ĐẠI HỌC QUỐC GIA TP. HCM

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

BÁO CÁO TỔNG KẾT ĐỀ TÀI ĐỒ ÁN

Môn: Chuyên đề thiết kế hệ thống nhúng 1 – CE437.P11

Tên đề tài:

Xe tự động phát hiện vật cản và tránh vật cản

***Giảng viên* hướng dẫn: Ths. Phạm Minh Quân**

Tham gia thực hiện:

STT	Họ và tên	Chịu trách nhiệm	MSSV
1.	**Nguyễn Thanh An**	**Nhóm trưởng**	**22520025**
2.	**Đào Tiến Hùng**	**Thành viên**	**22520499**
3.	**Hồ Đăng Mạnh Hưng**	**Thành viên**	**22520502**

MỤC LỤC

TÓM TẮT

Đề tài này nhằm xây dựng một hệ thống xe mô hình tự động nhận diện và tránh vật cản sử dụng vi điều khiển STM32F103C8T6, bao gồm hai node giao tiếp với nhau qua giao thức CAN. Hệ thống giải quyết các vấn đề của phương pháp điều khiển xe truyền thống như thiếu khả năng tự động nhận diện môi trường, dễ gây tai nạn, và yêu cầu sự can thiệp thủ công để điều chỉnh hướng di chuyển. Bằng cách ứng dụng công nghệ hiện đại, hệ thống cho phép xe mô hình tự động phát hiện vật cản thông qua cảm biến VL53L1X, xử lý dữ liệu và điều khiển động cơ cũng như servo để điều hướng xe một cách an toàn và hiệu quả.

Hệ thống bao gồm các thành phần chính như: 1 vi điều khiển ESP32 nhận nhiệm vụ là WifiNode, 2 vi điều khiển STM32F103C8T6 nhận nhiệm vụ là: ActuatorNode và SensorNode để xử lý dữ liệu và điều khiển, cảm biến VL53L1X để đo khoảng cách không tiếp xúc đến vật cản, module CAN để giao tiếp giữa hai node, động cơ DC và servo để điều khiển chuyển động và hướng di chuyển của xe. Ngoài ra, hệ thống còn sử dụng màn hình LCD để hiển thị thông tin trạng thái và các hệ thống mạch nguồn phục vụ cấp nguồn cho hệ thống.

Trong quá trình thử nghiệm, hệ thống hoạt động khá ổn định với khả năng tự động phát hiện vật cản và thực hiện các hành động tránh vật cản một cách khá chính xác. Hệ thống giảm đáng kể thời gian phản ứng so với phương pháp truyền thống, giảm thiểu sai sót trong quá trình điều khiển, và không yêu cầu sự can thiệp của người dùng để điều chỉnh hướng di chuyển. Dữ liệu từ cảm biến được xử lý nhanh chóng và gửi qua giao thức CAN, đảm bảo sự liên lạc hiệu quả giữa các node, từ đó tối ưu hóa quá trình điều khiển xe.

Tuy nhiên, một số hạn chế vẫn còn tồn tại như độ trễ trong phản hồi chưa tối ưu (khoảng 500ms – 1s), phụ thuộc vào độ ổn định của giao thức CAN và nguồn điện, cũng như khả năng đọc dữ liệu đôi lúc còn chưa chính xác đối với cảm biến VL53L1X trong các tình huống phức tạp. Đề tài vẫn có tiềm năng phát triển lớn với các định hướng như nâng cấp phần cứng để giảm độ trễ, cải thiện thuật toán xử lý dữ liệu cảm biến để tăng độ chính xác, tích hợp thêm các loại cảm biến khác như camera hoặc cảm biến siêu âm để mở rộng khả năng nhận diện môi trường, và xây dựng giao diện người dùng thông minh để theo dõi và điều khiển xe từ xa.

Hệ thống này không chỉ giải quyết bài toán điều khiển xe tránh vật cản một cách tự động mà còn mở rộng sang các lĩnh vực khác như robot di chuyển trong môi trường phức tạp, phương tiện giao thông tự hành nhỏ, và các hệ thống tự động hóa khác yêu cầu khả năng nhận diện và phản ứng nhanh với môi trường xung quanh.

Chương 1. TỔNG QUAN ĐỀ TÀI

1.1. Tổng quan các nghiên cứu liên quan

- **Bài báo "Autonomous Car Implementation Based on CAN Bus Protocol for IOT Applications"**. Nghiên cứu này tập trung vào việc sử dụng giao thức CAN trong các hệ thống điều khiển xe tự động. Bài báo mô tả cách CAN bus cung cấp một phương tiện giao tiếp tin cậy giữa các thành phần khác nhau của xe, bao gồm cảm biến, bộ xử lý trung tâm và các actuator.
  - **Nguồn tham khảo**: Autonomous Car Implementation Based on CAN Bus Protocol for IOT Applications

- **Bài báo "Design and Development of an Obstacle Avoidance Mobile-controlled Robot"** tập trung vào việc thiết kế và phát triển một robot di động có khả năng tự động nhận diện và tránh vật cản trong môi trường xung quanh. Mục tiêu chính của nghiên cứu là tạo ra một hệ thống robot hiệu quả, có thể hoạt động trong các điều kiện khác nhau mà không cần sự can thiệp thủ công, đồng thời cải thiện tính linh hoạt và khả năng tương tác với môi trường.
  - **Nguồn tham khảo**: Design and Development of an Obsstacle Avoidance Mobile-controlled Robot

1.2. Mục tiêu nghiên cứu

1.2.1. Mục tiêu tổng quát

Phát triển phần mềm điều khiển cho hệ thống xe mô hình tự động nhận diện và tránh vật cản sử dụng cảm biến VL51L1X, vi điều khiển STM32 và giao tiếp CAN, nhằm tối ưu hóa quá trình xử lý dữ liệu cảm biến, điều khiển các actuator một cách hiệu quả, đồng thời đảm bảo tính ổn định và khả năng mở rộng của hệ thống.

1.2.2. Mục tiêu cụ thể

- **Phát Triển Phần Mềm Xử Lý Dữ Liệu Cảm Biến:**

- Viết mã nguồn trên SensorNode để đọc và xử lý dữ liệu từ các cảm biến VL53L1X.
- Tối ưu hóa thuật toán xử lý tín hiệu để xác định chính xác khoảng cách đến vật cản trong thời gian thực.

- **Thiết Kế Giao Tiếp CAN:**

- Triển khai giao thức CAN giữa hai node SensorNode và ActuatorNode để truyền tải dữ liệu và lệnh điều khiển một cách hiệu quả.
- Đảm bảo tính ổn định và độ tin cậy trong giao tiếp giữa các node thông qua việc cấu hình các thông số CAN phù hợp.

- **Phát Triển Thuật Toán Điều Khiển và Tránh Vật Cản:**

- Xây dựng và triển khai các thuật toán điều khiển động cơ và servo dựa trên dữ liệu cảm biến nhận được trên ActuatorNode
- Đảm bảo xe mini có khả năng thực hiện các hành động tránh vật cản như rẽ trái, rẽ phải, tiến và lùi.

- **Kiểm Thử và Tối Ưu Hóa Hiệu Suất Phần Mềm:**

- Thực hiện các bài kiểm tra vận hành phần mềm trong các điều kiện môi trường khác nhau để đánh giá khả năng nhận diện và tránh vật cản.
- Tối ưu hóa mã nguồn để giảm độ trễ phản hồi và tăng tốc độ xử lý dữ liệu, đảm bảo hệ thống hoạt động mượt mà và hiệu quả.

1.2.3. Giới hạn của đề tài

- **Phạm Vi Kỹ Thuật:**

- Chỉ tập trung vào phát triển phần mềm điều khiển, không bao gồm việc thiết kế hoặc lắp ráp phần cứng của xe mô hình.
- Hệ thống sử dụng giao thức CAN duy nhất để giao tiếp giữa các node, không tích hợp thêm các giao thức khác như Bluetooth hoặc Wi-Fi.

