ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN ĐIỆN TỬ
----------o0o----------

LUẬN VĂN TỐT NGHIỆP ĐẠI HỌC

Hệ thống phát hiện chướng ngại vật trên xe ô tô sử dụng STM32 và RTOS

(Obstacle Detection System for Vehicles using STM32 and RTOS)

GVHD: PGS. TS. Trương Quang Vinh

SVTH: Võ Thành Danh

MSSV: 2110902

TP. HỒ CHÍ MINH, THÁNG 5 NĂM 2025

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

-----☆-----

Số: _____ /BKĐT
Khoa: **Điện – Điện tử**
Bộ Môn: **Điện Tử**

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập – Tự do – Hạnh phúc.

-----☆-----

NHIỆM VỤ LUẬN VĂN TỐT NGHIỆP

1. HỌ VÀ TÊN: VÕ THÀNH DANH MSSV: 2110902
2. NGÀNH: **ĐIỆN TỬ - VIỄN THÔNG** LỚP: DD21DV1
3. Đề tài: Hệ thống phát hiện chướng ngại vật trên xe ô tô sử dụng STM32 và RTOS.
4. Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):
   - Hiểu rõ bản chất và thiết kế thành công một RTOS có thể thực thi được trên vi điều khiển STM32.
   - Ứng dụng RTOS tự thiết kế để giải quyết các vấn đề về việc phát hiện chướng ngại vật sử dụng cảm biên siêu âm.
   - Thiết kế thành công mô hình và môi trường thử nghiệm đề tài.
   - Thiết kế hoàn chỉnh giao diện trực quan hoá đề tài.
5. Ngày giao nhiệm vụ luận văn: 10/02/2025
6. Ngày hoàn thành nhiệm vụ: 22/04/2025
7. Họ và tên người hướng dẫn: Phần hướng dẫn
   PGS. TS. Trương Quang Vinh

Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

Tp.HCM, ngày..... tháng..... năm 2025

CHỦ NHIỆM BỘ MÔN **NGƯỜI HƯỚNG DẪN CHÍNH**

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):.....
Đơn vị:.....
Ngày bảo vệ : .....
Điểm tổng kết: .....
Nơi lưu trữ luận văn: .....

LỜI CẢM ƠN

Lời đầu tiên, em xin chân thành cảm ơn thầy, PGS.TS. Trương Quang Vinh đã đồng ý cho em bảo vệ bài Đồ án "Hệ thống phát hiện chướng ngại vật trên xe ô tô sử dụng STM32 và RTOS" của em, đồng thời, thầy đã giúp đỡ em hết mình, tạo cho em những điều kiện tốt nhất cũng như đề xuất những hướng đi hợp lý và đúng đắn trong quá trình 3 tháng thực hiện Đồ án qua.

Em xin chân thành cảm ơn!

Tp. Hồ Chí Minh, ngày 23 tháng 5 năm 2025.

Sinh viên

TÓM TẮT ĐỒ ÁN

An toàn giao thông là một trong những yếu tố quan trọng trong quá trình vận hành xe ô tô, đặc biệt là trong môi trường đô thị với mật độ phương tiện cao. Để hỗ trợ người lái, hệ thống phát hiện chướng ngại vật được thiết kế nhằm cảnh báo về các vật thể xung quanh xe, giúp giảm thiểu nguy cơ va chạm khi di chuyển hoặc đỗ xe. Đề tài này tập trung nghiên cứu và phát triển một hệ thống phát hiện chướng ngại vật trên xe ô tô sử dụng kit phát triển từ vi điều khiển STM32 kết hợp với các cảm biến khoảng cách và hệ điều hành thời gian thực (RTOS) để xử lý và hiển thị thông tin theo thời gian thực.

Hệ thống có khả năng phát hiện vật cản trước, sau và hai bên xe, sử dụng cảm biến khoảng cách để đo lường khoảng cách đến các vật thể xung quanh. Dữ liệu thu thập sẽ được xử lý bởi vi điều khiển STM32 và hiển thị lên màn hình OLED hoặc gửi đến máy tính qua UART để mô phỏng trên phần mềm WinForms. Hệ thống cũng tích hợp cảnh báo bằng LED và buzzer để thông báo khi phát hiện vật cản quá gần, giúp người lái có thể phản ứng kịp thời.

Hệ thống sử dụng kit phát triển vi điều khiển STM32F103C8T6, kết nối với các cảm biến khoảng cách siêu âm HC-SR04, tích hợp trên một mạch PCB tùy chỉnh để xử lý dữ liệu một cách hiệu quả.

Vi điều khiển STM32 sẽ được lập trình để thu thập dữ liệu từ cảm biến. RTOS sẽ chia hệ thống thành các tác vụ nhỏ, bao gồm đọc cảm biến, xử lý dữ liệu, hiển thị thông tin và cảnh báo. Việc sử dụng RTOS giúp tối ưu hóa việc xử lý đa nhiệm, đảm bảo cảnh báo có độ trễ thấp ngay khi phát hiện vật cản. Dữ liệu cũng có thể được truyền đến giao diện được lập trình dựa vào framework Qt trên PC để hiển thị trực quan và phân tích chi tiết hơn. Thông tin khoảng cách sẽ được hiển thị trên màn hình OLED, đồng thời các vật cản sẽ được mô phỏng trên phần mềm Qt với cảnh báo trực quan.

Hệ thống phát hiện chướng ngại vật trên xe ô tô là một giải pháp hỗ trợ người lái trong việc đảm bảo an toàn khi di chuyển, đặc biệt trong các không gian hẹp như bãi đỗ xe hoặc đường phố đông đúc. Việc kết hợp STM32 với RTOS giúp hệ thống hoạt động ổn định, xử lý dữ liệu nhanh chóng và có khả năng mở rộng trong tương lai. Hệ thống này có thể được tích hợp thêm nhận diện vật thể bằng camera, cảm biến radar chính xác cao hơn, hoặc kết nối với hệ thống phanh tự động để phát triển thành một hệ thống an toàn tiên tiến hơn.

MỤC LỤC

DANH SÁCH HÌNH MINH HỌA

DANH SÁCH BẢNG SỐ LIỆU

DANH MỤC VIẾT TẮT

Ký hiệu và chữ viết tắt	Diễn giải
ADAS	Advanced Driver Assistance Systems – Hệ thống hỗ trợ tài xế nâng cao
AI	Artificial Intelligence – Trí tuệ nhân tạo
CPU	Central Processing Unit – Vi xử lý trung tâm
FIFO	First-In First-Out – Kiểu vùng nhớ dữ liệu nào vào trước sẽ được xử lý trước
GPIO	General-Purpose Input/Output – Chân đọc/ghi tín hiệu điện áp đa dụng
ID	Identification – Thông số nhận dạng
LiDAR	Light Detection and Ranging – Phát hiện vật thể bằng ánh sáng
LIFO	Last-In First-Out – Kiểu vùng nhớ dữ liệu nào vào sau sẽ được xử lý trước
ODS	Obstacle Detection System – Hệ thống phát hiện chướng ngại vật
PCB	Printed Circuit Board – Mạch điện in
RTOS	Real-Time Operating System – Hệ điều hành thời gian thực
TCB	Task Control Block – Khối điều khiển luồng
UART	Universal Asynchronous Receiver-Transmitter – Bộ truyền/nhận bất đồng bộ

1. GIỚI THIỆU

1.1 Định nghĩa đề tài

Mỗi khi mở các chương trình thời sự hay đọc các bài báo tin tức hằng ngày, ta đều xem qua các bài nói về tai nạn giao thông và thiệt hại của nó gây ra. Sự gia tăng nhanh chóng của phương tiện giao thông trên toàn cầu kéo theo những thách thức nghiêm trọng về an toàn đường bộ. Hàng năm, các vụ tai nạn giao thông do va chạm với chướng ngại vật trên đường chiếm một tỷ lệ đáng kể trong tổng số vụ tai nạn. Tai nạn giao thông do va chạm là một trong những nguyên nhân chính gây ra thương vong và thiệt hại về tài sản trên toàn thế giới. Theo Tổ chức Y tế Thế giới (WHO), mỗi năm có khoảng 1,19 triệu người tử vong do tai nạn giao thông, trong đó một tỷ lệ lớn xuất phát từ các vụ va chạm với chướng ngại vật như xe cộ, người đi bộ, động vật băng qua đường, hay các vật thể cố định trên đường[1]. Những vụ tai nạn này không chỉ gây nguy hiểm cho người lái mà còn ảnh hưởng nghiêm trọng đến người tham gia giao thông khác, làm gián đoạn giao thông và gây tổn thất kinh tế lớn cho xã hội.

Nguyên nhân của các vụ va chạm này có thể đến từ sự thiếu quan sát của tài xế, điều kiện ánh sáng kém, thời tiết xấu, hoặc sự xuất hiện bất ngờ của vật cản trên đường. Các tác nhân này thường xuất hiện rất nhanh và bất ngờ, có thể là chỉ trong vài tích tắc. Trong những tình huống như vậy, phản ứng của con người thường bị hạn chế bởi yếu tố thời gian và khả năng nhận thức, dẫn đến những tình huống nguy hiểm không thể tránh kịp. Chính vì vậy, nhu cầu về một hệ thống có khả năng phát hiện và cảnh báo chướng ngại vật tự động đang được các nhà phát triển các loại phương tiện giao thông để tâm đến và hiện tại đang coi như là một phần không thể thiếu trong các mẫu phương tiện giao thông hiện đại, đặc biệt là xe ô tô.

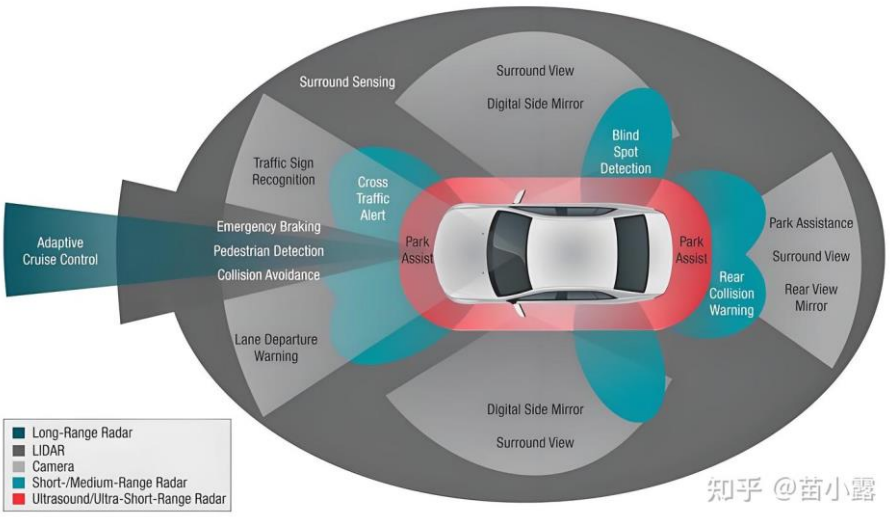
Hệ thống này ứng dụng các công nghệ cảm biến hiện đại như LiDAR, radar, camera và cảm biến siêu âm, giúp phương tiện có thể nhận diện vật cản theo thời gian thực. Khi phát hiện vật thể có nguy cơ va chạm, hệ thống sẽ đưa ra cảnh báo bằng tín hiệu âm thanh, hình ảnh hoặc rung động, thậm chí có thể can thiệp tự động bằng cách phanh khẩn cấp hoặc điều chỉnh hướng lái để tránh va chạm. Các công nghệ này không chỉ giúp xe phát hiện các chướng ngại vật tĩnh như biển báo giao thông, dải phân cách, cột đèn mà còn có thể nhận diện các vật thể di chuyển như người đi bộ, xe đạp, xe máy hoặc động vật băng qua đường.

[1] Trích từ *"Báo cáo về tình trạng an toàn đường bộ trên thế giới năm 2023"* của WHO

Nhờ đó, người lái có thể phản ứng kịp thời với những tình huống nguy hiểm, giảm thiểu đáng kể nguy cơ xảy ra tai nạn.

Cùng với sự phát triển của trí tuệ nhân tạo và các thuật toán xử lý hình ảnh, các hệ thống phát hiện chướng ngại vật ngày càng trở nên thông minh và chính xác hơn. Trước đây, các hệ thống này chủ yếu dựa vào cảm biến siêu âm để đo khoảng cách với vật cản ở tốc độ thấp, thường chỉ được sử dụng trong hỗ trợ đỗ xe. Tuy nhiên, với sự phát triển mạnh mẽ của công nghệ AI và Machine Learning, các hệ thống hiện đại có thể phân tích dữ liệu từ camera và LiDAR để dự đoán chuyển động của vật thể, giúp xe có thể phản ứng trước khi nguy hiểm xảy ra. Đây là nền tảng quan trọng cho sự phát triển của các phương tiện tự hành, nơi xe có thể tự động phát hiện và phản ứng với mọi tình huống mà không cần sự can thiệp của con người.

Hiện nay, hệ thống phát hiện chướng ngại vật là một phần không thể thiếu trong các hệ thống hỗ trợ lái xe tiên tiến (ADAS - Advanced Driver Assistance Systems). Các dòng xe hiện đại, từ phân khúc phổ thông đến cao cấp, đều được trang bị các công nghệ này để nâng cao mức độ an toàn. Ví dụ, các hệ thống phanh khẩn cấp tự động, cảnh báo điểm mù, phát hiện người đi bộ, hỗ trợ giữ làn đường đều sử dụng công nghệ phát hiện chướng ngại vật để hỗ trợ tài xế.



Hình 1.1: Tầm nhìn của ODS trong ADAS

Sự ra đời của hệ thống phát hiện chướng ngại vật không chỉ mang lại lợi ích cho cá nhân người lái mà còn đóng góp quan trọng vào sự phát triển của hệ thống giao thông thông minh. Khi ngày càng nhiều phương tiện được trang bị các hệ thống này, tai nạn giao thông sẽ

được giảm thiểu đáng kể, tạo ra một môi trường giao thông an toàn và hiệu quả hơn. Trong tương lai, với sự phát triển của xe tự lái, công nghệ phát hiện chướng ngại vật sẽ tiếp tục đóng vai trò then chốt, giúp phương tiện có thể nhận diện, phân tích và xử lý tình huống một cách hoàn toàn tự động, tiến gần hơn đến mục tiêu xây dựng hệ thống giao thông không tai nạn.

1.2 Nhiệm vụ đề tài

Đề tài này nhằm thiết kế và phát triển một hệ thống phát hiện chướng ngại vật trên xe ô tô, sử dụng vi điều khiển STM32 và RTOS kết hợp với cảm biến khoảng cách HC-SR04 để nhận diện vật cản trong quá trình di chuyển hoặc đỗ xe. Hệ thống sẽ giúp người lái phát hiện sớm các vật cản bằng cảnh báo trực quan và âm thanh, đồng thời cung cấp giao diện đồ họa trên máy tính để hiển thị dữ liệu đo được.

Các nhiệm vụ chính của đề tài này như sau:

* **Tìm hiểu nguyên lý hoạt động của cảm biến siêu âm HC-SR04**

Đề tài sẽ sử dụng cảm biến siêu âm HC-SR04 để đo khoảng cách từ mô hình xe cho tới các vật cản. Đề tài sẽ tìm hiểu chức năng, nguyên lý hoạt động và cách sử dụng cảm biến US-100. Đề tài dựa vào các kiến thức tìm được ở trên để viết chương trình để giao tiếp với cảm biến nhằm lấy được những thông số cần thiết cho đề tài.

* **Thiết kế phần vỏ ngoài sao chép hình dáng của xe ô tô**

Đề tài sẽ thiết kế vỏ ngoài để có thể giữ các cảm biến, kit STM32, các tín hiệu, dây điện, PCB,… thành 1 khối thống nhất. Đồng thời, vỏ ngoài này giúp đề tài hoạt động trong một khối gần giống với xe ô tô nhất có thể.

