

ĐẠI HỌC BÁCH KHOA HÀ NỘI



BÀI TẬP LỚN THIẾT KẾ HỆ THỐNG NHÚNG

PHẠM HUY TUYÊN

Tuyen.PH2021301@sis.hust.edu.vn

LƯU ĐÌNH TÚ

Tu.Ld213016@sis.hust.edu.vn

Ngành Kỹ thuật Điều khiển và Tự động hóa

Giảng viên hướng dẫn: TS. LÊ CÔNG CUỜNG

Chữ ký của GVHD

Khoa: Tự động hóa

Trường: Điện - Điện tử

Hà Nội, 1/2025

LỜI CẢM ƠN

Chúng em xin gửi đến thầy lời cảm ơn chân thành nhất vì sự tận tâm và nhiệt huyết mà thầy đã dành cho chúng em trong suốt kỳ học vừa qua. Chúc thầy luôn dồi dào sức khỏe, hạnh phúc và thành công trong sự nghiệp trồng người cao quý.

Hà Nội, ngày 22 tháng 1 năm 2025

Sinh viên thực hiện

Phạm Huy Tuyên-Lưu Đình Tú

MỤC LỤC

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT	i
DANH MỤC HÌNH VẼ	iii
DANH MỤC BẢNG BIỂU	iv
CHƯƠNG 1. Thông tin nhóm	1
CHƯƠNG 2. YÊU CẦU BÀI TẬP LỚN	2
2.1 Đề tài được giao	2
CHƯƠNG 3. GHÉP NỐI PHẦN CỨNG	3
3.1 Nguyên lý làm việc các module phần cứng	3
3.1.1 Cảm biến nhiệt độ , độ ẩm SHT21	3
3.1.2 Module thời gian thực DS3231	4
3.1.3 LCD 16X2	6
3.1.4 Module UART USB to TTL	6
3.1.5 Vi điều khiển STM32F401RE Nucleo	7
3.2 Các ngoại vi sử dụng	8
3.2.1 Timer	8
3.2.2 UART	9
3.2.3 Ngoại vi I2C	10
3.3 Sơ đồ ghép nối giữa STM32 và các moudle phần cứng	13
CHƯƠNG 4.THIẾT KẾ PHẦN MỀM	14
4.1 Phân tích yêu cầu xử lý để đảm bảo tính thời gian thực cho các đối tượng	14
4.2 Phân tích và tính toán các tham số cho các Task	14
4.2.1 Thông số các task trong các mô hình đa nhiệm	14
4.2.2 Thông số các task trong các mô hình đơn nhiệm	16
4.3 Phân tích và lập trình cho yêu cầu BTL theo 2 mô hình đơn nhiệm	18
4.3.1 Simple Periodic TT Scheduler	18
4.3.2 Non-Preemptive Event-Triggered Scheduling	23

4.4	Phân tích và lập trình cho yêu cầu BTL theo mô hình đa nhiệm	31
4.4.1	Mô hình đa nhiệm RMS	31
4.4.2	Lập trình các task hoạt động theo mô hình RMS	35
4.4.3	Mô hình đa nhiệm EDF	45
4.4.4	Lập trình các task hoạt động theo mô hình EDF	47
4.5	Time slice và Priority based Scheduling	54
4.5.1	Khái niệm	54
4.5.2	Phân tích tính toán	55
4.5.3	Triển khai lập trình	55
CHƯƠNG 5. KẾT QUẢ VÀ ĐÁNH GIÁ		57
5.1	Các kết quả đạt được	57
5.1.1	Simple Periodic TT Scheduler	57
5.1.2	Non-Preemptive Event-Triggered Scheduling	58
5.1.3	Mô hình đa nhiệm RMS	62
5.1.4	Mô hình đa nhiệm EDF	64
5.1.5	Mô hình đa nhiệm Time slice và Priority based Scheduling	66
5.2	Phân tích và đánh giá kết quả	67
5.2.1	Simple Periodic Time-Triggered Scheduler	67
5.2.2	Non-Preemptive Event-Triggered Scheduling	68
5.2.3	Mô hình EDF (Earliest Deadline First):	68
5.2.4	Mô hình RMS (Rate Monotonic Scheduling):	69
5.2.5	Mô hình đa nhiệm Time slice và Priority based Scheduling	69
5.3	Thông kê công việc và tự đánh giá mức độ đóng góp của các thành viên (thang 10)	70
CHƯƠNG 6. LINK SHARE THỦ MỤC		71
KẾT LUẬN		72
TÀI LIỆU THAM KHẢO		73
PHỤ LỤC		73

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

DANH MỤC HÌNH VẼ

Hình 3.1. Module SHT21	4
Hình 3.2. Module DS3231	5
Hình 3.3. LCD 16X2	6
Hình 3.4. USB to TTL	7
Hình 3.5. STM32 Nucleo	8
Hình 3.6. Khung dữ liệu truyền của I2C	11
Hình 3.7. Sơ đồ ghép nối giữa STM32 và các module	13
Hình 4.1. Lưu đồ thuật của Time-Triggered Cyclic Executive Scheduler . .	19
Hình 4.2. Bộ lập lịch dựa trên thông số các task	20
Hình 4.3. Cấu hình các ngoại vi cần thiết	21
Hình 4.4. Cấu hình timer 3	22
Hình 4.5. Lưu đồ thuật toán mô hình Non-Preemptive Event-Triggered Scheduling	25
Hình 4.6. Lưu đồ thuật toán thay đổi chu kỳ lập lịch	26
Hình 4.7. Định dạng bản tin thay đổi chu kỳ lập lịch	26
Hình 4.8. Cấu hình các ngoại vi cần thiết	27
Hình 4.9. Cấu hình các mức ưu tiên ngắn	28
Hình 4.10. Cấu hình timer	28
Hình 4.11. Thiết kế bộ lập lịch RMS dựa vào thông số task	34
Hình 4.12. Lưu đồ thuật toán hoạt động mô hình RMS	35
Hình 4.13. Hoạt động của task xử lý ngắn trì hoãn	36
Hình 4.14. Hoạt động của task đo	37
Hình 4.15. Hoạt động của task hiển thị, truyền thông số qua LCD, UART .	38
Hình 4.16. Cấu hình xung clock cho module vi điều khiển	39
Hình 4.17. Cấu hình các ngoại vi để giao tiếp với phần cứng.	39
Hình 4.18. Thiết kế thuật toán EDF	47
Hình 4.19. Lưu đồ thuật toán hoạt động mô hình EDF	47
Hình 4.20. Cấu trúc dữ liệu gửi xuống từ máy tính	48
Hình 4.21. Lưu đồ thuật toán xử lý ngắn	48
Hình 4.22. Lưu đồ thuật toán hoạt động task đọc dữ liệu từ cảm biến . .	49
Hình 4.23. Lưu đồ thuật toán hoạt động task hiển thị qua LCD và truyền qua UART	50

Hình 4.24. Chu trình thực hiện các task	55
Hình 4.25. Cấu hình freeRTOS	56
Hình 5.1. Các task đã chạy đúng theo thời gian cài đặt	57
Hình 5.2. Màn hình hiển thị LCD	58
Hình 5.3. Các task đã chạy đúng theo thời gian cài đặt	58
Hình 5.4. Điều chỉnh thành công chu kỳ của task1	59
Hình 5.5. Lấy thông số frame truyền	59
Hình 5.6. Lấy thông số của đối số truyền vào	60
Hình 5.7. Lấy thông số của tất cả các hàm đang có	60
Hình 5.8. Màn hình hiển thị LCD	61
Hình 5.9. Kết quả in ra khi các task hoạt động theo mô hình RMS	62
Hình 5.10. Khi nhận ngắt yêu cầu thay đổi thông số deadline và chu kỳ task 5	63
Hình 5.11. Kết quả hiển thị trên LCD	64
Hình 5.12. Kết quả hoạt động theo EDF	64
Hình 5.13. Kết quả nhận thay đổi thông số task từ ngắt UART trên màn hình	65
Hình 5.14. Kết quả hiển thị lên LCD	65
Hình 5.15. Màn hình hiển thị LCD	66
Hình 5.16. Đánh giá CPU và các task hoạt động	67

DANH MỤC BẢNG BIỂU

Bảng 4.1. Execution times for 4 tasks	17
Bảng 4.2. Thông số của các task.	33
Bảng 4.3. Bảng các tác vụ với chu kỳ, giá trị C và mức độ ưu tiên.	34
Bảng 4.4. Bảng các tác vụ với deadline, mức độ ưu tiên, chu kỳ và giá trị C. .	46
Bảng 4.5. Phân tích chu kỳ, deadline, thời gian thực thi và mức ưu tiên của các task	55

CHƯƠNG 1. THÔNG TIN NHÓM

Sinh viên thực hiện:

- Mã nhóm: 14
- Lưu Đình Tú: 20213016
- Phạm Huy Tuyên: 20213031

CHƯƠNG 2. YÊU CẦU BÀI TẬP LỚN

2.1 Đề tài được giao

Lập trình VĐK STM32 đọc cảm biến nhiệt độ, thời gian thực hiển thị LCD đồng thời truyền thông số lên máy tính thông qua UART. Cài đặt được thông số lập lịch và nhận lệnh điều khiển từ máy tính.

CHƯƠNG 3. GHÉP NỐI PHẦN CỨNG

3.1 Nguyên lý làm việc các module phần cứng

3.1.1 Cảm biến nhiệt độ, độ ẩm SHT21

3.1.1.1 Nguyên lý hoạt động.

Phần đo nhiệt độ: SHT21 sử dụng cảm biến nhiệt điện trở (thermistor) hoặc diode bán dẫn để đo nhiệt độ. Khi nhiệt độ môi trường thay đổi, điện trở hoặc dòng điện qua diode thay đổi, cảm biến chuyển đổi giá trị này thành tín hiệu kỹ thuật số.

Phần đo độ ẩm: Được tích hợp một lớp polymer nhạy cảm với độ ẩm. Lớp này hấp thụ hoặc nhả hơi nước từ môi trường, làm thay đổi hằng số điện môi, từ đó thay đổi điện dung. Giá trị điện dung được đo và chuyển đổi thành độ ẩm.

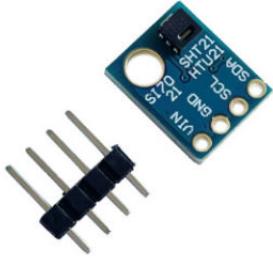
3.1.1.2 Ngoại vi kết nối.

SHT21 là một module đo nhiệt độ độ ẩm phổ biến và kết nối với vi điều khiển thông qua giao tiếp I²C.

3.1.1.3 Giao diện và chân kết nối.

Cảm biến SHT21 có 4 chân được mô tả như sau:

1. **VDD**: Chân nguồn cấp cho cảm biến, hoạt động ở mức điện áp từ 2.1 V đến 3.6 V, với điện áp tiêu chuẩn là 3.3 V.
2. **GND**: Chân nối đất (Ground), đóng vai trò làm mạch tham chiếu cho toàn bộ mạch điện.
3. **SCL**: Chân xung đồng hồ (*Serial Clock Line*), được sử dụng để đồng bộ dữ liệu khi giao tiếp qua giao thức I²C.
4. **SDA**: Chân truyền dữ liệu (*Serial Data Line*), đảm nhiệm việc truyền và nhận dữ liệu qua giao thức I²C.



Hình 3.1. Module SHT21

3.1.1.4 Kết nối với vi điều khiển.

Kết nối SCL và SDA với các chân tương ứng trên vi điều khiển hỗ trợ I²C.

Kết nối VCC và GND với nguồn cung cấp tương thích.

3.1.2 Module thời gian thực DS3231

Module thời gian thực DS3231 là một trong những chip RTC (Real-Time Clock) phổ biến, cung cấp chức năng duy trì thời gian chính xác. Dưới đây là chi tiết về nguyên lý hoạt động, ngoại vi kết nối, và các thông tin quan trọng liên quan:

3.1.2.1 Nguyên lý hoạt động.

DS3231 là một chip RTC sử dụng thạch anh tích hợp bên trong với độ chính xác cao để duy trì thời gian và ngày tháng. Thạch anh bên trong giúp loại bỏ sai số thường gặp trong các RTC khác, đặc biệt là ở nhiệt độ thay đổi.

- Tính năng chính: Đếm thời gian theo giây, phút, giờ, ngày, tháng, năm. Hỗ trợ lịch tự động, bao gồm năm nhuận.
- Chế độ 12 giờ hoặc 24 giờ, có hỗ trợ định dạng AM/PM.
- Nhiệt độ bù sai số: Chip có tích hợp cảm biến nhiệt độ để tự động hiệu chỉnh dao động thạch anh nhằm tăng độ chính xác.
- Pin dự phòng: Có thể kết nối với pin ngoài (thường là CR2032) để đảm bảo duy trì thời gian khi mất nguồn chính.
- Tích hợp giao tiếp I²C (Inter-Integrated Circuit) để giao tiếp với vi điều khiển.

3.1.2.2 *Ngoại vi kết nối.*

DS3231 thường được tích hợp trên các module RTC phổ biến và kết nối với vi điều khiển hoặc máy tính thông qua giao tiếp I²C.

3.1.2.3 *Giao diện và chân kết nối.*

Các chân cơ bản trên module DS3231 bao gồm:

- VCC: Cấp nguồn (3.3V hoặc 5V tùy module).
- GND: Chân nối đất.
- SCL (Serial Clock Line): Chân đồng hồ của giao tiếp I²C.
- SDA (Serial Data Line): Chân dữ liệu của giao tiếp I²C.
- BAT: Chân kết nối pin dự phòng.
- SQW (Square Wave Output): Cung cấp tín hiệu xung vuông có thể cấu hình (1Hz, 4kHz, 8kHz, 32kHz).



Hình 3.2. Module DS3231

3.1.2.4 *Kết nối với vi điều khiển.*

Kết nối SCL và SDA với các chân tương ứng trên vi điều khiển hỗ trợ I²C.

Kết nối VCC và GND với nguồn cung cấp tương thích.

Nếu cần, kết nối pin dự phòng vào chân BAT.

3.1.2.5 *Đặc điểm kỹ thuật chính.*

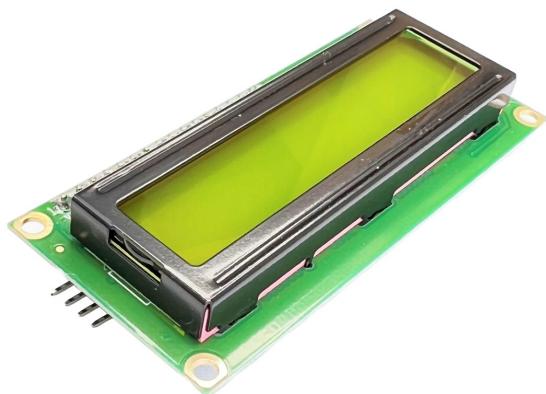
- Điện áp hoạt động: 2.3V - 5.5V.

- Dòng tiêu thụ: $3.5\mu\text{A}$ (ở chế độ pin dự phòng).
- Sai số: ± 2 ppm (phút mỗi năm) từ -40°C đến $+85^\circ\text{C}$.
- Nhiệt độ hoạt động: -40°C đến $+85^\circ\text{C}$.
- Tích hợp cảm biến nhiệt độ: Sai số $\pm 3^\circ\text{C}$.

3.1.3 LCD 16X2

Màn hình LCD 1602 (Liquid Crystal Display) được sử dụng trong rất nhiều ứng dụng của vi điều khiển. LCD 1602 có rất nhiều ưu điểm so với các dạng hiển thị khác như: khả năng hiển thị ký tự đa dạng (chữ, số, ký tự đồ họa); dễ dàng đưa vào mạch ứng dụng theo nhiều giao thức giao tiếp khác nhau. Dưới đây là một số thông số kỹ thuật quan trọng:

- Dải điện áp hoạt động: $2.7 \rightarrow 5.5$ VDC.
- Điện áp ra mức cao: > 2.4 VDC.
- Điện áp ra mức thấp: < 0.4 VDC.
- Dòng điện cấp nguồn: $350\mu\text{A} \rightarrow 600\mu\text{A}$.
- Nhiệt độ hoạt động: $-30 \rightarrow 75$ °C.



Hình 3.3. LCD 16X2

3.1.4 Module UART USB to TTL

Module thực hiện chức năng chuyển đổi tín hiệu USB sang tín hiệu nối tiếp tuần tự theo chuẩn TTL UART với một số thông số kỹ thuật sau:

- Điện áp hoạt động: 3.3 V hoặc 5 V.

- Dòng điện đầu ra: 100 mA.
- Nhiệt độ hoạt động: $-40^{\circ}\text{C} \rightarrow 85^{\circ}\text{C}$.
- Chuẩn giao tiếp: USB 2.0 và TTL.
- Hỗ trợ tốc độ truyền nhận trong khoảng 300 bps \rightarrow 1.5Mbps.
- Có cầu chì tự phục hồi để bảo vệ khi gặp sự cố ngắn mạch.
- Có 3 LED để hiển thị nguồn và chế độ truyền nhận tín hiệu.
- CP2102 không sử dụng thạch anh ngoài như các chip PL2303.

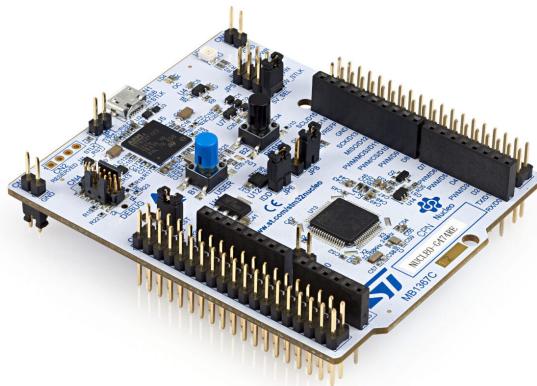


Hình 3.4. USB to TTL

3.1.5 Vi điều khiển STM32F401RE Nucleo

- Điện áp hoạt động: Điện áp hoạt động của kit dao động từ 2.7 V \rightarrow 3.6 V, thích hợp với các ứng dụng cần mức tiêu thụ điện năng thấp. Hỗ trợ mức tín hiệu logic chuẩn 3.3 V, phù hợp với nhiều loại cảm biến và module phổ biến.
- Bộ vi xử lý: Sử dụng ARM Cortex-M4, có FPU (Floating Point Unit) hỗ trợ xử lý dấu phẩy động, giúp tăng hiệu năng xử lý số liệu phức tạp. Tần số xung nhịp tối đa lên đến 84 MHz, cung cấp tốc độ xử lý mạnh mẽ. Bộ nhớ Flash tích hợp dung lượng 512 KB, đủ để lưu trữ firmware cho các ứng dụng phức tạp. RAM tích hợp 96 KB, đảm bảo khả năng quản lý dữ liệu linh hoạt.
- Thạch anh: Tích hợp thạch anh nội tần số 16 MHz, cung cấp độ chính xác cao cho hệ thống thời gian thực. Hỗ trợ kết nối thạch anh ngoài để sử dụng với các ứng dụng yêu cầu độ chính xác cao hơn hoặc nhiều nguồn clock khác nhau.

- Cổng USB: Tích hợp cổng USB 2.0 Full-Speed, cho phép sử dụng như một giao diện USB Device hoặc USB Host.
- Hỗ trợ nạp Bootloader qua USB, giúp nạp firmware một cách dễ dàng mà không cần sử dụng mạch nạp chuyên dụng.
- LED tích hợp: Led trên chân PC13, giúp người dùng có thể kiểm tra nhanh trạng thái hoạt động của vi điều khiển. Led này thường được dùng cho các bài thực hành cơ bản, như kiểm tra GPIO hoặc nháy led.
- Giao diện lập trình và nạp chương trình: Hỗ trợ chuẩn mạch nạp SWD (Serial Wire Debug), tương thích với các loại mạch nạp thông dụng như: ST-Link V2: Dễ dàng kết nối và nạp chương trình hoặc debug trực tiếp. J-Link: Hỗ trợ debug và lập trình với nhiều tính năng mạnh mẽ hơn. Tương thích với môi trường lập trình phổ biến như STM32CubeIDE, Keil uVision, IAR Embedded Workbench, hoặc sử dụng công cụ mã nguồn mở như PlatformIO.



Hình 3.5. STM32 Nucleo

3.2 Các ngoại vi sử dụng

3.2.1 Timer

Timer là một ngoại vi quan trọng trong vi điều khiển STM32, được sử dụng để thực hiện các chức năng liên quan đến thời gian, đo lường và điều khiển. Timer hỗ trợ nhiều tính năng như tạo tín hiệu PWM, đo thời gian, hoặc tạo các ngắt định kỳ.

Phân loại Timer:

STM32 cung cấp nhiều loại Timer phục vụ các mục đích khác nhau. **Basic Timer** (TIM6, TIM7) được thiết kế đơn giản để đếm thời gian. **General Purpose Timer** (TIM2 đến TIM5, TIM9 đến TIM14) là các Timer đa năng, hỗ trợ đo thời gian, tạo tín hiệu PWM và xử lý tín hiệu. Trong khi đó, **Advanced Timer** (TIM1, TIM8) có các tính năng nâng cao, thường được sử dụng trong các ứng dụng điều khiển động cơ.

Cấu trúc Timer :

Một Timer trong STM32 thường bao gồm các thành phần chính như sau:

- **Prescaler (PSC)**: Chia tần số xung nhịp để giảm tốc độ đếm.
- **Counter (CNT)**: Bộ đếm, thực hiện việc đếm các xung nhịp và tạo ra các sự kiện.
- **Auto-Reload Register (ARR)**: Xác định giá trị giới hạn của bộ đếm. Khi CNT đạt giá trị ARR, một sự kiện ngắt hoặc tín hiệu sẽ được tạo ra.
- **Capture/Compare Register (CCR)**: Dùng để đo tín hiệu đầu vào hoặc tạo tín hiệu đầu ra.
- **Interrupt and Event Generation**: Sinh ra các ngắt hoặc sự kiện khi Timer đạt đến các mốc nhất định.

Chế độ hoạt động của Timer:

STM32 hỗ trợ nhiều chế độ hoạt động khác nhau. Chế độ **Up Counter** đếm tăng từ 0 đến giá trị ARR, sau đó đặt lại về 0. Chế độ **Down Counter** đếm giảm từ giá trị ARR về 0. Trong khi đó, chế độ **Center-Aligned Mode** cho phép bộ đếm tăng và giảm đối xứng, thường dùng trong điều khiển động cơ. Ngoài ra, Timer còn hỗ trợ chế độ **PWM Mode (Pulse Width Modulation)** để tạo tín hiệu điều chỉnh rộng xung.

Chức năng của Timer :

Timer trong STM32 cung cấp nhiều chức năng hữu ích:

- **Delay chính xác**: Tạo khoảng thời gian trễ chính xác trong chương trình.
- **Tạo tín hiệu PWM**: Điều khiển thiết bị như động cơ, LED.
- **Đo thời gian**: Đo khoảng thời gian giữa các sự kiện hoặc tín hiệu đầu vào.
- **Tạo ngắt định kỳ**: Tạo ra các sự kiện định kỳ để thực thi một nhiệm vụ cụ thể.
- **Encoder Interface**: Đọc giá trị từ bộ mã hóa quay (encoder).
- **One Pulse Mode (OPM)**: Tạo một xung duy nhất sau khi một sự kiện kích hoạt.

3.2.2 **UART**

UART hay bộ thu-phát không đồng bộ đa năng là một trong những hình thức giao tiếp kỹ thuật số giữa thiết bị với thiết bị đơn giản và lâu đời nhất. Ta có thể tìm thấy các thiết bị UART trong một phần của mạch tích hợp (IC) hoặc dưới dạng các thành phần riêng lẻ. Các UART giao tiếp giữa hai nút riêng biệt bằng cách sử dụng một cặp dẫn và một nối đất chung. UART bao gồm chân TX và RX.

