Milkwood pine Flower

FreeRTOS in STM32F103C8T6

by Truong Nho Tuan

Ha Noi, 9/2023

Các khái niệm quan trọng

1. Task

a. Overview

- Là đơn vị thực thi cơ bản. Chúng đại diện cho các luồng thực thi riêng lẻ trong một hệ thống nhúng thời gian thực. Task được sử dụng để thực hiện các chức năng hoặc thao tác cụ thể trong ứng dụng dựa trên FreeRTOS.
- Trong 1 thời điểm, các Task luôn chạy xen kẽ nhau theo chứ k đợi hết hàm vTaskDelay()của Task có mức ưu tiên cao hơn hết.

b. Function

- xTaskCreate()

BaseType_t xTaskCreate(
TaskFunction_t pxTaskCode, // Pointer to the task function.
*const char * const pcName, // A descriptive name for the task.*
const configSTACK_DEPTH_TYPE usStackDepth, // The stack depth in words (usually specified in words, not bytes).
*void * const pvParameters, // A pointer to a parameter structure or data for the task.*
UBaseType_t uxPriority, // The priority of the task (0 is the lowest, with higher values being higher priority).
*TaskHandle_t * const pxCreatedTask // A pointer to a variable that will hold the task's handle.*
);

pxTaskCode(TaskFunction_t):

- Tham số này là một con trỏ tới hàm tác vụ xác định hành vi của tác vụ. Về cơ bản nó là điểm vào cho mã của nhiệm vụ. Hàm tác vụ phải có nguyên mẫu sau: void myTaskFunction(void *pvParameters).

pcName(const char * const):

- Đây là tên mà con người đặt cho nhiệm vụ. Nó được sử dụng cho mục đích gỡ lỗi và nhận dạng. Đó là một chuỗi mô tả nhiệm vụ và thường được sử dụng cho các công cụ giám sát và gỡ lỗi.

usStackDepth(cấu hìnhSTACK_DEPTH_TYPE):

- Tham usStackDepthsố chỉ định kích thước ngăn xếp cho tác vụ. Nó thường được chỉ định bằng từ, không phải byte. Điều này xác định lượng bộ nhớ được phân bổ cho ngăn xếp của tác vụ. Kích thước ngăn xếp thực tế tính bằng byte sẽ phụ thuộc vào kiến trúc và cổng bạn đang sử dụng.
- Giá trị của usStackDepththường được chỉ định bằng từ, không phải byte. Kích thước ngăn xếp thực tế tính bằng byte sẽ phụ thuộc vào kiến trúc và cổng FreeRTOS mà bạn đang sử dụng. Điều này có nghĩa là nếu bạn chỉ định usStackDepthas 128, nó sẽ phân bổ đủ bộ nhớ để chứa 128 từ chứ không phải 128 byte.
  Kích thước từ:

- "Từ" trong ngữ cảnh này thường có nghĩa là kích thước từ tự nhiên của bộ xử lý hoặc vi điều khiển mà bạn đang sử dụng. Ví dụ: nếu bạn đang sử dụng bộ vi điều khiển có kiến trúc 32 bit thì mỗi từ sẽ có 4 byte.

  pvParameters(void * const):
- Tham số này là một con trỏ tới bất kỳ dữ liệu nào bạn muốn chuyển tới tác vụ khi nó được tạo. Nó có thể được sử dụng để khởi tạo tác vụ hoặc cung cấp thông tin ngữ cảnh.

  uxPriority(UBaseType_t):
- Tham uxPrioritysố đặt mức độ ưu tiên của tác vụ. Giá trị số càng cao thì mức độ ưu tiên càng cao. Các nhiệm vụ có mức độ ưu tiên thấp hơn sẽ được ưu tiên bởi các nhiệm vụ có mức độ ưu tiên cao hơn. Trong cmsis_oh.h, o*sPriorityAboveNormal* ứng với mức ưu tiên là 0.

  pxCreatedTask(TaskHandle_t * const):

- Đây là một con trỏ tới một biến sẽ giữ điều khiển của tác vụ đã tạo. Có thể được sử dụng để tham chiếu và thao tác tác vụ sau này trong mã của bạn.

  Ví dụ

  xTaskCreate(AboveNormalTask, "Task01", 128, NULL, 3, &AboveNormalHandle);

2. Queue

a. Overview

- hàng đợi là cấu trúc dữ liệu cho phép một tác vụ gửi dữ liệu đến tác vụ khác hoặc truy xuất dữ liệu từ hàng đợi. Nó tuân theo thứ tự "nhập trước, xuất trước" (FIFO), nghĩa là các mục dữ liệu được xử lý theo thứ tự chúng được đặt vào hàng đợi.

b. fuction

- xQueueCreate

  QueueHandle_t xQueueCreate(
  UBaseType_t uxQueueLength, // Maximum number of items the queue can hold
  UBaseType_t uxItemSizeBytes // Size (in bytes) of each item in the queue
  );

  uxQueueLength: Tham số này chỉ định số lượng mục tối đa mà hàng đợi có thể chứa tại bất kỳ thời điểm nào. Nó xác định độ sâu hoặc sức chứa của hàng đợi. Hàng đợi sẽ có thể lưu trữ uxQueueLengthcác mặt hàng. Ví dụ: nếu bạn đặt uxQueueLengththành 5 thì hàng đợi có thể chứa tối đa 5 mục.

  uxItemSizeBytes: Tham số này chỉ định kích thước, tính bằng byte, của từng mục sẽ được lưu trong hàng đợi. Điều quan trọng cần lưu ý là tất cả các mục được lưu trữ trong hàng đợi phải có cùng kích thước. Ví dụ: nếu bạn đang lưu trữ số nguyên trong hàng đợi, bạn sẽ chỉ định sizeof(int)là uxItemSizeBytes.