- **Môi Trường Thử Nghiệm:**

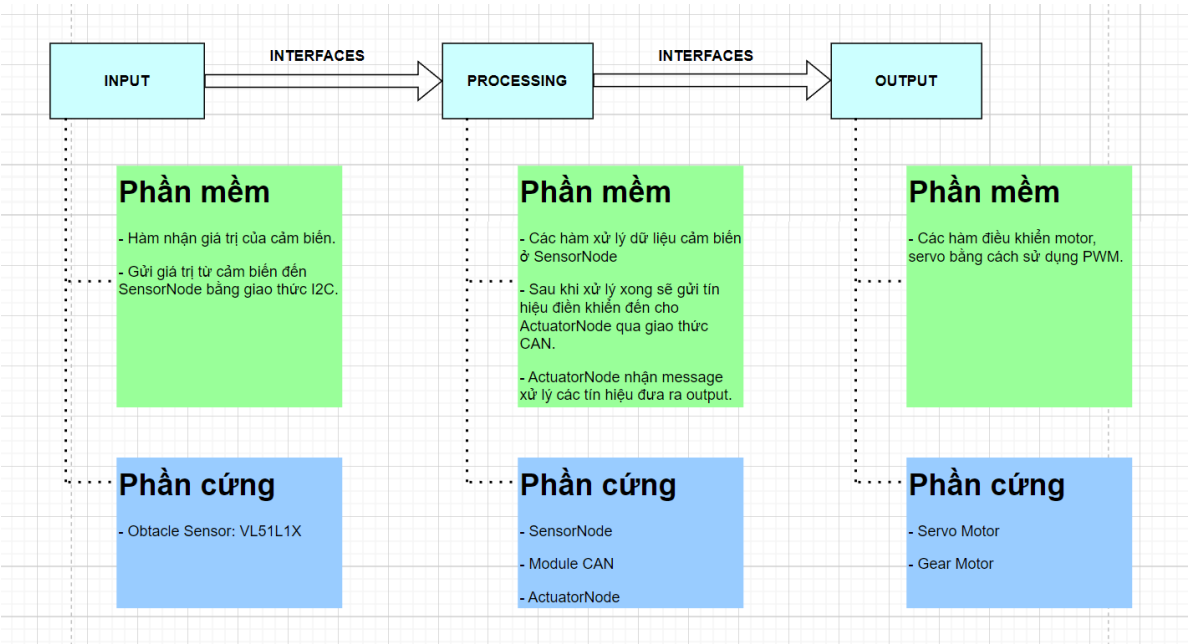
- Phần mềm được thử nghiệm trong môi trường kiểm soát với các vật cản cố định và di động, không áp dụng cho các môi trường ngoài trời có điều kiện khắc nghiệt như mưa, gió mạnh hoặc các bề mặt phức tạp.

- **Phần Cứng và Tương Thích:**

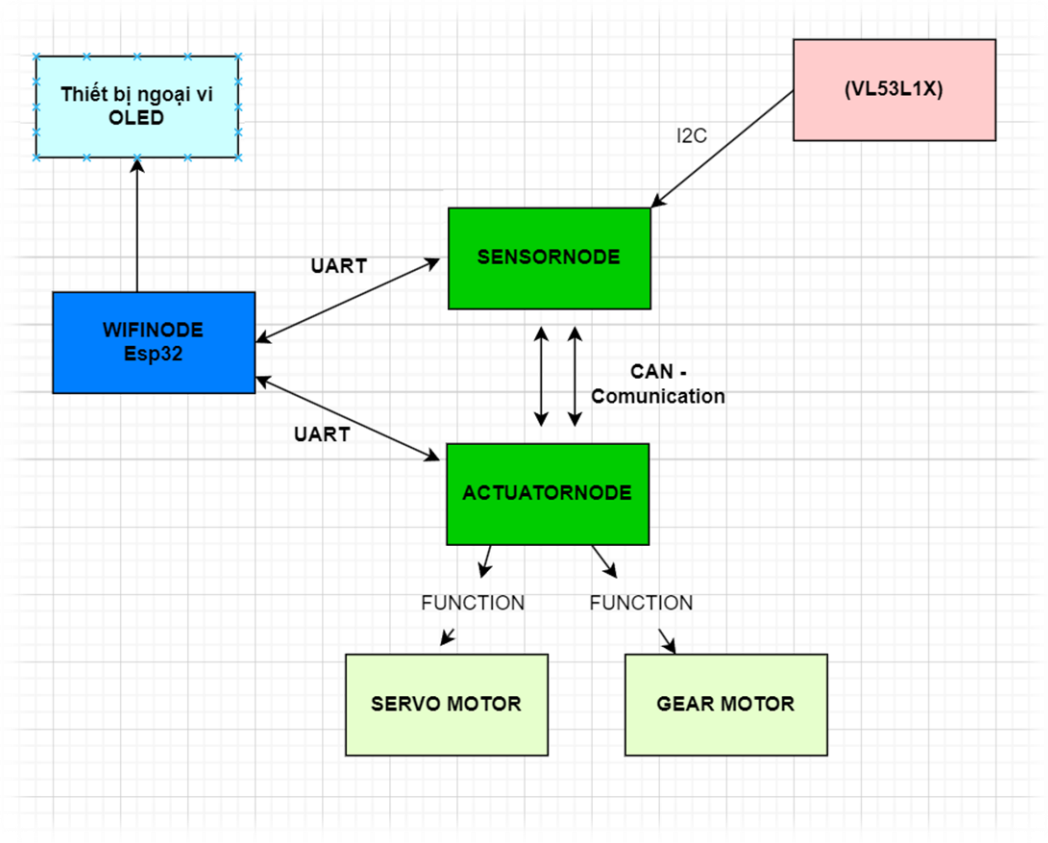
- Phần mềm được phát triển dựa trên cấu hình phần cứng hiện tại của STM32C8T6 và các module cảm biến VL53L1X, không hỗ trợ các loại cảm biến hoặc vi điều khiển khác.

Chương 2. THIẾT KẾ GIẢI PHÁP ĐỀ XUẤT

2.1. Kiến trúc hệ thống



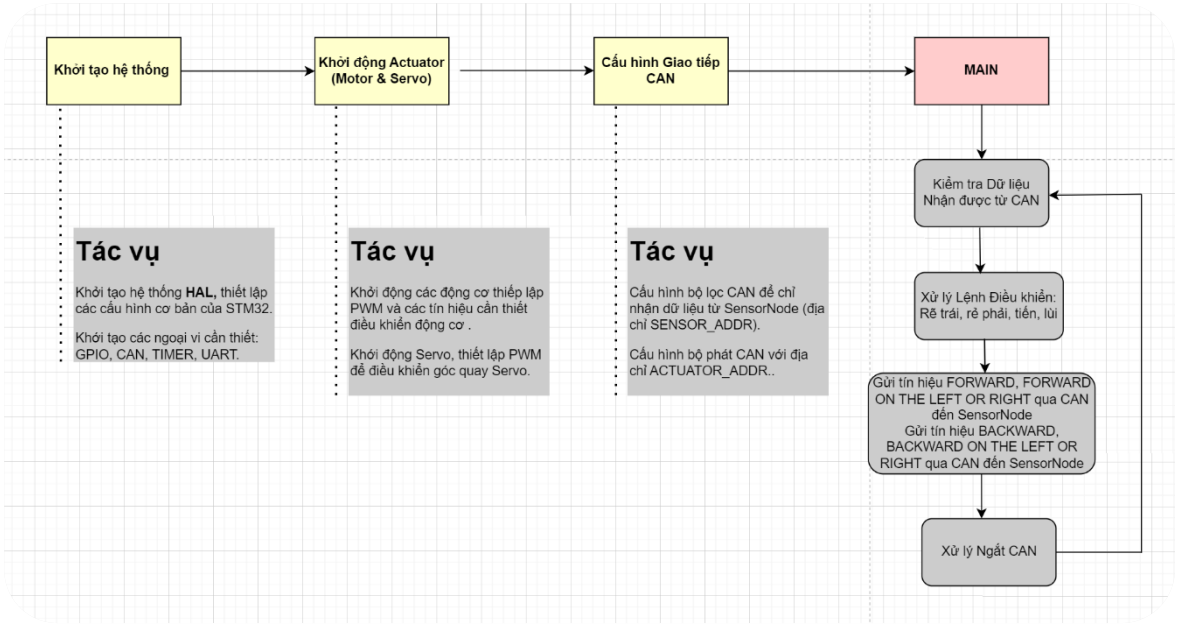
Hình 1. Kiến trúc hệ thống



Hình 2. Kiến trúc phần cứng hệ thống

2.2. Thiết kế chi tiết các thành phần phần mềm của hệ thống

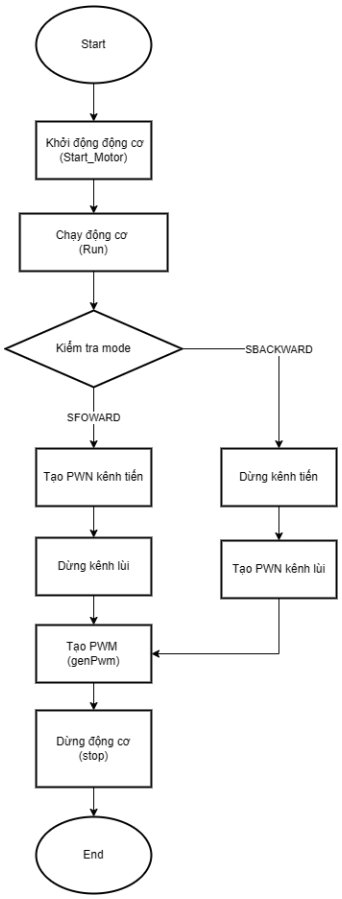
2.2.1. ActuatorNode



Hình 1. Sơ đồ tổng quan ActuatorNode

I. Motor và Servo

- **Đối với Motor (MOTOR.c)**



Hình 4. Lưu đồ giải thuật với Motor

a) Hàm *Start_Motor*

```
void Start_Motor()
{
    // Start PWM channels
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);
    // Set motor control pins to high
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, 1);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_15, 1);
}
```