- Chân TX: Có nhiệm vụ truyền dữ liệu đến thiết bị giao tiếp cùng qua UART
- Chân RX: Có nhiệm vụ nhận dữ liệu từ thiết bị đang giao tiếp cùng qua UART

UART truyền dữ liệu nối tiếp, theo một trong ba chế độ:

- Full duplex: Giao tiếp đồng thời đến và đi từ mỗi master và slave - Half duplex: Dữ liệu đi theo một hướng tại một thời điểm
- Simplex: Chỉ giao tiếp một chiều

Dữ liệu truyền qua UART được tổ chức thành các gói. Mỗi gói chứa 1 bit bắt đầu, 5 đến 9 bit dữ liệu (tùy thuộc vào UART), một bit chẵn lẻ tùy chọn và 1 hoặc 2 bit dừng. **Ưu điểm**

- Chỉ sử dụng hai dây
- Không cần tín hiệu clock
- Có một bit chẵn lẻ để cho phép kiểm tra lỗi
- Cấu trúc của gói dữ liệu có thể được thay đổi miễn là cả hai bên đều được thiết lập cho nó
- Phương pháp có nhiều tài liệu và được sử dụng rộng rãi

Nhược điểm

- Kích thước của khung dữ liệu được giới hạn tối đa là 9 bit
- Không hỗ trợ nhiều hệ thống slave hoặc nhiều hệ thống master
- Tốc độ truyền của mỗi UART phải nằm trong khoảng 10% của nhau

3.2.3 Ngoài vi I2C

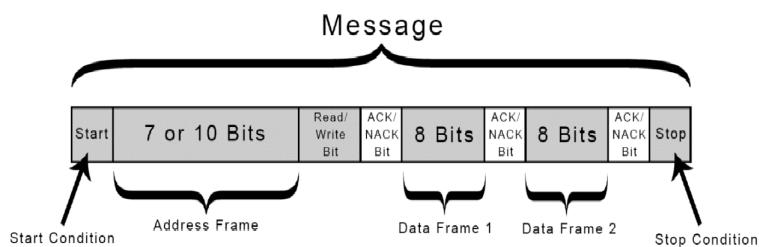
I2C kết hợp các tính năng tốt nhất của SPI và UART. Với I2C, bạn có thể kết nối nhiều slave với một master duy nhất (như SPI) và bạn có thể có nhiều master điều khiển một hoặc nhiều slave. Điều này thực sự hữu ích khi bạn muốn có nhiều hơn một vi điều khiển ghi dữ liệu vào một thẻ nhớ duy nhất hoặc hiển thị văn bản trên một màn hình LCD.

Giống như giao tiếp UART, I2C chỉ sử dụng hai dây để truyền dữ liệu giữa các thiết bị:

- SDA (Serial Data) - đường truyền cho master và slave để gửi và nhận dữ liệu.
- SCL (Serial Clock) - đường mang tín hiệu xung nhịp.

- I2C là một giao thức truyền thông nối tiếp, vì vậy dữ liệu được truyền từng bit đọc theo một đường duy nhất (đường SDA).

Giống như SPI, I2C là đồng bộ, do đó đầu ra của các bit được đồng bộ hóa với việc lấy mẫu các bit bởi một tín hiệu xung nhịp được chia sẻ giữa master và slave. Tín hiệu xung nhịp luôn được điều khiển bởi master. **Cách hoạt động của I2C** Với I2C, dữ liệu được truyền trong các tin nhắn. Tin nhắn được chia thành các khung dữ liệu. Mỗi tin nhắn có một khung địa chỉ chứa địa chỉ nhị phân của địa chỉ slave và một hoặc nhiều khung dữ liệu chứa dữ liệu đang được truyền. Thông điệp cũng bao gồm điều kiện khởi động và điều kiện dừng, các bit đọc / ghi và các bit ACK / NACK giữa mỗi khung dữ liệu:



Hình 3.6. Khung dữ liệu truyền của I2C

- Điều kiện khởi động: Đường SDA chuyển từ mức điện áp cao xuống mức điện áp thấp trước khi đường SCL chuyển từ mức cao xuống mức thấp.
- Điều kiện dừng: Đường SDA chuyển từ mức điện áp thấp sang mức điện áp cao sau khi đường SCL chuyển từ mức thấp lên mức cao.
- Khung địa chỉ: Một chuỗi 7 hoặc 10 bit duy nhất cho mỗi slave để xác định slave khi master muốn giao tiếp với nó.
- Bit Đọc / Ghi: Một bit duy nhất chỉ định master đang gửi dữ liệu đến slave (mức điện áp thấp) hay yêu cầu dữ liệu từ nó (mức điện áp cao).
- Bit ACK / NACK: Mỗi khung trong một tin nhắn được theo sau bởi một bit xác nhận / không xác nhận. Nếu một khung địa chỉ hoặc khung dữ liệu được nhận thành công, một bit ACK sẽ được trả lại cho thiết bị gửi từ thiết bị nhận.
- Địa chỉ: I2C không có các đường Slave Select như SPI, vì vậy cần một cách khác để cho slave biết rằng dữ liệu đang được gửi đến slave này chứ không phải slave khác. Nó thực hiện điều này bằng cách định địa chỉ. Khung địa chỉ luôn là khung đầu tiên sau bit khởi động trong một tin nhắn mới. Master gửi địa chỉ của slave mà nó muốn giao tiếp với mọi slave được kết nối với nó. Sau đó, mỗi slave sẽ so sánh địa chỉ được gửi từ master với địa chỉ của chính nó. Nếu địa chỉ phù hợp, nó sẽ gửi lại một bit ACK điện áp thấp cho master. Nếu địa chỉ không khớp, slave không làm gì cả và đường SDA vẫn ở mức cao.

- Bit đọc / ghi Khung địa chỉ bao gồm một bit duy nhất ở cuối tin nhắn cho slave biết master muốn ghi dữ liệu vào nó hay nhận dữ liệu từ nó. Nếu master muốn gửi dữ liệu đến slave, bit đọc / ghi ở mức điện áp thấp. Nếu master đang yêu cầu dữ liệu từ slave, thì bit ở mức điện áp cao. Sau khi master phát hiện bit ACK từ slave, khung dữ liệu đầu tiên đã sẵn sàng được gửi. Khung dữ liệu luôn có độ dài 8 bit và được gửi với bit quan trọng nhất trước. Mỗi khung dữ liệu ngay sau đó là một bit ACK / NACK để xác minh rằng khung đã được nhận thành công.
- Bit ACK phải được nhận bởi master hoặc slave (tùy thuộc vào cái nào đang gửi dữ liệu) trước khi khung dữ liệu tiếp theo có thể được gửi. Sau khi tất cả các khung dữ liệu đã được gửi, master có thể gửi một điều kiện dừng cho slave để tạm dừng quá trình truyền. Điều kiện dừng là sự chuyển đổi điện áp từ thấp lên cao trên đường SDA sau khi chuyển tiếp từ thấp lên cao trên đường SCL , với đường SCL vẫn ở mức cao.

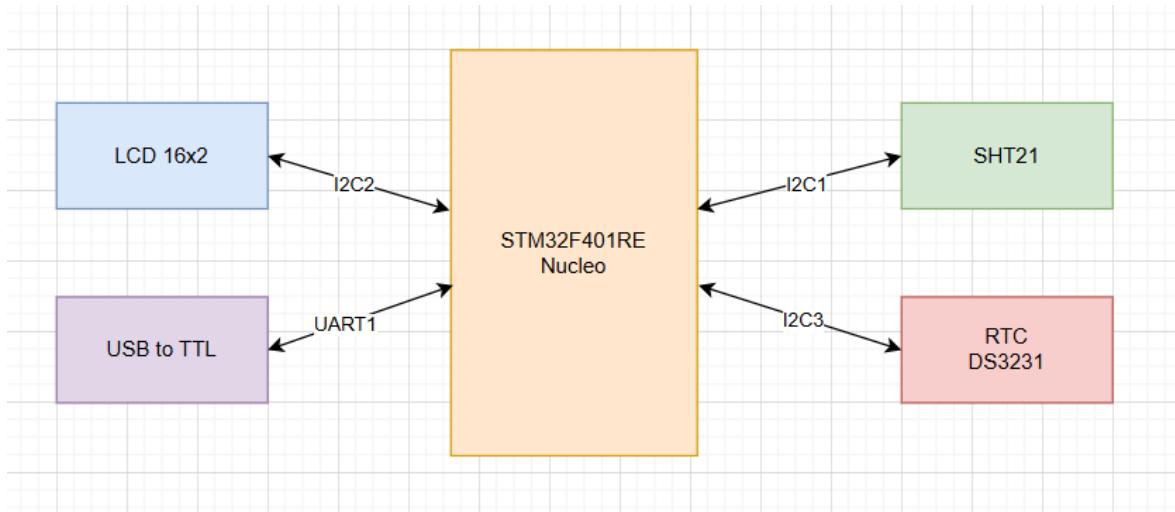
Ưu điểm

- Chỉ sử dụng hai dây
- Hỗ trợ nhiều master và nhiều slave
- Bit ACK / NACK xác nhận mỗi khung được chuyển thành công
- Phần cứng ít phức tạp hơn so với UART
- Giao thức nổi tiếng và được sử dụng rộng rãi

Nhược điểm

- Tốc độ truyền dữ liệu chậm hơn SPI
- Kích thước của khung dữ liệu bị giới hạn ở 8 bit
- Cần phần cứng phức tạp hơn để triển khai so với SPI

3.3 Sơ đồ ghép nối giữa STM32 và các module phần cứng



Hình 3.7. Sơ đồ ghép nối giữa STM32 và các module

CHƯƠNG 4. THIẾT KẾ PHẦN MỀM

4.1 Phân tích yêu cầu xử lý để đảm bảo tính thời gian thực cho các đối tượng

- Task đo nhiệt độ: Do nhiệt độ không thay đổi quá nhiều trong thời gian ngắn nên thời gian giữa 2 lần đo có thể để khoảng 2-3s
- Task đo thời gian thực: Do task này cần phải đo được thời gian thực trước khi gửi cho task hiển thị xử lý, để đảm bảo hiển thị chính xác thời gian thì task này chu kỳ có thể nhỏ hơn hoặc bằng 1 giây
- Task hiển thị nhiệt độ: Do phụ thuộc vào task đo nhiệt độ, chu kỳ giữa 2 lần gửi qua LCD và UART của nhiệt độ có thể để lớn hơn hoặc bằng task đo nhiệt độ
- Task hiển thị thời gian thực: Để đảm bảo luôn hiển thị thời gian chính xác, task hiển thị thời gian thực phải có chu kỳ thực thi cố định là 1 giây
- Task đo độ ẩm: tương tự như task đo nhiệt độ, độ ẩm thường không thay đổi quá nhiều trong thời gian ngắn nên có thể để task này khoảng 2-3s
- Task hiển thị độ ẩm qua LCD, UART: Do độ ẩm thường không thay đổi quá nhiều, có thể để task này khoảng 4-5s

4.2 Phân tích và tính toán các tham số cho các Task

Các thông số của các task:

- Chu kỳ T: Khoảng thời gian giữa 2 lần liên tiếp đối tượng được kích hoạt
- Deadline D: Khoảng thời gian tối đa của đối tượng kể từ khi kích hoạt cần phải hoàn thành để đảm bảo tính đúng đắn của đối tượng cần xử lý.
- Thời gian thực thi C: Thời gian cần thiết để đối tượng được xử lý hoàn toàn.
Thời gian thực thi của các task được tính toán dựa vào hàm osKernelGetTickCount() cho đa nhiệm, và HAL_GetTick() cho đơn nhiệm sau đó được đổi sang đơn vị milisecond.

4.2.1 Thông số các task trong các mô hình đa nhiệm

4.2.1.1 Đo nhiệt độ từ cảm biến SHT21.

Nhiệm vụ:

- Đọc nhiệt độ từ cảm biến SHT21 và xử lý.
- Cập nhật thông số các task nếu phát hiện có tín hiệu điều khiển từ máy tính
- Gửi vào hàng đợi sau khi đã đo được dữ liệu nhiệt độ thành công.

Thông số đối tượng cần xử lý:

- Chu kỳ: T_1 : 2250s.
Lý do: Dữ liệu nhiệt độ của 1 địa điểm tại 1 vùng thường thay đổi với tần suất không quá lớn, do đó có thể chọn 2s, nhiệt độ sẽ được cập nhật một lần.
- Deadline: D_1 : 100ms Lý do: Đảm bảo dữ liệu hoặc hành động không bị trì hoãn so với yêu cầu của hệ thống.
- Thời gian thực thi C_1 : 60ms.

4.2.1.2 Đo độ ẩm từ cảm biến SHT21.

Nhiệm vụ: Đọc dữ liệu độ ẩm từ cảm biến SHT21

Thông số đối tượng cần xử lý:

- Chu kỳ: T_1 : 1800s.
Lý do: Dữ liệu cảm biến trên SHT21 thường không cần đọc với tần suất quá nhiều do độ ẩm thay đổi chậm.
- Deadline: D_1 : 500ms
Lý do: Giá trị độ ẩm cần được cập nhật trước thời điểm kích hoạt của 1 task hiển thị
- Thời gian thực thi C_1 : 200ms.

4.2.1.3 Đo dữ liệu thời gian thực. Nhiệm vụ: Đọc dữ liệu thời gian thực từ cảm biến DS3231

Thông số đối tượng cần xử lý:

- Chu kỳ: T_1 : 900ms.
Lý do: Đồng hồ thời gian thực cần cập nhật mỗi giây để đảm bảo hiện thị giờ chính xác lên UART và LCD.
- Deadline: D_1 : 200 ms Lý do: Thời gian thực đọc được từ DS3231 cần được đọc trước khi task hiển thị được kích hoạt, đảm bảo dữ liệu hiện thị được giờ 1 cách chính xác.
- Thời gian thực thi C_1 : 60ms

4.2.1.4 Hiển thị dữ liệu nhiệt độ lên LCD và UART.

Nhiệm vụ: Hiện thị dữ liệu nhiệt độ đo được từ cảm biến SHT21 lên LCD và UART.

Thông số đối tượng cần xử lý:

- Chu kỳ: T_1 : 3000s.
Lý do: Nhiệt độ thường không thay đổi quá nhanh trong thời gian ngắn
- Deadline: D_1 : 300ms Lý do: Đảm bảo hoàn thành trước chu kỳ mới
- Thời gian thực thi C_1 : 30ms

4.2.1.5 Hiện thị dữ liệu độ ẩm lên LCD và UART. Nhiệm vụ: Hiện thị dữ liệu độ ẩm đo được từ cảm biến SHT21 lên LCD và UART.

Thông số đối tượng cần xử lý:

- Chu kỳ: T_1 : 4500s.
Lý do: Phụ thuộc vào chu kỳ đo độ ẩm, dữ liệu chỉ được đẩy lên LCD hoặc UART khi độ ẩm được đo thành công.
- Deadline: D_1 : 600ms Lý do: Thời gian để hiển thị lên LCD và UART
- Thời gian thực thi C_1 : 250ms

4.2.1.6 Hiển thị dữ liệu thời gian thực qua LCD và UART.

Nhiệm vụ: Hiện thị dữ liệu thời gian thực một cách chính xác lên LCD và UART.

- Chu kỳ: T_1 : 1s.
Lý do: Cần phải đồng bộ với đối tượng đọc thời gian thực, dữ liệu cần phải hiện thị 1s một lần để có thể quan sát được thời gian thực một cách chính xác nhất.
- Deadline: D_1 : 400ms.
Lý do: Dữ liệu thời gian thực cần phải được đẩy đi trước khi có dữ liệu mới tới, cũng để đảm bảo thời gian thực được cập nhật một cách chính xác.
- Thời gian thực thi C_1 : 60ms

4.2.2 Thông số các task trong các mô hình đơn nhiệm

Hệ thống sẽ gồm 4 task :

- Đọc nhiệt độ, độ ẩm từ cảm biến
- Đọc thời gian từ module thời gian thực
- Gửi các dữ liệu về thời gian, nhiệt độ, độ ẩm qua UART
- Gửi các dữ liệu về thời gian, nhiệt độ, độ ẩm lên LCD

Sử dụng hàm HAL_Gettick() đây là một hàm được sử dụng để lấy thời gian đã trôi qua tính bằng millisecond kể từ khi hệ thống khởi động. Đây là một phần của thư

viện HAL (Hardware Abstraction Layer) trong STM32, thường được sử dụng để đo thời gian hoặc tạo độ trễ trong các ứng dụng nhúng.

Ví dụ một đoạn chương trình lấy thời gian thực thi của task đọc nhiệt độ , độ ẩm từ cảm biến :

```

1 // Calculate execution time
2 float execute_time = 0;
3 uint32_t start_tick = HAL_GetTick();
4 read_temp_and_hum();
5 uint32_t end_tick = HAL_GetTick();
6 execute_time = (end_tick - start_tick) / 1000.f;

```

Ta có bảng thời gian thực thi của các task như sau :

Task ID	Task Name	Execution Time (s)
1	Đọc nhiệt độ , độ ẩm từ cảm biến	0.057
2	Đọc thời gian từ module thời gian thực	0.001
3	Gửi các dữ liệu về thời gian , nhiệt độ , độ ẩm qua UART	0.006
4	Gửi các dữ liệu về thời gian , nhiệt độ , độ ẩm lên LCD	0.112

Bảng 4.1. Execution times for 4 tasks

4.2.2.1 Đo nhiệt độ độ ẩm từ cảm biến.

Nhiệm vụ: Đo nhiệt độ độ ẩm từ cảm biến

Thông số đối tượng cần xử lý: • Chu kỳ: T1: 3s.

- Deadline: D1: 1s
- Thời gian thực thi C1: 0.057s.

4.2.2.2 Đọc giá trị thời gian của RTC.

Nhiệm vụ: Đọc giá trị thời gian từ thanh ghi của RTC : Thông số đối tượng cần xử lý:

- Chu kỳ: T1: 0.8s. Lý do: Không cần quá nhanh
- Deadline: D1: 0.5s Lý do: Để lấy được giá trị thời gian hiển thị lên LCD
- Thời gian thực thi C1: 0.001s

4.2.2.3 Gửi dữ liệu lên UART.

Nhiệm vụ: Hiện thị dữ liệu nhiệt độ độ ẩm và thời gian đo được qua UART :

Thông số đối tượng cần xử lý:

- Chu kỳ: T1: 5s. Lý do: Không cần quá nhanh
- Deadline: D1: 0.5s Lý do: Để hiển thị lên LCD
- Thời gian thực thi C1: 0.006s

4.2.2.4 Gửi dữ liệu lên LCD.

Nhiệm vụ: Hiện thị dữ liệu nhiệt độ độ ẩm và thời gian đo được lên LCD :

Thông số đối tượng cần xử lý:

- Chu kỳ: T1: 1s. Lý do: Phụ thuộc vào thời gian hiển thị lên LCD phải hiển thị đơn vị s
- Deadline: D1: 0.5s Lý do: Để hiển thị lên LCD
- Thời gian thực thi C1: 0.112s

4.3 Phân tích và lập trình cho yêu cầu BTL theo 2 mô hình đơn nhiệm

4.3.1 Simple Periodic TT Scheduler

4.3.1.1 Khái niệm

Simple Periodic TT Scheduler là một bộ lập lịch thuộc mô hình kích hoạt theo thời gian (Time-Triggered System) có các đặc điểm nổi bật như sau. Thứ nhất, các nhiệm vụ chỉ được kích hoạt bởi ngắt từ bộ định thời, và không có bất kỳ ngắt nào khác được phép, giúp đảm bảo tính nhất quán và ổn định của hệ thống. Thứ hai, lịch trình của các nhiệm vụ được tính toán ngoại tuyến, cho phép sử dụng các thuật toán phức tạp để tối ưu hóa trước khi hệ thống hoạt động. Khi hệ thống chạy, lịch trình này là cố định, mang tính quyết định, đảm bảo các nhiệm vụ luôn thực thi đúng thời gian và thứ tự đã định. Ngoài ra, hệ thống tương tác với môi trường thông qua cơ chế bỏ phiếu (polling), tức là kiểm tra dữ liệu từ môi trường tại các thời điểm xác định trước thay vì phản hồi ngay khi có sự kiện. Mô hình này đảm bảo độ tin cậy cao và phù hợp với các hệ thống nhúng quan trọng, nhưng hạn chế ở khả năng xử lý linh hoạt các sự kiện không dự đoán trước.

4.3.1.2 Nguyên lý hoạt động

Simple Periodic TT Scheduler cần có một bộ định thời (timer) tạo ngắt định kỳ với chu kỳ P. Tất cả các tác vụ (tasks) đều có cùng chu kỳ P.

Các tác vụ sau sẽ có thời điểm bắt đầu không xác định chính xác trước. Giao tiếp giữa các tác vụ hoặc sử dụng tài nguyên chung được đảm bảo an toàn nhờ trình tự được mặc định trước.

Điều kiện cần thiết: Tổng thời gian thực thi cực đại (WCET) của tất cả các tác vụ trong một chu kỳ phải nhỏ hơn hoặc bằng chu kỳ P.

4.3.1.3 Phân tích tính toán cho dự án

Ta có tổng thời gian thực thi của các task là :

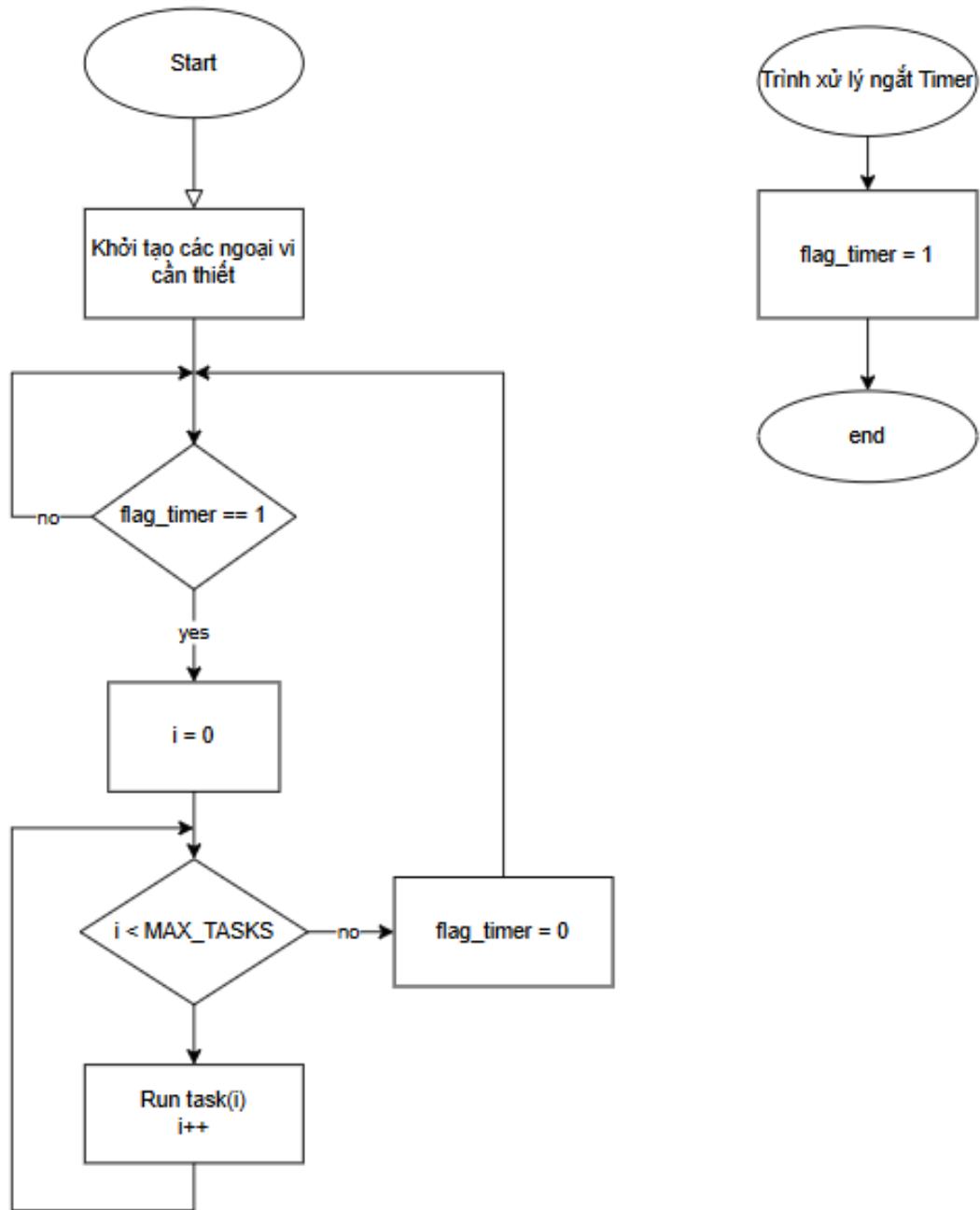
$$T_{\text{total}} = \sum_{i=1}^4 T_i = 0.057 + 0.001 + 0.006 + 0.112 = 0.176(s) \quad (4.1)$$

Vậy nên chu kỳ của Time-Triggered Cyclic Executive Scheduler phải lớn hơn giá trị này. Dựa vào chu kỳ nhỏ nhất của các task

$$P = 1(s) >> T_{\text{total}} \quad (4.2)$$

Vậy có thể sử dụng bộ lập lịch trên.