* **Thiết kế PCB để kết nối các cảm biến và các tín hiệu trực quan như LED hay buzzer với kit STM32**

Một PCB giúp cho hệ thống dây điện bên trong mô hình được gọn gàng hơn và chắc chắn hơn việc chỉ nối dây vào các mối nối. Ngoài ra, chiếc PCB còn giúp ta có thể mở rộng các tính năng nếu cần thiết hoặc thay thế các linh kiện tiện lợi hơn cách nối dây truyền thống. Đề tài sẽ thiết kế một PCB giúp cho các cảm biến, các LED, buzzer,… có nơi để kết nối tới vi điều khiển và giúp cho đề tài được ngăn nắp và chắc chắn hơn.

* **Thiết kế RTOS chạy được trên vi điều khiển STM32**

Một ODS rất coi trọng việc hệ thống sẽ phản ứng nhanh như thế nào với các yếu tố bất ngờ xuất hiện trên đường. Vì thế ODS là một hệ thống hard real-time. Để có thể phục vụ được yêu cầu này của đề tài, cũng như việc phải làm việc với nhiều cảm biến US-100 cùng lúc thì đề tài sẽ thiết kế một RTOS nhằm đáp ứng được yêu cầu hard real-time và làm được gần như song song các tác vụ khác nhau.

Đề tài này sẽ tự thiết kế và phát triển một RTOS có đủ những tính năng cơ bản và cần thiết đối với một RTOS như khả năng quản lý đa thread, sleep, periodic task, và inter-thread communication,… RTOS này sẽ được viết bằng ngôn ngữ C và Assembly để tối ưu hoá thời gian thực lệnh các lệnh để đáp ứng với yêu cầu real-time của hệ thống, và sẽ được kiểm tra và thử nghiệm trên môi trường KeilC và vi điều khiển STM32F103C8T6.

* **Viết chương trình UI hiển thị trên máy tính sử dụng Qt**

Hệ thống sẽ có chiếc màn hình OLED SSD1306 gắn trên mô hình để hiển thị các thông tin mà hệ thống thu được. Tuy nhiên chiếc màn hình này có một số nhược điểm đáng chú ý như hiển thị được khá ít điểm ảnh và màu so với nhu cầu của đề tài, nên đề tài sẽ có thêm một phương thức hiển thị là màn hình máy tính. Màn hình máy tính đương nhiên là lớn hơn, nhiều màu hơn và linh hoạt hơn màn hình OLED SSD1306, giúp ta quan sát rõ hơn các hoạt động của đề tài. Đề tài sẽ giao tiếp với máy tính thông qua UART và các thông tin hiển thị sẽ được trình chiếu thông qua framework Qt.

* **Tích hợp các phần cứng và phần mềm, thử nghiệm, kiểm tra quá trình chạy của hệ thống**

Các bước kể trên sẽ được ghép lại để thành một khối thống nhất và hoạt động hiệu quả. Người thực hiện đề tài có nhiệm vụ cho đề tài chạy thử nghiệm các trường hợp có thể có và quan sát quá trình hoạt động của đề tài, qua đó sửa lỗi, cải tiến và đưa ra các kết quả, nhận định về đề tài.

* **Viết báo cáo cho đề tài dựa vào hiểu biết bản thân và kết quả của đề tài**

1.3 Phạm vi đề tài

Với quy mô của một luận văn tốt nghiệp, đề tài này sẽ có một vài giới hạn so với một ODS ngoài thị trường. Đề tài này tập trung vào việc nhận diện và cảnh báo vật cản xung quanh xe, giúp người lái có thể đưa ra quyết định kịp thời để tránh va chạm trong quá trình di

chuyển hoặc đỗ xe. Vì lý do đó và khoảng thời gian cho phép để thực hiện thì đề tài có một số giới hạn sau

1.3.1 Giới hạn về công nghệ

Trong khuôn khổ của đề tài này, hệ thống phát hiện chướng ngại vật trên xe ô tô sẽ sử dụng vi điều khiển STM32 để thu thập, xử lý dữ liệu từ các cảm biến khoảng cách và đưa ra cảnh báo trực quan. Do giới hạn về thời gian, nguồn lực và khả năng triển khai, đề tài sẽ tập trung vào các công nghệ cơ bản nhưng hiệu quả, thay vì áp dụng những công nghệ phức tạp như thị giác máy hoặc AI. Ngoài ra, vì lý do kinh phí nên đề tài không đi quá xa với các lựa chọn công nghệ, cụ thể sau đây:

* Sử dụng vi điều khiển lõi ARM Cortex-M3 thay vì các vi điều khiển lõi cao cấp hơn bởi vì RTOS được sử dụng trong đề tài không quá phức tạp hay sử dụng quá nhiều tài nguyên của vi điều khiển nên sẽ chỉ cần vi điều khiển tầm trung, không quá mạnh hay quá yếu.
* Sử dụng các cảm biến siêu âm thay vì các công nghệ đo khoảng cách cao cấp hơn vì cảm biến siêu âm dễ sử dụng, dễ giao tiếp, dễ tích hợp vào hệ thống và đáp ứng đủ tốc độ cho hệ thống yêu cầu real-time.
* Sử dụng màn hình OLED đơn giản thay vì các màn hình Android ở ngoài thị trường. Các màn hình Android cần rất nhiều thời gian để lập trình và cho nó chạy theo đúng ý đề tài.
* Phần vỏ ngoài của đề tài chỉ làm bằng các vật liệu nhẹ, không sử dụng các vật liệu giống một chiếc xe ô tô ngoài thị trường.
* Phần nguồn mà đề tài sử dụng sẽ chỉ là các nguồn pin đơn giản chứ không sử dụng nguồn từ ắc quy như các hệ thống ngoài thị trường.

1.3.2 Giới hạn về thử nghiệm

Kích thước của đề tài có khả năng cao là sẽ nhỏ hơn mặt bàn làm việc cơ bản, vì thế không thể mang đề tài ra ngoài đường để thử nghiệm với các tình huống thực tế. Thay vào đó, đề tài sẽ được thử nghiệm trong môi trường trong nhà và các tình huống thực tế sẽ được thay thế bằng các hành động đơn giản như ném đồ vật ngang qua đề tài hay đặt một đồ vật bên cạnh đề tài. Các kết quả sẽ được quan sát qua màn hình OLED và màn hình máy tính, thay thế cho màn hình Android.

1.4 Ứng dụng của đề tài

Hệ thống phát hiện chướng ngại vật được thiết kế với mục tiêu hỗ trợ các phương tiện giao thông đường bộ, đặc biệt là ô tô cá nhân, xe tải nhỏ và xe mô hình thử nghiệm. Hệ thống này tập trung vào việc nhận diện và cảnh báo vật cản xung quanh xe, giúp người lái có thể đưa ra quyết định kịp thời để tránh va chạm trong quá trình di chuyển hoặc đỗ xe. Ngoài thị trường, ODS thường được tích hợp vào ADAS để hỗ trợ tài xế trong việc điều khiển phương tiện của họ.

Tuy nhiên với quy mô của đề tài thì nó có thể ứng dụng vào các phương tiện di chuyện mang tính cá nhân như xe đạp hay xe máy. Đề tài có thể hoạt động tốt trong các môi trường đường phố bình thường nếu như được bảo vệ và cài đặt đúng cách. Đề tài hoàn toàn có thể giúp cho người điều khiển phương tiện có cái nhìn tốt hơn về các vật thể xung quanh họ. Ngoài ra, đề tài còn có thể tích hợp vào các dự án di động khác như AGV (Automotive Guided Vehicle), robot hay các dự án hỗ trợ cho đời sống thường ngày như máy cắt cỏ, robot hút bụi,…

1.5 Hướng phát triển đề tài trong tương lai

Sự phát triển của công nghệ ô tô ngày càng đòi hỏi các hệ thống hỗ trợ lái xe thông minh và hiệu quả hơn nhằm giảm thiểu tai nạn giao thông, nâng cao trải nghiệm người dùng và hướng tới những phương tiện tự hành trong tương lai. Hệ thống phát hiện chướng ngại vật trên xe ô tô trong đề tài này, mặc dù đã đáp ứng được các yêu cầu cơ bản về phát hiện vật cản và cảnh báo nguy hiểm, nhưng vẫn còn nhiều tiềm năng để mở rộng và nâng cấp. Trong tương lai, hệ thống có thể được phát triển theo nhiều hướng khác nhau để tăng độ chính xác, tính chủ động và khả năng tích hợp với các công nghệ tiên tiến hơn.

Một trong những hướng phát triển quan trọng của hệ thống là nâng cấp và đa dạng hóa các loại cảm biến để tăng độ chính xác và khả năng nhận diện vật thể. Hiện tại, hệ thống chủ yếu sử dụng cảm biến siêu âm để đo khoảng cách và phát hiện vật cản tĩnh. Tuy nhiên, trong tương lai, có thể kết hợp thêm các loại cảm biến cao cấp hơn như cảm biến LiDAR, cho phép xe quét môi trường xung quanh bằng laser, tạo ra bản đồ 3D chi tiết, giúp nhận diện vật cản với độ chính xác cao, ngay cả trong điều kiện ánh sáng yếu hoặc thời tiết xấu.

Trong đa số các ADAS đều có công nghệ phân biệt làn đường cũng như môi trường xung quanh của phương tiện. Vì vậy hệ thống phải có một thiết bị giúp hệ thống có được "con mắt" để quan sát mọi thứ. Đề tài trong tương lai có thể tích hợp camera để giúp hệ thống

có được "con mắt" của riêng nó. Việc tích hợp camera độ phân giải cao cùng với các thuật toán xử lý ảnh sẽ giúp hệ thống không chỉ đo khoảng cách mà còn có thể nhận diện loại vật thể như người đi bộ, xe cộ, biển báo giao thông, động vật băng qua đường,…

Một trong những bước tiến quan trọng trong hệ thống hỗ trợ lái xe thông minh là ứng dụng AI và Machine Learning để cải thiện khả năng nhận diện và phân tích tình huống giao thông. Hiện tại, hệ thống phát hiện chướng ngại vật chủ yếu hoạt động dựa trên dữ liệu cảm biến thô, tức là đo khoảng cách và kích hoạt cảnh báo khi vật cản ở gần. Tuy nhiên, trong tương lai, AI có thể giúp hệ thống phân biệt các loại vật thể, dự đoán hành vi của vật thể di chuyển, tự động điều chỉnh mức độ cảnh báo. AI và Machine Learning cũng có thể được triển khai trên các nền tảng RTOS tiên tiến hơn, giúp hệ thống xử lý dữ liệu nhanh chóng và hiệu quả hơn.

Hệ thống phát hiện chướng ngại vật có thể được tích hợp vào hệ thống hỗ trợ lái xe tiên tiến ADAS để cung cấp khả năng hỗ trợ toàn diện hơn, trong đó có một số tính năng được thêm vào hệ thống như phanh khẩn cấp tự động, hỗ trợ giữ làn đường, kiểm soát hành trình thích ứng,…

Hiện tại, hệ thống đang sử dụng vi điều khiển STM32 để xử lý dữ liệu cảm biến, nhưng trong tương lai, có thể nâng cấp lên các nền tảng phần cứng mạnh mẽ hơn như bộ xử lý nhúng, FPGA,… Ngoài ra, việc tối ưu thuật toán trên RTOS cũng sẽ giúp hệ thống hoạt động nhanh hơn, giảm độ trễ khi xử lý dữ liệu từ cảm biến và đưa ra cảnh báo.

Hệ thống phát hiện chướng ngại vật trên xe ô tô vẫn còn nhiều tiềm năng phát triển trong tương lai, với các cải tiến về công nghệ cảm biến, ứng dụng trí tuệ nhân tạo, tích hợp với ADAS và kết nối với mạng lưới giao thông thông minh. Khi các công nghệ này ngày càng phát triển, hệ thống có thể tiến gần hơn đến mục tiêu tăng cường an toàn giao thông, hỗ trợ lái xe hiệu quả hơn và mở đường cho các phương tiện tự hành trong tương lai.

2. CƠ SỞ LÝ THUYẾT

2.1 Real-Time Operating System

2.1.1 Hệ thống thời gian thực

Một hệ thống thời gian thực là một hệ thống máy tính mà độ tin cậy của nó không chỉ phụ thuộc vào kết quả tính toán mà còn phụ thuộc vào thời điểm kết quả đó được tạo ra[1]. Trong các hệ thống này, yếu tố thời gian giữ vai trò quan trọng tương đương với tính chính xác logic. Điều này có nghĩa là một hệ thống có thể được xem là "sai" nếu nó trả về kết quả đúng nhưng vượt quá thời gian cho phép. Đặc điểm quan trọng nhất của hệ thống thời gian thực là khả năng tuân thủ các ràng buộc về thời gian một cách nghiêm ngặt và có thể dự đoán trước. Hệ thống phải phản hồi đúng lúc với các sự kiện phát sinh từ môi trường hoặc từ các thiết bị ngoại vi, và thời gian phản hồi này thường được quy định bởi các giới hạn rõ ràng gọi là deadline.

Trong thực tế, hệ thống thời gian thực thường được phân loại thành ba nhóm: hệ thống thời gian thực cứng (hard real-time), hệ thống thời gian thực mềm (soft real-time) và hệ thống thời gian thực vừa (firm real-time). Ở loại đầu tiên, mọi deadline đều phải được đáp ứng tuyệt đối. Nếu một deadline bị bỏ lỡ, hệ thống có thể rơi vào trạng thái nguy hiểm hoặc gây hậu quả nghiêm trọng như mất kiểm soát, thiệt hại tài sản, thậm chí đe dọa tính mạng con người – ví dụ điển hình là hệ thống điều khiển phanh trong ô tô hoặc hệ thống điều khiển tên lửa. Trong khi đó, hệ thống thời gian thực mềm cho phép một số deadline bị bỏ lỡ mà không gây hậu quả nghiêm trọng, tuy nhiên hiệu suất hoặc chất lượng dịch vụ sẽ giảm. Hệ thống thời gian thực vừa nằm giữa hai loại trên; việc bỏ lỡ deadline có thể được chấp nhận trong một giới hạn nhất định, nhưng nếu vượt quá ngưỡng đó, toàn bộ hệ thống có thể thất bại.

Các hệ thống thời gian thực hiện diện trong nhiều lĩnh vực quan trọng như hàng không, ô tô, y tế, tự động hóa công nghiệp và các thiết bị IoT. Trong lĩnh vực hàng không vũ trụ, hệ thống điều khiển chuyến bay hoặc dẫn đường cần đảm bảo chính xác tuyệt đối về mặt thời gian. Trong lĩnh vực y tế, các thiết bị như máy trợ tim hoặc máy thở cần có phản hồi tức thời để duy trì sự sống cho bệnh nhân. Còn trong công nghiệp, các dây chuyền sản xuất tự động hoặc robot điều khiển cũng yêu cầu xử lý dữ liệu trong thời gian giới hạn nhằm đảm bảo sự an toàn và tối ưu vận hành.

2.1.2 Giới thiệu về RTOS

Một hệ điều hành thời gian thực (Real-Time Operating System – RTOS) là một phần mềm hệ điều hành được thiết kế chuyên biệt để đáp ứng yêu cầu của các hệ thống thời gian thực. Khác với các hệ điều hành phổ thông như Windows hay Linux, RTOS cung cấp một môi trường thực thi đảm bảo rằng các tác vụ sẽ được xử lý trong khoảng thời gian cho phép, phù hợp với các ràng buộc thời gian đã định trước. Một RTOS thường được xây dựng theo mô hình đa nhiệm ưu tiên, cho phép nhiều tác vụ đồng thời tồn tại trong hệ thống, nhưng chỉ một tác vụ duy nhất được thực thi tại một thời điểm. Việc chọn tác vụ nào để thực hiện phụ thuộc vào mức độ ưu tiên được gán cho từng tác vụ và vào cơ chế lập lịch của RTOS.