Hình 5. Hàm Start_Motor()

- Chức năng: Khởi động động cơ bằng cách kích hoạt các kênh PWM và đặt tín hiệu điều khiển cho động cơ
- Cách hoạt động:
    - Bật tín hiệu PWM trên kênh *TIM_CHANNEL_1* và *TIM_CHANNEL_4*
    - Đặt các chân *GPIO GPIO_PIN_14* và *GPIO_PIN_15* lên mức cao để cung cấp tín hiệu điều khiển đến mạch điều khiển động cơ

b) Hàm *run*

```c
void run(uint8_t speed, uint8_t mode)
{
    if (mode == SFORWARD) {
        genPwm(&htim1,TIM_CHANNEL_4, speed);     // Forward channel
        genPwm(&htim1,TIM_CHANNEL_1, 0);         // Stop backward channel
    }
    else {  // SBACKWARD
        genPwm(&htim1,TIM_CHANNEL_4, 0);         // Stop forward channel
        genPwm(&htim1,TIM_CHANNEL_1, speed);     // Backward channel
    }
}
```

Hình 6. Hàm run()

- Chức năng: Điều khiển hướng và tốc độ của động cơ
- Tham số:
  - *speed:* Tốc độ động cơ (tỷ lệ PWM từ 0–100%)
  - *mode:* Hướng di chuyển của động cơ (tiến hoặc lùi: SFORWARD hoặc SBACKWARD)
- Cách hoạt động:
  - Nếu mode == *SFORWARD*:
  - Kênh *PWM TIM_CHANNEL_4* được cấp xung PWM với giá trị tương ứng speed
  - Kênh *PWM TIM_CHANNEL_1* được đặt về 0 để dừng hướng ngược lại
- Nếu mode != *SFORWARD* (tức là lùi):
  - Kênh *PWM TIM_CHANNEL_1* được cấp xung PWM với giá trị speed
  - Kênh *PWM TIM_CHANNEL_4* được đặt về 0 để dừng hướng ngược lại

c) Hàm *genPwm*

```c
void genPwm(TIM_HandleTypeDef *htim, uint32_t channel, float duty_cycle)
{
    // Calculate load value based on duty cycle
    float load_value = (duty_cycle / 100) * htim->Instance->ARR;
    // Set compare register value to generate PWM
    __HAL_TIM_SET_COMPARE(htim, channel, (uint16_t)load_value);
}
```

Hình 7. Hàm genPwm()

- Chức năng: Tạo tín hiệu PWM với tỷ lệ độ rộng xung (duty cycle) mong muốn
- Tham số:
  - *htim:* Bộ định thời (timer) được sử dụng để tạo tín hiệu PWM
  - *channel*: Kênh timer cụ thể xuất tín hiệu PWM
  - *duty_cycle:* Độ rộng xung của tín hiệu PWM (0–100%)

- Cách hoạt động:
  - Tính toán giá trị tải (load_value) dựa trên tỷ lệ duty_cycle và giá trị trong thanh ghi ARR của timer
  - Ghi giá trị này vào thanh ghi so sánh (CCR) của kênh timer thông qua macro *__HAL_TIM_SET_COMPARE*

d) Hàm *stop*

```
void stop()
{
    // Stop PWM signals
    genPwm(&htim1,TIM_CHANNEL_4, 0);
    genPwm(&htim1,TIM_CHANNEL_1, 0);
    // Short delay to ensure the motor stops completely
    HAL_Delay(200);
}
```

Hình 8. Hàm stop()

- Chức năng: Dừng động cơ
- Cách hoạt động:
  - Gọi hàm genPwm để đặt tín hiệu PWM của cả hai kênh *TIM_CHANNEL_1* và *TIM_CHANNEL_4* về 0
  - Gọi *HAL_Delay(200)* để thêm thời gian trễ 200ms, đảm bảo động cơ dừng hoàn toàn

- **Đối với Servo (SERVO.c)**

a) Hàm *Start_Servo*

```
void Start_Servo()
{
    HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
    To_Default();
    HAL_Delay(100);      // Allow time for the servo to move to the default position
}
```

Hình 9. Hàm Start_Servo()

- Chức năng: Khởi động servo bằng cách kích hoạt tín hiệu PWM và đưa servo về vị trí mặc định.
- Cách hoạt động:
  - *HAL_TIM_PWM_Start:* Bật kênh PWM trên *TIM_CHANNEL_1*
  - *To_Default:* Đưa servo về vị trí mặc định (góc 90 độ)
  - *HAL_Delay(100):* Tạo độ trễ 100ms để đảm bảo servo có đủ thời gian di chuyển

b) Hàm *Angle*

```c
uint16_t Angle(double angle)
{
    double temp = 250 + angle * 5.56;       // Conversion formula
    return (int)temp;
}
```

Hình 10. Hàm Angle()

- Chức năng: Chuyển đổi góc (đơn vị độ, từ 0–180) sang giá trị PWM tương ứng
- Công thức chuyển đổi:
    - PWM_value = 250 + angle * 5.56
    - 250: Giá trị PWM tương ứng với góc 0 độ
    - 5.56: Tỷ lệ chuyển đổi từ độ sang PWM (phụ thuộc vào cấu hình servo và timer)

c) Hàm *To_Default*

```c
void To_Default()
{
    __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, Angle(90));     // Set to 90 degrees
}
```

Hình 11. Hàm To_Default()

- Chức năng: Đưa servo về vị trí mặc định (góc 90 độ)
- Cách hoạt động:
    - Gọi hàm Angle(90) để tính giá trị PWM cho góc 90 độ
    - Ghi giá trị này vào thanh ghi so sánh (CCR) của TIM_CHANNEL_1

d) Hàm *To_Angle*

```c
void To_Angle(uint16_t CAngle, uint16_t DAngle, uint8_t mode, uint16_t inc)
{
    if (mode == FAST) {
        // Move directly to the desired angle
        __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, Angle(DAngle));
    }
    else {
        // Gradual movement with incremental steps
        while (CAngle < DAngle) {
            __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, Angle(CAngle));
            HAL_Delay(80);      // Delay between increments
            CAngle += inc;
        }
    }
}
```

Hình 12. Hàm To_Angle()

- Chức năng: Di chuyển servo đến góc mong muốn với 2 chế độ: di chuyển nhanh (FAST) hoặc di chuyển chậm (SLOW).
- Tham số:
    - *CAngle*: Góc hiện tại của servo
    - *DAngle*: Góc mong muốn

- *mode:* Chế độ di chuyển (FAST hoặc SLOW)
- *inc:* Bước nhảy góc (dùng trong chế độ SLOW)

- Cách hoạt động:
  - Nếu *mode == FAST:*
  - Servo được di chuyển trực tiếp đến DAngle
  - Nếu *mode == SLOW:*
  - Servo di chuyển từng bước từ *CAngle* đến *DAngle*, với mỗi bước có độ lớn inc
  - Thời gian trễ 80ms giữa mỗi bước được thêm để di chuyển ổn định

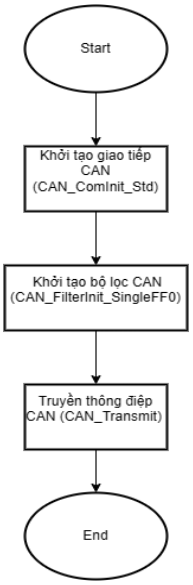
e) Hàm *Turn_Left* và *Turn_Right*

```
void Turn_Left()
{
    //To_Default();
    __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, Angle(48));
}
/**
 * @brief Turn the servo to the right (136 degrees).
 */
void Turn_Right()
{
    //To_Default();
    __HAL_TIM_SET_COMPARE(&htim4,TIM_CHANNEL_1, Angle(136));
}
```

Hình 13. Hàm Turn_Left() và Turn_Right()

- Chức năng: Xoay servo sang trái và phải
- Cách hoạt động:
  - Gọi hàm Angle(48) để tính giá trị PWM tương ứng với góc 48 độ
  - Gọi hàm Angle(136) để tính giá trị PWM tương ứng với góc 48 độ
  - Ghi giá trị này vào thanh ghi so sánh của TIM_CHANNEL_1

II. Giao tiếp CAN (CAN_Handler.c)



Hình 14. Sơ đồ CAN_Handler ở ActuatorNode

a) Các biến toàn cục

```
CAN_TxHeaderTypeDef TxHeader;
uint32_t Mailbox;
uint8_t TxBuffer[] = { 1, 2, 3, 4, 5, 6, 0, 9 };

CAN_RxHeaderTypeDef RxHeader;
CAN_FilterTypeDef canfilterconfig;
uint8_t RxData[8];
```