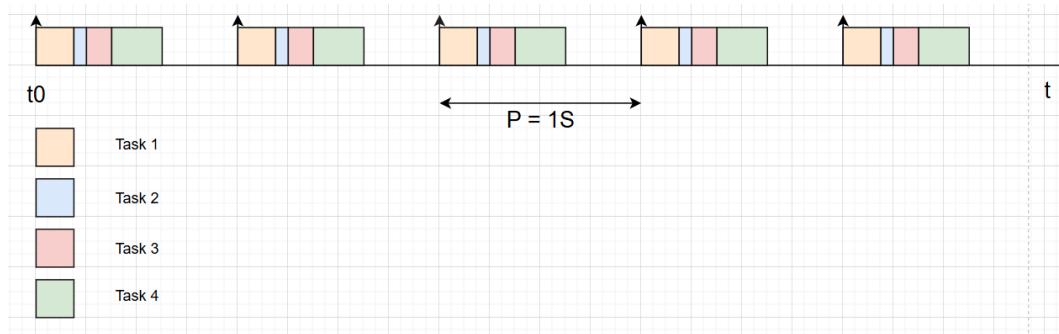
4.3.1.4 Lưu đồ thuật toán



Hình 4.1. Lưu đồ thuật của Time-Triggered Cyclic Executive Scheduler

Hệ thống hoạt động như sau: Ban đầu chương trình chạy và khởi tạo các ngoại vi cần thiết sử dụng như : UART, I2C , TIMER .Timer được khởi động và kích hoạt lần đầu tại thời điểm $t(0)$. Tại mỗi lần kích hoạt thứ i , bắt đầu từ $i = 0$, sẽ cho

còn flag_timer = 1 trong vòng while(1) là hàm handle_function kiểm tra liên tục cờ này. Nếu flag_timer = 1 thì sẽ chạy các nhiệm vụ theo thứ tự Task(0), Task(1), Task(2), ... cho đến hết khi các hàm . Sau khi hoàn thành các nhiệm vụ, timer được đặt lại để kích hoạt tại thời điểm $t = P + t(0)$, trong đó P là chu kỳ định trước. Khi tất cả nhiệm vụ đã hoàn tất, CPU sẽ chương trình chính nó có thể ngủ trong lúc chờ timer ngắt nếu thời gian chờ đủ dài.



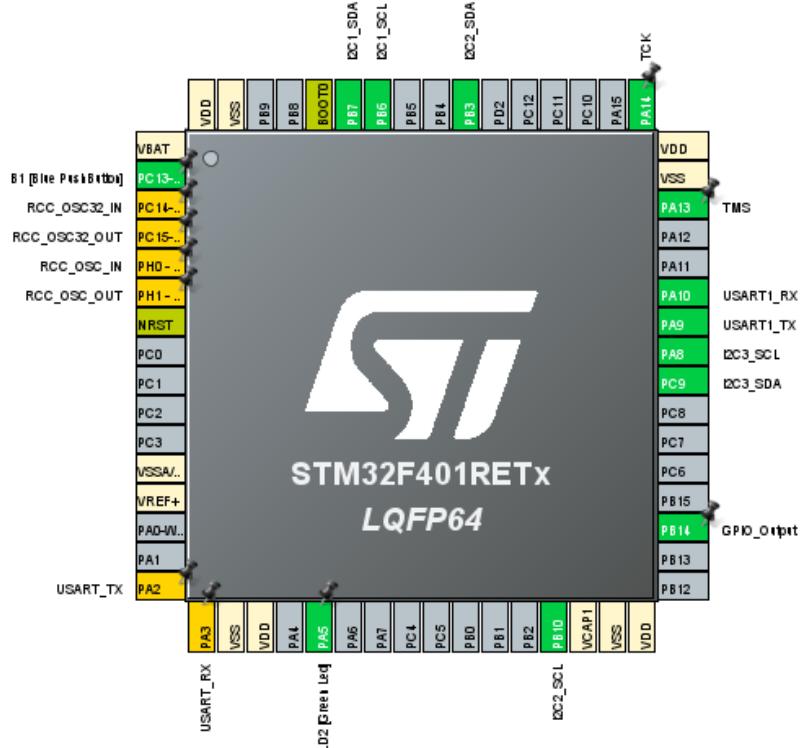
Hình 4.2. Bộ lập lịch dựa trên thông số các task

Với chu kỳ ngắn là 1 giây và có thời gian các task chạy như ở phần trước ta có bộ lập lịch dựa trên thông số các task như trên.

4.3.1.5 Triển khai lập trình

Cấu hình CubeMX

Trước hết ta phải cấu hình các ngoại vi sử dụng cần thiết như sau :



Hình 4.3. Cấu hình các ngoại vi cần thiết

Ta có công thức tính toán giá trị nạp vào thanh ghi bộ đếm (CNT) của ngắt timer được xác định như sau:

$$CNT = \frac{T_{\text{ngắt}} \cdot f_{\text{timer}}}{\text{Prescaler}} - 1$$

Trong đó:

- *CNT*: Giá trị nạp vào bộ đếm.
- $T_{\text{ngắt}}$: Thời gian mong muốn giữa các lần ngắt (giây).
- f_{timer} : Tần số clock đầu vào của timer (Hz).
- Prescaler: Giá trị chia tần số (*Prescaler*) được cấu hình trong timer.

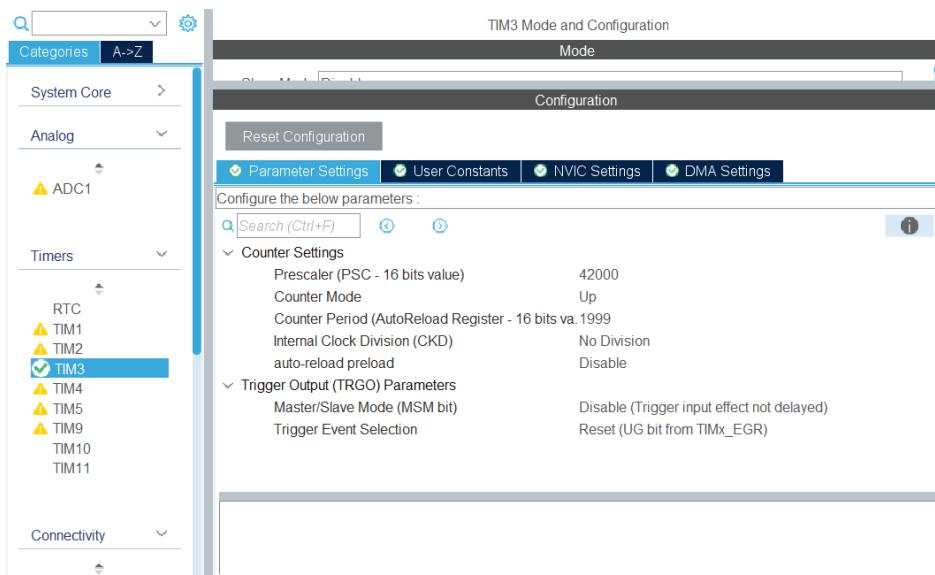
Với bài toán ta có :

- Tần số clock của timer: $f_{\text{timer}} = 84 \text{ MHz}$.
- Thời gian ngắt mong muốn: $T_{\text{ngắt}} = 1000 \text{ ms} = 1 \text{ s}$.
- Prescaler được cấu hình là 42000.

Áp dụng công thức:

$$CNT = \frac{1 \cdot 84 \times 10^6}{42000} - 1 = 1999$$

Do đó, giá trị nạp vào thanh ghi CNT là 1999.



Hình 4.4. Cấu hình timer 3

Chi tiết cấu hình timer 3 như hình 4.4. và sau đó chọn ngắt.

Quản lý các task theo con trỏ hàm

Để quản lý các hàm em sử dụng 1 cấu trúc dữ liệu như sau :

```
1  typedef struct {
2      void (*task_fn)(void);    // Function pointer
3      bool active;           // State of function / 1 or 0
4  } Task;
```

Với tham số đầu tiên sẽ là con trỏ hàm để quản lý các hàm. Tham số thứ 2 sẽ là trạng thái hoạt động của hàm đó. Tạo một mảng Task để quản lý các hàm như sau.

```
1  Task task_list[MAX_TASKS] = {
2      {read_temp_and_hum, true},
3      {read_time, true},
4      {send_uart, true},
5      {display_lcd, true},
6      {NULL, false}
7  };
```

function_handle() trong while được gọi liên tục để kiểm tra cờ và thực thi các hàm tương ứng trong mảng.

```

1 void function_handle(){
2     if(flag == 1){
3         for (int i = 0; i < MAX_TASKS; i++) {
4             if (task_list[i].active && task_list[i].task_fn
5                 != NULL) {
6                 task_list[i].task_fn();
7             }
8         }
9     }
10 }
```

4.3.2 Non-Preemptive Event-Triggered Scheduling

4.3.2.1 Khái niệm

Non-Preemptive Event-Triggered Scheduling là 1 bộ lập lịch dựa trên sự kiện (Event-Triggered Scheduling) tức sự kiện là người điều phối hoạt động của hệ thống, nơi lịch trình các nhiệm vụ được xác định bởi sự xuất hiện của các sự kiện nội bộ hoặc ngoại vi.

4.3.2.2 Nguyên lý hoạt động

Mỗi sự kiện được gán một nhiệm vụ tương ứng, nhiệm vụ này sẽ được thực thi khi sự kiện xảy ra. Các sự kiện có thể được phát ra bởi:

- Ngắt bên ngoài (*external interrupts*).
- Nhiệm vụ nội bộ (*tasks themselves*).

Tất cả các sự kiện đều được lưu trữ trong hàng đợi chung (*event queue*). Dựa trên quy tắc xử lý hàng đợi (*queuing discipline*), sự kiện tiếp theo được chọn để thực thi. Các nhiệm vụ không thể bị gián đoạn trong khi đang thực thi.

Khi hàng đợi sự kiện trống, một nhiệm vụ nền (background task) có thể được thực thi. Nhiệm vụ này sẽ bị gián đoạn nếu có sự kiện mới. Sự kiện định thời (timed events) chỉ sẵn sàng thực thi sau khi một khoảng thời gian nhất định đã trôi qua. Điều này cho phép các sự kiện xảy ra định kỳ. Một số vấn đề :

- **Giao tiếp giữa các nhiệm vụ:**

Không xảy ra truy cập đồng thời vào tài nguyên chia sẻ, nhưng ngắt có thể gây ra vấn đề nếu làm gián đoạn nhiệm vụ đang thực thi.

- **Tràn bộ đệm:**

Có nguy cơ xảy ra nếu môi trường hoặc nhiệm vụ tạo ra quá nhiều sự kiện mà không được xử lý kịp thời.

- **Nhiệm vụ dài:**

Nhiệm vụ với thời gian thực thi dài có thể ngăn chặn các nhiệm vụ khác thực thi và gây tràn bộ đếm do sự kiện không được xử lý.

- **Phân chia nhiệm vụ:**

Các nhiệm vụ lớn nên được chia thành các nhiệm vụ nhỏ hơn (*subtasks*) để giảm thời gian thực thi.

Ngữ cảnh cục bộ phải được lưu lại trước khi chuyển đổi giữa các nhiệm vụ để đảm bảo tính toàn vẹn.

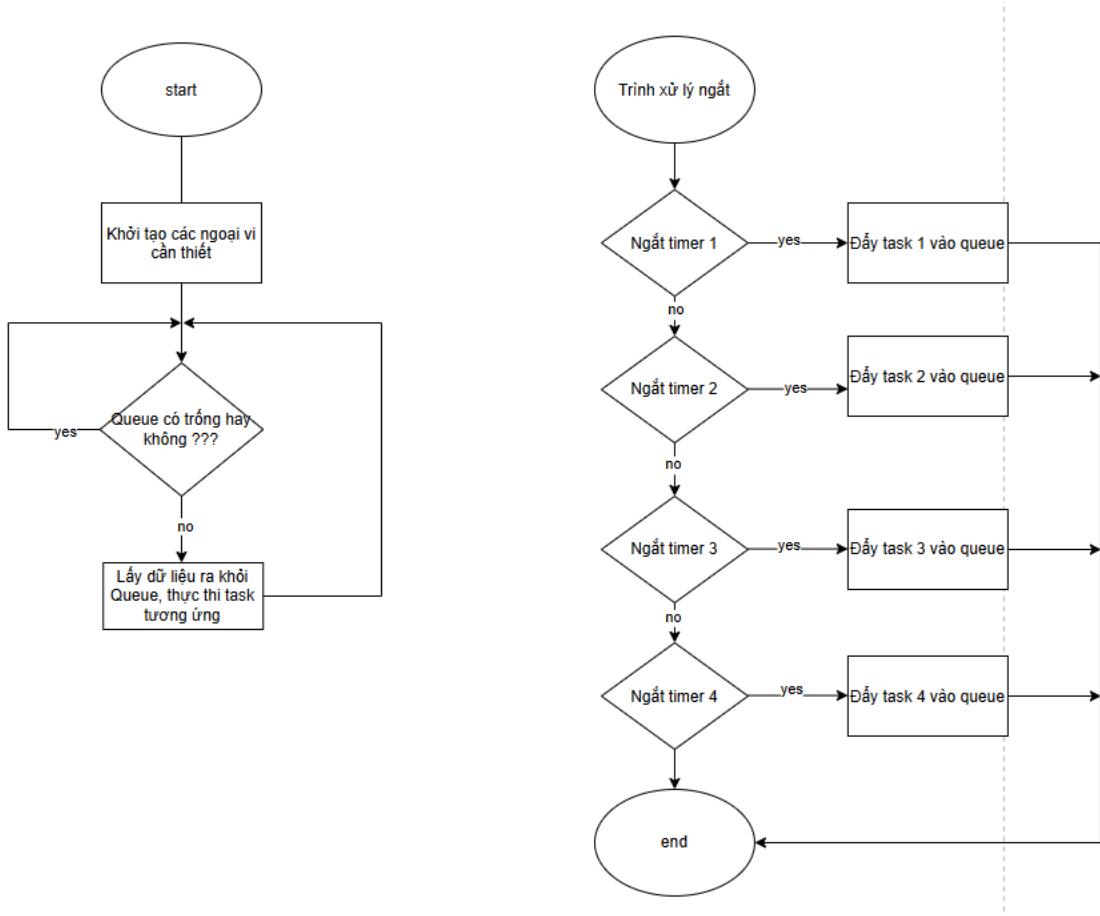
4.3.2.3 Phân tích tính dự án

Do có 4 task cần xử lý và các task này cần phải được xử lý kịp thời. Nên với mỗi task sẽ được cho vào hàm đợi nếu có ngắt từ timer với các timer tương ứng. Đây là sự kết hợp với việc sử dụng bộ lập lịch theo sự kiện định thời (Timed Event-Triggered Scheduling). Kết hợp các sự kiện định kỳ để đảm bảo nhiệm vụ được xử lý đúng lúc. Task nào vào queue trước sẽ được xử lý trước.

Việc thay đổi chu kỳ lập lịch sẽ dựa vào việc thay đổi chu kỳ của từng task - chính là chu kỳ ngắt của timer. Để thay đổi được chu kỳ của timer ta thay đổi 2 tham số đó chính là Period và Prescaler của cá timer tương ứng.

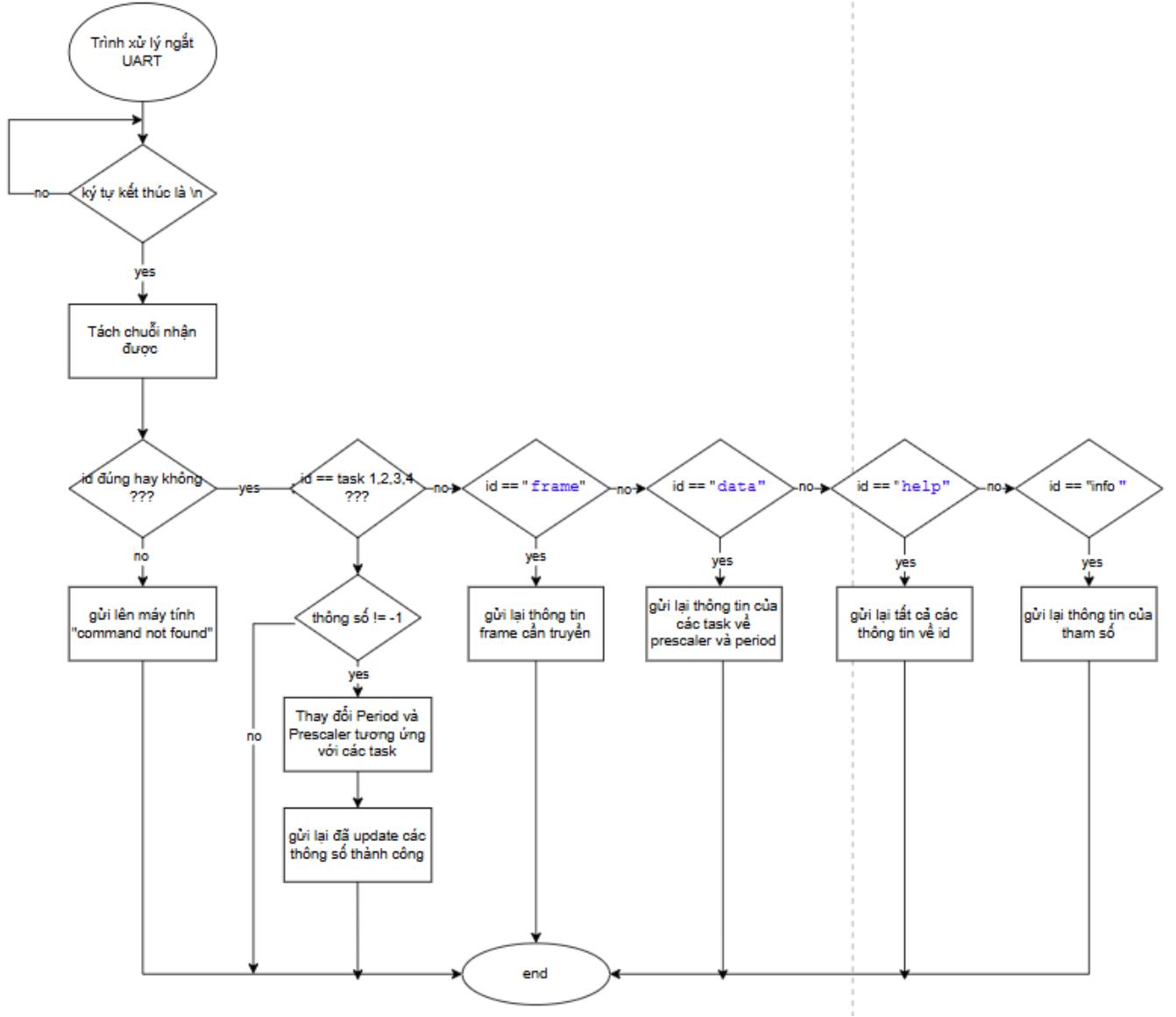
Prescaler (Bộ chia trước) là một tham số được sử dụng để chia tần số đầu vào của bộ Timer. Mục đích chính là giảm tần số đếm của Timer xuống mức dễ xử lý hơn, đặc biệt khi tần số xung nhịp của vi điều khiển (clock) rất cao. Period (Chu kỳ) xác định số lần Timer phải đếm trước khi tạo ra một sự kiện (ví dụ: ngắt). Nó quyết định khoảng thời gian thực tế của Timer dựa trên tần số đếm sau khi đã chia bởi Prescaler.

4.3.2.4 Lưu đồ thuật toán



Hình 4.5. Lưu đồ thuật toán mô hình Non-Preemptive Event-Triggered Scheduling

Trên đây là lưu đồ thuật toán của chương trình.Ban đầu khi chương trình được chạy sẽ khởi tạo các ngoại vi cần thiết và queue.Trong hàm while(1) sẽ luôn kiểm tra xem queue có trống hay không nếu trống tức là không có hàm nào cần được xử lý.Nếu queue không trống tức là có task cần phải được thực thi.Queue sẽ lấy hàm đó ra và xử lý. Với 4 task được yêu cầu mỗi task sẽ được đẩy vào queue nếu có ngắt timer tương ứng.



Hình 4.6. Lưu đồ thuật toán thay đổi chu kỳ lập lịch

Để thay đổi được chu kỳ lập lịch từ UART. Công việc này được viết dựa trên command Prompt trên máy tính. Định dạng bản tin truyền đi như sau :

ID	Period	Prescaler	<LF>
----	--------	-----------	------

Hình 4.7. Định dạng bản tin thay đổi chu kỳ lập lịch

- **ID:** Mã nhận diện của nhiệm vụ hoặc task. Đây là một định danh duy nhất để nhận biết task mà bạn muốn thay đổi chu kỳ hoặc hàm muốn gọi.
- **Period:** Giá trị chu kỳ, xác định số lần Timer sẽ đếm trước khi tạo ra một sự kiện (ngắt hoặc thông báo).
- **Prescaler:** Giá trị chia tần số, xác định số lần giảm tốc độ xung nhịp của Timer.

- <LF>: Ký tự xuống dòng (Line Feed), để báo hiệu 1 bản tin đã truyền xong.

Đối với các ngắt UART chính là ngắt có mức ưu tiên cao nhất.Khi có ngắt từ UART hàm xử lý sẽ liên tục kiểm tra xem ký tự đó có phải là ký tự xuống dòng <LF> hay không.Nếu đúng thì nó sẽ đi xử lý.