Mục tiêu chính của một RTOS là tối ưu hóa khả năng đáp ứng đúng hạn, đảm bảo rằng các tác vụ quan trọng nhất luôn được thực thi kịp thời. Điều này đạt được thông qua các thành phần chính như bộ lập lịch (scheduler), các cơ chế đồng bộ hóa (semaphore, mutex), các công cụ truyền thông giữa các tiến trình (message queue, mailbox) và quản lý thời gian (timers, system tick). Scheduler là trung tâm điều phối việc thực thi tác vụ, đảm bảo tác vụ có mức độ ưu tiên cao nhất trong hàng đợi sẵn sàng luôn được chọn để thực thi. Trong khi đó, semaphore và mutex giúp điều phối truy cập tới các tài nguyên dùng chung, tránh tình trạng tranh chấp và đảm bảo tính nhất quán dữ liệu.

Một điểm quan trọng trong thiết kế của RTOS là tính xác định. Tính chất này đòi hỏi rằng mọi dịch vụ do RTOS cung cấp (như tạo task, chuyển trạng thái, chờ tín hiệu) đều có thời gian thực thi giới hạn trên rõ ràng và không phụ thuộc vào trạng thái bên ngoài. Điều này khác biệt với hệ điều hành phổ thông, nơi độ trễ có thể biến thiên đáng kể. Bên cạnh đó, RTOS còn tối ưu hóa thời gian chuyển đổi ngữ cảnh (context switching), thời gian phản hồi ngắt và chiếm dụng bộ nhớ ở mức tối thiểu, giúp hệ thống vận hành hiệu quả ngay cả trên các vi điều khiển có tài nguyên hạn chế.

2.1.3 Luồng (thread)

Trong RTOS, luồng (thread) hay còn gọi là tác vụ (task) là đơn vị thực thi nhỏ nhất được lập lịch bởi scheduler của hệ điều hành. Mỗi luồng là một đoạn chương trình độc lập, có thể thực hiện song song hoặc ngắt quãng với các luồng khác trong hệ thống. RTOS quản lý nhiều luồng để xử lý các phần việc riêng biệt như đọc cảm biến, truyền dữ liệu, xử lý tín hiệu, hoặc giao tiếp với người dùng.

Theo định nghĩa trong tài liệu của Silicon Labs, mỗi luồng bao gồm ba thành phần chính: mã thực thi, dữ liệu và biến riêng và vùng stack độc lập[2]. Mỗi luồng thường được lập trình dưới dạng vòng lặp vô hạn và hoạt động độc lập với các luồng khác, chỉ bị gián đoạn bởi việc chờ sự kiện, nhận tín hiệu hoặc bị hệ điều hành dừng lại để chuyển sang luồng khác có mức ưu tiên cao hơn.

Tương tự như trong các hệ điều hành hiện đại, một luồng trong RTOS cũng có thể nằm ở một trong các trạng thái sau, tùy thuộc vào điều kiện thực thi và tài nguyên sẵn có. Các trạng thái cơ bản này tạo nên mô hình vòng đời của một luồng, như sau:

- **Idle:** Luồng đã được tạo nhưng chưa được kích hoạt hoặc đang ở trạng thái không hoạt động. Luồng trong trạng thái này không tiêu tốn CPU.
- **Ready:** Luồng sẵn sàng để được thực thi, nhưng chưa được CPU cấp thời gian. Các luồng trong trạng thái này được xếp hàng theo mức độ ưu tiên.
- **Running:** Luồng đang được thực thi. RTOS đảm bảo chỉ một luồng được thực thi tại một thời điểm trên mỗi nhân CPU.
- **Waiting (Blocked):** Luồng tạm dừng, đang chờ một sự kiện cụ thể xảy ra, ví dụ: chờ semaphore, chờ I/O, hoặc chờ hết thời gian delay.
- **Suspended:** Luồng bị hệ thống hoặc người dùng tạm dừng chủ động, thường dùng để tiết kiệm năng lượng hoặc thay đổi lịch trình.
- **Terminated:** Luồng kết thúc và bị loại khỏi hệ thống. Trạng thái này thường ít gặp hơn trong RTOS vì hầu hết các luồng đều là vòng lặp vô hạn.

Mỗi luồng được quản lý một cách tách biệt thông qua bộ điều khiển luồng (Task Control Block – TCB) và đặc biệt là vùng stack riêng. Stack trong mỗi luồng là một vùng nhớ dạng Last In First Out (LIFO), dùng để lưu trữ các biến cục bộ trong hàm, con trỏ return và dữ liệu từ các lời gọi hàm lồng nhau, thông tin ngữ cảnh CPU như giá trị các thanh ghi (R0 – R15, xPSR) khi xảy ra ngắt hoặc khi thực hiện context switching.Việc tách stack cho từng luồng là một nguyên tắc thiết yếu trong RTOS nhằm đảm bảo tính độc lập, tránh ghi đè chéo dữ liệu giữa các luồng, và giúp quá trình lưu, phục hồi ngữ cảnh thực hiện chính xác và nhanh chóng.

Trong kiến trúc Cortex-M, khi chuyển đổi ngữ cảnh xảy ra, Stack Pointer (SP) sẽ được cập nhật để trỏ tới vùng stack tương ứng của luồng mới. Toàn bộ các thanh ghi quan trọng sẽ được lưu vào vùng stack hiện tại của luồng đang bị tạm dừng và phục hồi từ stack của luồng sắp được chạy. Điều này đòi hỏi mỗi stack phải được cấp phát đủ và đúng địa chỉ, nếu không sẽ dẫn tới lỗi không khôi phục được trạng thái trước đó của CPU.

2.1.4 Bộ lập lịch (scheduler)

Trong mọi hệ điều hành đa nhiệm, đặc biệt là RTOS, scheduler là thành phần đóng vai trò trung tâm. Scheduler quyết định tác vụ nào được phép sử dụng CPU tại một thời điểm, từ đó đảm bảo tính đúng hạn, tính đáp ứng và hiệu quả xử lý tài nguyên hệ thống. Khác với các hệ điều hành truyền thống vốn hướng đến hiệu năng tổng thể hoặc trải nghiệm người dùng, scheduler trong RTOS cần phải đảm bảo các ràng buộc về thời gian, phải hoàn thành trước deadline một cách chắc chắn và có thể tiên đoán được. Do đó, cơ chế lập lịch thường đơn giản, rõ ràng, với độ trễ lập lịch thấp và có thể tùy chỉnh theo đặc tính ứng dụng.

Có nhiều thuật toán lập lịch khác nhau trong RTOS, có thể phân chia theo tiêu chí có sử dụng mức ưu tiên cho từng tác vụ hay không. Ví dụ nếu scheduler được thiết kế để ưu tiên xử lý các tác vụ quan trọng hơn thì hệ điều hành này có thể phục vụ các công việc khẩn cấp, hay là nếu sử dụng thuật toán xử lý task có deadline sớm nhất thì có thể dùng cho các công việc yêu cầu hard real-time khắt khe hơn.

Có một thuật toán scheduler đơn giản và dễ triển khai đó là Round Robin, là một trong những thuật toán lập lịch cổ điển và đơn giản nhất, được sử dụng rộng rãi trong các hệ thống không yêu cầu tính ưu tiên giữa các luồng. Trong Round Robin, mỗi task được cấp CPU trong một khoảng thời gian cố định gọi là time slice (hay quantum, quanta). Toàn bộ các task đang ở trạng thái Ready sẽ được sắp xếp trong một hàng đợi FIFO (First In First Out). Scheduler lần lượt lấy task ở đầu hàng đợi ra chạy. Nếu task chưa hoàn thành trong thời gian đó, nó sẽ bị dừng lại và đặt lại vào cuối danh sách ready, scheduler sẽ chọn task kế tiếp theo thứ tự vòng tròn. Ví dụ: giả sử có 3 task (A, B, C), mỗi task được cấp 10ms. Thứ tự xử lý sẽ là: A → B → C → A → B → C → ...

Trong Round Robin, mọi task đều có mức ưu tiên như nhau, cơ hội để sử dụng CPU là như nhau. Vì thế nên triển khai Round Robin trên một con vi điều khiển là khá hợp lý. Một tham số quan trọng của Round Robin là time slice. Tham số này là cố định trong cả quá trình hệ thống đang thực thi và sẽ được quản lý bởi một ngắt timer nào đó. Việc lựa chọn độ dài

time slice ảnh hưởng mạnh tới hành vi của hệ thống. Nếu time slice quá lớn, task chiếm CPU lâu làm ảnh hưởng độ trễ phản hồi. Ngược lại, nếu time slice quá nhỏ thì sẽ tăng số lần context switching, làm hao phí CPU, ảnh hưởng hiệu năng. Vì vậy, phải cân đối giữa tính phản hồi và hiệu quả sử dụng CPU.

2.1.5 Context switching

Trong một RTOS, khi nhiều task đồng thời tồn tại trong hệ thống, chỉ một task được thực thi tại một thời điểm trên một đơn vị xử lý, ở đây là số lượng lõi CPU có trong vi điều khiển. Do đó, để đạt được sự đa nhiệm, RTOS phải liên tục chuyển đổi giữa các task theo một chiến lược lập lịch nhất định. Quá trình chuyển quyền điều khiển CPU từ task đang chạy sang task khác được gọi là context switching. Đây là một trong những hoạt động trọng yếu và phức tạp nhất của nhân RTOS, có ảnh hưởng trực tiếp đến hiệu suất và tính đúng hạn của hệ thống.

Về mặt bản chất, mỗi task trong RTOS được xem là một thực thể độc lập, có trạng thái thực thi riêng và được gắn với một TCB để điều khiển nó. Trong TCB, hệ điều hành lưu trữ các thông tin cần thiết để quản lý và khôi phục trạng thái thực thi của task, bao gồm con trỏ stack, giá trị các thanh ghi CPU, giá trị bộ đếm chương trình (thanh ghi PC), và cờ trạng thái chương trình (xPSR trên lõi ARM Cortex-M). Khi một context switch xảy ra, hệ điều hành phải đảm bảo rằng toàn bộ trạng thái hiện tại của task đang bị tạm dừng sẽ được lưu lại đầy đủ vào TCB của nó, và sau đó, trạng thái của task kế tiếp được scheduler lựa chọn sẽ được nạp trở lại vào CPU để tiếp tục thực thi.

Cơ chế chuyển ngữ cảnh được khởi động bởi nhiều nguyên nhân như là hết time slice trong thuật toán Round Robin, có một task khác trở nên ready với mức ưu tiên cao hơn (trong RTOS hỗ trợ mức ưu tiên), hoặc khi một task chủ động nhường CPU do thực hiện các hoạt động như sleep, yeild, hoặc wait. Trên vi kiến trúc ARM Cortex-M, việc chuyển ngữ cảnh thường được thực hiện bên trong một trình xử lý ngắt đặc biệt tên là PendSV (Pendable Service Call). Khi cần thực hiện context switch, RTOS sẽ kích hoạt ngắt này. Bên trong ISR, nội dung của các thanh ghi hiện tại sẽ được đẩy lên stack, stack pointer của task hiện tại được cập nhật vào TCB, sau đó RTOS sẽ trích xuất stack pointer của task kế tiếp từ TCB tương ứng và phục hồi lại toàn bộ trạng thái CPU từ vùng stack này.

Một điểm đáng lưu ý là khoảng thời gian của context switch không chỉ dừng ở số chu kỳ CPU cần thiết để lưu và phục hồi ngữ cảnh, mà còn bao gồm cả các tác động không trực

tiếp như mất cache locality, giảm hiệu năng khi luồng mới vào CPU phải làm nóng lại pipeline, và độ trễ tiềm ẩn nếu context switch xảy ra quá thường xuyên. Trong một RTOS được thiết kế tốt, quá trình context switch cần được tối ưu để thực hiện trong thời gian ngắn nhất có thể (thường dưới vài μs), nhằm đảm bảo các ràng buộc thời gian của hệ thống không bị phá vỡ.

Tóm lại, context switching là cầu nối giữa sự đa nhiệm và thực tế của đơn luồng trên CPU. Một RTOS có hiệu năng tốt không chỉ cần bộ lập lịch hiệu quả, mà còn phải có cơ chế context switch nhẹ, nhanh và ổn định. Khả năng thực hiện chuyển ngữ cảnh chính xác, không gây nhiễu thời gian hệ thống là yếu tố quan trọng đảm bảo tính xác định và độ tin cậy của các hệ thống nhúng thời gian thực.

2.1.6 Đồng bộ hoá trong RTOS

Trong RTOS, khi nhiều task cùng tồn tại và thực thi song song trong cùng một không gian địa chỉ, việc truy cập và thao tác lên tài nguyên dùng chung như biến toàn cục, vùng nhớ, thiết bị ngoại vi, hoặc vùng buffer trở nên đặc biệt nhạy cảm. Nếu không có biện pháp kiểm soát, hệ thống có thể rơi vào các trạng thái không xác định như sai lệch dữ liệu, lỗi ghi đè bộ nhớ, hoặc thậm chí treo toàn bộ hệ thống. Chính vì vậy, các cơ chế đồng bộ hóa được thiết lập như một thành phần cốt lõi trong mọi RTOS để đảm bảo tính toàn vẹn dữ liệu và tính đúng đắn trong thực thi giữa các task.

Hai công cụ đồng bộ hóa phổ biến nhất trong RTOS là mutex và semaphore, mỗi công cụ được thiết kế để phục vụ những mục tiêu sử dụng khác nhau. Trong đó, mutex là phương tiện bảo vệ tài nguyên dùng chung khỏi bị truy cập đồng thời, trong khi semaphore là công cụ đồng bộ hóa tín hiệu giữa các luồng hoặc giữa ngắt và luồng.

Mutex (mutual exclusion) là một cơ chế loại trừ lẫn nhau giúp đảm bảo rằng tại một thời điểm chỉ có một task được phép truy cập vào một tài nguyên cụ thể. Khi một task muốn sử dụng một tài nguyên được bảo vệ bởi mutex, nó phải thực hiện thao tác khóa mutex. Nếu mutex đang bị khóa bởi task khác, task hiện tại sẽ bị chặn lại cho đến khi mutex được giải phóng. Đây là cơ chế đồng bộ hóa mang tính sở hữu, nghĩa là chỉ có task đã khóa mutex mới có quyền mở khóa nó. Một số RTOS còn có thể cung cấp thêm tính năng priority inheritance để xử lý tình huống task ưu tiên thấp giữ mutex mà một task ưu tiên cao đang cần tài nguyên đó. Cơ chế này tạm thời nâng mức ưu tiên của task đang giữ mutex nhằm tránh tình trạng priority inversion, một vấn đề nghiêm trọng có thể phá vỡ tính xác định của hệ thống.

Semaphore, mặt khác, là một biến đếm được bảo vệ đặc biệt, thường được dùng để báo hiệu trạng thái sẵn sàng hoặc điều kiện đồng bộ hóa giữa các thực thể thực thi. Semaphore không mang tính sở hữu, và thường được phân thành hai loại: binary semaphore (chỉ có hai trạng thái 0 và 1) và counting semaphore (cho phép giá trị đếm lớn hơn 1). Binary semaphore thường được dùng để đồng bộ hóa giữa một ISR và một task, chẳng hạn như khi ISR hoàn thành một chu kỳ lấy mẫu ADC và cần báo cho một task xử lý dữ liệu. Counting semaphore phù hợp hơn cho các bài toán quản lý tài nguyên số lượng giới hạn, ví dụ như số lượng kết nối UART đồng thời hoặc số lượng buffer còn trống trong một hàng đợi dữ liệu.