Hình 12. Các biến toàn cục trong ActuatorNode

- *TxHeader:* Cấu trúc định nghĩa header cho các tin nhắn gửi (Transmit) trên giao thức CAN
- *Mailbox:* Biến lưu địa chỉ mailbox được sử dụng khi gửi một message
- *TxBuffer:* Mảng dữ liệu dùng để truyền qua giao thức CAN
- *RxHeader:* Cấu trúc lưu header của các tin nhắn nhận (Receive)
- *canfilterconfig:* Cấu trúc dùng để cấu hình bộ lọc CAN
- *RxData:* Mảng lưu dữ liệu nhận được từ giao thức CAN

b) Hàm *CAN_ComInit_Std*

```
void CAN_ComInit_Std(CAN_TxHeaderTypeDef *Tx_Header, CAN_HandleTypeDef *hcan, uint32_t id, uint8_t dlc)
{
    Tx_Header->IDE = CAN_ID_STD;     // Use Standard ID
    Tx_Header->DLC = dlc;            // Set data length
    Tx_Header->StdId = id;           // Set Standard ID
    Tx_Header->RTR = CAN_RTR_DATA;   // Data Frame
    HAL_CAN_Start(hcan);             // Start the CAN peripheral
}
```

Hình 13. Hàm CAN_ComInit_Std()

- Chức năng: Cấu hình tin nhắn chuẩn (Standard Frame) cho giao thức CAN và khởi động module CAN
- Tham số:
    - *Tx_Header:* Con trỏ đến cấu trúc CAN_TxHeaderTypeDef để lưu thông tin cấu hình header
    - *hcan:* Con trỏ đến handle của module CAN đang được sử dụng
    - *id:* ID chuẩn (Standard ID) của tin nhắn CAN
    - *dlc:* Độ dài dữ liệu (Data Length Code) trong tin nhắn
- Các cài đặt:
    - *CAN_ID_STD:* Sử dụng ID chuẩn
    - *CAN_RTR_DATA:* Gửi tin nhắn loại "Data Frame"
    - *HAL_CAN_Start:* Bắt đầu hoạt động module CAN

c) Hàm *CAN_FilterInit_SingleFF0*

```
void CAN_FilterInit_SingleFF0(CAN_HandleTypeDef *hcan, CAN_FilterTypeDef *canfilterconfig, uint32_t addr)
{
    canfilterconfig->FilterActivation = CAN_FILTER_ENABLE;       // Enable filter
    canfilterconfig->FilterBank = 0;                             // Use filter bank 0
    canfilterconfig->FilterFIFOAssignment = CAN_FILTER_FIFO0;    // Assign to FIFO0
    canfilterconfig->FilterIdHigh = addr << 5;                   // Set filter ID (high part)
    canfilterconfig->FilterIdLow = 0x0000;                       // No additional bits
    canfilterconfig->FilterMaskIdHigh = addr << 5;               // Set mask ID (high part)
    canfilterconfig->FilterMaskIdLow = 0x0000;                   // No additional bits
    canfilterconfig->FilterMode = CAN_FILTERMODE_IDMASK;         // Use ID masking
    canfilterconfig->FilterScale = CAN_FILTERSCALE_32BIT;        // 32-bit filter scale

    HAL_CAN_ConfigFilter(hcan, canfilterconfig);                    // Configure the filter
    HAL_CAN_ActivateNotification(hcan, CAN_IT_RX_FIFO0_MSG_PENDING);// Enable RX FIFO0 interrupt
}
```

Hình 14. Hàm CAN_FilterInit_SingleFF0()

- Chức năng: Cấu hình bộ lọc CAN để chỉ nhận những tin nhắn có ID phù hợp với ID và mặt nạ (mask) đã cấu hình
- Tham số:
    - *hcan:* Con trỏ đến handle của module CAN
    - *canfilterconfig:* Con trỏ đến cấu trúc cấu hình bộ lọc CAN
    - *addr:* ID được cấu hình cho bộ lọc
- Các cài đặt:
    - *CAN_FILTER_ENABLE:* Kích hoạt bộ lọc
    - *FilterBank:* Sử dụng bank 0
    - *FilterFIFOAssignment:* Kết nối bộ lọc với FIFO0

- *FilterIdHigh/Low:* Xác định ID bộ lọc
- *FilterMaskIdHigh/Low:* Mặt nạ ID để xác định tin nhắn hợp lệ
- *FilterMode:* Sử dụng chế độ mặt nạ ID
- *FilterScale:* Bộ lọc sử dụng 32-bit
- *HAL_CAN_ActivateNotification:* Kích hoạt ngắt nhận tin nhắn từ FIFO0

d) Hàm *CAN_Transmit*

```
uint32_t CAN_Transmit(CAN_HandleTypeDef *hcan, CAN_TxHeaderTypeDef *Tx_Header, uint8_t* buffer)
{
    HAL_CAN_AddTxMessage(hcan, Tx_Header, buffer, &Mailbox);
    return Mailbox;     // Return mailbox if transmission is successful
}
```

Hình 15. Hàm CAN_Transmit()

- Chức năng: Gửi tin nhắn CAN thông qua module CAN
- Tham số:
  - *hcan:* Con trỏ đến handle của module CAN
  - *Tx_Header:* Con trỏ đến cấu trúc header chứa thông tin tin nhắn
  - *buffer:* Con trỏ đến mảng chứa dữ liệu cần gửi
- Cách hoạt động:
  - *HAL_CAN_AddTxMessage:* Thêm tin nhắn vào hàng đợi truyền của CAN và lấy mailbox để kiểm tra trạng thái
  - *return Mailbox:* Trả về mailbox đã sử dụng

III. Xử lý tín hiệu (SIGNAL.c)

a) Các biến hằng số

```
/* Constants -----------------------------------------
uint8_t LEFT[]      = { 1, 0, 0, 0, 0, 0, 0, 0 };
uint8_t RIGHT[]     = { 2, 0, 0, 0, 0, 0, 0, 0 };
uint8_t FORWARD[]   = { 3, 0, 0, 0, 0, 0, 0, 0 };
uint8_t BACKWARD[]  = { 4, 0, 0, 0, 0, 0, 0, 0 };
/**
```

Hình 19. Các biến hằng số

- Chức năng: Các mảng dữ liệu tượng trưng cho tín hiệu điều khiển tương ứng với các hướng di chuyển:
  - LEFT: Tín hiệu cho xe di chuyển sang trái
  - RIGHT: Tín hiệu cho xe di chuyển sang phải
  - FORWARD: Tín hiệu cho xe di chuyển về phía trước
  - BACKWARD: Tín hiệu cho xe di chuyển về phía sau

- Cấu trúc:
  - Byte đầu tiên trong mảng đại diện cho lệnh di chuyển
  - Các byte còn lại (0)

b) Hàm *Get_State*

```
uint8_t Get_State(int distance1, int distance2)
{
    if (distance1 <= 600 && distance2 <= 600)
        return BACKWARD_STATE;

    if (distance2 < distance1) {
        if (distance2 <= 900)
            return LEFT_STATE;
    }
    else if (distance1 < distance2) {
        if (distance1 <= 900)
            return RIGHT_STATE;
    }
    return FORWARD_STATE;
}
```

Hình 20. Hàm Get_State()

- Chức năng:
  - Xác định trạng thái di chuyển của xe dựa trên khoảng cách đo được từ hai cảm biến
- Tham số:
  - *distance1*: Khoảng cách từ cảm biến thứ nhất
  - *distance2*: Khoảng cách từ cảm biến thứ hai
- Hoạt động:
  - Điều kiện cả hai cảm biến gần (≤ 600):
    - Xe sẽ di chuyển lùi (*BACKWARD_STATE*)
    - Điều kiện cảm biến thứ hai gần hơn cảm biến thứ nhất:
    - Nếu distance2 ≤ 900, xe sẽ rẽ trái (*LEFT_STATE*).
  - Điều kiện cảm biến thứ nhất gần hơn cảm biến thứ hai:
    - Nếu distance1 ≤ 900, xe sẽ rẽ phải (*RIGHT_STATE*).
  - Trường hợp không thỏa mãn điều kiện nào ở trên:
    - Xe di chuyển về phía trước (*FORWARD_STATE*)

c) Hàm *Get_CounterState*

```c
uint8_t Get_CounterState(uint8_t state)
{
    if (state == RIGHT_STATE) {
        return LEFT_STATE;
    }
    else if (state == LEFT_STATE) {
        return RIGHT_STATE;
    }
    else {
        return LEFT_STATE;  // Default fallback for invalid state
    }
}
```

Hình 21. Hàm Get_CounterState()

