheel\_torques\_, 0, sizeof(wheel\_torques\_));

        std::memset(magnetorquer\_, 0, sizeof(magnetorquer\_));

        return;

    }

    for (int i = 0; i < 3; ++i) {

        gyro\_rates\_[i] = sensor\_data.gyro[i];

        magnetometer\_[i] = sensor\_data.magnetometer[i];

    }

    sun\_angle\_ = sensor\_data.sun\_angle;

    consecutive\_faults\_ = 0;

    performFaultDetection();

    adcs\_controller\_->computeControl(gyro\_rates\_, magnetometer\_, sun\_angle\_,

                                     wheel\_torques\_, magnetorquer\_, control\_mode\_);

    for (int i = 0; i < 3; ++i) {

        wheel\_torques\_[i] = std::max(-max\_torque\_, std::min(max\_torque\_, wheel\_torques\_[i]));

        magnetorquer\_[i] = std::max(-1.0f, std::min(1.0f, magnetorquer\_[i]));

    }

    control\_cycles\_++;

    updateHealthMonitoring();

}

void Microcontroller::getActuatorCommands(ActuatorCommands& commands) {

    for (int i = 0; i < 3; ++i) {

        commands.wheel\_torques[i] = wheel\_torques\_[i];

        commands.magnetorquer[i] = magnetorquer\_[i];

    }

    commands.timestamp = control\_cycles\_;

}

void Microcontroller::setControlMode(int mode) {

    if (mode >= 0 && mode <= 3) {

        control\_mode\_ = mode;

        std::cout << "Control mode set to: " << mode << std::endl;

    }

}

void Microcontroller::setFaultThreshold(float threshold) {

    if (threshold > 0.0f) {

        fault\_threshold\_ = threshold;

    }

}

void Microcontroller::performFaultDetection() {

    fault\_flags\_ = 0;

    for (int i = 0; i < 3; ++i) {

        if (std::abs(gyro\_rates\_[i]) > fault\_threshold\_) {

            fault\_flags\_ |= (1 << i);

        }

    }

    bool sensor\_fault = false;

    for (int i = 0; i < 3; ++i) {

        if (std::isnan(gyro\_rates\_[i]) || std::isinf(gyro\_rates\_[i]) ||

            std::isnan(magnetometer\_[i]) || std::isinf(magnetometer\_[i])) {

            sensor\_fault = true;

            break;

        }

    }

    if (sensor\_fault) {

        fault\_flags\_ |= 0x08;

    }

    if (fault\_flags\_ != 0) {

        fault\_count\_++;

        if (fault\_count\_ > 3 && control\_mode\_ != 0) {

            control\_mode\_ = 0;

            std::cout << "Multiple faults detected. Entering safe mode." << std::endl;

        }

    } else {

        fault\_count\_ = 0;

    }

}

void Microcontroller::updateHealthMonitoring() {

    if (control\_cycles\_ % 10 == 0) {

        uptime\_seconds\_++;

    }

}

**Microcontroller wrapper**

#include "microcontroller.h"

#include <cstring>

static Microcontroller\* g\_microcontroller = nullptr;

extern "C" {

// Initialize the microcontroller instance

void Microcontroller\_Init() {

    if (!g\_microcontroller) {

        g\_microcontroller = new Microcontroller();

    }

}

// Sensor data struct matching cFS telemetry payload

typedef struct {

    float gyro[3];

    float magnetometer[3];

    float sun\_angle;

    unsigned int timestamp;

    unsigned char valid;

} SensorData\_t;

// Actuator command struct for output

typedef struct {

    float wheel\_torques[3];

    float magnetorquer[3];

    unsigned int timestamp;

} ActuatorCommands\_t;

// Process incoming sensor data (called from cFS app)

void Microcontroller\_ProcessSensor(const SensorData\_t\* sensor) {

    if (g\_microcontroller && sensor) {

        Microcontroller::SensorData sensorData;

        // Explicit assignment ensures compatibility

        for (int i = 0; i < 3; i++) {

            sensorData.gyro[i] = sensor->gyro[i];

            sensorData.magnetometer[i] = sensor->magnetometer[i];

        }

        sensorData.sun\_angle = sensor->sun\_angle;

        sensorData.timestamp = sensor->timestamp;

        sensorData.valid = (sensor->valid != 0);

        g\_microcontroller->processSensorData(sensorData);

    }

}

// Get actuator commands from controller

void Microcontroller\_GetActuatorCommands(ActuatorCommands\_t\* commands) {

    if (g\_microcontroller && commands) {

        Microcontroller::ActuatorCommands actuators;

        g\_microcontroller->getActuatorCommands(actuators);

        std::memcpy(commands->wheel\_torques, actuators.wheel\_torques, sizeof(actuators.wheel\_torques));

        std::memcpy(commands->magnetorquer, actuators.magnetorquer, sizeof(actuators.magnetorquer));

        commands->timestamp = actuators.timestamp;

    }

}

// Cleanup microcontroller instance

void Microcontroller\_Cleanup() {

    delete g\_microcontroller;

    g\_microcontroller = nullptr;

}

**}**