Một vấn đề kinh điển khi sử dụng các công cụ đồng bộ là deadlock, xảy ra khi hai hoặc nhiều luồng chờ tài nguyên do nhau giữ, tạo thành một chu trình không có lối thoát. Deadlock thường phát sinh khi các luồng khóa nhiều mutex theo thứ tự khác nhau, hoặc khi tài nguyên không được giải phóng đúng cách. Để tránh deadlock, một số quy tắc thiết kế được khuyến nghị như: luôn khóa tài nguyên theo cùng một thứ tự trên tất cả các luồng, tránh giữ mutex trong thời gian dài hoặc khi đang chờ tài nguyên khác, và sử dụng timeout trong các thao tác chờ semaphore/mutex để có thể thoát ra trong trường hợp bất thường.

Tóm lại, đồng bộ hóa là một phần không thể thiếu trong mọi hệ điều hành thời gian thực, đặc biệt khi hệ thống có nhiều luồng hoặc khi luồng và ngắt cùng truy cập các tài nguyên dùng chung. Việc sử dụng đúng cách mutex và semaphore không chỉ đảm bảo tính đúng đắn của dữ liệu, mà còn góp phần duy trì tính ổn định, hiệu năng và khả năng mở rộng của toàn bộ hệ thống RTOS.

2.1.7 Giao tiếp giữa các luồng

Trong RTOS, các task thường được phân chia theo chức năng, ví dụ như một task đọc cảm biến, một task xử lý dữ liệu, một task giao tiếp UART, v.v. Do mỗi task chỉ đảm nhiệm một phần trong chuỗi hoạt động của hệ thống, việc trao đổi dữ liệu giữa các task là điều tất yếu để đảm bảo luồng thông tin được liên tục và hệ thống hoạt động đúng logic. Phương pháp đơn giản nhất để giao tiếp giữa các task là sử dụng các biến toàn cục để lưu dữ liệu của chúng. Tuy nhiên, cách này lại không hiệu quả vì trong các hệ thống, dữ liệu tới liên tục và nhiều, đồng thời không phải task nhận sẽ xử lý nhanh bằng task gửi để không bị mất dữ liệu. Ngoài ra còn các vấn đề về việc nhiều task truy cập đến các biến toàn cục đó và có thể gây ra xung đột.

Khác với các hệ điều hành lớn như Linux hay Windows, nơi có thể sử dụng các kỹ thuật giao tiếp task phức tạp như socket, shared memory hay named pipes, thì trong RTOS, các phương pháp giao tiếp cần đảm bảo ba tiêu chí nhẹ, xác định và phù hợp với tài nguyên hạn chế. Do đó, các cơ chế giao tiếp liên luồng trong RTOS thường được cung cấp sẵn dưới dạng dịch vụ hệ thống, tiêu biểu là message queues, mailboxes và pipes.

Message Queue là một trong những cơ chế phổ biến nhất và linh hoạt nhất để truyền thông tin giữa các task. Về bản chất, message queue là một hàng đợi có kích thước xác định, nơi các message được gửi từ task gửi và được đọc bởi task nhận theo cơ chế FIFO (First-In, First-Out). Khi một task gửi message, nếu queue còn chỗ trống, message sẽ được thêm vào cuối hàng đợi; nếu queue đầy, tùy theo cấu hình, luồng có thể bị chặn, hủy bỏ thao tác, hoặc ghi đè dữ liệu cũ. Ngược lại, task nhận sẽ bị block nếu queue trống và chỉ được đánh thức khi có dữ liệu mới được đẩy vào.

Một trong những ưu điểm quan trọng của message queue là sự độc lập giữa task gửi và task nhận. Task gửi không cần đợi task nhận xử lý xong mới tiếp tục thực thi, điều này giúp hệ thống hoạt động hiệu quả hơn và giảm thiểu độ trễ trong các hệ thống có task chạy không đồng tốc. Tuy nhiên, chính điều này cũng đặt ra yêu cầu phải định kích thước hàng đợi một cách hợp lý, tránh hiện tượng queue tràn khi task gửi hoạt động nhanh hơn task nhận, dẫn đến mất dữ liệu hoặc chặn hệ thống.

Một phương án nhẹ hơn message queue là mailbox, thường chỉ chứa một message tại một thời điểm. Đây là công cụ phù hợp với các hệ thống đơn giản, nơi task nhận chỉ cần biết một sự kiện hoặc dữ liệu đơn lẻ từ task gửi. Mailbox thường có chi phí xử lý thấp hơn, và do chỉ chứa một phần tử nên dễ quản lý, nhưng lại không phù hợp với các hệ thống cần truyền tải luồng dữ liệu liên tục.

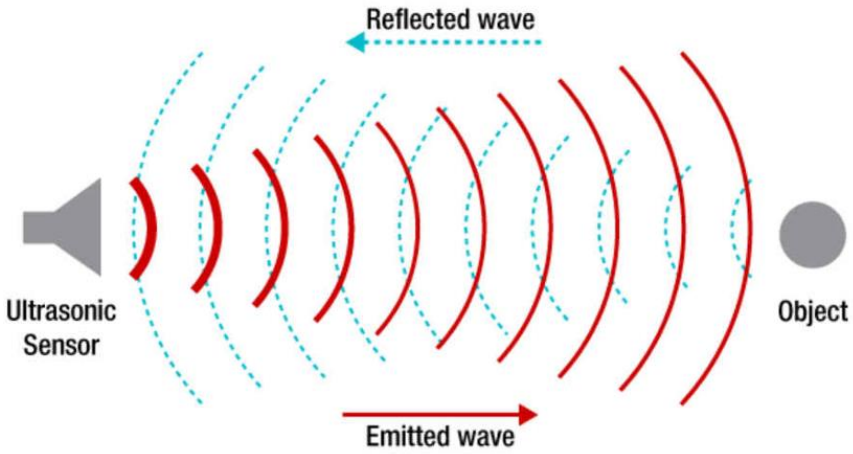
Một cơ chế khác ít phổ biến hơn nhưng vẫn tồn tại trong một số RTOS là pipes, một dạng giao tiếp buffer tuần tự theo byte hoặc word. Pipes cho phép một task ghi dữ liệu vào đầu buffer, trong khi task khác đọc từ cuối buffer, tương tự như một đường ống truyền dữ liệu. So với message queue vốn truyền từng khối dữ liệu rời rạc, pipes phù hợp với các ứng dụng truyền chuỗi liên tục, ví dụ như giao tiếp UART, âm thanh hoặc truyền file. Tuy nhiên, quản lý pipes phức tạp hơn và yêu cầu cơ chế đọc/ghi chính xác để tránh vượt giới hạn buffer.

Tổng kết lại, lựa chọn cơ chế giao tiếp liên luồng phù hợp trong RTOS là một quyết định quan trọng ảnh hưởng đến hiệu năng, độ trễ và tính ổn định của hệ thống. Trong khi message queue mang lại sự linh hoạt và tách biệt thời gian rõ ràng, thì mailbox phù hợp cho những giao tiếp đơn giản, còn pipes thích hợp cho các kịch bản truyền dữ liệu tuần tự. Việc lựa chọn đúng công cụ giao tiếp phải dựa trên đặc điểm truyền dữ liệu, tần suất, kích thước, và yêu cầu độ trễ giữa các tác vụ trong hệ thống thời gian thực.

2.2 Cảm biến siêu âm

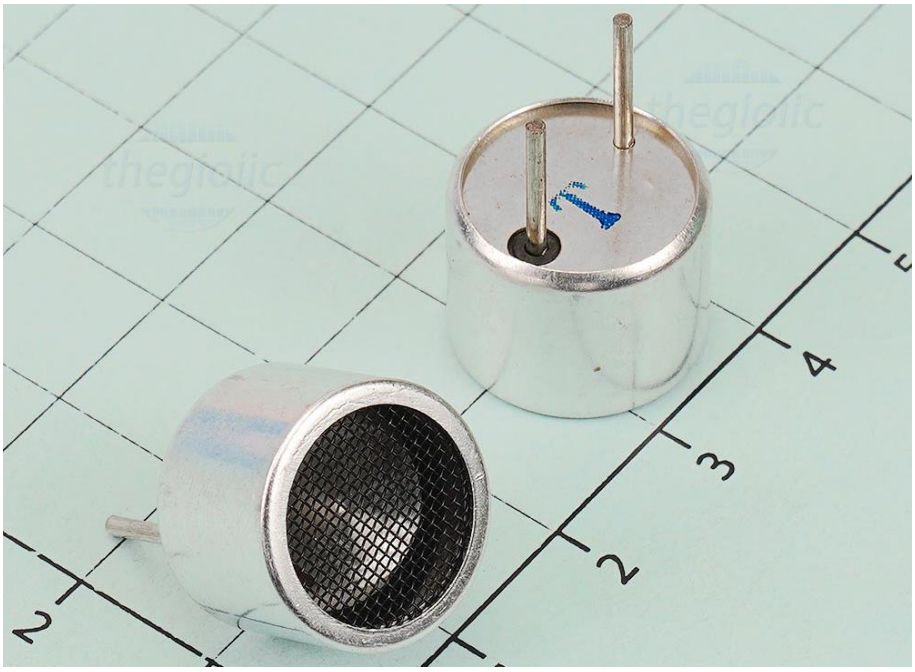
2.2.1 Giới thiệu và nguyên lý hoạt động

Cảm biến siêu âm có thể đo khoảng cách và phát hiện sự hiện diện của một vật thể mà không cần tiếp xúc với vật thể đó. Chúng thực hiện điều này bằng cách phát ra và theo dõi sự phản xạ của sóng siêu âm. Tùy thuộc vào loại cảm biến và đặc tính của vật thể, phạm vi hoạt động hiệu quả trong không khí có thể từ vài cm đến vài m. Cảm biến siêu âm (hay đầu dò siêu âm) tạo và phát ra các xung siêu âm, các xung này sẽ được phản xạ trở lại về phía cảm biến bởi một vật thể nằm trong vùng quan sát của cảm biến[3].



Hình 2.1: Cách cảm biến siêu âm phát hiện vật thể

Cảm biến siêu âm là một bộ chuyển đổi áp điện, có khả năng chuyển đổi tín hiệu điện thành dao động cơ học, tạo thành những sóng xung và ngược lại. Vì vậy, trong cấu hình đơn tĩnh, cảm biến siêu âm hoạt động như một bộ thu phát – vừa là loa vừa là micro tại một tần số duy nhất.



Hình 2.2: Đầu phát sóng siêu âm trong cảm biến siêu âm

Cảm biến có thể đo được khoảng thời gian giữa xung phát ra và xung phản xạ về. Vì vận tốc của âm thanh là một đại lượng đã biết, nên thời gian truyền – nhận thu được có thể được sử dụng để tính khoảng cách giữa cảm biến và vật thể. Phương trình (1) minh họa cách tính khoảng cách bằng sóng siêu âm.

$$d = \frac{\Delta t . v_{sound}}{2} \quad (1)$$

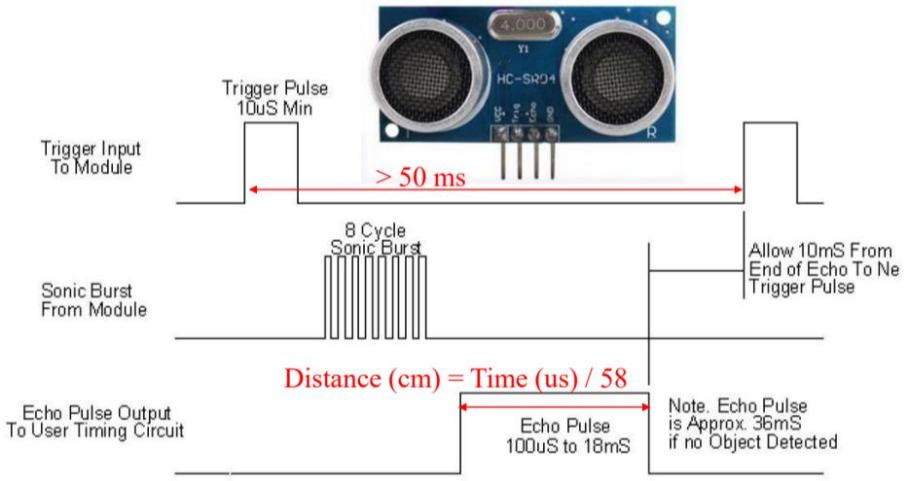
Với: d là khoảng cách giữa cảm biến và vật thể

Δt là khoảng thời gian từ lúc phát sóng siêu âm cho tới khi nhận được sóng phản xạ

v_{sound} là vận tốc âm thanh trong không khí. Giá trị này thay đổi theo nhiệt độ. Trong không khí khô ở 20°C, vận tốc âm thanh là 343 m/s.

2.2.2 Cách làm việc với cảm biến siêu âm sử dụng vi điều khiển

Trong thực tế, các module cảm biến siêu âm phổ biến như HC-SR04, US-100, hay các module công nghiệp sử dụng hai chân tín hiệu chính để thực hiện chức năng này, đó là Trigger và Echo. Đây là hai giao diện điện tử quan trọng cho phép cảm biến tương tác với vi điều khiển để thực hiện quá trình đo khoảng cách.



Hình 2.3: Cách làm việc với module HC-SR04 thông qua tín hiệu 2 chân Trigger và Echo

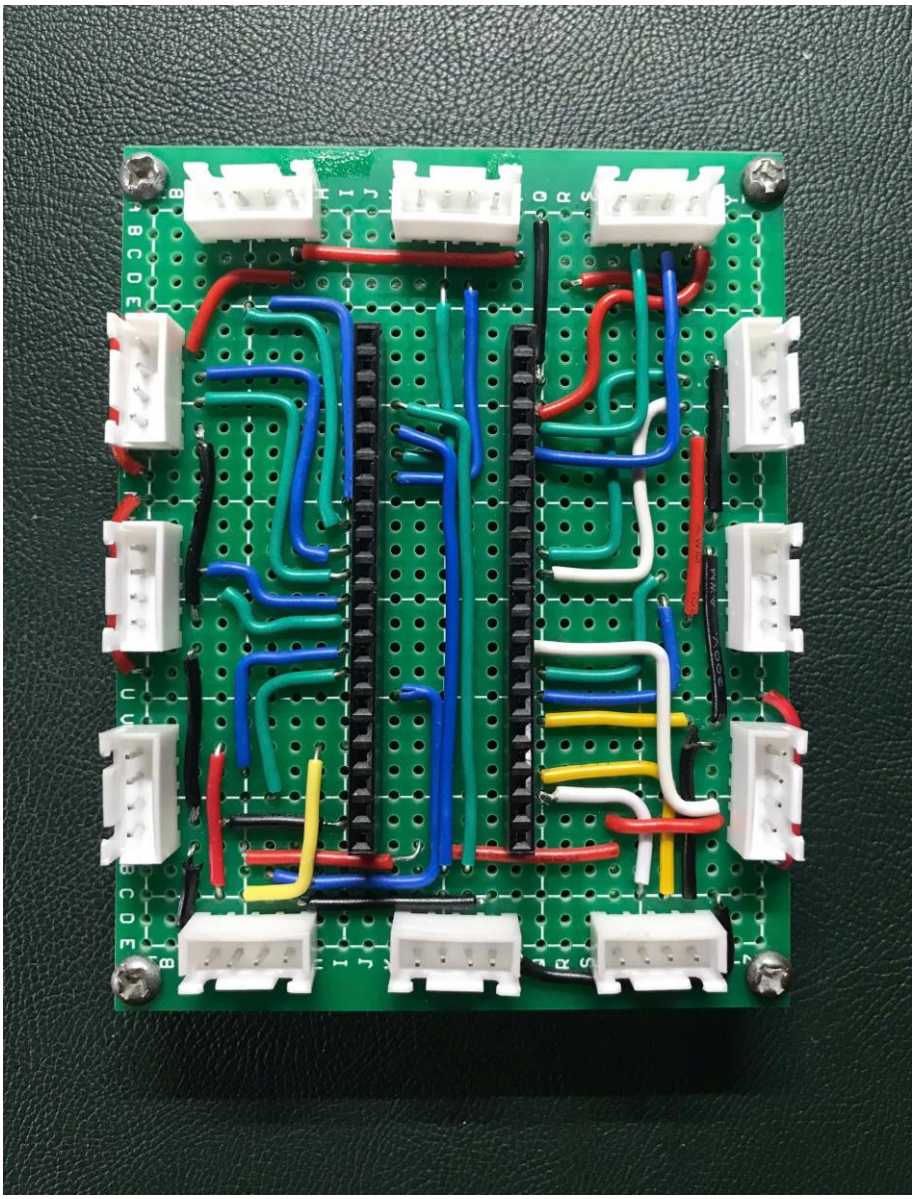
Chân Trigger đóng vai trò kích hoạt cảm biến phát sóng siêu âm. Từ phía vi điều khiển, để yêu cầu cảm biến bắt đầu một phép đo, người lập trình cần tạo ra một xung điện mức cao trong thời gian rất ngắn, thường là 10µs. Khi cảm biến nhận được xung này, nó sẽ phát ra một chùm xung siêu âm với tần số cố định (thường là 40 kHz) vào môi trường. Xung Trigger chỉ là tín hiệu điều khiển khởi động, hoàn toàn không chứa dữ liệu đo. Nó là yêu cầu đơn giản: “Hãy phát sóng và bắt đầu quá trình đo lường”. Trên vi điều khiển, chân Trigger được cấu hình là output. Lập trình viên chỉ cần đặt mức logic HIGH trong 10µs, sau đó kéo về mức LOW và chờ đợi tín hiệu phản hồi từ chân Echo.