- Chức năng:
  - Lấy trạng thái đối lập (phản ứng) cho trạng thái hiện tại của xe
- Tham số:
  - state: Trạng thái hiện tại của xe, có thể là:
  - LEFT_STATE: Xe đang di chuyển sang trái
  - RIGHT_STATE: Xe đang di chuyển sang phải
  - Các giá trị khác (không hợp lệ)
- Hoạt động:
  - Nếu trạng thái hiện tại là RIGHT_STATE:
    - Trả về trạng thái đối lập là LEFT_STATE.
  - Nếu trạng thái hiện tại là LEFT_STATE:
    - Trả về trạng thái đối lập là RIGHT_STATE.
  - Nếu trạng thái không hợp lệ:
    - Trả về giá trị mặc định là LEFT_STATE

IV. Chương trình chính (main.c)

a) Khởi tạo hệ thống

```c
HAL_Init();

/* Configure the system clock */
SystemClock_Config();

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_TIM1_Init();
MX_TIM2_Init();
MX_TIM3_Init();
MX_USART1_UART_Init();
MX_TIM4_Init();
MX_CAN_Init();
```

Hình 22. Khởi tạo hệ thống()

- HAL_Init: Khởi tạo thư viện HAL

- SystemClock_Config: Cấu hình xung nhịp hệ thống với nguồn HSE và bộ PLL nhân lên để đạt tần số mong muốn
- MX_GPIO_Init, MX_TIMx_Init, MX_USART1_UART_Init, MX_CAN_Init: Khởi tạo các module GPIO, Timer, UART, và CAN

b) Cấu hình và khởi động ứng dụng

```
Start_Motor();
Start_Servo();
CAN_FilterInit_SingleFF0(&hcan, &canfilterconfig, SENSOR_ADDR);
CAN_ComInit_Std(&TxHeader, &hcan, ACTUATOR_ADDR, 1);
/* USER CODE END 2 */
```

Hình 23. Cấu hình và khởi động ứng dụng()

- Start_Motor, Start_Servo: Khởi động các module điều khiển động cơ và servo
- CAN_FilterInit_SingleFF0: Cấu hình bộ lọc CAN để nhận các tin nhắn có ID phù hợp với SENSOR_ADDR
- CAN_ComInit_Std: Cấu hình giao tiếp CAN để gửi tin nhắn với ID là ACTUATOR_ADDR

c) Biến toàn cục và callback

```
uint8_t g_receiveFlag = 0;
uint8_t g_message = 7;
uint8_t g_subMessage = 7;
```

Hình 24. Biến toàn cục

- g_receiveFlag: Cờ báo hiệu có dữ liệu CAN mới được nhận
- g_message, g_subMessage: Lưu trạng thái và tín hiệu phụ (sub-message) được nhận từ CAN

Call back:

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef* hcan)
{
    HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData);
    mess = RxData[0];
    mess2 = RxData[1];
    receive = 1;
}
```

Hình 25. Callback

- Được gọi khi có Message mới trong FIFO0 của CAN
- Gán giá trị RxData[0] và RxData[1] vào mess và mess2, đồng thời đặt cờ receive = 1

d) Vòng lặp chính

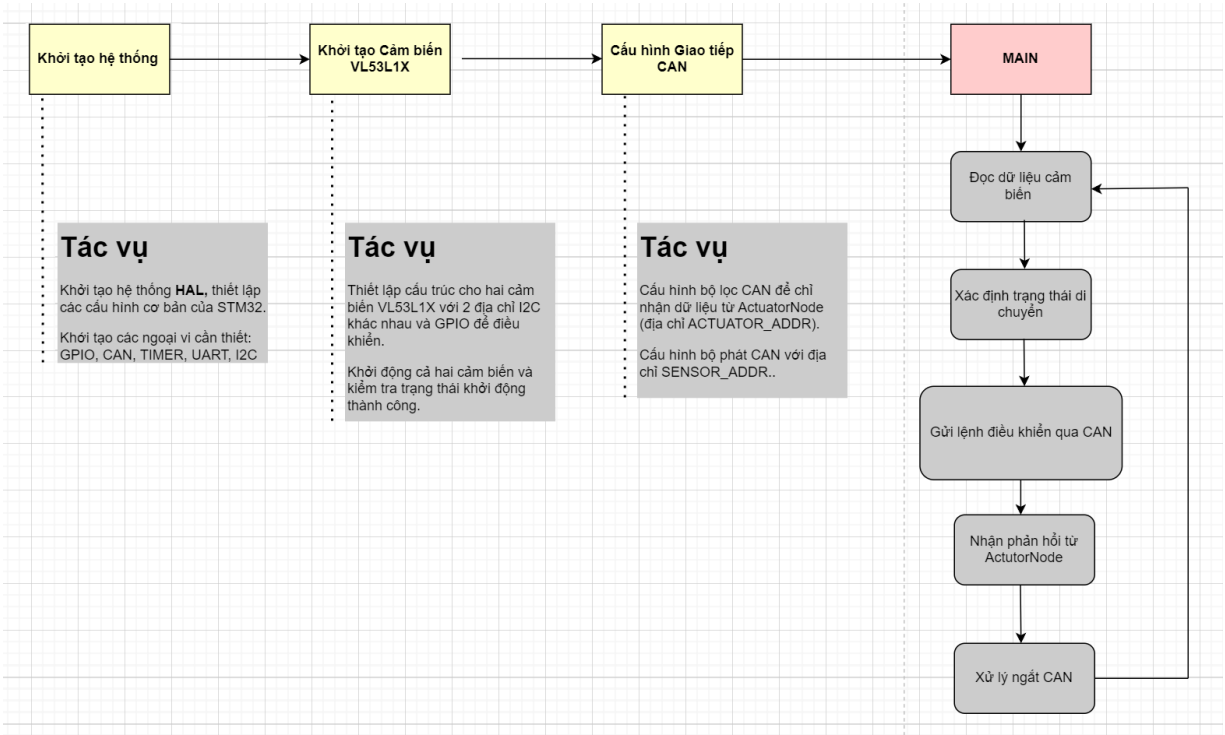
```
while (1) {
    // Force change state if necessary
    if (forceChange != 0) {
        if (forceChange == BACKWARD_RIGHT_STATE) {
            state = LEFT_STATE;
        } else {
            state = RIGHT_STATE;
        }
        forceChange = 0;
    } else {
        // Read distance from sensors
        VL53L1_GetDistance(&sensor1);
        VL53L1_GetDistance(&sensor2);
        // Adjust sensor data for specific scenarios
        if (sensor2.distance >= 55) {
            sensor2.distance += 50;          // Calibration for specific conditions
        }
        // Determine next state based on sensor data
        state = Get_State(sensor1.distance, sensor2.distance, 0, prevstate);
    }
    // State machine to handle system behavior
    switch (state) {
    case LEFT_STATE:
        forceChange = 0;
        prevstate = state;
        HAL_Delay(50);
        while (returnSignal != LEFT_STATE) {
            CAN_Transmit(&hcan, &TxHeader, LEFT);    // Send LEFT command
        }
        break;

    case RIGHT_STATE:
        forceChange = 0;
        prevstate = state;
```

Hình 26. Vòng lặp chính

- Hoạt động:
  - Kiểm tra cờ receive (có tin nhắn mới từ CAN)
  - Dựa vào giá trị mess (tín hiệu chính) và mess2 (tín hiệu phụ) để xác định hành động:
    - *case 1:* Chạy tới và rẽ trái
    - *case 2:* Chạy tới và rẽ phải
    - *case 3:* Chạy thẳng, nhưng có thể điều chỉnh trái/phải dựa vào mess2
    - *case 4:* Chạy lùi và điều chỉnh hướng dựa vào mess2
  - Sau khi xử lý, đặt lại cờ receive = 0
- Hành động chính:
  - *run*(speed, direction): Chạy động cơ với tốc độ và hướng cụ thể
  - *Turn_Left*, *Turn_Right*: Điều khiển servo quay trái/phải
  - *CAN_Transmit*: Gửi tín hiệu phản hồi qua CAN

2.2.2. SensorNode



Hình 6. Sơ đồ tổng quan SensorNode

I. Khai báo SIGNAL.h và CAN_Handler.

File SIGNAL.h.