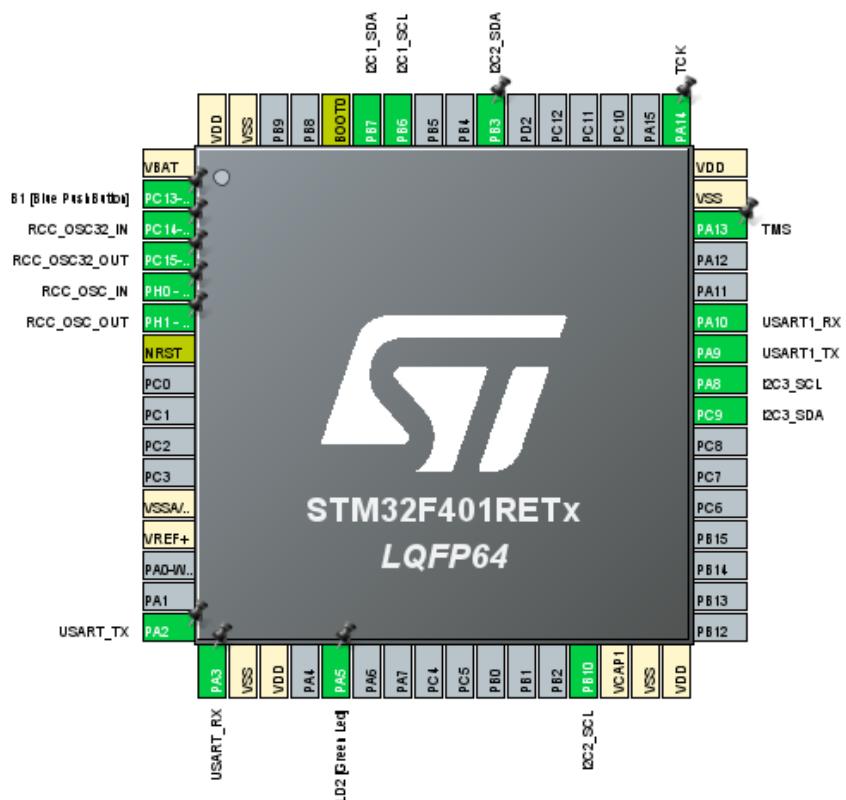
Bước xử lý sẽ tách chuỗi nhận được thành các phần tương ứng và kiểm tra. Nếu mã ID là đúng thì sẽ tiếp tục xử lý còn nếu sai thì sẽ gửi lại "command not found". Nếu id là các task 1 hoặc task 2 , task 3 ,task 4 thì lấy các giá trị đó để thay vào Prescale , Period của timer tương ứng.

Nếu là các hàm lấy thông tin như : frame , data , help , info . Thì sẽ gửi lại thông tin để người dùng xác nhận.

4.3.2.5 Triển khai lập trình

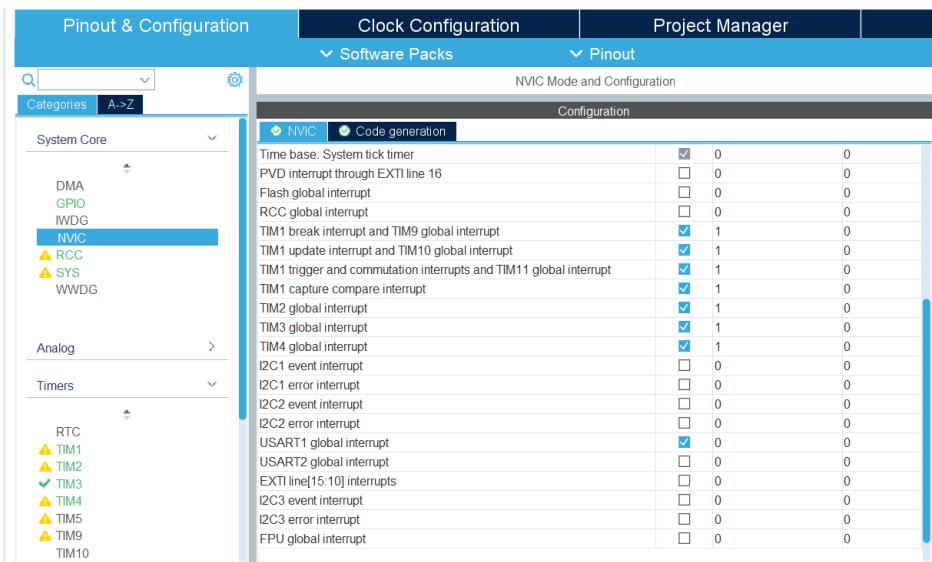
Cấu hình CubeMX

Cấu hình các ngoại vi cần thiết sử dụng như sau:



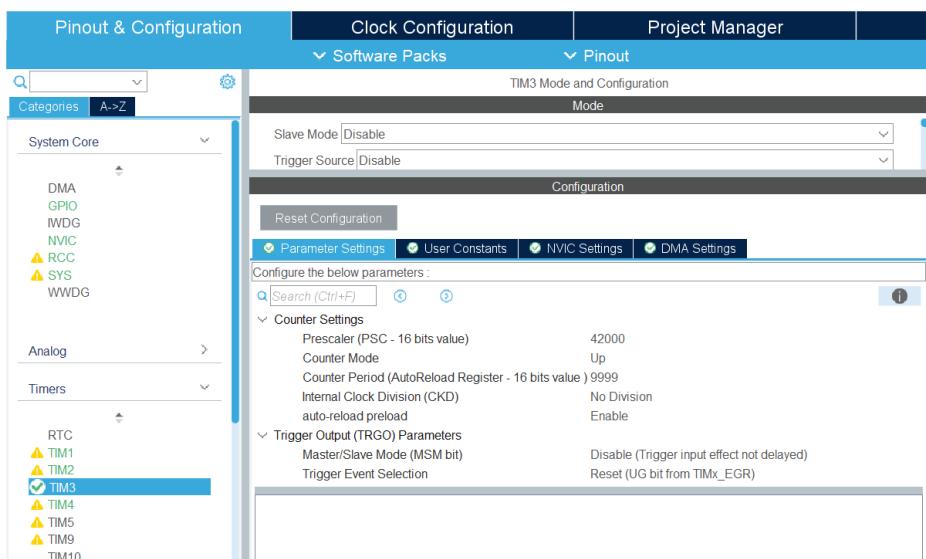
Hình 4.8. Cấu hình các ngoại vi cần thiết

Do UART có mức ưu tiên cao nhất nên ta đặt mức ưu tiên là 0. Với các timer ngắt còn lại đặt là 1.(Do trong STM32 giá trị ưu tiên càng thấp thì mức ưu tiên càng cao)



Hình 4.9. Cấu hình các mức ưu tiên ngắt

Tương tự như các timer ở trên ta áp dụng công thức và tính được các giá trị tương ứng để nạp vào Prescaler và Period



Hình 4.10. Cấu hình timer

Các hàm xử lý Queue

Queue hay hàng đợi là một cấu trúc dữ liệu phổ biến được sử dụng nhiều trong khoa học máy tính và lập trình. Queues đôi lập với Stack, các phần tử của hàng đợi chỉ được loại bỏ từ đầu hàng(head). Các phần tử hàng đợi chỉ được thêm vào cuối hàng (tail). Vì lý do này, hàng đợi được gọi là cấu trúc dữ liệu "vào trước ra trước - FIFO (First In, First Out)" Thêm sự kiện vào hàng đợi enqueue_event

```

1 bool enqueue_event(EventQueue *queue, EventType event) {
2     if (queue->count < MAX_EVENTS) {
3         queue->events[queue->end] = event;

```

```

4         queue->end = (queue->end + 1) % MAX_EVENTS;
5         queue->count++;
6         return true;
7     }
8     return false;
9 }
```

Đẩy 1 sự kiện ra khỏi hàng đợi dequeue_event

```

1 bool dequeue_event(EventQueue *queue, EventType *event) {
2     if (queue->count > 0) {
3         *event = queue->events[queue->begin];
4         queue->begin = (queue->begin + 1) % MAX_EVENTS;
5         queue->count--;
6         return true;
7     }
8     return false;
9 }
```

Trong hàng đợi với mảng tĩnh như trên, khi các phần tử bị xóa ở đầu (dequeue), vùng bộ nhớ tại các chỉ số ban đầu của mảng vẫn tồn tại nhưng không còn được sử dụng nữa, dẫn đến lãng phí bộ nhớ. Khi hàng đợi thông thường sử dụng mảng, khi rear đạt đến chỉ số cuối cùng của mảng (mặc dù phía trước mảng có thể còn trống do các phần tử đã bị xóa), hàng đợi vẫn được coi là đầy. Điều này gây lãng phí không gian. Tuy nhiên ưu điểm là cấu trúc dữ liệu đơn giản và dễ sử dụng. Quản lý chỉ số dễ dàng: Các thao tác thêm và xóa phần tử chỉ yêu cầu cập nhật chỉ số begin và end.

Quản lý các task

Để quản lý các task sử dụng 1 struct enum như sau :

```

1 typedef enum {
2     EVENT_TIMER1,
3     EVENT_TIMER2,
4     EVENT_TIMER3,
5     EVENT_TIMER4,
6     EVENT_NONE
7 } EventType;
```

Thêm task vào hàng đợi và xử lý

Mỗi khi có ngắt trình xử lý ngắt sẽ đẩy các event tương ứng vào ngắt:

```

1 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
2     //uint32_t currentTime;
3     if(htim == &htim1){
4         //currentTime = HAL_GetTick()/1000 ;
5         enqueue_event(&event_queue, EVENT_TIMER1);
```

```

6      //print_cli("task 1 vao hang doi -
7          %us\n", currentTime);
8  }
9  else if(htim == &htim2){
10     //currentTime = HAL_GetTick()/1000 ;
11     enqueue_event(&event_queue , EVENT_TIMER2);
12     //print_cli("task 2 vao hang doi-
13         %us\n", currentTime);
14 }
15 else if(htim == &htim3){
16     //currentTime = HAL_GetTick()/1000 ;
17     enqueue_event(&event_queue , EVENT_TIMER3);
18     //print_cli("task 3 vao hang doi-
19         %us\n", currentTime);
20 }
21 else if(htim == &htim4){
22     //currentTime = HAL_GetTick()/1000 ;
23     enqueue_event(&event_queue , EVENT_TIMER4);
24 }

```

Trong hàm while(1) luôn kiểm tra hàng đợi nếu nó không trống thì sẽ thực thi các hàm ở trong queue:

```

1 void handle_event(EventType event) {
2     float currentTime;
3     switch (event) {
4         case EVENT_TIMER1:
5             currentTime = HAL_GetTick()/1000.f ;
6             print_cli("The time run task 1 -
7                 read_temp_and_hum() : %.3fs\n" , currentTime );
8             read_temp_and_hum();
9             break;
10        case EVENT_TIMER2:
11            currentTime = HAL_GetTick()/1000.f ;
12            print_cli("The time run task 2 - read_time() :
13                %.3fs\n" , currentTime );
14            read_time();
15            break;
16        case EVENT_TIMER3:
17            currentTime = HAL_GetTick()/1000.f ;
18            print_cli("The time run task 3 - send_uart() :
19                %.3fs\n" , currentTime );
20            send_uart();
21            break;
22    }
23 }

```

```

19     case EVENT_TIMER4:
20         currentTime = HAL_GetTick()/1000.f ;
21         print_cli("The time run task 4 - display_lcd() :
22             %.3fs\n", currentTime );
23         display_lcd();
24         break;
25     default:
26         break;
27 }

```

Xử lý các hàm từ UART

Để xử lý các hàm từ UART tạo 1 struct chung mỗi khi kiểm tra id nếu id đúng thì nó sẽ chạy các hàm tương ứng.

```

1 cli_command_t list_command []={
2     {"task1",update_task1,"task1 : read temperature and
3         humidity"}, ,
4     {"task2",update_task2,"task2 : read DS3231"}, ,
5     {"task3",update_task3,"task3 : send time,data and
6         temperature,humidity to COM"}, ,
7     {"task4",update_task4,"task5 : send time,data and
8         temperature,humidity LCD"}, ,
9     {"frame",data_frame,"frame : give data of frame"}, ,
10    {"data",data_task,"data : give data of tasks"}, ,
11    {"help",help,"help : give data of all function"}, ,
12    {"info",info,"info : info of parameter"}, ,
13    {NULL , NULL , NULL}
14 };

```

4.4 Phân tích và lập trình cho yêu cầu BTL theo mô hình đa nhiệm

4.4.1 Mô hình đa nhiệm RMS

4.4.1.1 Định nghĩa.

Rate Monotonic Scheduling (RMS) là một thuật toán lập lịch ưu tiên tĩnh được sử dụng trong các hệ thống thời gian thực với các tác vụ định kỳ. RMS dựa trên giả định rằng các tác vụ có chu kỳ thực thi ngắn hơn (T) quan trọng hơn và cần được ưu tiên cao hơn trong quá trình thực thi.

4.4.1.2 Nguyên lý lập lịch của RMS.

Ưu tiên cố định dựa trên chu kỳ (T):

- RMS sử dụng ưu tiên cố định, trong đó mỗi tác vụ được gán mức ưu tiên dựa trên chu kỳ T của nó.

- Tác vụ có chu kỳ ngắn hơn sẽ được ưu tiên cao hơn vì cần hoàn thành thường xuyên hơn

$$\text{Priority} \propto \frac{1}{T} \quad (4.3)$$

Chu kỳ và deadline: RMS giả định rằng deadline của mỗi tác vụ bằng với chu kỳ của nó ($D = T$). Điều này có nghĩa là một tác vụ phải hoàn thành trước khi nó được phát hành lại.

Tiền nhiệm (Preemption): Khi một tác vụ có mức ưu tiên cao hơn trở nên sẵn sàng, nó có thể tạm dừng một tác vụ ưu tiên thấp hơn khi đang thực thi.

Khả năng lập lịch: RMS đảm bảo tất cả các tác vụ có thể được lập lịch đúng hạn nếu tổng thời gian sử dụng CPU không vượt quá giới hạn khả năng lập lịch:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1) \quad (4.4)$$

- C_i : Thời gian thực thi của tác vụ i .
- T_i : Chu kỳ thực thi của tác vụ i .
- n : Tổng số tác vụ.

Với:

- $n = 1$: $U \leq 100\%$.
- $n = 2$: $U \leq 83.3\%$.
- $n \rightarrow \infty$: $U \leq 69.3\%$.

4.4.1.3 Đặc điểm của mô hình RMS.

Ưu điểm:

- **Đơn giản:** RMS sử dụng ưu tiên cố định, không cần thay đổi trong suốt quá trình thực thi.
- **Hiệu quả:** RMS tối ưu trong trường hợp deadline bằng chu kỳ ($D = T$).
- **Phổ biến:** Được sử dụng rộng rãi trong các hệ thống nhúng và thời gian thực.

Hạn chế:

- **Không tối ưu trong mọi trường hợp:** RMS không hiệu quả nếu deadline nhỏ hơn chu kỳ ($D < T$).
- **Giới hạn khả năng lập lịch:** RMS không đảm bảo lập lịch thành công nếu tổng thời gian sử dụng CPU vượt quá giới hạn khả năng lập lịch.

- **Yêu cầu tiền nhiệm:** RMS cần cơ chế hỗ trợ tiền nhiệm để đảm bảo các tác vụ ưu tiên cao không bị chặn.

4.4.1.4 Quy trình hoạt động.

Gán mức ưu tiên:

- Xác định chu kỳ T_i của từng tác vụ.
- Gán mức ưu tiên cố định, tác vụ có chu kỳ ngắn nhất được ưu tiên cao nhất.

Kiểm tra khả năng lập lịch:

- Tính tổng thời gian sử dụng CPU (U):

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} \quad (4.5)$$

- Kiểm tra điều kiện lập lịch:

$$U \leq n \cdot (2^{1/n} - 1) \quad (4.6)$$

Nếu điều kiện này không được thỏa mãn, lập lịch có nguy cơ thất bại.

Thực thi tác vụ:

- Các tác vụ được thực thi theo mức ưu tiên.
- Tác vụ có mức ưu tiên cao hơn được tiền nhiệm nếu cần.

Task	Execution Time (C)	Period (T)	Deadline (D)
Đọc dữ liệu nhiệt độ	60 ms	2250 ms	100 ms
Đọc dữ liệu thời gian thực	20 ms	900 ms	200 ms
Hiển thị dữ liệu nhiệt độ	200 ms	3000 ms	300 ms
Hiển thị thời gian thực	250 ms	1000 ms	400 ms
Đọc dữ liệu độ ẩm	30 ms	1800 ms	500 ms
Hiển thị dữ liệu độ ẩm	250 ms	4500 ms	600 ms

Bảng 4.2. Thông số của các task.

4.4.1.5 Phân tích các thông số task

4.4.1.6 Thiết kế thuật toán RMS

- Chu kỳ tổng $P = 9s$. Là bội chung của tất cả chu kỳ

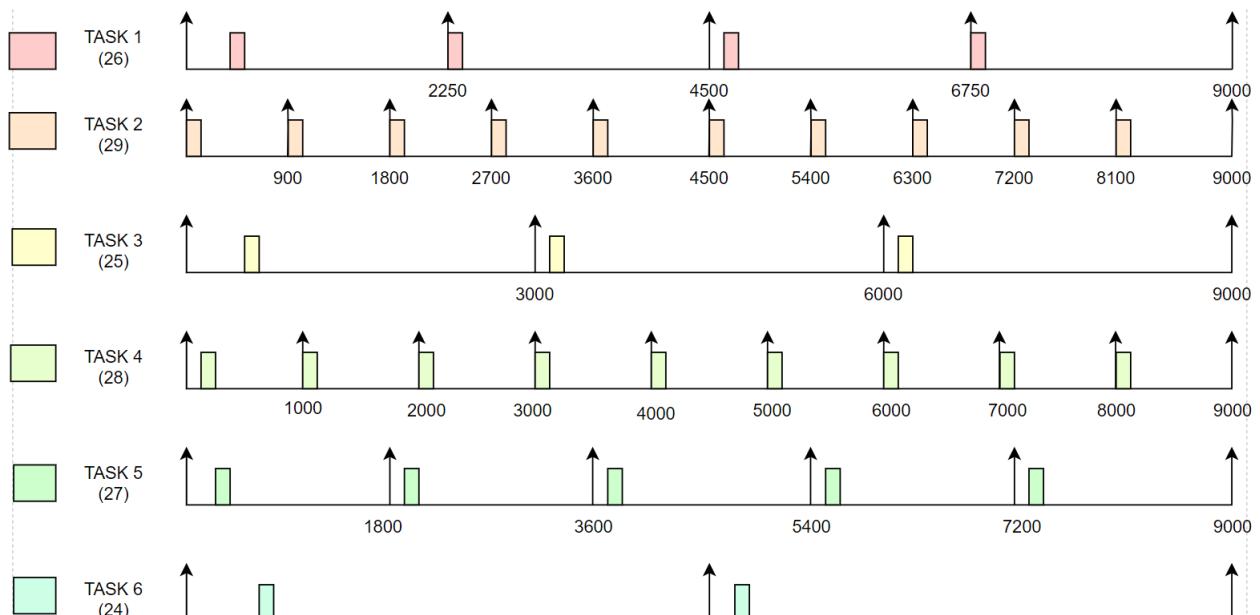
$$P = \text{LCM}(T_1, T_2, T_3, T_4, T_5, T_6) \quad (4.7)$$

- **Ưu tiên:** Gán mức ưu tiên cho mỗi task sao cho T nhỏ nhất được ưu tiên cao nhất.
- Trong CMSIS V2, giá trị cao hơn đại diện cho mức ưu tiên cao hơn.

Từ bảng các thông số task ở trên, ta có thể gán các mức ưu tiên cho task như sau:

Task	Chu kỳ (T) [ms]	C [ms]	Priority
Đọc dữ liệu thời gian thực	900	60	29
Hiển thị thời gian thực	1000	20	28
Đọc dữ liệu độ ẩm	1800	200	27
Đọc dữ liệu nhiệt độ	2250	250	26
Hiển thị dữ liệu nhiệt độ	3000	30	25
Hiển thị dữ liệu độ ẩm	4500	250	24

Bảng 4.3. Bảng các tác vụ với chu kỳ, giá trị C và mức độ ưu tiên.



Hình 4.11. Thiết kế bộ lập lịch RMS dựa vào thông số task

4.4.1.7 Kiểm tra khả năng lập lịch.

Tính tổng sử dụng CPU (U) để kiểm tra xem hệ thống có thể lập lịch đúng hạn hay không:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (4.8)$$

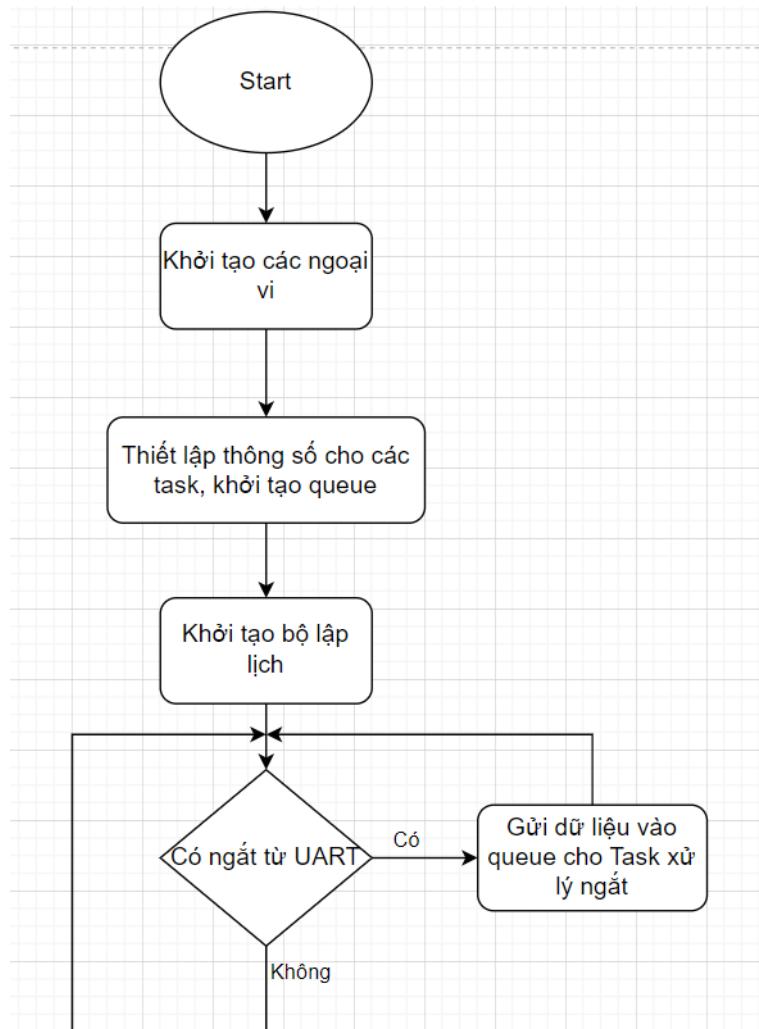
Trong ví dụ:

$$U = \frac{60}{2250} + \frac{20}{900} + \frac{200}{3000} + \frac{250}{1000} + \frac{30}{1800} + \frac{250}{4500} = 0.42 \quad (4.9)$$

Vì $U < 69.3\%$ (giới hạn cho 6 task), lập lịch RMS khả thi.