Sau khi phát sóng, cảm biến chuyển sang trạng thái “nghe”, nghĩa là nó sẽ chờ sóng siêu âm phản xạ trở lại từ vật thể. Khi sóng phản xạ được thu về, cảm biến sẽ tạo ra một tín hiệu dạng xung tại chân Echo, với độ rộng của xung tỉ lệ thuận với khoảng cách đo được. Tín hiệu Echo được kéo lên mức HIGH ngay sau khi xung siêu âm được phát đi. Echo được giữ ở mức HIGH cho đến khi sóng phản xạ quay trở về cảm biến. Thời gian giữ mức cao chính là thời gian truyền – nhận của sóng siêu âm. Trên vi điều khiển, chân Echo phải được cấu hình là input. Nhiệm vụ của vi điều khiển là đo chính xác độ rộng của xung Echo, từ đó tính toán được khoảng cách bằng công thức (1).

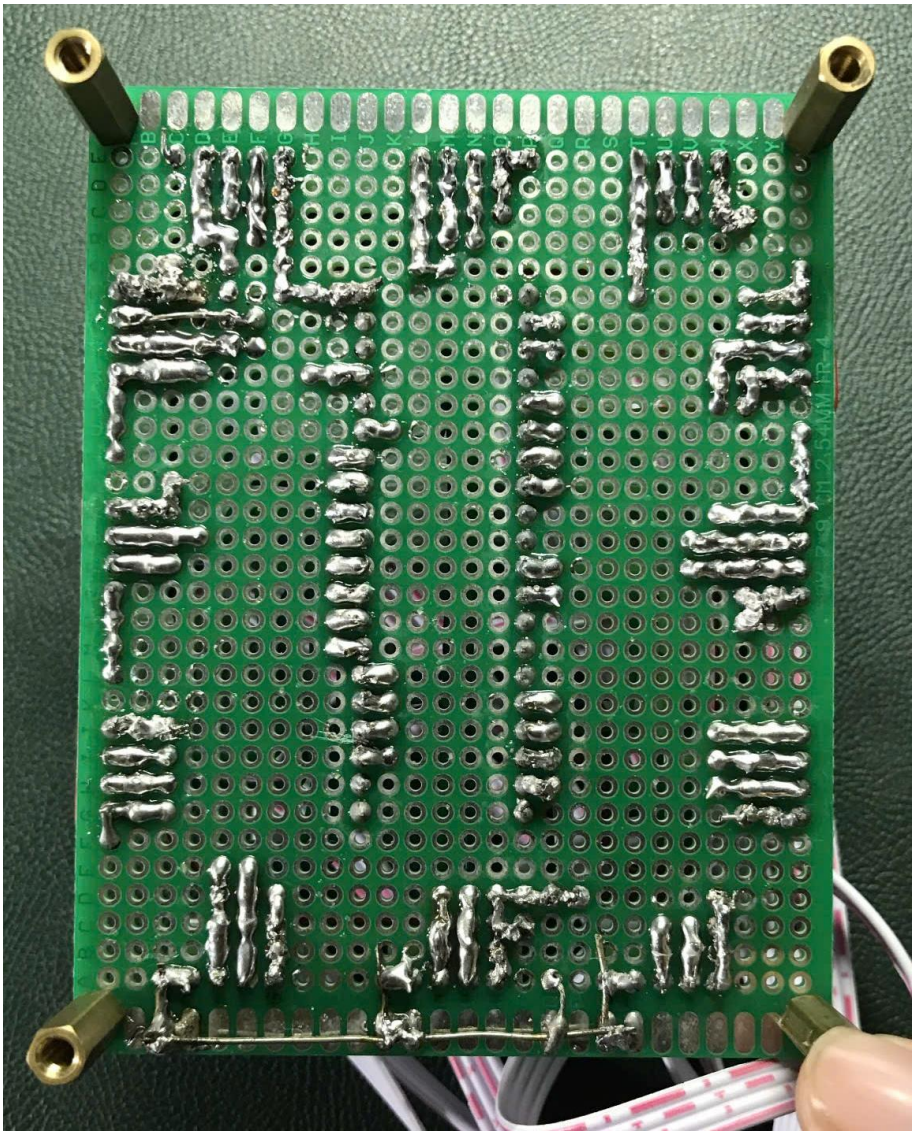
3. THỰC HIỆN ĐỀ TÀI

3.1 Thiết kế phần cứng

Phần cứng duy nhất của đề tài là mạch ra chân GPIO của kit vi điều khiển. Mỗi cảm biến sẽ có 2 chân tín hiệu và 2 chân nguồn và GND. Các chân nguồn của các cảm biến sẽ được nối chung và nối vào 5V, tương tự với các chân GND. Các chân tín hiệu sẽ được kết nối tới các chân GPIO có thể sử dụng được và thuận tiện trong việc di dây. Tổ hợp 4 chân lại vào 1 pin header, các cảm biến sẽ được nối vào các bus cái – cái 4 tiếp điểm và kết nối tới các pin header trên mạch, giúp cho kết nối được chắc chắn hơn.



Hình 3.1: Mặt trên mạch điện của đề tài



Hình 3.2: Mặt dưới mạch điện của đề tài

3.2 Lập trình firmware

3.2.1 Thiết kế và phát triển RTOS

RTOS do người viết tự thiết kế có cấu trúc gọn nhẹ, được xây dựng nhằm quản lý và điều phối nhiều tác vụ chạy đồng thời trên vi điều khiển STM32. RTOS này bao gồm các thành phần cốt lõi như quản lý thread, lập lịch vòng, sleep, semaphore, tác vụ định kỳ và hàng đợi thông điệp. Mỗi thành phần đóng vai trò cụ thể trong việc đảm bảo tính phản ứng nhanh, ổn định và có khả năng mở rộng. Bảng dưới đây là mô tả chi tiết về các thành phần chính của hệ thống:

Thành phần	Mô tả chức năng
Task Control Block (TCB)	Đây là cấu trúc dữ liệu trung tâm dùng để quản lý các thread trong hệ thống. Mỗi thread được gắn với một TCB riêng, trong đó lưu trữ con trỏ stack hiện tại, thời gian sleep còn lại (nếu có), cờ active (trạng thái sống/chết của thread), và con trỏ liên kết đến TCB tiếp theo (tạo thành danh sách liên kết vòng).
Stack Initialization	Khi thêm thread mới vào hệ thống, hệ thống sẽ tự động khởi tạo stack cho thread đó bằng cách nạp sẵn các thanh ghi cần thiết (R0 – R12, LR, PC, xPSR) theo chuẩn ARM Cortex-M3. Việc này cho phép khi xảy ra context switching, CPU có thể quay lại đúng điểm bắt đầu thực thi của thread.
Context Switching	Việc context switching giữa các thread được thực hiện bằng ngắt PendSV được viết bằng hợp ngữ. PendSV_Handler thực hiện việc lưu trạng thái của thread hiện tại, sau đó tải lại trạng thái của thread tiếp theo, cho phép hệ thống chuyển đổi mượt mà giữa các task.
System Tick	RTOS sử dụng bộ đếm SysTick tích hợp trong ARM Cortex-M để tạo ra mốc thời gian định kỳ. Tín hiệu ngắt từ SysTick là cơ sở để kích hoạt scheduler, giảm thời gian sleep, cập nhật timeout và thực thi các tác vụ định kỳ. Tần số ngắt có thể cấu hình tùy theo yêu cầu ứng dụng.
Thread Scheduler	Lập lịch đơn giản theo thuật toán Round Robin. Scheduler thực hiện lựa chọn luân phiên các thread để thực thi. Nếu một thread đang trong trạng thái sleep, scheduler sẽ bỏ qua và chọn thread tiếp theo. Phần lập lịch được thực thi thông qua ngắt SysTick và PendSV.
Semaphore	RTOS hỗ trợ semaphore dạng cooperative để đồng bộ giữa các thread. Semaphore hoạt động như một biến đếm. Khi một thread gọi Wait, nếu giá trị semaphore > 0, nó được tiếp tục; nếu không, thread đó bị chặn tạm thời. Khi một thread khác gọi Give, giá trị được tăng lên để giải phóng thread đang chờ.

Periodic Task	RTOS hỗ trợ thêm các tác vụ định kỳ bằng cách sử dụng timer TIM2 của STM32. Một hoặc nhiều hàm định kỳ có thể được đăng ký thực thi với chu kỳ cố định. Tính năng này rất hữu ích cho các tác vụ như đọc cảm biến định kỳ, cập nhật giao diện, hoặc thực hiện điều khiển thời gian thực.
Message Queue	Hệ thống có hỗ trợ message queue, một hàng đợi dữ liệu nhằm cho phép các thread giao tiếp với nhau. Queue sử dụng buffer vòng cùng với semaphore và mutex để đảm bảo dữ liệu không bị ghi đè hoặc đọc sai khi có nhiều thread truy cập đồng thời.

Bảng 3-1: Các thành phần được thiết kế và phát triển trong RTOS của đề tài

3.2.1.1 Task Control Block

Trong một RTOS, **Task Control Block (TCB)** đóng vai trò là cấu trúc dữ liệu cốt lõi, dùng để quản lý thông tin của từng thread đang hoạt động trong hệ thống. Mỗi thread sẽ tương ứng với một TCB riêng biệt, lưu trữ các thông tin như con trỏ stack, thời gian sleep, trạng thái hoạt động, và liên kết tới thread kế tiếp. Việc sử dụng danh sách liên kết vòng giúp cho quá trình scheduling theo cơ chế Round Robin được thực hiện đơn giản và hiệu quả.

Cấu trúc TCB được định nghĩa như sau:

```
struct tcb{
 uint32_t *stackPtr;
 struct tcb* nextPtr;
 uint32_t sleepTime;
 uint8_t isActive;
};
```

Cấu trúc này bao gồm:

- **`stackPtr`**: là địa chỉ hiện tại của stack mà thread đang sử dụng. Mỗi lần context switching xảy ra, giá trị này sẽ được lưu lại và khôi phục để thread có thể tiếp tục thực thi tại đúng thời điểm trước đó.
- **`nextPtr`**: là địa chỉ của thread tiếp theo trong danh sách, giúp kết nối các thread lại với nhau trong một danh sách liên kết vòng. Điều này rất quan trọng để scheduler có thể lặp qua từng thread một cách tuần tự.
- **`sleepTime`**: nếu thread gọi hàm **`osThreadSleep`**, hệ thống sẽ ghi nhận thời gian cần sleep vào trường này. Mỗi lần tick timer xảy ra, **`sleepTime`** sẽ giảm đi 1 đơn vị. Khi **`sleepTime`** về 0, thread sẽ quay trở lại trạng thái hoạt động bình thường.

- **`isActive`**: dùng để kiểm soát trạng thái hoạt động của thread. Nếu một thread bị vô hiệu hóa, nó sẽ bị bỏ qua trong quá trình lập lịch.

Để tạo một thread nào đó, người dùng sẽ sử dụng hàm **`osKernelAddThread`**. Hàm này thực hiện các thao tác sau:

- Khởi tạo và gán TCB cho thread.
- Gán liên kết cho thread với thread được tạo trước đó và với thread đầu tiên, tạo thành danh sách liên kết vòng khép kín.
- Gọi **`osKernelStackInit`** để chuẩn bị vùng stack cho mỗi thread.
- Gán các giá trị mặc định như **`sleepTime`** = 0, **`isActive`** = 1, **`stackPtr`** trỏ đến đúng vị trí stack đã chuẩn bị.
- Gán địa chỉ của thread cần tạo cho thanh ghi PC trong stack của thread.

3.2.1.2 Stack Initialization

Một đặc điểm quan trọng trong RTOS là mỗi thread cần có vùng stack riêng. Stack này không chỉ lưu dữ liệu tạm mà còn mô phỏng đầy đủ trạng thái CPU tại thời điểm thread bắt đầu thực thi. Khi khởi tạo một thread mới, thread này chưa từng được CPU thực thi, vì vậy hệ thống phải giả lập sẵn một context frame ngay trên stack của thread đó, giống như nó vừa bị ngắt giữa chừng. Nhờ vậy, khi thực hiện context switch, CPU sẽ "tưởng như" thread này chỉ đang tiếp tục công việc của mình.

Hệ điều hành sử dụng hàm **`osKernelStackInit`** để nạp sẵn các giá trị ngữ cảnh vào stack của thread. Việc nạp này dựa trên cấu trúc context frame chuẩn của ARM Cortex-M, được tạo ra khi xảy ra một ngắt hoặc context switch. Cấu trúc context frame khởi tạo bao gồm như sau (theo thứ tự phần cứng sẽ đẩy lên stack trước rồi đến sau):

- **xPSR**: trạng thái CPU. Bit T trong thanh ghi xPSR (bit số 24) xác định CPU đang chạy ở chế độ Thumb hay ARM. Trong dòng STM32F1/Cortex-M3, chỉ hỗ trợ chế độ Thumb, nên bit này bắt buộc phải = 1. Nếu quên không cho bit T lên 1, CPU sẽ HardFault ngay lập tức khi thực thi thread.
- **PC**: là thanh chứa địa chỉ mà chương trình đang thực thi, nó sẽ là địa chỉ bắt đầu thực thi của thread.

- **LR**: là thanh ghi dùng để lưu địa chỉ quay về khi thực hiện lệnh gọi hàm như BL, BLX hoặc các lệnh trả về như `return`. Vì thread không được thiết kế để return, nên ta không muốn CPU thực sự dùng giá trị trong LR. Vì lý do đó nên ta sẽ dùng giá trị giả gán vào LR để đảm bảo khởi tạo đầy đủ context frame.
- **R12, R3, R2**: là thanh ghi tạm thời, chứa các giá trị trung gian để thực hiện mục đích nào đó và sẽ bị ghi đè khi các hàm chạy. Vì thế nên ta cũng sẽ gán giá trị giả cho các thanh ghi này.
- **R1, R0**: là thanh ghi dùng để chứa tham số truyền vào và trả về của các hàm. Các thread thường sẽ không sử dụng tham số truyền vào nên 2 thanh ghi này ta cũng gán giá trị giả vào khi khởi tạo stack.