```
#ifndef INC_SIGNAL_H_
#define INC_SIGNAL_H_

#include "main.h"

#define LEFT_STATE              1
#define RIGHT_STATE             2
#define FORWARD_STATE           3
#define BACKWARD_LEFT_STATE     4
#define BACKWARD_RIGHT_STATE    5
#define FORWARD_LEFT_STATE      6
#define FORWARD_RIGHT_STATE     7
#define TEST_SENSORS            8

extern uint8_t LEFT[8];
extern uint8_t RIGHT[8];
extern uint8_t FORWARD[8];
extern uint8_t BACKWARD_LEFT[8];
extern uint8_t BACKWARD_RIGHT[8];
extern uint8_t FORWARD_LEFT[8];
extern uint8_t FORWARD_RIGHT[8];

extern uint8_t returnSignal;

uint8_t Get_State(int d1, int d2, uint8_t *forceChange, uint8_t state);
uint8_t Get_CounterState(uint8_t state);
#endif /* INC_SIGNAL_H_ */
```

Giải thích:

Các giá trị định nghĩa trạng thái di chuyển (movement states):

LEFT_STATE = 1: Trạng thái di chuyển sang trái.

RIGHT_STATE = 2: Trạng thái di chuyển sang phải.

FORWARD_STATE = 3: Trạng thái di chuyển thẳng.

BACKWARD_LEFT_STATE = 4: Trạng thái lùi sang trái.

BACKWARD_RIGHT_STATE = 5: Trạng thái lùi sang phải.

FORWARD_LEFT_STATE = 6: Trạng thái tiến sang trái.

FORWARD_RIGHT_STATE = 7: Trạng thái tiến sang phải.

TEST_SENSORS = 8: Trạng thái kiểm tra cảm biến, thường được dùng trong các trường hợp thử nghiệm hoặc gỡ lỗi.

Các mảng tín hiệu điều khiển.

uint8_t LEFT[8]: Lưu tín hiệu điều khiển cho trạng thái di chuyển sang trái.

uint8_t RIGHT[8]: Lưu tín hiệu điều khiển cho trạng thái di chuyển sang phải.

uint8_t FORWARD[8]: Lưu tín hiệu điều khiển cho trạng thái di chuyển thẳng.

uint8_t BACKWARD_LEFT[8]: Lưu tín hiệu điều khiển cho trạng thái lùi sang trái.

uint8_t BACKWARD_RIGHT[8]: Lưu tín hiệu điều khiển cho trạng thái lùi sang phải.

uint8_t FORWARD_LEFT[8]: Lưu tín hiệu điều khiển cho trạng thái tiến sang trái.

uint8_t FORWARD_RIGHT[8]: Lưu tín hiệu điều khiển cho trạng thái tiến sang phải

Chức năng của chúng là mảng 8 phần tử dùng để lưu thông tin liên quan đến tín hiệu điều khiển. Lưu giá trị từ Acutor và từ Sensor.

Hàm **uint8_t Get_State**(int d1, int d2, uint8_t *forceChange, uint8_t state): Xác định trạng thái di chuyển tiếp theo của xe dựa trên các khoảng cách đó được trả về từ Sensor.

Hàm **uint8_t Get_CounterState**(uint8_t state): Trả về trạng thái đối nghịch của trạng thái hiện tại, hỗ trợ chuyển trạng thái nếu cần.

File CAN_Handler.h.

```c
#ifndef INC_CAN_HANDLER_H_
#define INC_CAN_HANDLER_H_

#include "can.h"
#include "main.h"

#define ACTUATOR_ADDR   0x0A2
#define SENSOR_ADDR     0X012

extern CAN_TxHeaderTypeDef TxHeader;
extern uint8_t TxBuffer[8];
extern uint32_t Mailbox;

extern CAN_RxHeaderTypeDef RxHeader;
extern CAN_FilterTypeDef canfilterconfig;
extern uint8_t RxData[8];

void CAN_ComInit_Std(CAN_TxHeaderTypeDef *Tx_Header, CAN_HandleTypeDef *hcan, uint32_t id, uint8_t dlc);
void CAN_FilterInit_SingleFF0(CAN_HandleTypeDef *hcan, CAN_FilterTypeDef *canfilterconfig, uint32_t addr);
uint32_t CAN_Transmit(CAN_HandleTypeDef *hcan, CAN_TxHeaderTypeDef *Tx_Header, uint8_t* buffer);

#endif /* INC_CAN_HANDLER_H_ */

```

Giải thích:

Macro định nghĩa địa chỉ:

#define ACTUATOR_ADDR 0x0A2: địa chỉ CAN của Actutor sử dụng để truyền hoặc nhận dữ liệu từ Actutor.

#define SENSOR_ADDR 0x012: địa chỉ CAN của cảm biến Sensor được sử dụng để trao đổi dữ liệu qua CAN bus.

Biến toàn cục:

CAN_TxHeaderTypeDef TxHeader: Cấu trúc chứa thông tin tiêu đề của khung dữ liệu CAN khi truyền.

uint8_t TxBuffer[8]: Bộ đệm lưu trữ dữ liệu truyền qua CAN bus tối đa 8byte dữ liệu.

CAN_RxHeaderTypeDef RxHeader: Cấu trúc chứa thông tin tiêu đề của CAN khi nhận dữ liệu.

CAN_FilterTypeDef canfilterconfig: cấu hình bộ lọc CAN , cho phép nhận địa chỉ hoặc thông số nhất định

uint8_t RxData[8]: Bộ đệm lưu trữ dữ liệu nhận được từ CAN bus.

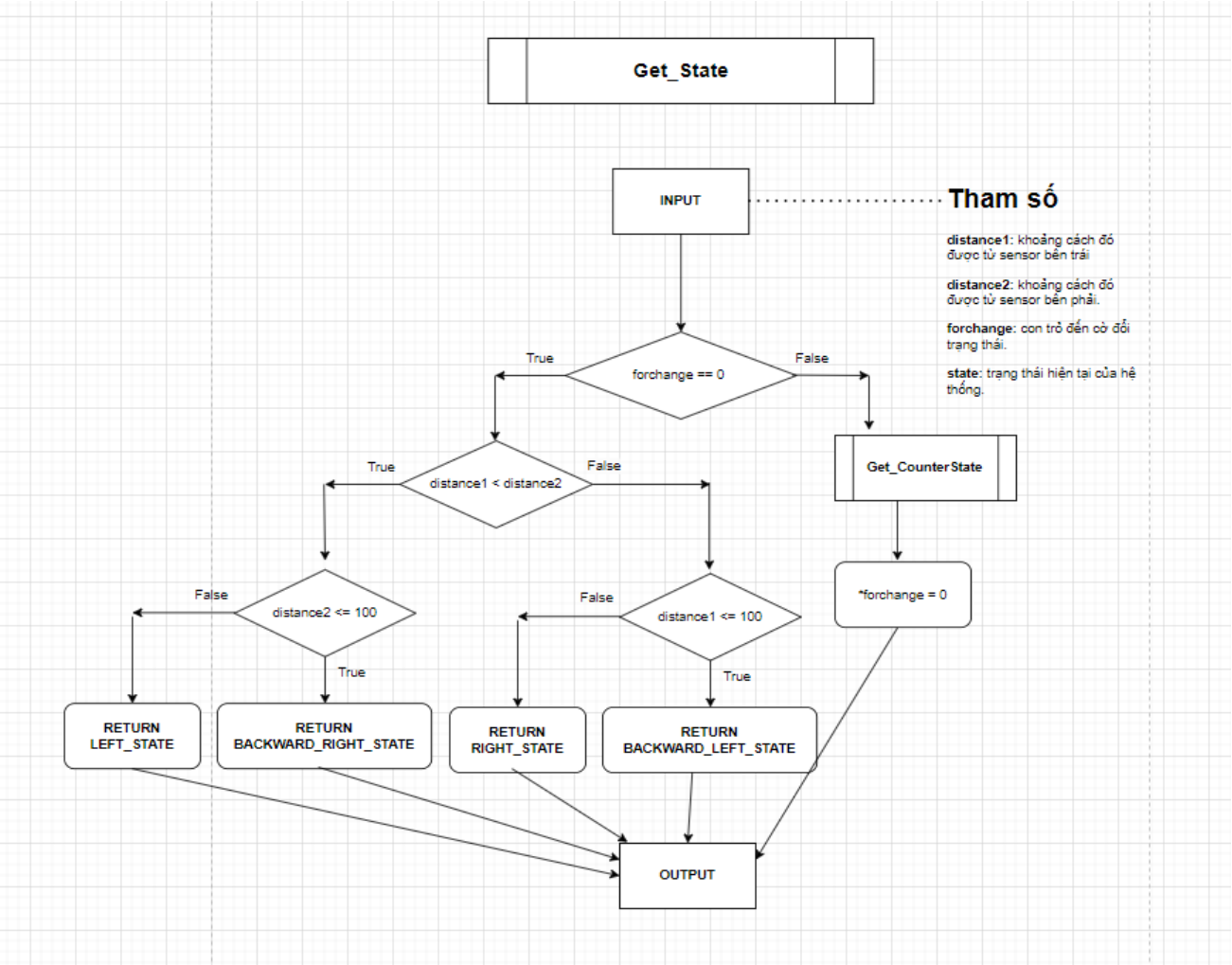
Hàm **CAN_ComInit_Std**(CAN_TxHeaderTypeDef *Tx_Header, CAN_HandleTypeDef *hcan, uint32_t id, uint8_t dlc):Khởi tạo header cho CAN truyền đi với các tham số cơ bản.

void **CAN_FilterInit_SingleFF0**(CAN_HandleTypeDef *hcan, CAN_FilterTypeDef *canfilterconfig, uint32_t addr): Khởi tạo bộ lọc CAN nhận gói tin trên địa chỉ xác định cụ thể.

uint32_t **CAN_Transmit**(CAN_HandleTypeDef *hcan, CAN_TxHeaderTypeDef *Tx_Header, uint8_t* buffer): Thực hiện gửi dữ liệu thông qua CAN bus.

II. Xử lý tín hiệu (SIGNAL.c)

a) Hàm Get_State:



```c
#include "SIGNAL.h"
// Define movement signals for different states
uint8_t LEFT[]          = { 1, 0, 0, 0, 0, 0, 0, 0 };          // Move left
uint8_t RIGHT[]         = { 2, 0, 0, 0, 0, 0, 0, 0 };          // Move right
uint8_t FORWARD[]       = { 3, 0, 0, 0, 0, 0, 0, 0 };          // Move forward
uint8_t BACKWARD_LEFT[] = { 4, 1, 0, 0, 0, 0, 0, 0 };          // Move backward left
uint8_t BACKWARD_RIGHT[]= { 4, 2, 0, 0, 0, 0, 0, 0 };          // Move backward right
uint8_t FORWARD_LEFT[]  = { 3, 1, 0, 0, 0, 0, 0, 0 };          // Move forward left
uint8_t FORWARD_RIGHT[] = { 3, 2, 0, 0, 0, 0, 0, 0 };          // Move forward right

uint8_t returnSignal = 0;   // Return signal from the actuator system
```

```c
uint8_t Get_State(int distance1, int distance2, uint8_t *forceChange, uint8_t state)
{
    if (*forceChange == 0) {
        // No forced state change
        if (distance2 < distance1) {
            // Right side is closer
            if (distance2 <= 100)
                return BACKWARD_RIGHT_STATE;    // Object too close on the right
            else if (distance2 <= 900)
                return LEFT_STATE;              // Turn left to avoid
        }
        else if (distance1 < distance2) {
            // Left side is closer
            if (distance1 <= 100)
                return BACKWARD_LEFT_STATE;     // Object too close on the left
            else if (distance1 <= 900)
                return RIGHT_STATE;             // Turn right to avoid
        }

    }
    else if (*forceChange == 1) {
        // Enforce a counter-state transition
        *forceChange = 0;                       // Reset force change flag
        return Get_CounterState(state);
    }
    return FORWARD_STATE;                       // Default to forward movement
}
```

**Phân tích:*

Các biến định nghĩa như sau:

Mỗi mảng đại diện cho một tín hiệu di chuyển tương ứng với trạng thái cụ thể. Các giá trị bên trong có ý nghĩa tùy theo cách bạn thiết kế giao thức tín hiệu, chẳng hạn byte đầu tiên là mã trạng thái chính, các byte còn lại có thể chứa dữ liệu bổ sung.

uint8_t LEFT[] = { 1, 0, 0, 0, 0, 0, 0, 0 };

Tín hiệu cho trạng thái di chuyển sang trái.

Byte đầu tiên 1: Mã trạng thái di chuyển sang trái.

uint8_t RIGHT[] = { 2, 0, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái di chuyển sang phải.

Byte đầu tiên 2: Mã trạng thái di chuyển sang phải.

uint8_t FORWARD[] = { 3, 0, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái di chuyển thẳng tiến.

Byte đầu tiên 3: Mã trạng thái di chuyển thẳng.

uint8_t BACKWARD_LEFT[] = { 4, 1, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái lùi sang trái.

Byte đầu tiên 4: Mã trạng thái lùi.

Byte thứ hai 1: Hướng lùi sang trái.

uint8_t BACKWARD_RIGHT[] = { 4, 2, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái lùi sang phải.

Byte thứ hai 2: Hướng lùi sang phải.

uint8_t FORWARD_LEFT[] = { 3, 1, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái tiến sang trái.

uint8_t FORWARD_RIGHT[] = { 3, 2, 0, 0, 0, 0, 0, 0 };:

Tín hiệu cho trạng thái tiến sang phải.

Chức năng hàm

Xác định trạng thái di chuyển tiếp theo dựa trên khoảng cách từ cảm biến, trạng thái cưỡng chế (**forceChange**), và trạng thái hiện tại.

Khi ta có (*forceChange == 0):

So sánh khoảng cách từ cảm biến:

Nếu **khoảng cách bên phải** (**distance2**) gần hơn:

-Nếu khoảng cách <= 100, trả về trạng thái **BACKWARD_RIGHT_STATE** để lùi sang phải.

-Nếu khoảng cách <= 900, trả về trạng thái **LEFT_STATE** để tránh va chạm bằng cách di chuyển sang trái.

Nếu **khoảng cách bên trái** (**distance1**) gần hơn:

-Nếu khoảng cách <= 100, trả về trạng thái **BACKWARD_LEFT_STATE.**

-Nếu khoảng cách <= 900, trả về trạng thái RIGHT_STATE.

Khi ta có (*forceChange == 1):

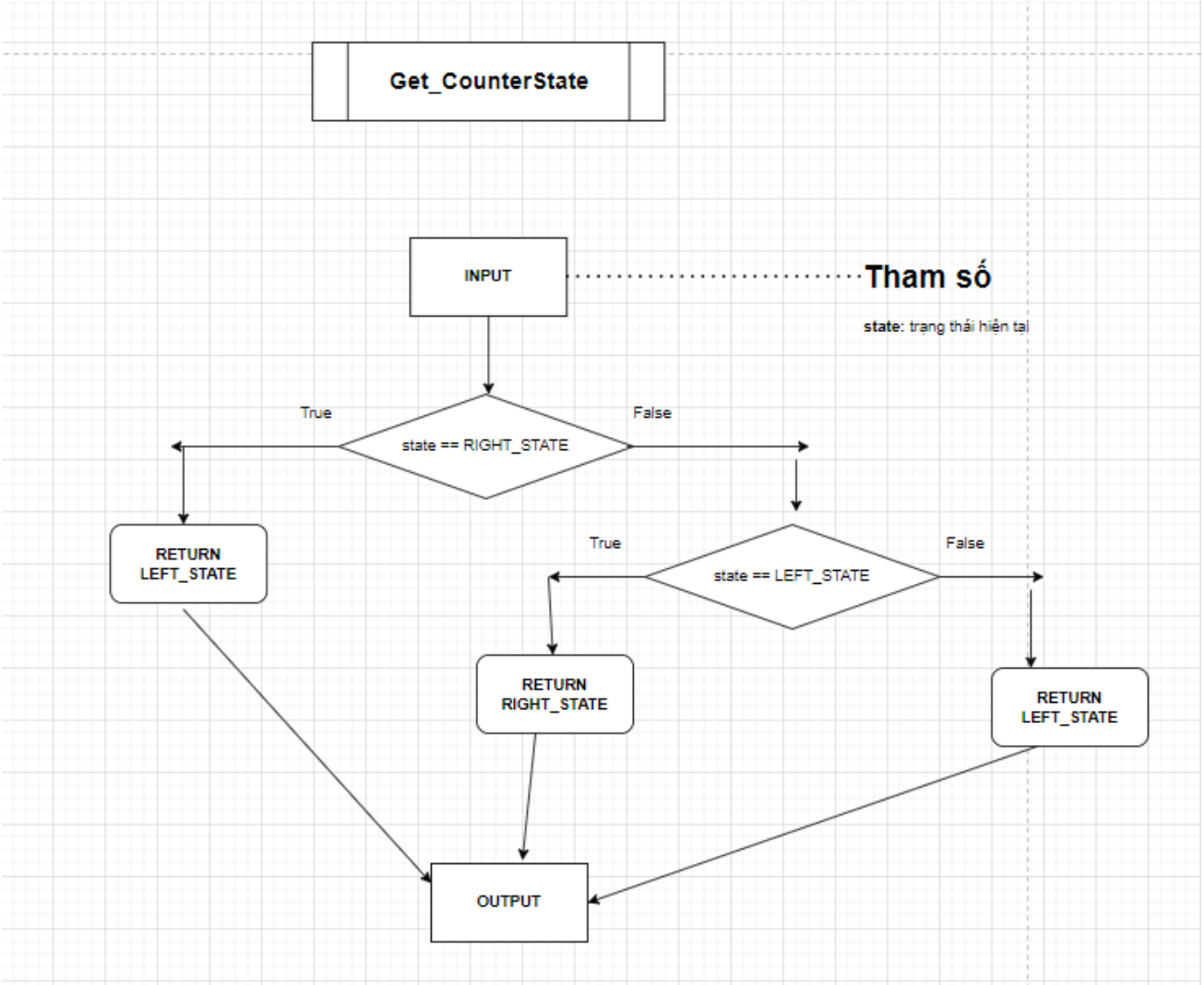
Gán lại: *forceChange = 0.

Trả về trạng thái đối nghịch của trạng thái hiện tại (dùng hàm **Get_CounterState**).

Mặc định:

Trả về trạng thái **FORWARD_STATE** (tiếp tục tiến về phía trước).

b) Hàm Get_CounterState:



```
uint8_t Get_CounterState(uint8_t state)
{
    if (state == RIGHT_STATE) {
        return LEFT_STATE;                  // Opposite of RIGHT is LEFT
    } else if (state == LEFT_STATE) {
        return RIGHT_STATE;                 // Opposite of LEFT is RIGHT
    } else {
        return LEFT_STATE;                  // Default counter-state is LEFT
    }
}
```

Phân tích:

Hàm Get_CounterState có nhiệm vụ xác định trạng thái đối nghịch (counter-state) của trạng thái hiện tại. Trạng thái đối nghịch được sử dụng trong trường hợp cần thay đổi hướng di chuyển một cách cưỡng chế (thường được kích hoạt bởi cờ forceChange trong hệ thống điều khiển).

Kiểm tra trạng thái hiện tại (state)

Nếu trạng thái hiện tại là RIGHT_STATE (di chuyển sang phải):

Hàm trả về LEFT_STATE (di chuyển sang trái).

Đây là trạng thái đối nghịch của RIGHT_STATE.

Nếu trạng thái hiện tại là LEFT_STATE (di chuyển sang trái):

Hàm trả về RIGHT_STATE (di chuyển sang phải).

Đây là trạng thái đối nghịch của LEFT_STATE.

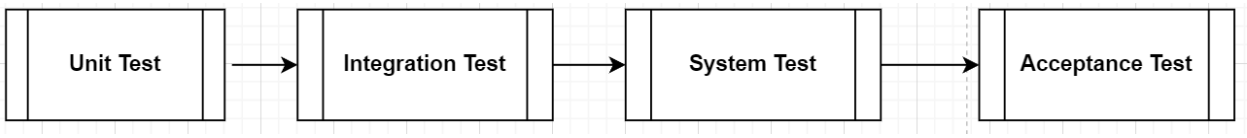
Trường hợp không nằm trong LEFT_STATE hoặc RIGHT_STATE:

Hàm trả về giá trị mặc định là LEFT_STATE.

Điều này đảm bảo hệ thống luôn có một trạng thái hợp lệ, dù tham số đầu vào không thuộc các trạng thái định nghĩa.

Chương 3. KIỂM THỬ

3.1. Quy trình kiểm thử



Hình 7. Quy trình kiểm thử

Quy trình kiểm thử được xây dựng như hình bên trên.

3.1.1. Unit Test

Thực hiện kiểm tra các hàm nằm trong **ActutorNode và SensorNode** để kiểm tra đánh giá các hàm đơn vị cơ bản.

3.1.2. Integration Test.

Thực hiện kết hợp 2 Node là **ActutorNode và SensorNode** để giao tiếp qua lại xử lý các chức năng cụ thể.

3.1.3. System Test.

Ta thực hiện kiểm tra toàn hệ thống bao gồm hiển thị LCD, module Wifi ESP32 và thực hiện nhận tín hiệu từ SenSor gửi vào SenSorNode để xử lý tín hiệu và truyền vào Actutor tương ứng với những tín hiệu sẽ điều khiển Motor quay và đánh lái Savor theo hướng xác định.

3.1.4. Acceptance Test.

Kiểm tra thực tiễn trên mô hình xe của Khoa Kỹ thuật Máy tính bằng cách nạp code qua ESP32 để đưa vào vi xử lý và kiểm tra. Kết quả mô phỏng thực hiện được đánh giá qua buổi báo cáo và thông qua video demo của nhóm.

Chương 4. KẾT LUẬN

4.1. Kết quả tổng quát

Phần mềm nhận diện vật cản và tránh vật cản trên xe mô hình của Khoa cung cấp có thể nhận diện được vật cản và đưa ra xử lý phù hợp trong một số điều kiện nhất định. Tuy nhiên vẫn có những khuyết điểm về mặt xử lý tín hiệu, thuật toán chưa tối ưu dẫn đến thời gian phản hồi chưa thực sự nhanh chóng

4.2. Ưu điểm và hạn chế của đồ án

4.2.1. Ưu điểm:

1. **Tự động hoá:**

   Giảm Sự Can Thiệp Thủ Công: Hệ thống xe mô hình hoạt động tự động trong việc nhận diện và tránh vật cản mà không cần sự điều khiển trực tiếp từ người dùng.

2. **Xử lý dữ liệu:**

   Cảm Biến VL53L1X: Cảm biến đo khoảng cách không tiếp xúc cung cấp dữ liệu về môi trường xung quanh xe, giúp hệ thống phản ứng kịp thời

3. **Giao tiếp tin cậy qua CAN Bus**

   Ổn định và tin cậy: Giao thức CAN bus cho phép truyền tải dữ liệu nhanh chóng và chính xác giữa các node SensorNode và ActuatorNode, đảm bảo sự liên lạc mạch lạc và ít bị nhiễu. Đây là giao tiếp phổ biến được sử dụng trong các phương tiện giao thông ngày nay

4.2.2. Hạn Chế

1. **Độ Trễ Phản Hồi:**

   Phản hồi chưa tối ưu: Độ trễ phản hồi hệ thống hiện tại có thể gây ra sự chậm trễ trong việc tránh vật cản, đặc biệt trong môi trường có vật cản di động nhanh

2. **Phụ thuộc vào kết nối và nguồn điện:**

   Phụ thuộc vào giao thức CAN: Hệ thống phụ thuộc vào độ ổn định của giao thức CAN bus. Sự cố về giao tiếp CAN có thể ảnh hưởng đến khả năng điều khiển và nhận diện của xe

Nguồn điện: Mất nguồn điện hoặc dao động điện có thể gây gián đoạn hoạt động của hệ thống; với việc 2 pin cho 2 phần khác nhau. Có thể xảy ra chênh lệch áp 2 bên dẫn đến việc nhận diện vật cản và đưa ra output xử lý không ổn định

3. **Khả năng xử lý:**

Giới hạn của thuật toán: Thuật toán hiện tại đôi lúc vẫn chưa ổn định lúc đưa ra quyết định di chuyển

Chưa Tích Hợp Các Cảm Biến Khác: Hệ thống chỉ sử dụng cảm biến VL53L1X, chưa tích hợp thêm các cảm biến như camera hoặc cảm biến siêu âm để tăng cường khả năng nhận diện môi trường

4. **Giới Hạn Trong Về Quy Mô và Ứng Dụng:**

Quy mô nhỏ: Đồ án tập trung vào phát triển phần mềm cho một mẫu thử nghiệm xe mini, chưa mở rộng ra các quy mô lớn hơn hoặc các ứng dụng thực tế trong các lĩnh vực khác như robot công nghiệp hay phương tiện giao thông tự hành

4.3. Hướng phát triển

4.3.1. Nâng cấp phần cứng:

Sử dụng cảm biến nhanh hơn và chính xác hơn: Tích hợp thêm các loại cảm biến như LIDAR, camera hoặc cảm biến siêu âm để tăng cường khả năng nhận diện và giảm độ trễ phản hồi

Cải thiện vi điều khiển: Nâng cấp lên các phiên bản vi điều khiển mạnh mẽ hơn hoặc sử dụng vi điều khiển đa nhân để tăng khả năng xử lý dữ liệu nhanh chóng và hiệu quả

4.3.2. Cải Thiện Thuật Toán Xử Lý Dữ Liệu:

Tối Ưu Hóa Thuật Toán Tránh Vật Cản: Phát triển các thuật toán xử lý tín hiệu và điều khiển thông minh hơn để giảm độ trễ và tăng độ chính xác trong việc tránh vật cản

Áp dụng trí tuệ nhân tạo: Sử dụng các kỹ thuật học máy (machine learning) để cải thiện khả năng nhận diện và dự đoán vị trí của vật cản, từ đó đưa ra các phản ứng điều khiển tối ưu; kèm theo đó là nâng cấp phần cứng để có thể đáp ứng được

4.3.3. Tích Hợp Thêm Công Nghệ:

Giao diện người dùng thông minh: Phát triển ứng dụng di động hoặc giao diện web để người dùng có thể theo dõi và điều khiển xe từ xa một cách dễ dàng thay vì nạp code qua OTA với WifiNode hiện tại

Lập bản đồ môi trường (SLAM): Tích hợp các thuật toán lập bản đồ và định vị để xe có thể tự tạo và cập nhật bản đồ môi trường xung quanh, tăng cường khả năng điều hướng tự động

LINK SOURCE CODE và VIDEO DEMO:

Link Github	Link github
Link Video Demo	Link Video