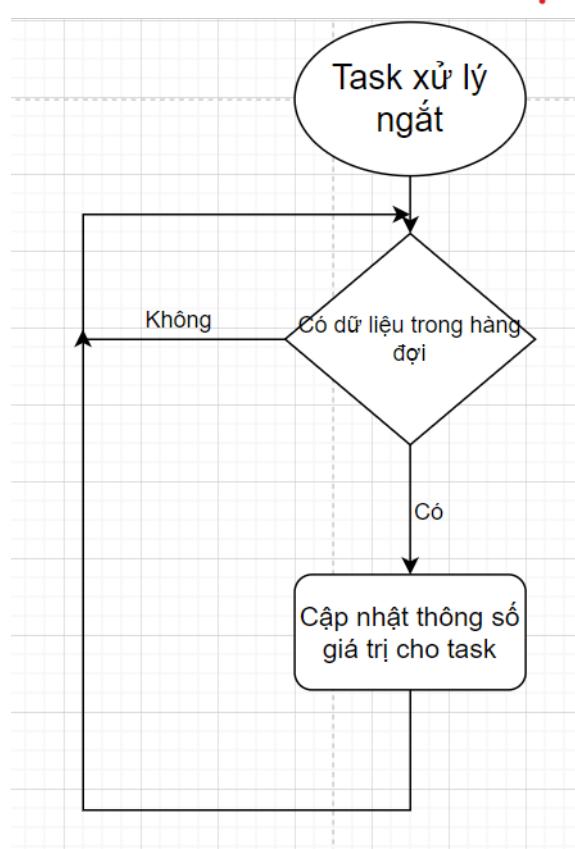
4.4.2 Lập trình các task hoạt động theo mô hình RMS

4.4.2.1 Lưu đồ thuật toán hoạt động.



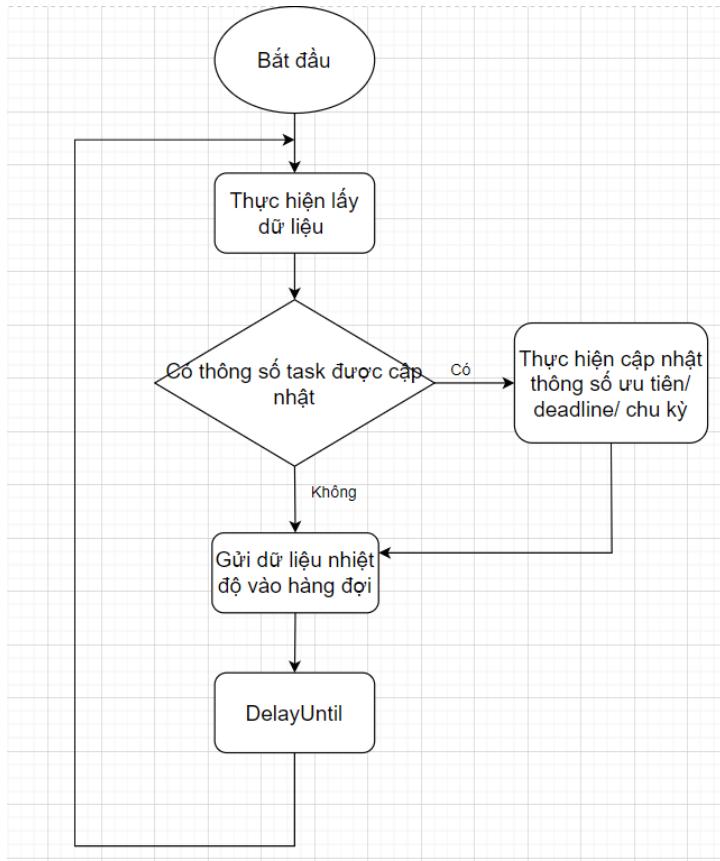
Hình 4.12. Lưu đồ thuật toán hoạt động mô hình RMS

Giải thích: Bước đầu tiên, vi điều khiển sẽ tiến hành khởi tạo các ngoại vi cần thiết, sau đó là bước thiết lập các thông số cho task và khởi tạo queue. Vì là lập lịch theo mô hình RMS nên task nào có chu kỳ thấp nhất là task sẽ có mức ưu tiên cao nhất. Thông số chu kỳ task đã được biết, do đó có thể tiến hành thiết lập mức ưu tiên ban đầu cho tất cả các task theo thứ tự chu kỳ tăng dần.



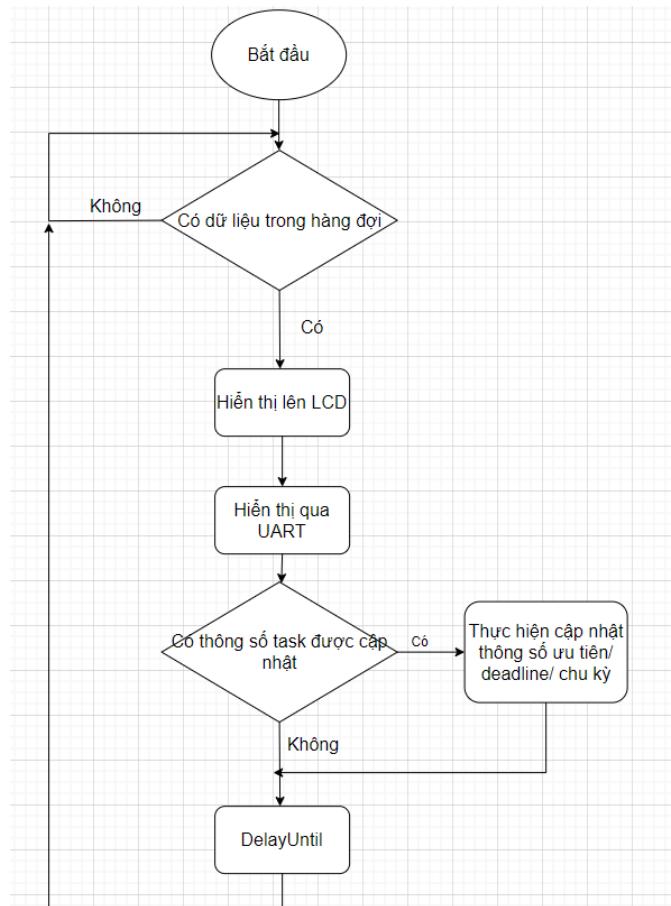
Hình 4.13. Hoạt động của task xử lý ngắt trì hoãn

Giải thích: Ngắt nhận UART nhận biết dữ liệu đã được gửi hoàn toàn bằng cách nhận biết ký tự kết thúc chuỗi trong chuỗi nhận được. Chuỗi nhận được sau đó được gửi vào một hàm đợi để cho task trì hoãn ngắt xử lý. Task này luôn ở mức ưu tiên cao nhất, hơn tất cả những task còn lại. Sau khi phân tách ra, nếu thấy thông số của task được yêu cầu nhận được khác với thông số của task sẵn có thì tiến hành cập nhật thông số cho task.



Hình 4.14. Hoạt động của task đo

Giải thích: Luồng hoạt động của task lấy dữ liệu từ cảm biến đo nhiệt độ, cảm biến độ ẩm và đồng hồ thời gian thực là tương tự nhau. Đầu tiên, khi task được kích hoạt, task đọc dữ liệu từ cảm biến sau đó kiểm tra xem có thông số nào của task được cập nhật từ máy tính không, nếu có task tiến hành cập nhật thông số task, nếu không hoặc đã cập nhật xong thì task tiến hành gửi dữ liệu vào hàm đợi cho task hiển thị xử lý, rồi chờ đến chu kỳ tiếp theo.

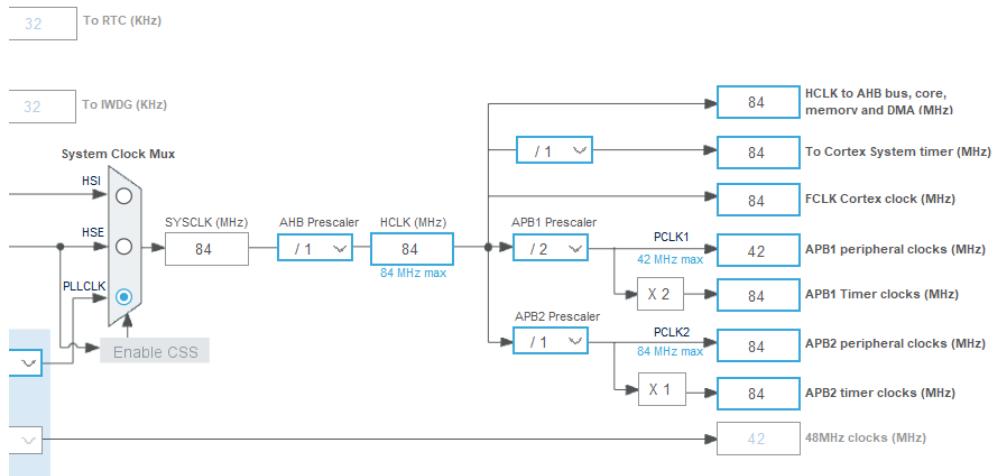


Hình 4.15. Hoạt động của task hiển thị, truyền thông số qua LCD, UART

Giải thích: Luồng hoạt động của task hiển thị nhiệt độ, độ ẩm và thời gian thực lên LCD và UART là tương tự nhau. Task sẽ luôn ở trạng thái block cho đến khi có dữ liệu từ cảm biến bên trong hàng đợi do task đó gửi đến. Nếu có dữ liệu từ hàng đợi, đồng thời task được kích hoạt. Tiến hành gửi dữ liệu lên LCD, sau đó gửi dữ liệu qua UART lên máy tính. Sau đó sẽ kiểm tra xem có thông số nào của task được cập nhật không. Nếu có thì tiến hành cập nhật, còn nếu không hoặc đã cập nhật xong thì đợi đến chu kỳ hoạt động tiếp theo.

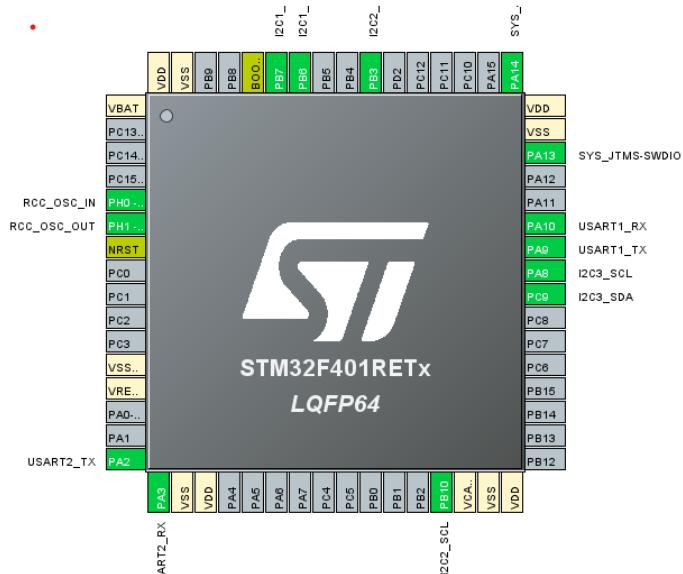
4.4.2.2 Cấu hình thiết bị trên phần mềm STM32CUBEIDE.

- Clock và Debug:** Đầu tiên, để tiến hành cấu hình đồng hồ xung nhịp cho chip bằng cách vào mục System Core phần RCC, ta chọn chế độ sử dụng thạch anh ngoài High Speed Clock. Trong mục Clock Configuration, ta đặt tốc độ HCLK ở mức tối đa (84MHz)



Hình 4.16. Cấu hình xung clock cho module vi điều khiển

- Cấu hình các ngoại vi



Hình 4.17. Cấu hình các ngoại vi để giao tiếp với phần cứng.

Các ngoại vi sử dụng bao gồm:

- I2C1, I2C2, I2C3: để giao tiếp lần lượt với các phần cứng SHT21, LCD, DS3231.
 - UART1 : Giao tiếp với máy tính, nhận cập nhật thông số từ máy tính, kích hoạt ngắt UART1
 - CMSISV2: Lập trình sử dụng 7 task với 6 task như bảng 4.3, task 07 có mức ưu tiên cao nhất so với những task còn lại nhằm mục đích trì hoãn xử lý ngắt

4.4.2.3 Triển khai lập trình

- Cấu trúc mảng để lưu thông số của tất cả 6 task. Trong đó biến changed được dùng để phát hiện khi nào task nhận được yêu cầu thay đổi thông số từ máy tính, biến sẽ được đặt là True

```
1 typedef struct{
2     int id;
3     int period;
4     int execTime;
5     int deadline;
6     int priority;
7     int allPeriod;
8     float wcet;
9     bool changed;
10 } Task;
11 Task tasks[6] = {
12     {0, 2250, 50, 100, 5, 10, 0, false},
13     {1, 900, 4, 200, 11, 10, 0, false},
14     {2, 3000, 4, 300, 17, 10, 0, false},
15     {3, 1000, 4, 400, 23, 10, 0, false},
16     {4, 1800, 50, 500, 29, 10, 0, false},
17     {5, 4500, 4, 600, 35, 10, 0, false}
18 };
```

- Cập nhật ưu tiên các task khi có yêu cầu thay đổi chu kỳ. Sau khi phát hiện có sự thay đổi chu kỳ của một task nào đó, hàm này sẽ được gọi để đặt lại giá trị ưu tiên của các task dựa vào chu kỳ mới nhất của các task

```
1 void update_task_priorities(Task *tasks, int num_tasks)
2 {
3     int periods[num_tasks];
4     for (int i = 0; i < num_tasks; i++)
5         periods[i] = tasks[i].period;
6
7     for (int i = 0; i < num_tasks; i++)
8     {
9         int priority = 0;
10
11         for (int j = 0; j < num_tasks; j++)
12         {
13             if (tasks[i].period < tasks[j].period)
14                 priority++;
15         }
16         tasks[i].priority = 29 - priority; // Muc uu tien
17         // tu 29 den 24
18     }}
```

- Hàm phát hiện có thông số của task thay đổi

```

1 bool detectChange(Task *current, Task *previous){
2     if (current->period != previous->period ||
3         current->execTime != previous->execTime ||
4         current->deadline != previous->deadline ||
5         current->priotity != previous->priotity ||
6         current->allPeriod != previous->allPeriod) {
7         current->changed = true;
8         return true;
9     }
10    current->changed = false;
11    return false;
12 }
```

- Task lấy dữ liệu từ cảm biến đo nhiệt độ. Task lấy dữ liệu từ đồng hồ thời gian thực và cảm biến đo độ ẩm có luồng hoạt động tương tự

```

1 void StartDefaultTask(void *argument)
2 {
3     //The parameters of task
4     TickType_t lastWakeTime = osKernelGetTickCount();
5     taskReadTempDeadline = lastWakeTime +
6                             pdMS_TO_TICKS(tasks[0].deadline);
7
8     float temp;
9     float timeInSeconds;
10    for(;;)
11    {
12        timeInSeconds = (float)osKernelGetTickCount() /
13                      configTICK_RATE_HZ;
14
15        print_cli("The start time of get temperature task:
16                  %.3f\n", timeInSeconds);
17        temp = SHT2x_GetTemperature(1);
18        //Check deadline
19        if (osKernelGetTickCount() > taskReadTempDeadline)
20        {
21            print_cli("Task Read Temperature missed
22                      deadline\n");
23        }
24        if (detectChange(&tasks[0], &prev_tasks[0]))
25        {
26            if(tasks[0].deadline != prev_tasks[0].deadline)
27            {
28                taskReadTempDeadline +=
29                    pdMS_TO_TICKS(tasks[0].period)
30            }
31        }
32    }
33 }
```

```

25
26             pdMS_TO_TICKS(prev_tasks[0].deadline)
27             + pdMS_TO_TICKS(tasks[0].deadline));
28         print_cli("TASK 01 WAS CHANGED DEADLINE\n");
29     }
30     if(tasks[0].priority != prev_tasks[0].priority)
31     {
32         osThreadSetPriority(MeasureTempHandle,
33             tasks[0].priority);
34         print_cli("TASK 01 WAS CHANGED PRIORITY\n");
35     }
36     updatePrevious(&prev_tasks[0], &tasks[0]);
37 }else
38 {
39     taskReadTempDeadline +=
40         pdMS_TO_TICKS(tasks[0].period);
41 }
42
43 timeInSeconds = (float)osKernelGetTickCount() /
44 configTICK_RATE_HZ;
print_cli("The end time of get temperature task:
    %.3f\n", timeInSeconds);
osMessageQueuePut(myQueue01Handle, &temp, 0U, 0U);
vTaskDelayUntil(&lastWakeTime,
    pdMS_TO_TICKS(tasks[0].period));
}
}

```

- Task hiển thị lên LCD và UART

```

1 void StartTask02(void *argument)
2 {
3     TickType_t lastWakeTime = osKernelGetTickCount();
4     taskReadTimeDeadline = lastWakeTime +
5         pdMS_TO_TICKS(tasks[1].deadline);
6     DateTime dateTimeSend;
7     float timeInSeconds;
8     for(;;)
9     {
10         timeInSeconds = (float)osKernelGetTickCount() /
11             configTICK_RATE_HZ;
12         print_cli("The start time of get real time task:
13             %.3f\n", timeInSeconds);
14         rtc_read_time(&dateTimeSend);
15
16         //Check Deadline
17         if (osKernelGetTickCount() > taskReadTimeDeadline)
18             pdMS_TO_TICKS(tasks[1].deadline));
19     }
20 }

```

```

15     {
16         print_cli("Task Read Time missed deadline\n");
17     }
18     if (detectChange(&tasks[1], &prev_tasks[1]))
19     {
20         // C p   n h t   g i   t r   t r   c
21         if(tasks[1].deadline != tasks[1].deadline)
22         {
23             taskReadTimeDeadline +=
24                 pdMS_TO_TICKS(tasks[1].period)
25             - pdMS_TO_TICKS(prev_tasks[1].deadline) +
26                 pdMS_TO_TICKS(tasks[1].deadline);
27             print_cli("TASK 2 WAS CHANGED DEADLINE\n");
28         }
29         if(tasks[1].priority != tasks[1].priority)
30         {
31             osThreadSetPriority(RealtimeHandle,
32                                 tasks[1].priority);
33             print_cli("TASK 2 WAS CHANGED PRIORITY\n");
34         }
35         updatePrevious(&prev_tasks[1], &tasks[1]);
36     }else
37     {
38         taskReadTimeDeadline +=
39             pdMS_TO_TICKS(tasks[1].period);
40     }
41     timeInSeconds = (float)osKernelGetTickCount() /
42         configTICK_RATE_HZ;
43     print_cli("The end time of get real time task:
44         %.3f\n", timeInSeconds);
45     osMessageQueuePut(myQueue02Handle, &dateTimeSend, 0U,
46                       0U);
47     //Delay until next period
48     vTaskDelayUntil(&lastWakeTime,
49                     pdMS_TO_TICKS(pdMS_TO_TICKS(tasks[1].period)));
50 }

```

- Task xử lý trì hoãn ngắt ngắt

```

1 void StartTask07(void *argument)
2 {
3     uint8_t Recv_ISR[20];
4     TaskData taskData;
5     int size;
6     char m;
7     int fieldRecv[12] = {0};

```

```

8     int field[6] = {0};
9     int idx;
10    for(;;)
11    {
12        osMessageQueueGet(myQueue04Handle, &Recv_ISR, NULL,
13                           osWaitForever);
14        print_cli("Nhan tu ngat yeu cau thay doi thong so lap
15                  lich\n");
16        for (int i = 0; i < 12; i++) {
17            fieldRecv[i] = Recv_ISR[i] - '0';
18        }
19        for (int j = 0; j < 6; j++) {
20            field[j] = fieldRecv[2 * j] * 10 +
21                        fieldRecv[2 * j + 1];
22        }
23        idx = field[0];
24        if(field[1] != 0 && tasks[idx].period != field[1]
25           *100)
26        {
27            tasks[idx].period = field[1] * 100;
28            update_task_priorities(tasks, 6);
29            print_cli("TASK %d WAS CHANGED PERIOD\n",idx + 1);
30        }
31        if(field[2] != 0)
32        {
33            print_cli("Thay doi thoi gian thuc thi\n");
34        }
35        if(field[3] != 0 && tasks[idx].deadline != field[3] *
36           100)
37        {
38            print_cli("TASK %d WAS CHANGED DEADLINE\n", idx +
39                      1);
40            tasks[idx].deadline = field[3] * 100;
41        }
42        if(field[4] != 0 && tasks[idx].allPeriod != field[4]
43           * 100)
44        {
45            print_cli("Thay doi chu ky toan bo\n");
46            for (int i = 0; i <= 5; i++)
47                tasks[i].allPeriod = field[4] * 100;
48        }if(field[5] != 0 && field[5] != tasks[idx].priority)
49        {
50            print_cli("Muc uu tien phu thuoc vao deadline\n");
51            tasks[idx].priority = field[5];
52        }
53        for (int i = 0; i <= 5; i++)
54            field[i] = 0;

```

```

48     for (int i = 0; i <= 11; i++)
49         fieldRecv[i] = 0;
50     osThreadYield();
51 }
52 }
```

4.4.3 Mô hình đa nhiệm EDF

4.4.3.1 Định nghĩa.

Earliest Deadline First (EDF) là một thuật toán lập lịch ưu tiên động cho hệ thống thời gian thực. Trong EDF, mỗi tác vụ sẽ được ưu tiên dựa trên thời hạn hoàn thành (*deadline*) của nó. Tác vụ có deadline gần nhất sẽ được ưu tiên cao hơn và được thực thi trước.

4.4.3.2 Nguyên lý lập lịch của EDF

- **Xác định deadline cho mỗi tác vụ:** Mỗi tác vụ τ_i được định nghĩa bởi:
 - T_i : Chu kỳ thực hiện của tác vụ.
 - C_i : Thời gian thực thi của tác vụ.
 - D_i : Thời hạn hoàn thành của tác vụ.
- **Ưu tiên theo deadline:** Tác vụ nào có deadline sớm nhất tại bất kỳ thời điểm nào sẽ được ưu tiên thực thi.
- **Kiểm tra khả năng lập lịch:** Hệ thống được gọi là khả thi nếu tổng mức sử dụng CPU U không vượt quá 100%:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (4.10)$$

với n là số lượng tác vụ trong hệ thống.

- **Tái lập lịch khi có thay đổi:** Khi một tác vụ mới được thêm vào hàng đợi hoặc khi một tác vụ kết thúc, thuật toán EDF sẽ tái đánh giá để xác định tác vụ tiếp theo được thực thi.

4.4.3.3 Ưu nhược điểm của mô hình EDF.

Ưu điểm:

- **Tối ưu hóa sử dụng CPU:** EDF là thuật toán lập lịch tối ưu cho hệ thống thời gian thực mềm, vì nó có thể tận dụng tài nguyên CPU tối đa nếu $U \leq 1$.
- **Hỗ trợ linh hoạt:** EDF phù hợp với cả hệ thống thời gian thực cứng và mềm, đồng thời dễ dàng thích ứng với các thay đổi trong hệ thống.

Nhược điểm của EDF:

- Chi phí xử lý cao:** Do cần phải sắp xếp lại các tác vụ theo thời gian thực, chi phí tính toán của EDF cao hơn so với các thuật toán ưu tiên tĩnh như RMS.
- Khả năng bị quá tải:** Trong trường hợp $U > 1$, hệ thống không đảm bảo đáp ứng được tất cả các deadline, dẫn đến tình trạng trễ hạn.

4.4.3.4 Phân tích các thông số task Như đã phân tích thông số các task ở trên phần lập trình đa nhiệm sử dụng RMS (bảng 4.2), với 6 task đã đưa ra, tổng mức sử dụng chỉ khoảng 42 %, do đó vẫn thỏa mãn để thực hiện các task theo mô hình đa nhiệm EDF

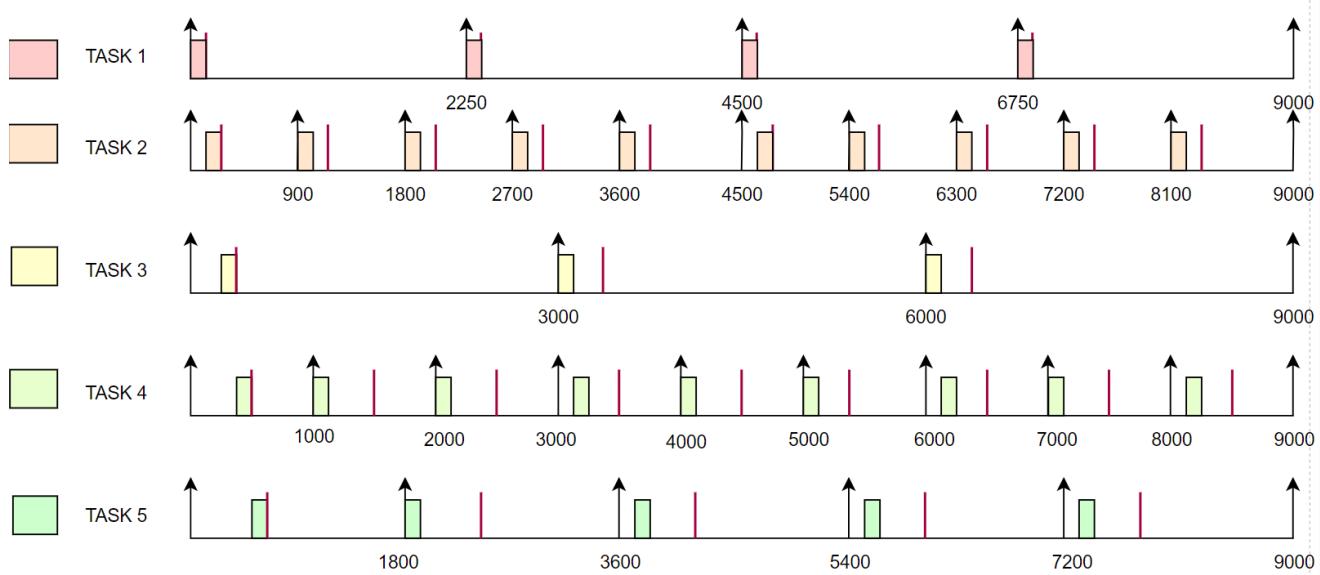
4.4.3.5 Thiết kế thuật toán EDF Như ở phần phân tích thông số các task ở trên (bảng 4.2), task có deadline thấp nhất và gần nhất thời điểm hiện tại sẽ là task có ưu tiên cao nhất.