Sau đó, ta thêm các giá trị giả vào các thanh ghi R4 – R11 để hoàn thành đầy đủ stack cho 1 thread. Khi context switching, giá trị của các thanh ghi sẽ được lưu lại để bảo toàn cho lần gọi thread sau. Các giá trị mới sẽ được thêm vào để tiến hành chạy thread mới.

```
void osKernelStackInit(int i){
 tcbs[i].stackPtr = &TCB_STACK[i][STACK_SIZE - 16];

 /*Set bit21 (T-Bit) in PSR to 1 to operate it*/
 /*Thumb Mode*/
 TCB_STACK[i][STACK_SIZE - 1] = 1 << 24; /*PSR*/

 /*Skip the PC*/

 /*Dumming Stack Content*/
 TCB_STACK[i][STACK_SIZE - 3] = 0xAAAAAAAA;   /*R14
(LR)*/
 TCB_STACK[i][STACK_SIZE - 4] = 0xAAAAAAAA;   /*R12*/
 TCB_STACK[i][STACK_SIZE - 5] = 0xAAAAAAAA;   /*R3*/
 TCB_STACK[i][STACK_SIZE - 6] = 0xAAAAAAAA;   /*R2*/
 TCB_STACK[i][STACK_SIZE - 7] = 0xAAAAAAAA;   /*R1*/
 TCB_STACK[i][STACK_SIZE - 8] = 0xAAAAAAAA;   /*R0*/

 TCB_STACK[i][STACK_SIZE - 9] = 0xAAAAAAAA;   /*R11*/
 TCB_STACK[i][STACK_SIZE - 10] = 0xAAAAAAAA;  /*R10*/
 TCB_STACK[i][STACK_SIZE - 11] = 0xAAAAAAAA;  /*R9*/
 TCB_STACK[i][STACK_SIZE - 12] = 0xAAAAAAAA;  /*R8*/
 TCB_STACK[i][STACK_SIZE - 13] = 0xAAAAAAAA;  /*R7*/
 TCB_STACK[i][STACK_SIZE - 14] = 0xAAAAAAAA;  /*R6*/
 TCB_STACK[i][STACK_SIZE - 15] = 0xAAAAAAAA;  /*R5*/
 TCB_STACK[i][STACK_SIZE - 16] = 0xAAAAAAAA;  /*R4*/
}
```

3.2.1.3 Context Switching

Context switching là quá trình mà hệ điều hành tạm dừng việc thực thi của một thread đang chạy, lưu lại toàn bộ trạng thái của nó, và sau đó khôi phục trạng thái của một thread khác để CPU tiếp tục thực thi thread mới đó. Quá trình này sẽ được điều khiển bởi ngắt PendSV và sẽ được xử lý trong ISR của nó là PendSV_Handler. ISR này được viết bằng hợp ngữ để tối đa hoá tốc độ context switching.

```
PendSV_Handler
 CPSID I
 PUSH {R4-R11}
 LDR R0, =currentPtr
 LDR R1, [R0]

 STR SP, [R1]

 PUSH {R0, LR}
 BL osSchedulerPeriodicRR
 POP {R0, LR}
 LDR R1, [R0]

 LDR SP, [R1]
 POP {R4-R11}
 CPSIE I
 BX LR

 END
```

Mỗi khi xảy ra ngắt, phần cứng ARM Cortex-M sẽ tự động lưu một phần context vào stack, theo đúng chuẩn của kiến trúc ARM. Đó chính là các thành phần trong một context frame. RTOS sẽ lợi dụng cơ chế này, áp dụng vào ngắt PendSV để lưu lại context frame vào stack, sau đó ISR này sẽ lưu lại các thanh ghi còn lại để hoàn toàn bảo tồn trạng thái của thread khi context switching. Thanh ghi SP (Stack pointer) lúc này đang trỏ vào vùng stack sau khi đã PUSH tất cả context, nên ISR sẽ ghi lại giá trị của SP này vào TCB, để thread này có thể quay lại đúng trạng thái cũ trong lần chạy tiếp theo.

Sau đó, ISR sẽ gọi scheduler để cập nhật giá trị của `currentPtr` thành địa chỉ của thread tiếp theo cần chạy. Tiếp theo, lấy giá trị của thành viên `stackPtr` của thread cần chạy gán vào thanh ghi SP để CPU chạy tới vùng stack của thread cần chạy. Cuối cùng, phục hồi lại giá trị của các thanh ghi R4 – R11. Các thanh ghi còn lại sẽ được tự động POP ra khi thoát khỏi ngắt.

3.2.1.4 System Tick

Trong RTOS của đề tài, SysTick được dùng để kích hoạt ngắt, yêu cầu context switch, đồng thời hỗ trợ sleep và xử lý các periodic task. Người dùng sẽ sử dụng hàm `osKernelLaunch` để khởi tạo SysTick timer và khởi chạy thread đầu tiên. Vì scheduler hoạt động theo thuật toán Round Robin nên SysTick cần kích hoạt ngắt sau một khoảng thời gian được người dùng chọn. Khoảng thời gian này phải là bội số của 1ms, theo tính toán từ tần số xung clock mà CPU dùng để hoạt động

```
void osKernelInit(void){
    MILLIS_PRESCALER = (BUS_FREQ / 1000); // 1ms
}
```

Khi SysTick timer tràn, CPU sẽ thực thi các lệnh trong `SysTick_Handler`. ISR này không xử lý các logic để tránh làm chồng ngắt. Thay vào đó, ISR sẽ chỉ bật cờ `PENDSVSET`, để kích hoạt ngắt PendSV, nơi xử lý context switching. Nói tóm lại, `SysTick_Handler` chỉ vào để kích hoạt việc context switching xảy ra bên trong `PendSV_Handler`.

```
void SysTick_Handler(void){
 INTCTRL |= (1 << 28);
}
```

Ngoài ra, SysTick timer khi phối hợp với một timer ngoại vi sẽ giúp điều khiển các periodic task. Xem chi tiết ở mục

3.2.1.5 Thread Scheduler

Trong một RTOS, scheduler là thành phần quan trọng nhất, chịu trách nhiệm lựa chọn task nào được thực thi tiếp theo, context switching và đảm bảo các task được luân phiên CPU một cách công bằng hoặc theo mức độ ưu tiên. Ở đề tài này, scheduler sẽ được lập trình theo thuật toán Round Robin. Thời gian mà scheduler cho phép thread được thực thi đã được khởi tạo cùng với SysTick timer và cũng chính SysTick timer sẽ báo hiệu cho CPU khi hết thời gian.

Dựa vào cấu trúc của TCB có chứa thành viên `nextPtr` trỏ tới TCB của thread kế tiếp được thực thi, scheduler lúc này chỉ cần cập nhật biến toàn cục `currentPtr`, là biến trỏ tới TCB của thread hiện tại đang thực thi bằng với giá trị thành viên `nextPtr` của TCB đó để có thể giúp cho PendSV thực hiện context switching.

```
void osSchedulerPeriodicRR(void){
 currentPtr = currentPtr->nextPtr;
}
```

Bên cạnh đó, scheduler còn có cơ chế sleep và vô hiệu hoá task. Sleep là cơ chế mà thread muốn đợi 1 khoảng thời gian để có thể tiếp tục thực thi các lệnh tiếp theo. Khi sleep thì thread sẽ cập nhật thành viên sleepTime trong TCB và sẽ nhường lại CPU cho các thread khác.

```
void osThreadsSleep(uint32_t sleeptime){
 __disable_irq();

 currentPtr->sleepTime = sleeptime;
 __enable_irq();
 osThreadYield();
}
```

Bản chất là scheduler sẽ chuyển tới thread kế tiếp theo vòng sau khi thread trước nhường lại CPU. Tuy nhiên, scheduler sẽ bỏ qua thread nào đang có thành viên `sleepTime > 0` hoặc `isActive = 0` trong TCB của thread đó. Chỉ khi nào cả 2 điều kiện trên không xảy ra thì thread đó mới được thực thi như thường.

```
void osSchedulerRRSleep(void){
 currentPtr = currentPtr->nextPtr;
 while(currentPtr->sleepTime > 0 || currentPtr-
>isActive == 0) {
    currentPtr = currentPtr->nextPtr;
 }
}
```

Ngoài ra, scheduler còn hỗ trợ cơ chế nhường CPU để thực thi thread tiếp theo, kể cả khi thread chưa sử dụng hết thời gian mà scheduler cho phép.

```
void osThreadYield(void){
 SysTick->VAL = 0;
 INTCTRL |= PENDSTSET;
}
```

3.2.1.6 Semaphore

Trong RTOS này, semaphore được triển khai theo mô hình counting semaphore kết hợp cơ chế polling và nhường CPU thủ công. Thiết kế này đơn giản nhưng hiệu quả cho các hệ thống nhỏ, nơi không sử dụng ưu tiên hoặc chờ thụ động.

Semaphore trong hệ thống được khai báo dưới dạng một biến kiểu `uint32_t`, được gọi là key và có giá trị biểu diễn trạng thái mở (giá trị lớn hơn 0) hoặc đóng (giá trị nhỏ hơn hoặc bằng 0) của tài nguyên.

```
void osSemaphore_Init(uint32_t *semaphore, uint32_t
val){
 *semaphore = val;
}
```

Khi 1 task sử dụng tài nguyên, nó giảm key đi 1 đơn vị, đây là hành động Take Semaphore.

```
void osSemaphore_Give(uint32_t *semaphore){
 __disable_irq();
 *semaphore += 1;
 __enable_irq();
}
```

Khi sử dụng xong, task đó trả lại tài nguyên, nó tăng key lên 1, đây là hành động Give Semaphore. Khi 1 task muốn sử dụng tài nguyên, nó phải đợi để được Take Semaphore, đây là hành động Wait Semaphore.

```
void osCooperative_Wait(uint32_t *semaphore){
 __disable_irq();

 while(*semaphore <= 0){
      __enable_irq();
      osThreadYield();
      __disable_irq();
 }
 *semaphore -= 1;

 __enable_irq();
}
```

3.2.1.7 Periodic Task

Periodic task trong RTOS này được thiết kế này tập trung vào việc cho phép người dùng đăng ký một hoặc nhiều hàm xử lý định kỳ, được thực thi song song với scheduler chính, nhưng không cạnh tranh quyền CPU như các thread thông thường. Các hàm định kỳ này thường được sử dụng để đọc cảm biến, cập nhật dữ liệu, hoặc thực hiện kiểm tra trạng thái hệ thống với chu kỳ ổn định, ví dụ mỗi 1ms hoặc 10ms.

Các thành phần của một periodic task được thiết kế như sau:

```
typedef void(*taskT)(void);
typedef struct{
```

```
 taskT task;
 uint32_t period;
 uint32_t timeLeft;
} periodicTaskT;
```

Cấu trúc này bao gồm:

- `taskT`: là con trỏ hàm void không có tham số truyền vào. Đây chính là dạng khai báo chuẩn của các task trong RTOS.
- `task`: thành viên chứa con trỏ hàm tới hàm đã được định nghĩa trong hệ thống.
- `period`: chu kỳ lặp lại của task.
- `timeLeft`: một bộ đếm thời gian còn lại trước khi task được chạy lại lần tiếp theo.

Danh sách các periodic task được lưu vào một mảng static:

```
static periodicTaskT
PeriodicTasks[NUM_OF_PERIODIC_TASKS];
```

Cốt lõi của periodic task trong hệ thống ở ba phần: khai báo, khởi tạo và thực thi định kỳ. Trong phần khai báo, hệ thống định nghĩa các con trỏ hàm ở phạm vi global. Mỗi con trỏ sẽ lưu địa chỉ của một hàm do người dùng định nghĩa và truyền vào. Việc đăng ký các hàm này được thực hiện thông qua hàm `osKernelAddPeriodicThreads`, với hai tham số truyền vào là con trỏ đến hai hàm void không đối số và chuu kỳ của task. Hàm này chỉ đơn giản là gán các giá trị của các tham số truyền vào hàm cho các thành viên trong struct và thêm task đó vào mảng `PeriodicTasks[NUM_OF_PERIODIC_TASKS]`.

```
uint8_t osKernelAddPeriodicThreads(void(*task)(void),
uint32_t period){
 if(numberofPeriodicThreads == NUM_OF_PERIODIC_TASKS
|| period == 0){
      return 0;
 }
 PeriodicTasks[numberofPeriodicThreads].task = task;
 PeriodicTasks[numberofPeriodicThreads].period =
period;
 PeriodicTasks[numberofPeriodicThreads].timeLeft =
period - 1;

 numberofPeriodicThreads++;
 return 1;
}
```

Điểm đặc biệt trong thiết kế periodic task của bạn không nằm ở mức độ trừu tượng cao mà ở cách tích hợp nó vào luồng thời gian của hệ thống. Cụ thể, bạn không tạo thread độc lập

cho các hàm periodic này, mà lựa chọn thực thi chúng ngay trong hàm `periodic_events_exe()`. Hàm này được gọi trong ISR của ngắt timer ngoại vi (hệ thống sử dụng TIM2), hệ thống sẽ tự động gọi `periodic_events_exe`. Trong `periodic_events_exe`, nếu periodic task có thành viên `timeLeft == 0` thì mới vào thực thi hàm đó, không thì giảm giá trị `timeLeft`. Ngoài ra, hàm này còn giúp giảm thời gian ngủ của 1 task bằng cách giảm giá trị của thành viên `sleepTime` trong TCB.

```
void periodic_events_exe(){
 for(int i = 0; i < numberofPeriodicThreads; i++){
      if(PeriodicTasks[i].timeLeft == 0){
           PeriodicTasks[i].task();
           PeriodicTasks[i].timeLeft =
PeriodicTasks[i].period - 1;
      }
      else PeriodicTasks[i].timeLeft --;
 }
 for(int i = 0; i < current_num_of_thread; i++){
      if(tcbs[i].sleepTime > 0){
           tcbs[i].sleepTime--;
      }
 }
}
```

Trước khi các periodic được thêm vào thì phải khởi tạo timer quản lý các periodic task trước rồi mới khởi tạo các periodic task sau. Sử dụng hàm `osPeriodicTask_Init` để khởi tạo TIM2, ngắt TIM2 để mỗi lần ngắt xảy ra, chương trình sẽ vào ISR và thực thi hàm `periodic_events_exe` để thực hiện các hành động điều khiển các periodic task.

```
void osPeriodicTask_Init(void(*task)(void), uint32_t
freq, uint8_t priority){
 __disable_irq();
 PeriodicTask = task;
 RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
 TIM2->PSC = 72 - 1;
 TIM2->ARR = 1000000/freq - 1;
 TIM2->CR1 = 1;
 TIM2->DIER |= 1;
 NVIC_SetPriority(TIM2_IRQn , priority);
 NVIC_EnableIRQ(TIM2_IRQn);
}

void TIM2_IRQHandler(){
 TIM2->SR = 0;
 if (PeriodicTask != NULL) {
      PeriodicTask();
 }
```

```
}
```

3.2.1.8 Message Queue

Message queue được thiết kế theo cấu trúc hàng đợi vòng có giới hạn kích thước cố định, kèm theo semaphore dạng **mutex** để bảo vệ quyền truy cập đồng thời tới queue giữa các thread. Queue được khai báo dưới dạng con trỏ tới struct **HCSR04_Queue**, với 3 thành viên trạng thái **head**, **tail** và **count** lần lượt đại diện cho vị trí đọc, vị trí ghi và số lượng phần tử hiện tại trong hàng đợi. Mảng **buffer[HCSR04_QUEUE_SIZE]** chứa các phần tử kiểu **HCSR04_DataFrame**, mỗi phần tử tương ứng với một khung dữ liệu thu từ cảm biến siêu âm chứa ID của cảm biến và khoảng cách sau khi tính toán.

```
typedef struct {
 HCSR04_TypeDef *HCSR04_Struct;
 uint16_t distance_cm;
} HCSR04_DataFrame;

typedef struct {
 HCSR04_DataFrame buffer[HCSR04_QUEUE_SIZE];
 uint32_t head;
 uint32_t tail;
 uint32_t count;
 uint32_t mutex;
} HCSR04_Queue;
```