- Ưu tiên:** Gán mức ưu tiên cho mỗi task sao cho D nhỏ nhất được ưu tiên cao nhất.
- Trong CMSIS V2, giá trị cao hơn đại diện cho mức ưu tiên cao hơn.

Từ bảng các thông số task ở trên, ta có thể gán các mức ưu tiên cho task như sau:

Task	Deadline (ban đầu)	Priority	Chu kỳ (ms)	C (ms)
Đọc dữ liệu nhiệt độ	100	29	2250	60
Đọc dữ liệu thời gian	200	28	900	20
Hiển thị dữ liệu nhiệt độ	300	27	3000	200
Hiển thị thời gian thực	400	26	1000	250
Đọc dữ liệu độ ẩm	500	25	1800	30
Hiển thị dữ liệu độ ẩm	600	24	4500	250

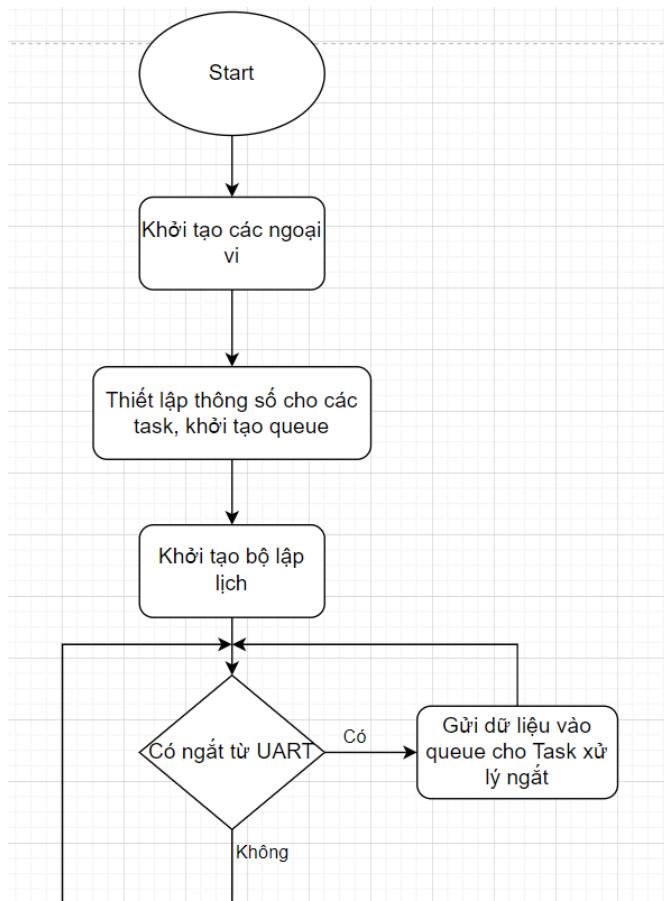
Bảng 4.4. Bảng các tác vụ với deadline, mức độ ưu tiên, chu kỳ và giá trị C.



Hình 4.18. Thiết kế thuật toán EDF

4.4.4 Lập trình các task hoạt động theo mô hình EDF

4.4.4.1 Lưu đồ thuật toán hoạt động



Hình 4.19. Lưu đồ thuật toán hoạt động mô hình EDF

Giải thích: Đầu tiên, module vi điều khiển tiến hành khởi tạo các ngoại vi, thiết lập các thông số cho task. Vì đã biết thông số của các task từ trước nên ta có thể dễ

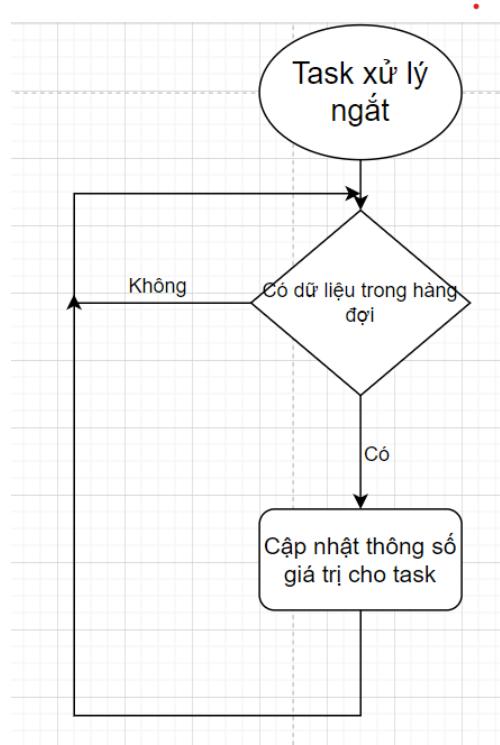
dòng xác định được mức độ ưu tiên của từng task theo chu kỳ với điều kiện task có chu kỳ càng nhỏ thì mức ưu tiên càng lớn. Sau đó sẽ tùy theo điều khiển từ máy tính xuông để thay đổi thông số lập lịch. Sau khi đã khởi tạo thông số cho các task thì tiến hành khởi tạo bộ lập lịch, chương trình bắt đầu hoạt động. Task có chu kỳ thấp nhất sẽ hoạt động đầu tiên và chương trình cũng bắt đầu chờ đợi ngắt thay đổi thông số các task từ máy tính.

Dữ liệu thay đổi thông số lập lịch các task được gửi từ máy tính xuông có cấu trúc như sau:

Task number	Chu kỳ	Thời gian thực thi	Deadline	Chu kỳ toàn bộ	Mức ưu tiên
-------------	--------	--------------------	----------	----------------	-------------

Hình 4.20. Cấu trúc dữ liệu gửi xuông từ máy tính

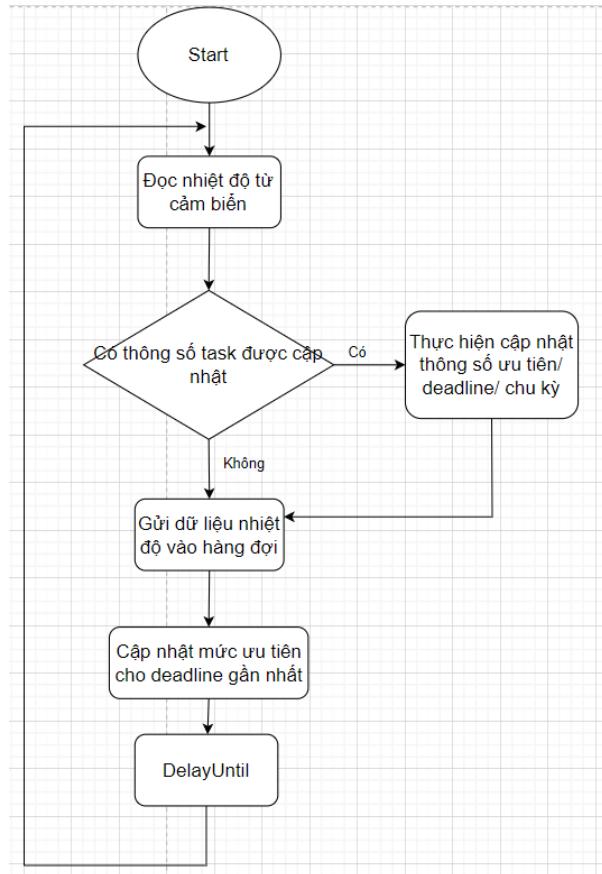
Giải thích: Dữ liệu gửi là 1 chuỗi gồm chứa các ký tự là các chữ số và kết thúc bằng ký tự <LF> để xác định toàn bộ chuỗi đã nhận được thành công. Gồm 12 ký tự chữ số như sau "041000080020<LF>". Chuỗi trên sau đó được tách để lấy ra 6 giá trị int tương ứng với 6 trường ở hình trên.



Hình 4.21. Lưu đồ thuật toán xử lý ngắt

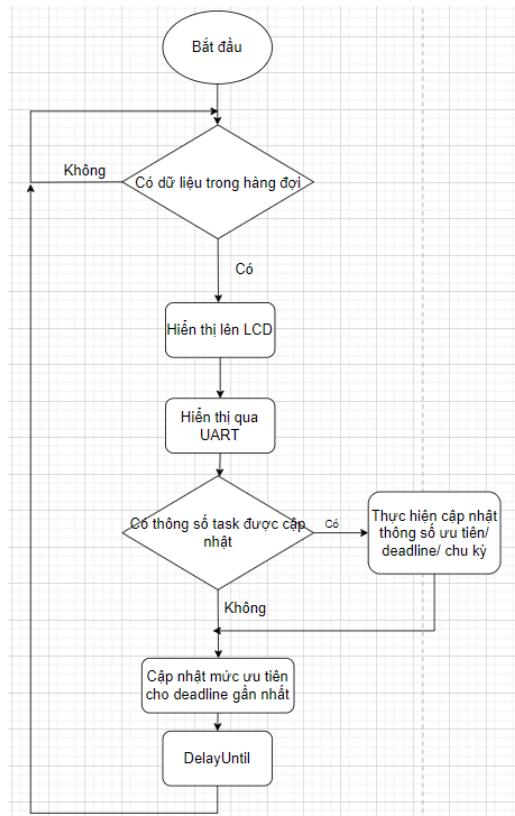
Giải thích: Ngắt nhận UART nhận biết dữ liệu đã được gửi hoàn toàn bằng cách nhận biết ký tự kết thúc chuỗi trong chuỗi nhận được. Chuỗi nhận được sau đó được

gửi vào một hàm đợi để cho task trì hoãn ngắt xử lý. Task này luôn ở mức ưu tiên cao nhất, hơn tất cả những task còn lại. Sau khi phân tách ra, nếu thấy thông số của task được yêu cầu nhận được khác với thông số của task sẵn có thì tiến hành cập nhật thông số cho task.



Hình 4.22. Lưu đồ thuật toán hoạt động task đọc dữ liệu từ cảm biến

Giải thích: Luồng hoạt động của task đọc dữ liệu từ cảm biến nhiệt độ, đồng hồ thời gian thực và cảm biến độ ẩm là tương tự nhau. Đầu tiên sẽ đọc dữ liệu từ cảm biến. Sau đó kiểm tra xem thông số task có được cập nhật từ máy tính qua UART không. Nếu có thì thực hiện dùng các API của CMSIS V2 để cập nhật thông số cho task như chu kỳ, thứ tự ưu tiên và thay đổi các giá trị liên quan trực tiếp trong task. Sau khi cập nhật xong, hoặc phát hiện không có thông số nào của task được cập nhật thì tiến hành gửi dữ liệu đọc được từ cảm biến vào hàng đợi. Sau đó cập nhật mức độ ưu tiên cao nhất (nhưng vẫn phải thấp hơn task trì hoãn). Rồi đợi cho lần kích hoạt chu kỳ tới.



Hình 4.23. Lưu đồ thuật toán hoạt động task hiển thị qua LCD và truyền qua UART

Giải thích: Task hiển thị dữ liệu qua LCD và UART sẽ luôn chờ hàng đợi, một khi có dữ liệu từ cảm biến, task sẽ tiến hành gửi dữ liệu nhận được trong hàng đợi lên LCD và UART. Sau đó task tiếp tục kiểm tra xem có thông số nào đã được thay đổi hoặc cập nhật không. Nếu có thì tiến hành cập nhật thông số cho task ngay trực tiếp trong task, nếu không hoặc đã cập nhật thông số xong thì tiến hành cập nhật mức ưu tiên cao nhất (nhưng vẫn thấp hơn task xử lý trì hoãn ngắn) rồi đợi đến chu kỳ làm việc tiếp theo.

4.4.4.2 Cấu hình thiết bị phần mềm STM32CUBEIDE.

Tương tự cấu hình ở lập trình đa nhiệm bằng mô hình RMS.

4.4.4.3 Lập trình

- Cấu trúc mảng để lưu thông số của tất cả 6 task

```

1  typedef struct{
2      int id;
3      int period;
4      int execTime;
5      int deadline;
6      int priority;
7      int allPeriod;
8      float wcet;
9      bool changed;
10 } Task;

```

```

11 Task tasks[6] = {
12     {0, 2250, 50, 100, 5, 10, 0, false},
13     {1, 900, 4, 200, 11, 10, 0, false},
14     {2, 3000, 4, 300, 17, 10, 0, false},
15     {3, 1000, 4, 400, 23, 10, 0, false},
16     {4, 1800, 50, 500, 29, 10, 0, false},
17     {5, 4500, 4, 600, 35, 10, 0, false}
18 };

```

- Task lấy dữ liệu từ cảm biến

```

1     void StartDefaultTask(void *argument)
2 {
3     TickType_t lastWakeTime = osKernelGetTickCount();
4     taskReadTempDeadline = lastWakeTime +
5         pdMS_TO_TICKS(tasks[0].deadline);
6
7     float temp;
8     float timeInSeconds;
9     for(;;)
10    {
11        timeInSeconds = (float)osKernelGetTickCount() /
12            configTICK_RATE_HZ;
13        print_cli("The start time of get temperature task:
14            %.3f\n", timeInSeconds);
15        temp = SHT2x_GetTemperature(1);
16
17        //Check Deadline
18        if (osKernelGetTickCount() > taskReadTempDeadline)
19        {
20            print_cli("Task Read Temperature missed
21                deadline\n");
22        }
23        if (detectChange(&tasks[0], &prev_tasks[0]))
24        {
25            if(tasks[0].deadline != prev_tasks[0].deadline)
26            {
27                taskReadTempDeadline +=
28                    pdMS_TO_TICKS(tasks[0].period)
29                    -
30                    pdMS_TO_TICKS(prev_tasks[0].deadline)
31                    +
32                    pdMS_TO_TICKS(tasks[0].deadline);
33                print_cli("TASK 01 THAY DOI DEADLINE\n");
34            }
35            updatePrevious(&prev_tasks[0], &tasks[0]);
36        } else

```

```

29         {
30             taskReadTempDeadline +=
31                 pdMS_TO_TICKS(tasks[0].period);
32         }
33         timeInSeconds = (float)osKernelGetTickCount() /
34             configTICK_RATE_HZ;
35         print_cli("The end time of get temperature task:
36             %.3f\n", timeInSeconds);
37         osMessageQueuePut(myQueue01Handle, &temp, 0U, 0U);
38         updatePriorities();
39         vTaskDelayUntil(&lastWakeTime,
40                         pdMS_TO_TICKS(tasks[0].period));
41     }
42     /* USER CODE END 5 */
43 }
```

- Task hiển thị dữ liệu qua LCD, UART

```

1 void StartTask06(void *argument)
2 {
3     TickType_t lastWakeTime = osKernelGetTickCount();
4     taskDisplayRhDeadline = lastWakeTime +
5         pdMS_TO_TICKS(tasks[5].deadline);
6     float rhRecv;
7     char rhStr[6];
8     float timeInSeconds;
9     for(;;)
10    {
11        osMessageQueueGet(myQueue03Handle, &rhRecv, NULL,
12            osWaitForever);
13        timeInSeconds = (float)osKernelGetTickCount() /
14            configTICK_RATE_HZ;
15        print_cli("The start time of display rh over UART, LCD:
16            %.3f\n", timeInSeconds);
17        sprintf(rhStr, "r:%.1f", rhRecv);
18        lcd_put_cur(1, 10);
19        lcd_send_string(&rhStr);
20        print_cli("DO AM LA %s\n", rhStr);
21        // Check deadline
22        if (xTaskGetTickCount() > taskDisplayRhDeadline) {
23            print_cli("Task Display Temperature missed
24                deadline\n");
25        }
26        if (detectChange(&tasks[5], &prev_tasks[5]))
27        {
28            if(tasks[5].deadline != prev_tasks[5].deadline)
29            {
```

```

25         taskDisplayRhDeadline +=
26             pdMS_TO_TICKS(tasks[5].period)
27             -
28             pdMS_TO_TICKS(prev_tasks[5].deadline)
29             +
30             pdMS_TO_TICKS(tasks[5].deadline);
31         print_cli("TASK 06 THAY DOI DEADLINE\n");
32     }
33     updatePrevious(&prev_tasks[5], &tasks[5]);
34 } else {
35     taskDisplayRhDeadline +=
36         pdMS_TO_TICKS(tasks[5].period);
37 }
38 timeInSeconds = (float)osKernelGetTickCount() /
39     configTICK_RATE_HZ;
40 print_cli("The end time of display rh over UART, LCD:
41     %.3f\n", timeInSeconds);
42 updatePriorities();
43 // Delay until next period
44 vTaskDelayUntil(&lastWakeTime,
45     pdMS_TO_TICKS(tasks[5].period));
46 }
47 }
```

- Cập nhật ưu tiên cao nhất cho task có deadline gần nhất. Hàm này được thực hiện mỗi khi có một task bắt kỳ trong 6 task thực thi xong, nó sẽ xác định task có deadline gần nhất để gán cho mức ưu tiên cao nhất trong 6 task.

```

1 void updatePriorities(void) {
2     TickType_t now = xTaskGetTickCount();
3     // Calculate time remaining for each task
4     TickType_t timeToDeadlineTemp = taskReadTempDeadline
5         - now;
6     TickType_t timeToDeadlineTime = taskReadTimeDeadline
7         - now;
8     TickType_t timeToDeadlineDispTemp =
9         taskDisplayTempDeadline - now;
10    TickType_t timeToDeadlineDispTime =
11        taskDisplayTimeDeadline - now;
12    TickType_t timeToDeadlineRh = taskReadRhDeadline -
13        now;
14    TickType_t timeToDeadlineDispRh =
15        taskDisplayRhDeadline - now;
16
17    // Update priorities dynamically
18 }
```

```

12     vTaskPrioritySet(NULL, osPriorityLow); // Default
13         priority for the current task
14
15     if (timeToDeadlineTemp <= timeToDeadlineTime &&
16         timeToDeadlineTemp <= timeToDeadlineDispTemp &&
17         timeToDeadlineTemp <= timeToDeadlineDispTime &&
18         timeToDeadlineTemp <= timeToDeadlineDispRh &&
19         timeToDeadlineTemp <= timeToDeadlineRh) {
20         vTaskPrioritySet(defaultTaskHandle,
21             osPriorityHigh);
22     } else if (timeToDeadlineTime <=
23             timeToDeadlineDispTemp &&
24             timeToDeadlineTime <=
25                 timeToDeadlineDispTime &&
26                 timeToDeadlineTime <=
27                     timeToDeadlineDispRh) {
28         vTaskPrioritySet(RealTimeHandle, osPriorityHigh);
29     } else if (timeToDeadlineDispTemp <=
30             timeToDeadlineDispTime &&
31             timeToDeadlineDispTemp <=
32                 timeToDeadlineDispRh &&
33                 timeToDeadlineDispTemp <= timeToDeadlineRh) {
34         vTaskPrioritySet(DisplayTempHandle,
35             osPriorityHigh);
36     } else if (timeToDeadlineRh <= timeToDeadlineDispRh &&
37                 timeToDeadlineRh <= timeToDeadlineDispTime){
38         vTaskPrioritySet(myTask05Handle, osPriorityHigh);
39     } else if (timeToDeadlineDispRh <=
40                 timeToDeadlineDispTime){
41         vTaskPrioritySet(myTask06Handle, osPriorityHigh);
42     } else {
43         vTaskPrioritySet(DisplayTimeHandle,
44             osPriorityHigh);
45     }
46 }

```

- Hàm phát hiện có thông số của task thay đổi: Tương tự phần lập trình với RMS
- Task trì hoãn xử lý ngắt: tương tự phần lập trình đa nhiệm với RMS

4.5 Time slice và Priority based Scheduling

4.5.1 Khái niệm

Lập lịch dựa trên mức ưu tiên (Priority-Based Scheduling) là phương pháp mà mỗi task trong hệ thống được gán một mức ưu tiên cụ thể. Task có mức ưu tiên cao hơn sẽ được chọn chạy trước task có mức ưu tiên thấp hơn. Khi một task

có mức ưu tiên cao trở nên sẵn sàng, nó sẽ ngay lập tức tranh chấp CPU và ngắt task đang chạy có mức ưu tiên thấp hơn. Điều này đảm bảo rằng các tác vụ quan trọng được thực thi ngay lập tức.

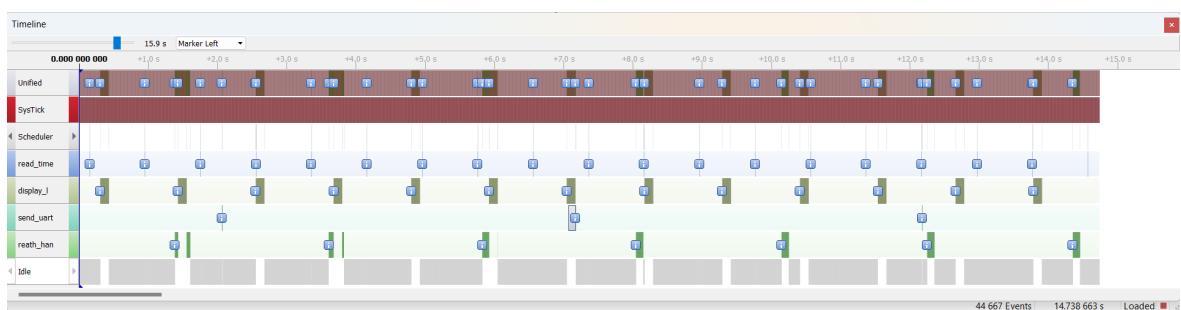
Trong khi đó, lập lịch chia thời gian (Time Slice Scheduling) được áp dụng khi có nhiều task có cùng mức ưu tiên. Lúc này, CPU sẽ chia sẻ thời gian chạy cho từng task trong khoảng thời gian cố định (time slice). Sau khi hết time slice, task sẽ nhường quyền CPU cho task tiếp theo trong danh sách có cùng mức ưu tiên, đảm bảo công bằng giữa các task.

Khi kết hợp hai phương pháp này, hệ thống sẽ ưu tiên xử lý các task dựa trên mức ưu tiên trước. Task có mức ưu tiên cao nhất luôn được chọn để chạy đầu tiên. Nếu có nhiều task có cùng mức ưu tiên, hệ thống sẽ sử dụng cơ chế chia thời gian (time slice) để luân phiên xử lý các task này. Điều này đảm bảo rằng các task có cùng mức ưu tiên đều được cấp CPU trong một khoảng thời gian nhất định trước khi chuyển sang task tiếp theo.

4.5.2 Phân tích tính toán

Task	Chu kỳ	Deadline	Execution C	Mức ưu tiên
read_time_handle	800ms	900ms	0.057s	3(cao)
display_lcd_handle	1 giây	950ms	0.001s	3(cao)
readth_handle	2 giây	1.8 giây	0.006s	2(Trung bình)
send_uart_handle	5 giây	3 giây	0.112s	1(Thấp)

Bảng 4.5. Phân tích chu kỳ, deadline, thời gian thực thi và mức ưu tiên của các task



Hình 4.24. Chu trình thực hiện các task

4.5.3 Triển khai lập trình

Đầu tiên, ta cấu hình configUSE_PREEMPTION. Khi giá trị được đặt là 1, tính năng *preemptive scheduling* (lập lịch ưu tiên ngắt quãng) sẽ được kích hoạt. Điều này cho phép FreeRTOS tạm dừng (*preempt*) task hiện tại và chuyển quyền điều khiển CPU ngay lập tức sang một task có mức ưu tiên cao hơn khi task đó trở nên sẵn sàng (*ready*). Điều này đảm bảo các task quan trọng được xử lý đúng lúc.

Tiếp theo, cấu hình configTICK_RATE_HZ xác định tần số tick (số lần ngắt hệ thống mỗi giây), tương ứng với độ phân giải của hệ điều hành. Khi giá trị này được đặt là 1000, mỗi tick sẽ xảy ra mỗi 1ms (1 giây = 1000 ticks). Đây là khoảng thời gian nhỏ nhất mà FreeRTOS có thể chuyển giữa các task cùng mức ưu tiên, đảm bảo thực hiện cơ chế *time-slicing*.

```

50 #define configUSE_PREEMPTION           1
51 #define configUSE_IDLE_HOOK            0
52 #define configUSE_TICK_HOOK             0
53 #define configCPU_CLOCK_HZ              ( SystemCoreClock )
54 #define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
55 #define configMAX_PRIORITIES          ( 5 )
56 #define configMINIMAL_STACK_SIZE        ( ( unsigned short ) 130 )
57 #define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 75 * 1024 ) )
58 #define configMAX_TASK_NAME_LEN         ( 10 )
59 #define configUSE_TRACE_FACILITY        1
60 #define configUSE_16_BIT TICKS           0
61 #define configIDLE_SHOULD_YIELD        1
62 #define configUSE_MUTEXES               1
63 #define configQUEUE_REGISTRY_SIZE        8
64 #define configCHECK_FOR_STACK_OVERFLOW   0
65 #define configUSE_RECURSIVE_MUTEXES      1
66 #define configUSE_MALLOC_FAILED_HOOK     0
67 #define configUSE_APPLICATION_TASK_TAG    0
68 #define configUSE_COUNTING_SEMAPHORES     1
69 #define configGENERATE_RUN_TIME_STATS    0
70

```

Hình 4.25. Cấu hình freeRTOS

Tiếp theo ta tạo các task và thiết lập mức ưu tiên cho chúng :

```

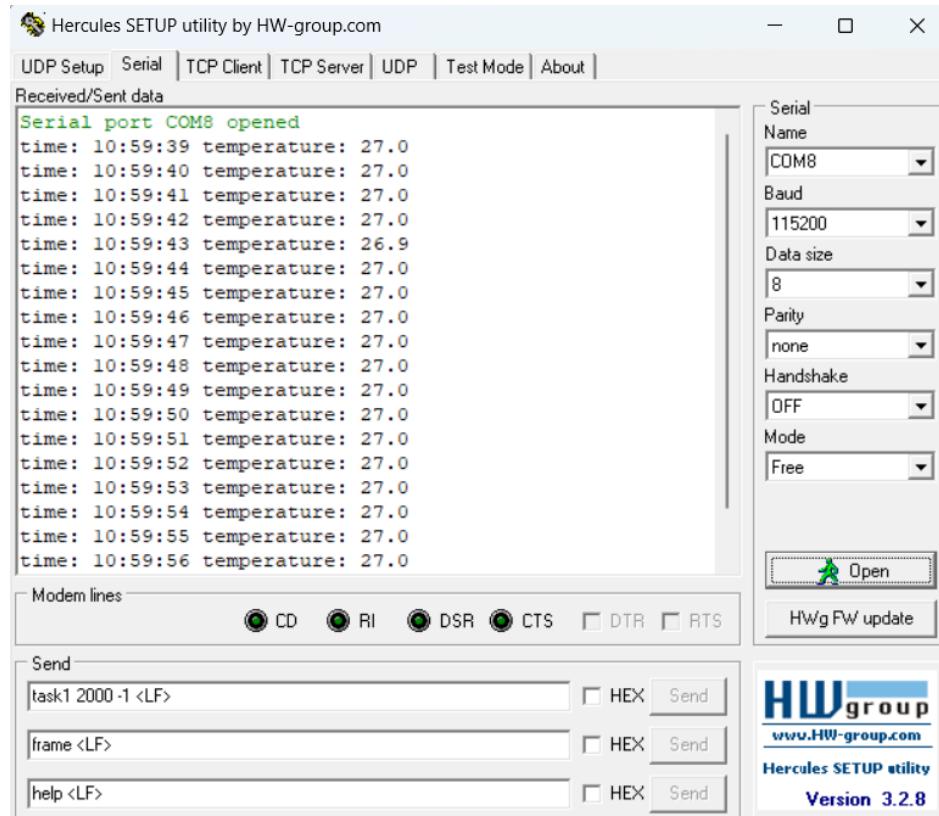
1 //create tasks
2
3     status = xTaskCreate(readth_handle,
4                           "readth_handle", 512, NULL, 1,
5                           &read_temp_and_hum);
6
7     configASSERT(status == pdPASS);
8
9
10    status = xTaskCreate(read_time_handle,
11                          "read_time", 512, NULL, 3, &read_time);
12
13    configASSERT(status == pdPASS);
14
15    status = xTaskCreate(send_uart_handle,
16                          "send_uart", 512, NULL, 2, &send_uart);
17
18    configASSERT(status == pdPASS);
19
20    status = xTaskCreate(display_lcd_handle,
21                          "display_lcd", 512, NULL, 3, &display_lcd);
22
23    configASSERT(status == pdPASS);

```

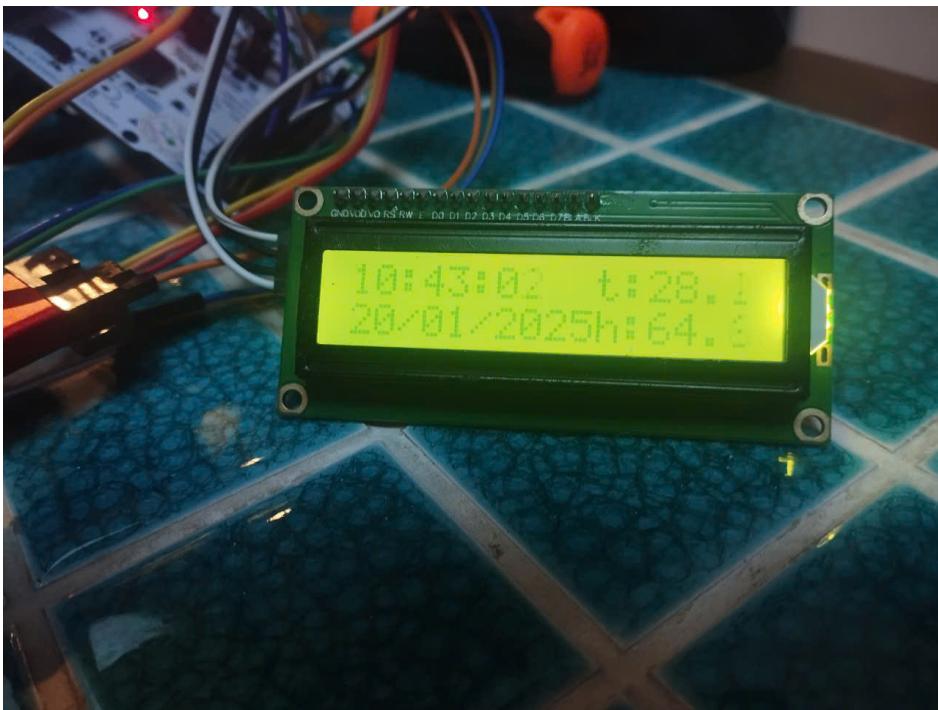
CHƯƠNG 5. KẾT QUẢ VÀ ĐÁNH GIÁ

5.1 Các kết quả đạt được

5.1.1 Simple Periodic TT Scheduler

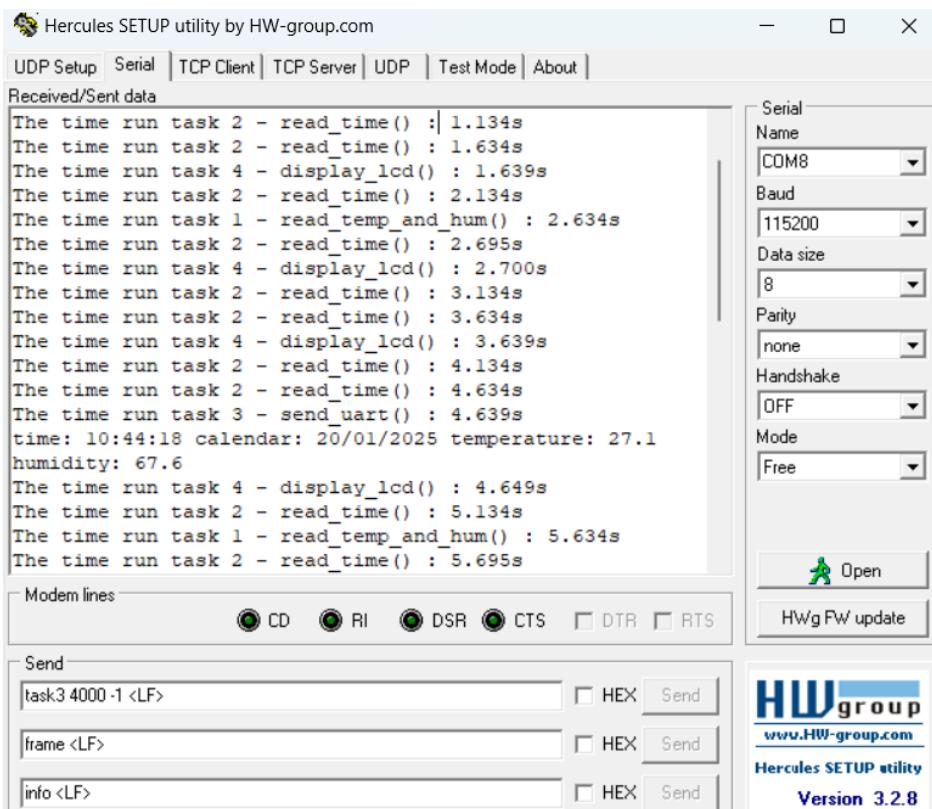


Hình 5.1. Các task đã chạy đúng theo thời gian cài đặt

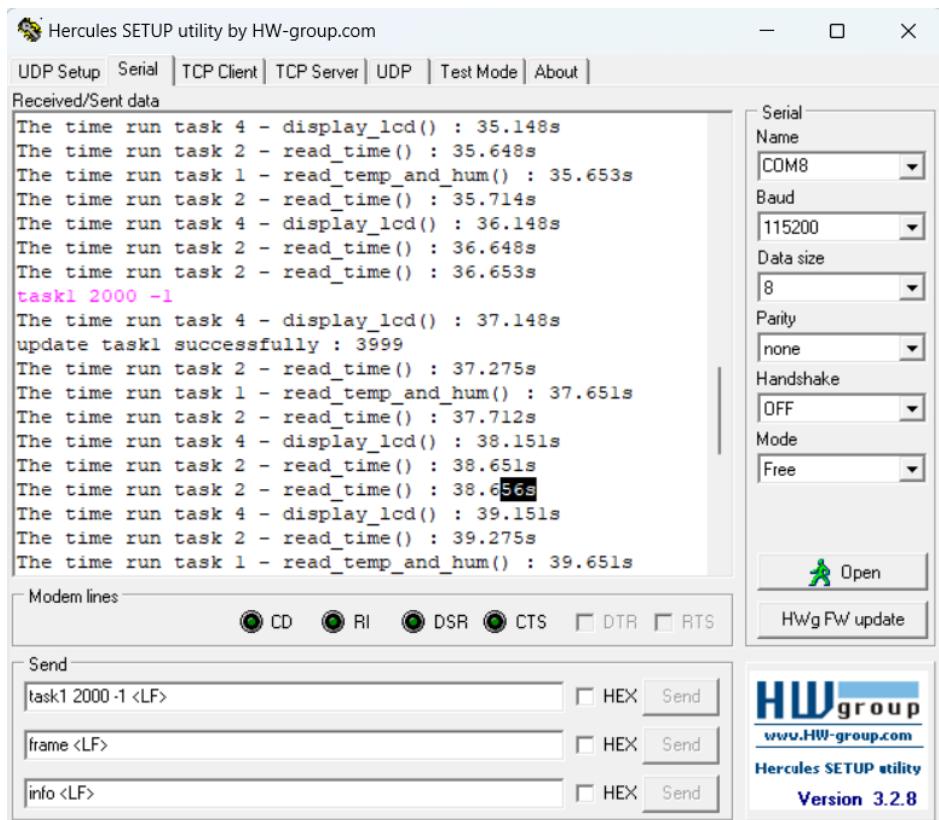


Hình 5.2. Màn hình hiển thị LCD

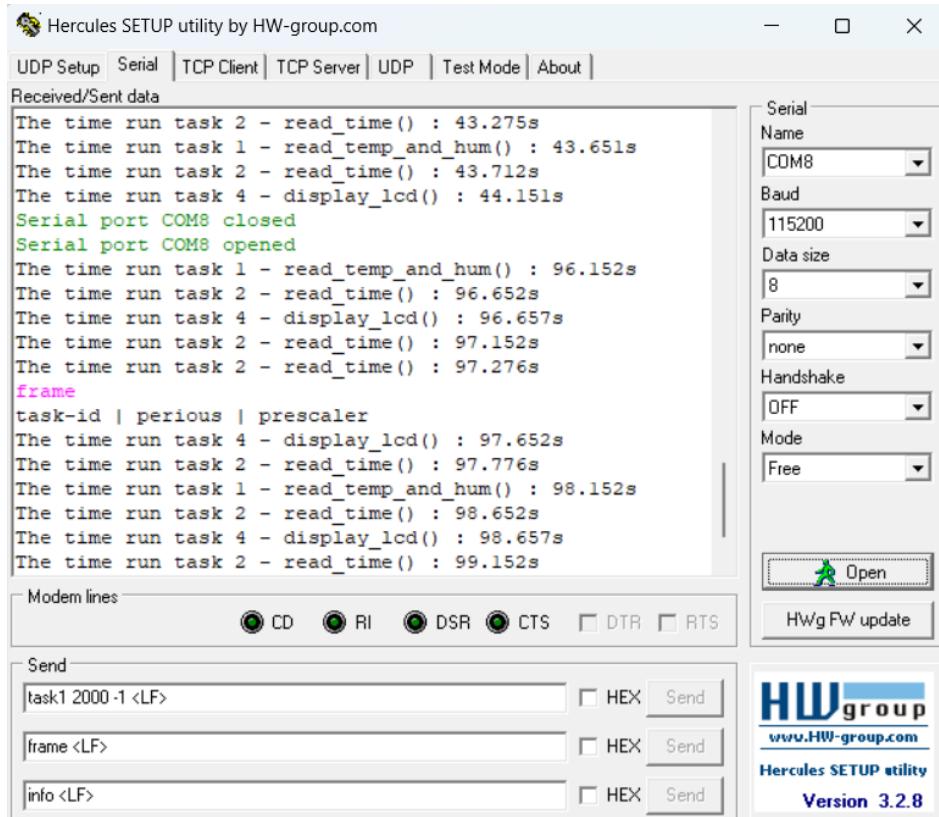
5.1.2 Non-Preemptive Event-Triggered Scheduling



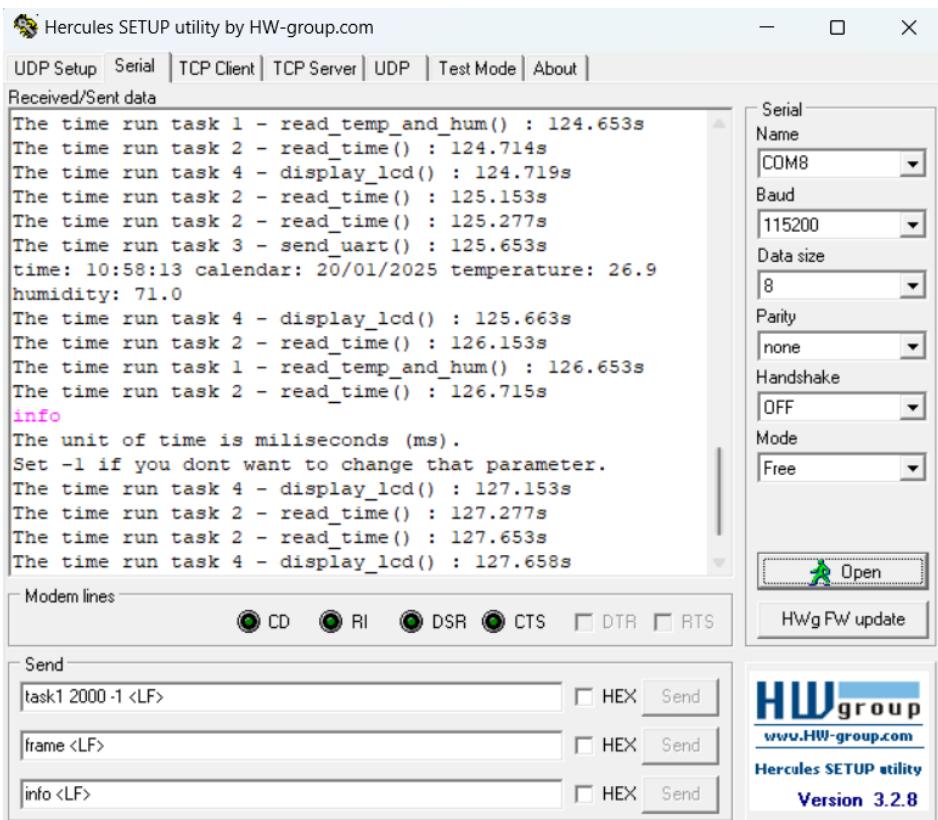
Hình 5.3. Các task đã chạy đúng theo thời gian cài đặt



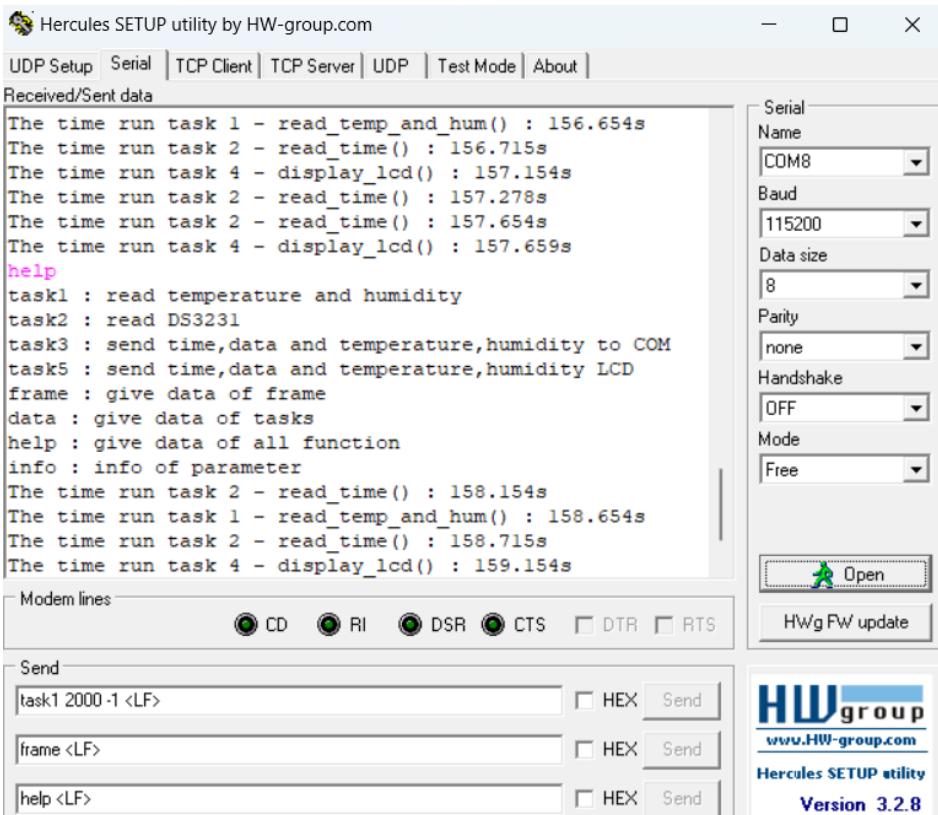
Hình 5.4. Điều chỉnh thành công chu kỳ của task1