Để thêm dữ liệu vào queue, sử dụng hàm **HCSR04_QueuePut**. Trong hàm này, thread gọi đầu tiên sẽ chờ lấy quyền mutex thông qua **osCooperative_Wait(&queue->mutex)**. Sau khi có quyền truy cập, hàm kiểm tra xem queue có còn chỗ trống không. Nếu còn, dữ liệu mới sẽ được ghi vào vị trí **tail**, sau đó **tail** được cập nhật bằng cách tăng lên một đơn vị và sử dụng phép chia dư để quay vòng lại đầu nếu vượt quá giới hạn mảng. Biến **count** cũng được tăng để phản ánh số phần tử hiện có trong queue. Cuối cùng, thread trả lại mutex bằng cách gọi **osSemaphore_Give**, cho phép thread khác có thể tiếp tục thao tác với queue.

```
void HCSR04_QueuePut(HCSR04_Queue *queue,
HCSR04_DataFrame data){
 osCooperative_Wait(&(queue->mutex));

 if (queue->count < HCSR04_QUEUE_SIZE) {
      queue->buffer[queue->tail] = data;
      queue->tail = (queue->tail + 1) %
HCSR04_QUEUE_SIZE;
      queue->count++;
```

```
 }

 osSemaphore_Give(&(queue->mutex));
}
```

Để lấy dữ liệu từ queue, sử dụng hàm `HCSR04_QueueGet`. Trong hàm này, RTOS sử dụng một vòng lặp vô hạn để liên tục kiểm tra xem có dữ liệu trong queue không. Thread đầu tiên cũng phải chờ lấy `mutex` giống như trong hàm ghi. Nếu `queue->count > 0`, dữ liệu tại vị trí `head` sẽ được đọc ra và gán vào biến `*data`, `head` được cập nhật bằng cách tăng tuần tự và quay vòng nếu cần, `count` giảm đi 1. Thread sau đó giải phóng mutex và trả về 1 để báo hiệu đọc thành công. Nếu queue đang trống, thread cũng phải giải phóng mutex nhưng thay vì chờ bận, nó sẽ gọi `osThreadsSleep` để tự động ngủ trong 1ms, sau đó lặp lại.

```
uint8_t HCSR04_QueueGet(HCSR04_Queue *queue,
HCSR04_DataFrame *data){
 while(1){
      osCooperative_Wait(&(queue->mutex));

      if (queue->count > 0) {
           *data = queue->buffer[queue->head];
           queue->head = (queue->head + 1) %
HCSR04_QUEUE_SIZE;
           queue->count--;
           osSemaphore_Give(&(queue->mutex));
           return 1;
      }

      osSemaphore_Give(&(queue->mutex));
      osThreadsSleep(1);
 }
}
```

3.2.2 Đo dữ liệu từ cảm biến HC-SR04

Hệ thống sử dụng 10 cảm biến để đo khoảng cách xung quanh và hệ thống phải phản ứng nhanh với sự thay đổi của dữ liệu. Vì thế, yêu cầu của thread đo dữ liệu của cảm biến là phải làm việc đồng thời với 10 cảm biến, đưa dữ liệu sau tính toán vào queue để truyền đi và phải có độ trễ thấp.

Mỗi cảm biến sẽ được định nghĩa với các thành phần như sau:

```
typedef struct {
 uint8_t id;
 uint16_t echoPin;
```

```
 uint16_t triggerPin;
 GPIO_TypeDef *gpioPort;
 uint8_t lastEchoState;
 uint32_t echoStart;
 uint32_t echoStop;
 HCSR04_State state;
 uint32_t timeout;
} HCSR04_TypeDef;
```

- **id**: dùng để phân biệt các cảm biến với nhau và dùng để đánh dấu vị trí của cảm biến trong hệ thống, với vị trí trước trái là số 1, đi theo chiều ngược chiều kim đồng hồ tới vị trí trước sẽ là số 10.
- **echoPin**: số thứ tự chân GPIO của vi điều khiển dùng để nối đến chân echo của cảm biến.
- **triggerPin**: số thứ tự chân GPIO của vi điều khiển dùng để nối đến chân trigger của cảm biến.
- **gpioPort**: địa chỉ tới struct chứa các thanh ghi của ngoại vi GPIO của 2 chân trigger và echo thuộc về. Nói nôm na thành viên này là ngoại vi GPIO chứa 2 chân của 2 thành viên **echoPin** và **triggerPin**.
- **lastEchoState**: trạng thái chân echo của lần ghi nhận trước đó, dùng để nhận biết sự thay đổi của chân echo.
- **echoStart**: thời điểm mà chân echo bắt đầu lên mức HIGH, dùng để tính toán độ rộng xung echo.
- **echoStop**: thời điểm mà chân echo bắt đầu xuống mức LOW, dùng để tính toán độ rộng xung echo.
- **state**: trạng thái của cảm biến, dùng cho state machine.
- **timeout**: thời gian tạm nghỉ của cảm biến, dùng để tạo delay trong thread mà không khiến thread phải nhường CPU hay trễ deadline.

Danh sách các cài đặt của cảm biến sẽ được lưu vào một mảng, ví dụ như:

```
HCSR04_TypeDef HCSR04List[NUM_OF_HCSR04] = {
 {0, GPIO_Pin_14, GPIO_Pin_13, GPIOB},
 {1, GPIO_Pin_8, GPIO_Pin_11, GPIOA},
 {2, GPIO_Pin_4, GPIO_Pin_3, GPIOB},
 {3, GPIO_Pin_9, GPIO_Pin_8, GPIOB},
 {4, GPIO_Pin_0, GPIO_Pin_1, GPIOA},
 {5, GPIO_Pin_3, GPIO_Pin_2, GPIOA},
 {6, GPIO_Pin_5, GPIO_Pin_4, GPIOA},
 {7, GPIO_Pin_7, GPIO_Pin_6, GPIOA},
```

```
 {8, GPIO_Pin_1, GPIO_Pin_0, GPIOB},
 {9, GPIO_Pin_14, GPIO_Pin_15, GPIOC}
};
```

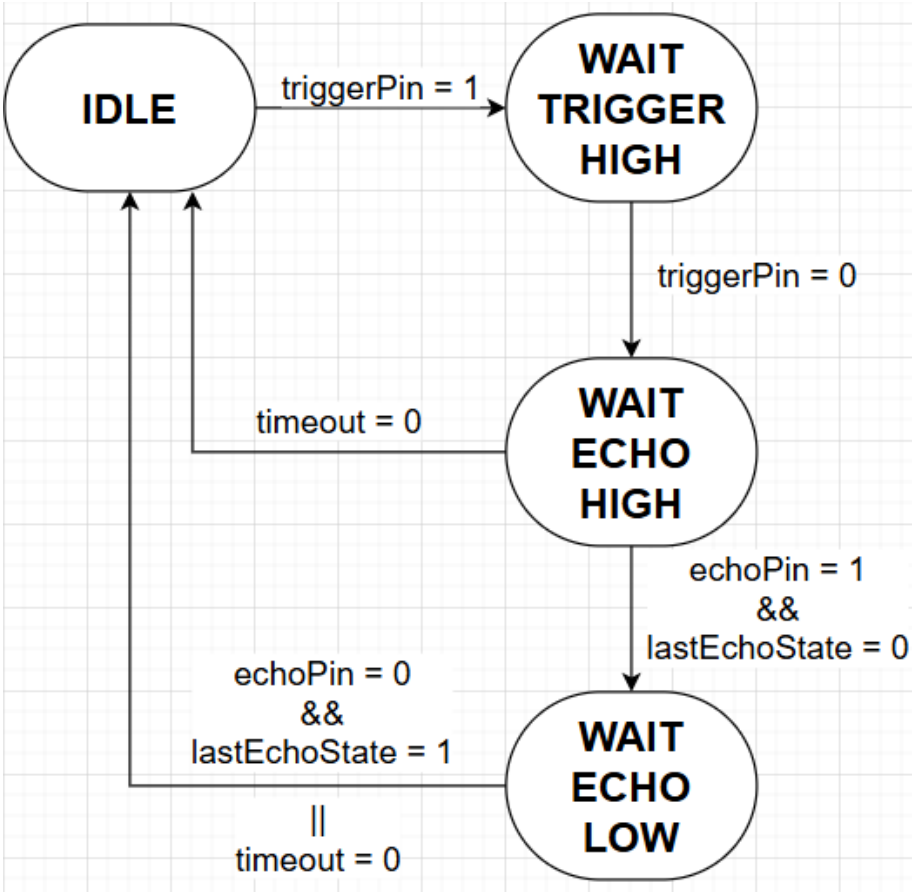
Các thành viên sẽ luôn có giá trị là 0 khi khởi tạo các chân GPIO cần thiết để làm việc với cảm biến

```
for (int i = 0; i < NUM_OF_HCSR04; i++) {
      // Trigger - Output
      GPIO_InitStruct.GPIO_Pin =
HCSR04List[i].triggerPin;
      GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
      GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(HCSR04List[i].gpioPort,
&GPIO_InitStruct);

      // Echo - Input
      GPIO_InitStruct.GPIO_Pin =
HCSR04List[i].echoPin;
      GPIO_InitStruct.GPIO_Mode =
GPIO_Mode_IN_FLOATING;
      GPIO_Init(HCSR04List[i].gpioPort,
&GPIO_InitStruct);

      HCSR04List[i].lastEchoState = 0;
      HCSR04List[i].echoStart = 0;
      HCSR04List[i].echoStop = 0;
      HCSR04List[i].state = HCSR04_IDLE;
      HCSR04List[i].timeout = 0;
 }
```

Đề tài thiết kế thread này theo mô hình FSM (Finite State Machine) bao gồm 4 trạng thái:

```
typedef enum {
 HCSR04_IDLE,
 HCSR04_WAIT_FOR_TRIGGER,
 HCSR04_WAIT_FOR_ECHO_HIGH,
 HCSR04_WAIT_FOR_ECHO_LOW,
} HCSR04_State;
```

- `HCSR04_IDLE`: là trạng thái cảm biến đã sẵn sàng cho lần đo tiếp theo
- `HCSR04_WAIT_FOR_TRIGGER`: là trạng thái mà vi điều khiển đã kích chân trigger lên mức HIGH và chờ cho tới khi đủ 10µs (timeout = 0).
- `HCSR04_WAIT_FOR_ECHO_HIGH`: là trạng thái chờ cho chân echo lên mức HIGH sau khi đã hoàn thành kích chân trigger.
- `HCSR04_WAIT_FOR_ECHO_LOW`: là trạng thái chờ cho chân echo xuống mức LOW để đo độ rộng xung echo.



Hình 3.3: Sơ đồ state machine của thread đo dữ liệu cảm biến

Thread này hoạt động theo vòng lặp như sau:

```
while(1){
 for(int i = 0; i < NUM_OF_HCSR04; i++){
      switch(HCSR04List[i].state){
           // xử lý theo từng trạng thái
      }
 }
}
```

Tức là thread sẽ làm việc với từng cảm biến một và tuỳ vào trạng thái của cảm biến đó mà xử lý theo. Cụ thể như sau:

- Khi cảm biến ở trạng thái `HCSR04_IDLE`, nếu `timeout = 0`, tức là đã đến lúc có thể kích hoạt lại một quá trình đo mới, thì thread sẽ gửi tín hiệu trigger bằng cách set chân trigger lên mức HIGH. Sau đó nó đặt `timeout = 1` (tương đương với 1ms) và chuyển sang trạng thái `HCSR04_WAIT_FOR_TRIGGER`. Như vậy, trong vòng lặp tiếp theo, thread sẽ không xử lý lại trigger ngay lập tức mà sẽ đợi `timeout` giảm về 0. Điều này tạo ra một khoảng delay nhỏ để đảm bảo trigger pulse đủ dài.

- Khi `timeout = 0` trong trạng thái `HCSR04_WAIT_FOR_TRIGGER`, thread đưa chân trigger về mức LOW và chuyển sang trạng thái `HCSR04_WAIT_FOR_ECHO_HIGH`, đồng thời thiết lập `timeout = 50` (giới hạn thời gian chờ tín hiệu echo lên mức cao). Đây là giai đoạn thread bắt đầu "nghe" phản hồi của cảm biến từ môi trường. Giá trị `lastEchoState` được reset tại đây để chuẩn bị cho phát hiện cạnh lên.
- Ở trạng thái `HCSR04_WAIT_FOR_ECHO_HIGH`, thread không chỉ đơn thuần kiểm tra chân echo đang ở mức HIGH hay không. Thay vào đó, nó sử dụng kỹ thuật phát hiện cạnh lên bằng cách so sánh `echoNow` (một biến lưu mức điện áp hiện tại của chân echo) và `lastEchoState`. Khi một cạnh lên thực sự xảy ra (tức là `echoNow = 1 && lastEchoState = 0`), thread ghi lại thời điểm bắt đầu echo vào thành viên `echoStart` bằng giá trị đếm hiện tại của timer TIM3, rồi chuyển sang trạng thái `HCSR04_WAIT_FOR_ECHO_LOW` và đặt lại `timeout`. Tuy nhiên, nếu sau một khoảng thời gian timeout mà echo không lên mức HIGH (tức không có phản hồi từ môi trường), thread sẽ bỏ qua quá trình đo đó và quay lại trạng thái `HCSR04_IDLE`, sẵn sàng cho một lần đo khác. Điều này giúp hệ thống không bị kẹt mãi trong một trạng thái nếu cảm biến không phản hồi.
- Sau khi phát hiện cạnh lên, thread sẽ chuyển sang `HCSR04_WAIT_FOR_ECHO_LOW`. Tại đây, thread tiếp tục quan sát chân echo nhưng lần này là để phát hiện cạnh xuống. Khi chân echo từ mức HIGH về LOW (tức là cạnh xuống được phát hiện qua điều kiện `echoNow = 0 && lastEchoState = 1`), thread ghi lại thời điểm kết thúc echo vào thành viên echoStop, tính toán khoảng cách bằng công thức `(echoStop - echoStart)/58`, đóng gói dữ liệu vào một frame, và đưa vào 2 hàng đợi UART và OLED. Sau đó, nó quay lại trạng thái `HCSR04_IDLE`, đặt `timeout = 10` để tạo một khoảng nghỉ trước khi cho phép cảm biến hoạt động lại. Trong trường hợp echo không xuống trong thời gian chờ, thread cũng quay về `HCSR04_IDLE` để tránh kẹt vòng đo.

Hệ thống có thêm 1 periodic task để xử lý chính xác timeout. Task này đơn thuần chỉ là truy cập vào thành viên timeout của từng cảm biến và giảm dần giá trị của nó.

```
void HCSR04_TimeoutTick(void){
 for(int i = 0; i < NUM_OF_HCSR04; i++){
     if(HCSR04List[i].timeout > 0)
     HCSR04List[i].timeout--;
```

```
  }
}
```

Vì task này được khởi tạo với chu kỳ là 1ms nên cứ sau mỗi 1ms thì thành viên timeout của tất cả các cảm biến đều bị trừ đi 1.

```
osKernelAddPeriodicThreads(HCSR04_TimeoutTick, 1);
```

3.2.3 Truyền dữ liệu ra giao diện

Để có thể hiển thị các dữ liệu đo được, đề tài sử dụng giao thức UART để truyền dữ liệu tới PC. Sau khi thu được các dữ liệu của cảm biến, dữ liệu sẽ được lưu vào queue để thread này có thể gửi đi. Nhiệm vụ chính của thread này là lấy dữ liệu từ queue, chuyển dữ liệu từ dạng số sang dạng chuỗi ký tự, đóng gói dữ liệu và truyền đi.

Việc lấy dữ liệu từ queue thì khá đơn giản, chỉ cần gọi hàm `HCSR04_QueueGet`. Khung dữ liệu nhận từ queue sẽ gồm 2 thành viên là ID của cảm biến và khoảng cách mà cảm biến đó thu được. Dựa vào ID của cảm biến, ta lưu khoảng cách mà cảm biến đó thu được vào một mảng với vị trí tương ứng với ID của cảm biến đó, tạo thành một chuỗi số.

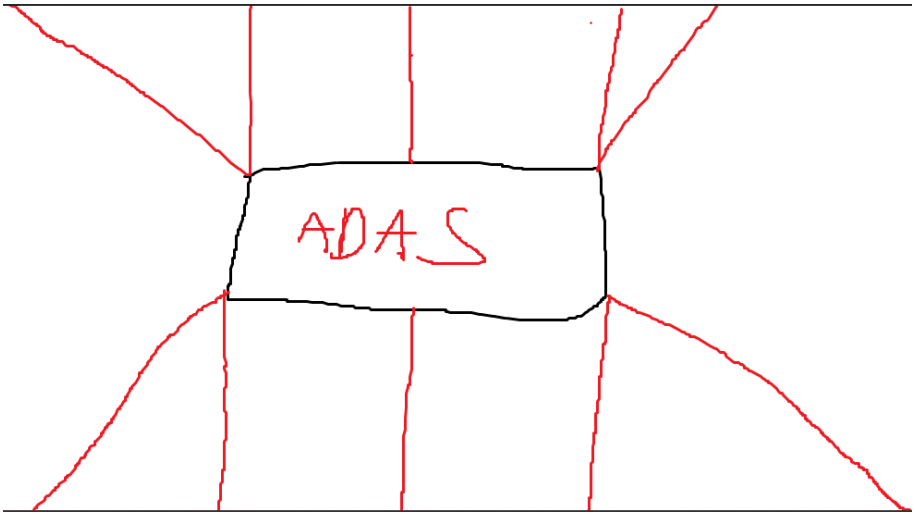
Thread này sẽ chuyển đổi các dữ liệu số này thành chuỗi ký tự bằng hàm `UART_ConvertIntToString`. Hàm này chuyển đổi số nguyên bằng cách lấy từng chữ số từ phải sang trái thông qua phép chia 10 lấy dư, sau đó lưu vào chuỗi. Khi số đã tách hết, hàm đảo ngược chuỗi lại để được đúng thứ tự từ trái sang phải.

```
void UART_ConvertIntToString(uint32_t num, char *str){
 int i = 0;
 if (num == 0){
      str[i++] = '0';
      str[i] = '\0';
      return;
 }
 while (num > 0){
      str[i++] = (num % 10) + '0';
      num /= 10;
 }
 str[i] = '\0';

 // đảo chuỗi lại
 for (int j = 0; j < i/2; j++) {
      char tmp = str[j];
      str[j] = str[i - j - 1];
      str[i - j - 1] = tmp;
 }
}
```

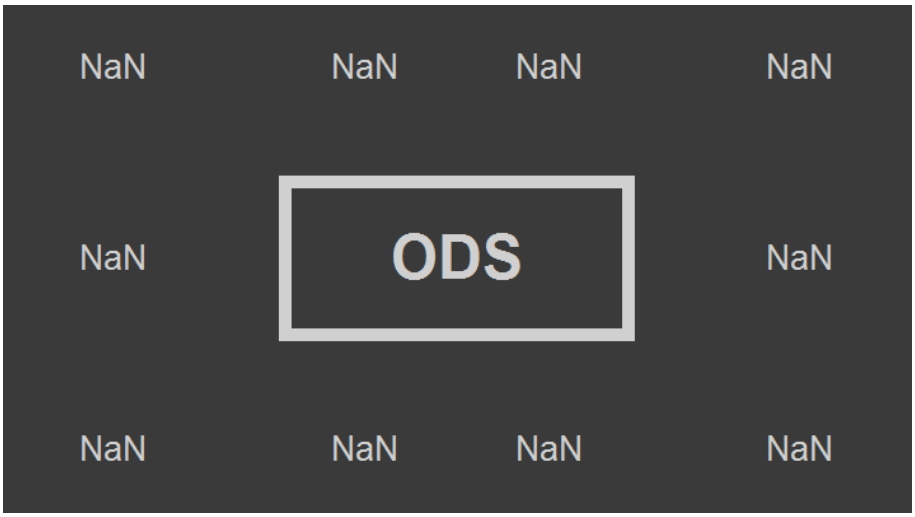
Mảng sau khi được chuyển đổi sẽ có format là *"distance0,distance1,distance2,...,distance9\r\n"* và sẽ được truyền đi sử dụng hàm `UART_SendString` (bản chất là gửi 1 ký tự nhưng lặp lại nhiều lần).

3.3 Thiết kế và lập trình phần mềm giao diện

Đề tài sẽ sử dụng phần mềm Qt để làm framework cho giao diện. Phần mềm này sử dụng ngôn ngữ C++ để lập trình logic cho giao diện.



Hình 3.4: Hình ảnh giao diện được phác thảo bằng tay



Hình 3.5: Giao diện của đề tài

Hình chữ nhật trung tâm đại diện cho 1 chiếc xe ô tô (hướng bên trái là đầu xe). Các label xung quanh sẽ hiển thị các dữ liệu khoảng cách tương ứng với các vị trí được đặt cảm biến. Giao diện được lập trình để nhận dữ liệu từ cổng serial, phân tách dữ liệu thành các số và gán vào vị trí tương ứng.

Cổng serial được cài đặt là baud rate là 115200, dữ liệu 8 bit, không sử dụng parity và 1 stop bit. Giao diện sử dụng nút nhỏ để kết nối tới cổng serial. Dữ liệu sau khi thu được sẽ được phân tách (bỏ \r\n, bỏ dấu phẩy, bỏ các khoảng trắng thừa) và chuyển đổi thành tập hợp các số khác nhau. Sau đó gán chúng vào các vị trí tương ứng đã quy định ở mục 3.2.2.

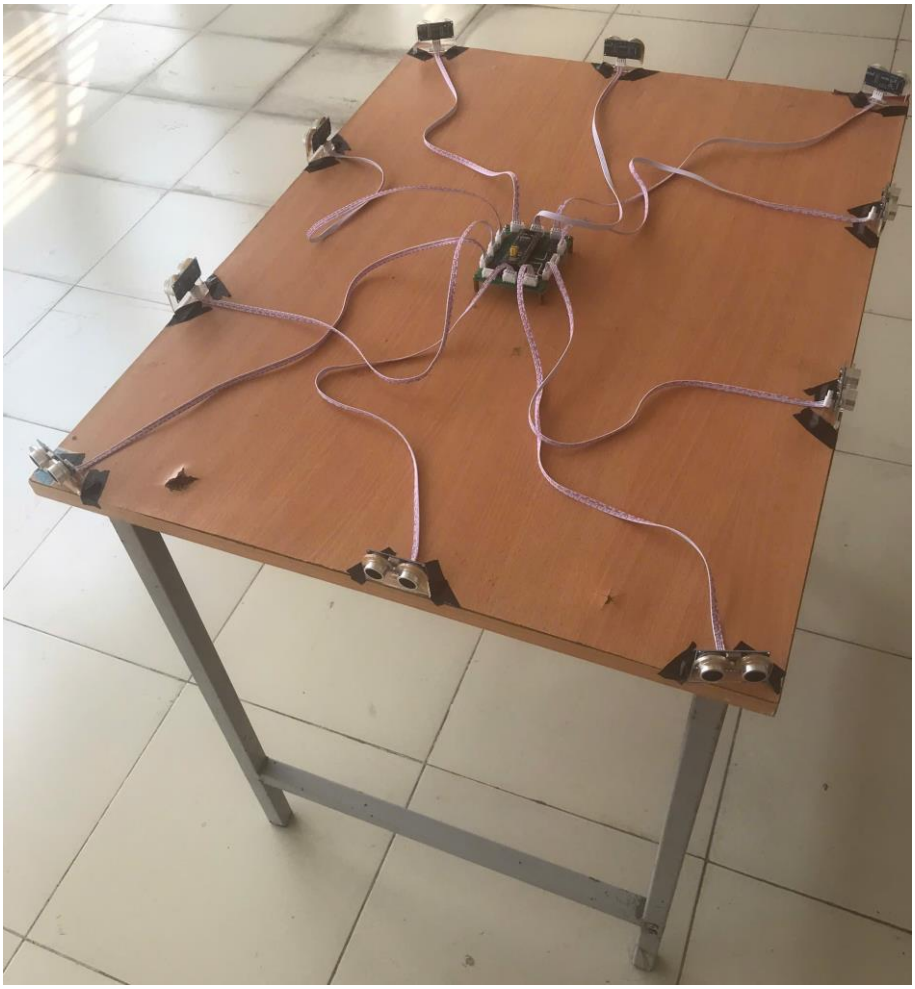
```
labelList[0] = ui->rearLeft;
labelList[1] = ui->left1;
labelList[2] = ui->left2;
labelList[3] = ui->backLeft;
labelList[4] = ui->back;
labelList[5] = ui->backRight;
labelList[6] = ui->right2;
labelList[7] = ui->right1;
labelList[8] = ui->rearRight;
labelList[9] = ui->rear;
```

(rear: phía trước của xe, back: phía sau của xe).

4. KẾT QUẢ THỰC HIỆN ĐỀ TÀI

4.1 Thiết lập môi trường thử nghiệm

Để có thể mô phỏng một cách thực tế nhất có thể, các cảm biến sẽ được đặt trên mặt bàn học cơ bản. Các vị trí của các cảm biến được quy định ở mục 3.2.2 cũng sẽ được đặt ở các vị trí tương ứng trên mặt bàn (xem Hình 4.1).



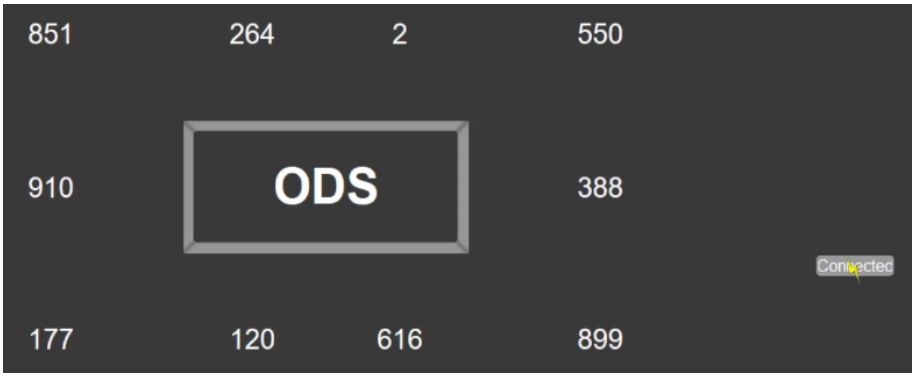
Hình 4.1: Thiết lập môi trường thử nghiệm trên mặt bàn

Để có thể cấp nguồn cho hệ thống và quan sát được dữ liệu của hệ thống, ta kết nối module chuyển đổi UART-USB với máy tính. Lúc này, người thử nghiệm phải lựa chọn vị trí thử nghiệm sao cho không làm ảnh hưởng đến môi trường xung quanh hệ thống.

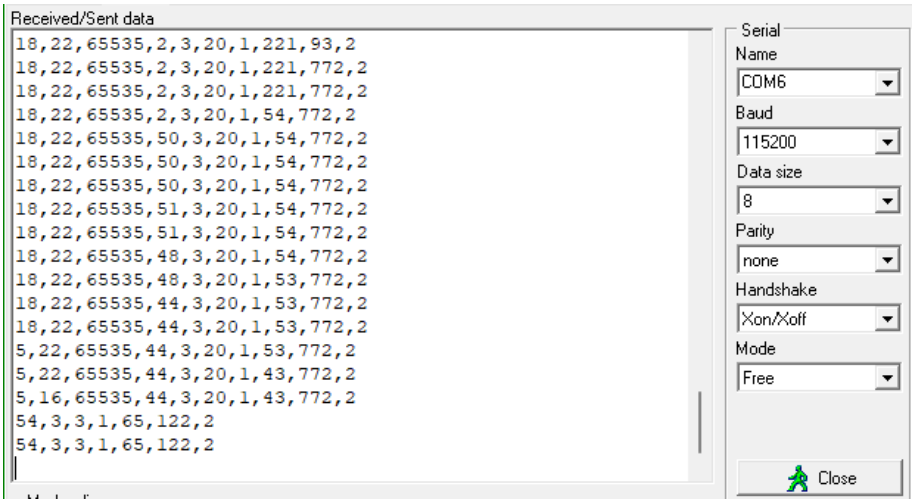
Hình 4.2: Người thử nghiệm đang thử nghiệm hệ thống

4.2 Kết quả thử nghiệm của hệ thống

Sau khi thử nghiệm và quan sát giao diện, ta có kết quả thử nghiệm như sau:



Hình 4.3: Dữ liệu thu được của các cảm biến khi quan sát qua giao diện



Hình 4.4: Kết quả quan sát được khi vi điều khiển truyền dữ liệu đi thông qua UART và phần mềm Hercules

Kết luận thu được sau quá trình thử nghiệm đề tài:

- 8/10 dữ liệu của các cảm biến hoạt động bình thường, dữ liệu chưa quá ổn định nhưng hiển thị đúng với những gì mong đợi.
- RTOS hoạt động bình thường, trong quá trình thử nghiệm không bị crash chương trình hay bị treo ở bất kỳ đâu.
- Các cảm biến hay mạch điện của đề tài hoạt động bình thường trong quá trình thử nghiệm và không bị hư hỏng hay ngắn mạch trong quá trình thử nghiệm.
- Dữ liệu hiển thị trên giao diện phản ứng đủ nhanh so với yêu cầu của hệ thống.
- Giao diện hiển thị đúng với những gì mà vi điều khiển truyền đi. Tuy nhiên, trong quá trình xử lý dữ liệu còn bị dư ở dữ liệu cuối (xem).

Hình 4.5: Dữ liệu cuối chuỗi (nằm ở vị trí trước cảu mô hình) bị lỗi

4.3 Hướng phát triển của đề tài

Đề tài nằm ở phạm vi nền tảng của những sản phẩm ngoài thị trường nên sẽ có rất nhiều hướng phát triển khác nhau. Cụ thể như sau:

- Thêm camera hay AI để giúp mô hình có thể nhận diện chính xác vật thể trong môi trường. Qua đó, nâng cao độ tin cậy của hệ thống khi bây giờ, nó có thể phát hiện làn đường giúp cho tài xế kiểm soát phương tiện của mình hoặc có thể cho tài xế cái nhìn trực quan hơn về môi trường xung quanh.
- Thay thế cảm biến siêu âm bằng các cảm biến phát hiện vật thể nhanh và chính xác hơn như cảm biến LiDAR. Điều này giúp cho hệ thống sẽ phản ứng nhanh hơn khi có sự thay đổi dữ liệu của các cảm biến.
- Phát triển sâu hơn RTOS như thêm các tính năng như quản lý bộ nhớ, sự ưu tiên của các task,… Hoặc phát triển RTOS sao cho nó có thể tương thích với mọi dòng vi điều khiển. Đây là điều mà các RTOS phổ biến như FreeRTOS có thể làm được.

5. TÀI LIỆU THAM KHẢO

[1] Richard E. Kowalski (10/2016), *Real-Time Operating Systems (RTOS) 101*. Truy cập từ: www.nasa.gov.

[2] Jean Labrosse (26 – 28/02/2019), *Everything You Need to Know about RTOSs in 30 Minutes*. Truy cập từ: www.silabs.com.

[3] Mubina Toa, Akeem Whitehead (12/2021), *Ultrasonic Sensing Basics*. Truy cập từ: www.ti.com.