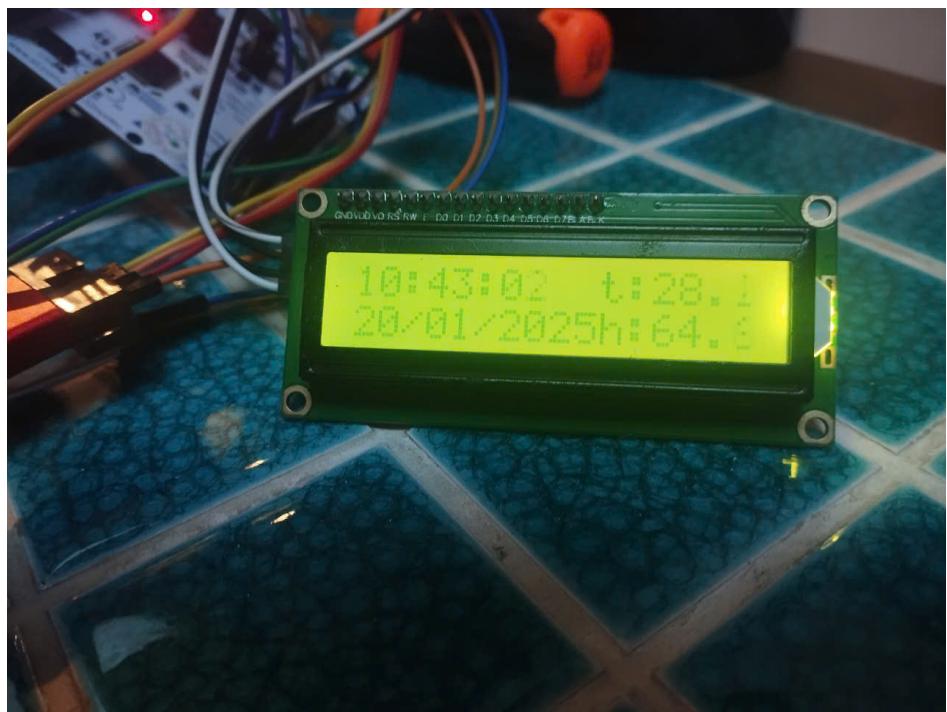
Hình 5.5. Lấy thông số frame truyền



Hình 5.6. Lấy thông số của đối số truyền vào

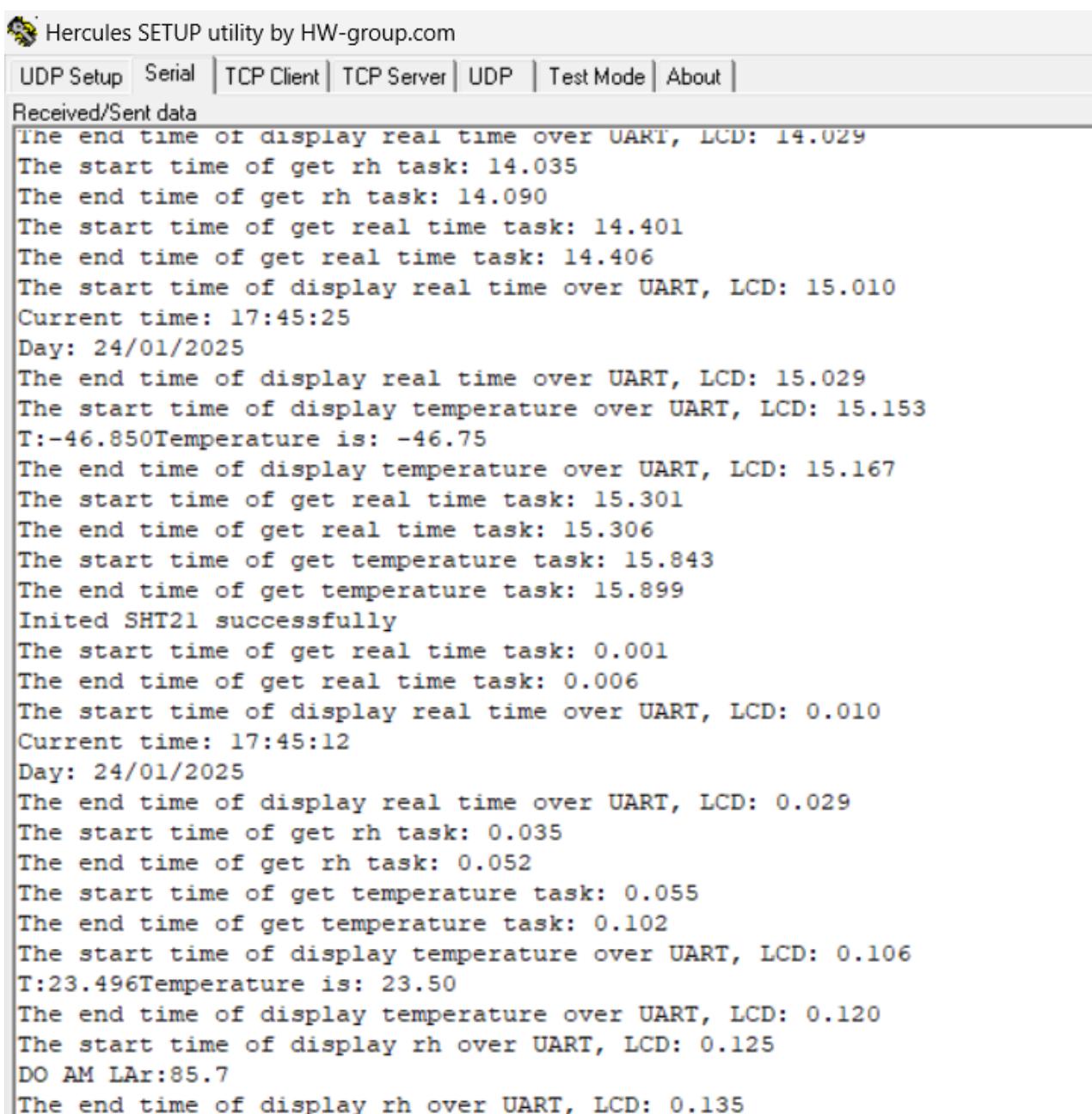


Hình 5.7. Lấy thông số của tất cả các hàm đang có



Hình 5.8. Màn hình hiển thị LCD

5.1.3 Mô hình đa nhiệm RMS



Hercules SETUP utility by HW-group.com

UDP Setup Serial TCP Client TCP Server UDP Test Mode About

Received/Sent data

```
The end time of display real time over UART, LCD: 14.029
The start time of get rh task: 14.035
The end time of get rh task: 14.090
The start time of get real time task: 14.401
The end time of get real time task: 14.406
The start time of display real time over UART, LCD: 15.010
Current time: 17:45:25
Day: 24/01/2025
The end time of display real time over UART, LCD: 15.029
The start time of display temperature over UART, LCD: 15.153
T:-46.850Temperature is: -46.75
The end time of display temperature over UART, LCD: 15.167
The start time of get real time task: 15.301
The end time of get real time task: 15.306
The start time of get temperature task: 15.843
The end time of get temperature task: 15.899
Initiated SHT21 successfully
The start time of get real time task: 0.001
The end time of get real time task: 0.006
The start time of display real time over UART, LCD: 0.010
Current time: 17:45:12
Day: 24/01/2025
The end time of display real time over UART, LCD: 0.029
The start time of get rh task: 0.035
The end time of get rh task: 0.052
The start time of get temperature task: 0.055
The end time of get temperature task: 0.102
The start time of display temperature over UART, LCD: 0.106
T:23.496Temperature is: 23.50
The end time of display temperature over UART, LCD: 0.120
The start time of display rh over UART, LCD: 0.125
DO AM LAr:85.7
The end time of display rh over UART, LCD: 0.135
```

Hình 5.9. Kết quả in ra khi các task hoạt động theo mô hình RMS



Hercules SETUP utility by HW-group.com

UDP Setup | Serial | TCP Client | TCP Server | UDP | Test Mode | About |

Received/Sent data

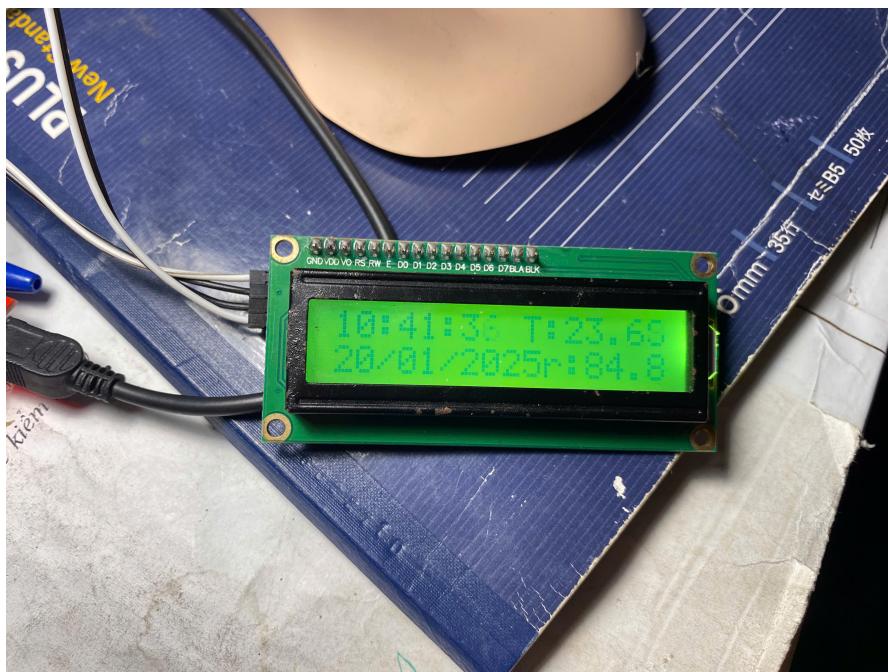
```

The end time of get temperature task: 2.351
The start time of get real time task: 2.701
The end time of get real time task: 2.706
The start time of display real time over UART, LCD: 3.010
Current time: 17:45:15
Day: 24/01/2025
The end time of display real time over UART, LCD: 3.029
The start time of display temperature over UART, LCD: 3.106
T:23.625Temperature is: 23.62
The end time of display temperature over UART, LCD: 3.119
The start time of get real time task045000080000
: 3.601
The end time of get real time task: 3.606
Nhan tu ngat yeu cau thay doi thong so lap lich
TASK 5
Period is 50
EXECUTION TIME is 0
DEADLINE is 8
AllPERIOD is 0
PRIORITY is 0
TASK 5 WAS CHANGED PERIOD
TASK 5 WAS CHANGED DEADLINE
The start time of display real time over UART, LCD: 4.010
Current time: 17:45:15
Day: 24/01/2025
TASK 04 WAS CHANGED PRIORITY
The end time of display real time over UART, LCD: 4.032
The start time of get real time task: 4.501
The end time of get real time task: 4.506
The start time of get temperature task: 4.555
TASK 01 WAS CHANGED PRIORITY
The end time of get temperature task: 4.604
The start time of display rh over UART, LCD: 4.625
DO AM LAr:84.9
TASK 06 WAS CHANGED PRIORITY

```

Hình 5.10. Khi nhận ngắt yêu cầu thay đổi thông số deadline và chu kỳ task 5

Khi task 5 nhận được yêu cầu thay đổi chu kỳ, những task còn lại cũng sẽ cập nhật lại mức độ ưu tiên của chúng dựa vào thứ tự chu kỳ hiện tại



Hình 5.11. Kết quả hiển thị trên LCD

5.1.4 Mô hình đa nhiệm EDF

```

Hercules SETUP utility by HW-group.com
UDP Setup | Serial | TCP Client | TCP Server | UDP | Test Mode | About |
Received/Sent data
The start time of display rh over UART, LCD: 4.118
DO AM LA r:83.5
The end time of display rh over UART, LCD: 4.127
The start time of get real time task: 5.002
The end time of get real time task: 5.006
Task 4 cao nhat
The start time of display real time over UART, LCD: 5.069
Current time: 10:41:17
Day: 20/01/2025
The end time of display real time over UART, LCD: 5.087
The start time of get temperature task: 6.001
The end time of get temperature task: 6.047
The start time of get real time task: 6.051
The end time of get real time task: 6.056
The start time of display temperature over UART, LCD: 6.060
Temperature is: 23.73
The end time of display temperature over UART, LCD: 6.071
The start time of display real time over UART, LCD: 6.077
Current time: 10:41:18
Day: 20/01/2025
The end time of display real time over UART, LCD: 6.095
Task 3 cao nhat
Task 6 cao nhat
The end time of get rh task: 6.114
The start time of display rh over UART, LCD: 6.118
DO AM LA r:83.6
The end time of display rh over UART, LCD: 6.127
The start time of get real time task: 7.002
The end time of get real time task: 7.006
Task 4 cao nhat
The start time of display real time over UART, LCD: 7.069
Current time: 10:41:18
Day: 20/01/2025
The end time of display real time over UART, LCD: 7.087

```

Hình 5.12. Kết quả hoạt động theo EDF

Hercules SETUP utility by HW-group.com

UDP Setup Serial | TCP Client | TCP Server | UDP | Test Mode | About |

Received/Sent data

```
Day: 20/01/2025
The end time of display real time over UART, LCD: 79.087
045000080000
Nhan tu ngat yeu cau thay doi thong so lap lich
TASK 5
Period is 50
EXECUTION TIME is 0
DEADLINE is 8
AllPERIOD is 0
PRIORITY is 0
TASK 5 THAY DOI CHU KY
The start time of get temperature task: 80.001
The end time of get temperature task: 80.047
The start time of get real time task: 80.051
The end time of get real time task: 80.056
The start time of display temperature over UART, LCD: 80.060
Temperature is: 23.86
The end time of display temperature over UART, LCD: 80.072
The start time of display real time over UART, LCD: 80.077
Current time: 10:42:30
Day: 20/01/2025
The end time of display real time over UART, LCD: 80.095
Task 3 cao nhat
TASK 05 THAY DOI DEADLINE
The end time of get rh task: 80.117
```

Hình 5.13. Kết quả nhận thay đổi thông số task từ ngắt UART trên màn hình



Hình 5.14. Kết quả hiển thị lên LCD

5.1.5 Mô hình đa nhiệm Time slice và Priority based Scheduling

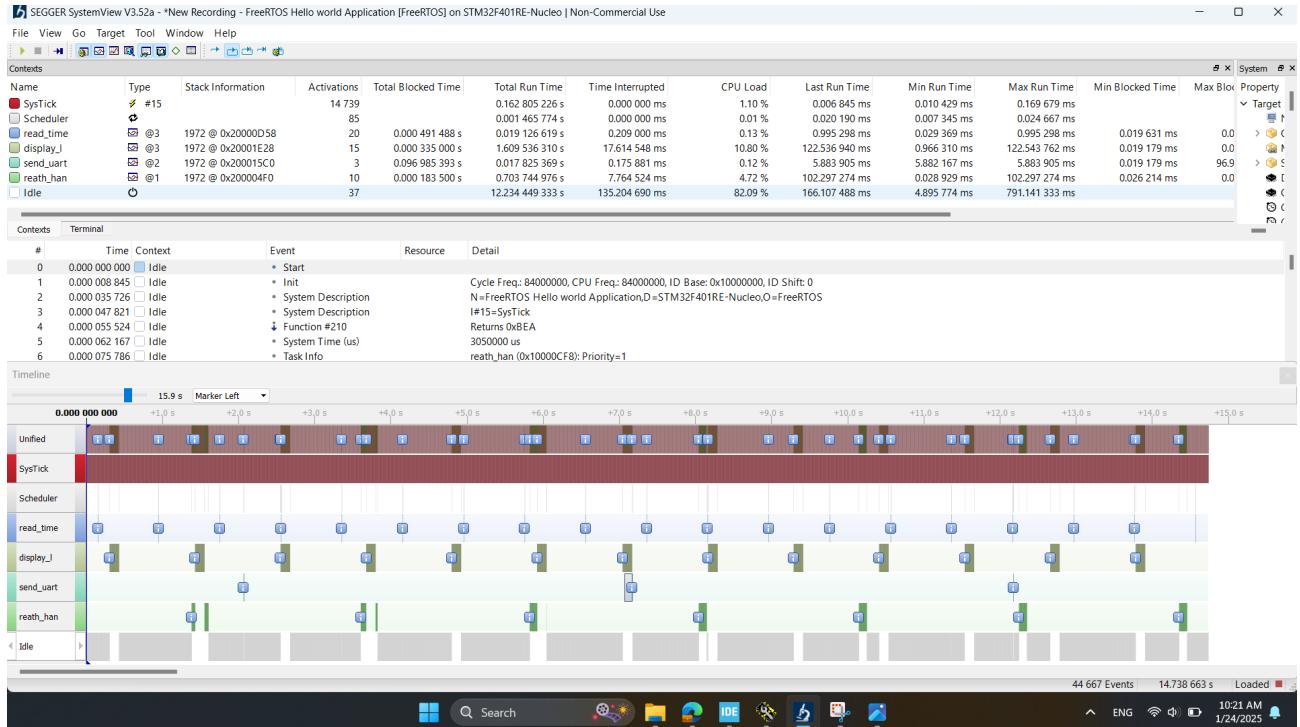


Hình 5.15. Màn hình hiển thị LCD

Để đánh giá CPU và các task hoạt động với mô hình này nhóm sử dụng SEGGER System View. Segger SystemView là một công cụ chuyên dụng để giám sát và phân tích các hệ thống nhúng thời gian thực, đặc biệt là các hệ thống sử dụng RTOS như FreeRTOS. Công cụ này được thiết kế nhằm cung cấp thông tin chi tiết về hành vi của hệ thống và giúp cải thiện hiệu năng thông qua các tính năng nổi bật sau:

- **Theo dõi sự kiện thời gian thực:** Segger SystemView ghi lại các sự kiện như chuyển đổi giữa các task, xử lý ngắn (*interrupts*), và hoạt động của các hàng đợi (*queues*) hoặc semaphore. Những sự kiện này được thu thập với độ phân giải cao, giúp phát hiện các vấn đề trong thời gian thực.
- **Phân tích chi tiết:** Công cụ cung cấp thông tin về thời gian thực thi, độ trễ và trạng thái của từng task. Dữ liệu này giúp xác định các điểm nghẽn hoặc task không tối ưu, từ đó đưa ra các cải tiến cho hệ thống.
- **Giao diện đồ họa trực quan:** Segger SystemView hiển thị dữ liệu qua giao diện trực quan, bao gồm biểu đồ thời gian và dòng sự kiện. Điều này giúp nhà phát triển dễ dàng nhận biết sự tương tác giữa các task và tài nguyên hệ thống.
- **Hỗ trợ đa RTOS:** Ngoài FreeRTOS, Segger SystemView còn hỗ trợ nhiều hệ điều hành thời gian thực khác, mang lại sự linh hoạt cao khi sử dụng cho các dự án khác nhau.
- **Tích hợp dễ dàng:** Công cụ yêu cầu cấu hình tối thiểu và có thể tích hợp nhanh chóng vào dự án, với tác động không đáng kể đến hiệu năng của hệ thống.

Segger SystemView là một giải pháp mạnh mẽ và hiệu quả để giám sát, phân tích và tối ưu hóa các hệ thống nhúng thời gian thực. Công cụ này giúp các nhà phát triển không chỉ đảm bảo hiệu năng hệ thống mà còn phát hiện và xử lý các vấn đề tiềm ẩn một cách nhanh chóng.



Hình 5.16. Đánh giá CPU và các task hoạt động

5.2 Phân tích và đánh giá kết quả

5.2.1 Simple Periodic Time-Triggered Scheduler

Các nhiệm vụ được thực thi theo chu kỳ cố định và được kích hoạt tại các thời điểm cụ thể. Mỗi nhiệm vụ phải hoàn thành trong một thời gian nhất định và sẽ được tái kích hoạt tự động sau mỗi chu kỳ. Ưu điểm:

- Dễ triển khai và quản lý: Mô hình đơn giản và dễ cài đặt vì bạn chỉ cần xác định chu kỳ và thời gian bắt đầu của các nhiệm vụ.
- Dự đoán và ổn định: Vì nhiệm vụ được thực thi tại các thời điểm cố định, dễ dàng dự đoán được hành vi của hệ thống và đảm bảo tính ổn định.
- Không bị ảnh hưởng bởi sự kiện bên ngoài: Các nhiệm vụ được thực thi theo lịch trình, không phụ thuộc vào các sự kiện bên ngoài, do đó hệ thống không bị ảnh hưởng bởi thay đổi đột ngột.

Nhược điểm:

- Không linh hoạt: Nếu nhiệm vụ không yêu cầu thực thi trong chu kỳ cố định, nó vẫn phải chạy, gây lãng phí tài nguyên.

- Không tối ưu hiệu suất tài nguyên: Một số nhiệm vụ có thể không cần thiết phải chạy trong mỗi chu kỳ, nhưng vẫn phải chạy, dẫn đến việc sử dụng tài nguyên không tối ưu.
- Độ trễ lớn khi có nhiệm vụ cần xử lý ngay lập tức: Nếu một nhiệm vụ cần được thực thi ngay lập tức, hệ thống phải đợi đến chu kỳ tiếp theo, gây ra độ trễ.

5.2.2 Non-Preemptive Event-Triggered Scheduling

Trong mô hình Non-Preemptive Event-Triggered Scheduling, nhiệm vụ được kích hoạt khi có một sự kiện xảy ra (ví dụ: một tín hiệu ngoại vi hoặc thay đổi trạng thái của hệ thống). Tuy nhiên, mô hình này là non-preemptive, tức là nhiệm vụ không thể bị gián đoạn trong khi đang thực hiện, và phải hoàn thành trước khi một nhiệm vụ khác có thể được thực hiện. Ưu điểm:

- Linh hoạt cao: Nhiệm vụ chỉ được thực hiện khi có yêu cầu thực sự, giúp tiết kiệm tài nguyên hệ thống.
- Đáp ứng nhanh với sự kiện: Hệ thống có thể phản hồi ngay lập tức khi có sự kiện cần xử lý mà không cần chờ đợi như mô hình time-trigger.
- Tối ưu hóa tài nguyên: Vì nhiệm vụ chỉ chạy khi có sự kiện, mô hình này giúp giảm thiểu việc sử dụng tài nguyên khi không cần thiết.

Nhược điểm:

- Không thể dự đoán chính xác: Vì nhiệm vụ được kích hoạt theo sự kiện, độ trễ có thể không được dự đoán chính xác, đặc biệt nếu có quá nhiều sự kiện chờ trong queue.
- Khó triển khai và quản lý: Việc quản lý và xử lý sự kiện có thể phức tạp, đặc biệt khi có nhiều sự kiện cần xử lý đồng thời.
- Quản lý queue có thể tạo overhead: Quản lý queue và đảm bảo rằng các sự kiện được xử lý đúng thứ tự có thể gây ra overhead, đặc biệt khi số lượng sự kiện rất lớn.

5.2.3 Mô hình EDF (Earliest Deadline First):

Ưu điểm:

- Các task đã chạy đúng như lý thuyết mô hình đưa ra, hoạt động một cách ổn định
- Đảm bảo có thể thay đổi thông số các task ngay lập tức từ máy tính
- Tối ưu hóa sử dụng CPU: Có khả năng đạt được hiệu quả sử dụng CPU tối đa (lên đến 100%

- Linh hoạt: Có thể đáp ứng các thay đổi trong thời gian thực và thích nghi với hệ thống thời gian thực mềm hoặc cứng.

Nhược điểm:

- Chi phí tính toán cao: Sắp xếp lại các tác vụ theo thời gian thực đòi hỏi tài nguyên tính toán lớn hơn so với các mô hình ưu tiên tĩnh.
- Nguy cơ quá tải: Trong trường hợp tổng mức sử dụng CPU vượt quá 100%

5.2.4 Mô hình RMS (Rate Monotonic Scheduling):

Ưu điểm:

- Các task đã chạy đúng như lý thuyết mô hình đưa ra một cách ổn định
- Có thể thay đổi thông số các task ngay lập tức qua UART từ máy tính.
- Dễ triển khai: Là thuật toán lập lịch ưu tiên tĩnh với các quy tắc đơn giản dựa trên chu kỳ tác vụ.
- Đáp ứng tốt thời gian thực cứng: Thích hợp với các hệ thống có yêu cầu deadline cứng nếu tổng mức sử dụng CPU không vượt quá 69.3
- Ổn định: Không dễ bị quá tải nếu điều kiện khả năng lập lịch được tuân thủ.

Nhược điểm:

- Không tối ưu hóa CPU: Không tận dụng được tài nguyên CPU khi tải thấp.
- Giới hạn khả năng lập lịch: Chỉ khả thi nếu điều kiện không chiếm quá 69% CPU.

5.2.5 Mô hình đa nhiệm Time slice và Priority based Scheduling

Ưu điểm:

- **Đảm bảo thực thi nhiệm vụ quan trọng:** Nhiệm vụ có mức độ ưu tiên cao hơn sẽ luôn được thực thi trước, đảm bảo rằng các tác vụ quan trọng không bị trì hoãn.
- **Sử dụng hiệu quả tài nguyên:** Time-slicing giúp các nhiệm vụ cùng mức ưu tiên chia sẻ CPU công bằng, tránh tình trạng "starvation" (nhiệm vụ không được thực thi).
- **Kết hợp tính linh hoạt và hiệu suất:** Mô hình kết hợp cả ưu tiên và time-slicing, mang lại sự cân bằng giữa hiệu suất hệ thống và đáp ứng thời gian thực.

Nhược điểm:

- Phức tạp trong triển khai:** Việc cài đặt mô hình này yêu cầu phải xác định chính xác mức độ ưu tiên và thời lượng time-slice phù hợp, có thể dẫn đến tăng độ phức tạp trong thiết kế.
- Khó dự đoán hành vi hệ thống:** Nếu nhiều nhiệm vụ có mức độ ưu tiên cao hoặc hệ thống có tải cao, thời gian thực thi có thể trở nên khó dự đoán.
- Tiềm ẩn độ trễ:** Mặc dù nhiệm vụ ưu tiên cao được xử lý trước, nhưng các nhiệm vụ ưu tiên thấp có thể gặp độ trễ nếu hệ thống bị quá tải bởi các nhiệm vụ quan trọng.

5.3 Thông kê công việc và tự đánh giá mức độ đóng góp của các thành viên (thang 10)

Tên	Đóng góp	Thang điểm
Lưu Đình Tú	Lập trình và thiết kế 2 mô hình đa nhiệm RMS và EDF. Lập trình giao tiếp với ngoại vi SHT21. Lập trình xử lý ngắn thay đổi thông số trong mô hình đa nhiệm. Thủ nghiệm và đánh giá kết quả.	9.5/10
Phạm Huy Tuyên	Lập trình và thiết kế 2 mô hình đơn nhiệm. Lập trình mô hình đa nhiệm Time slice và Priority based Scheduling Lập trình giao tiếp với RTC, LCD. Lập trình xử lý ngắn thay đổi thông số lập lịch. Thủ nghiệm và đánh giá kết quả.	9.5/10

CHƯƠNG 6. LINK SHARE THU MỤC

https://github.com/tuyenXuly/embedded_system

KẾT LUẬN

PHỤ LỤC