  Hàm xQueueCreatetrả về a QueueHandle_tlà một con trỏ tới hàng đợi đã tạo. Bạn sẽ sử dụng tay cầm này để tham chiếu và thao tác với hàng đợi trong các tác vụ của mình.
  VD: // Create a queue with a capacity of 10 items, each item being an integer (4 bytes)
  -> myQueue = xQueueCreate(10, sizeof(int));
- xQueueSend()

  BaseType_t xQueueSend(

QueueHandle_t xQueue, // Handle to the queue to which data is being sent
*const void *pvItemToQueue, // Pointer to the data being sent*
TickType_t xTicksToWait // Maximum time to wait if the queue is full
);

- **xQueue**: Đây là phần xử lý của hàng đợi mà bạn muốn gửi dữ liệu. Bạn có được thẻ điều khiển này khi tạo hàng đợi bằng cách sử dụng xQueueCreate.
- **pvItemToQueue**: Đây là con trỏ tới dữ liệu bạn muốn gửi đến hàng đợi. Dữ liệu có thể thuộc bất kỳ loại nào, nhưng nó phải cùng loại và kích thước với các mục mà hàng đợi mong đợi mà bạn đã chỉ định khi tạo hàng đợi. Ví dụ: nếu bạn đã tạo một hàng đợi cho intcác giá trị, bạn nên chuyển một con trỏ tới một intdữ liệu.
- **xTicksToWait**: Tham số này xác định thời gian tác vụ sẽ đợi nếu hàng đợi đầy. Nó chỉ định thời gian tối đa (tính theo dấu tích FreeRTOS) mà tác vụ sẵn sàng chờ khoảng trống trong hàng đợi nếu nó hiện đã đầy. Bạn có thể sử dụng hằng portMAX_DELAYsố để chỉ ra rằng tác vụ sẽ đợi vô thời hạn cho đến khi có đủ chỗ trống. Ngoài ra, bạn có thể chỉ định số lượng tích tắc hữu hạn để chờ.
  VD : xQueueSend(QueuexHandle,&adc_value[0],NULL);

- xQueueReceive()
  BaseType_t xQueueReceive(
  QueueHandle_t xQueue, // Handle to the queue from which data is being received
  *void *pvBuffer, // Pointer to the buffer where the received data will be stored*
  TickType_t xTicksToWait // Maximum time to wait if the queue is empty
  );
- **xQueue:** Đây là phần điều khiển của hàng đợi mà bạn muốn nhận dữ liệu. Bạn có được thẻ điều khiển này khi tạo hàng đợi bằng cách sử dụng xQueueCreate.
- **pvBuffer**: Đây là con trỏ tới bộ đệm nơi dữ liệu nhận được sẽ được lưu trữ. Dữ liệu nhận được từ hàng đợi sẽ được sao chép vào bộ đệm này. Loại và kích thước của bộ đệm phải khớp với loại và kích thước của các mục được lưu trong hàng đợi. Ví dụ: nếu hàng đợi chứa intcác giá trị, bạn nên cung cấp một con trỏ tới một intbiến.
- **xTicksToWait**: Tham số này xác định thời gian tác vụ sẽ đợi nếu hàng đợi trống. Nó chỉ định thời gian tối đa (tính theo dấu tích FreeRTOS) mà tác vụ sẵn sàng chờ dữ liệu có sẵn trong hàng đợi. Bạn có thể sử dụng hằng portMAX_DELAYsố để chỉ ra rằng tác vụ sẽ đợi vô thời hạn cho đến khi có sẵn dữ liệu. Ngoài ra, bạn có thể chỉ định số lượng tích tắc hữu hạn để chờ.
  VD : xQueueReceive(QueuexHandle,&data_receive[1],osWaitForever);

3. MailQueue

a. Overview

một loại hàng đợi cụ thể được thiết kế để lưu trữ con trỏ tới cấu trúc dữ liệu, thường được sử dụng để truyền tin nhắn hoặc dữ liệu giữa các tác vụ. Hàng đợi thư đặc biệt hữu ích khi các tác vụ cần trao đổi các cấu trúc hoặc đối tượng dữ liệu lớn hơn thay vì chỉ dữ liệu số hoặc ký tự đơn giản

b. Function

- xQueueCreate()
- xQueueSend()
- xQueueReceive()

4. Semaphore

a. Overview

- Các nguyên hàm đồng bộ hóa được sử dụng để đồng bộ hóa tác vụ và liên lạc trong các hệ điều hành thời gian thực như FreeRTOS. Chúng là một công cụ thiết yếu để quản lý quyền truy cập vào các tài nguyên được chia sẻ và kiểm soát việc thực hiện các tác vụ. FreeRTOS cung cấp cả ngữ nghĩa nhị phân và ngữ nghĩa đếm.

• Binary Semaphore

- Một semaphore nhị phân là một nguyên thủy đồng bộ hóa có hai trạng thái: đã lấy (hoặc "đã khóa") và đã phát hành (hoặc "đã mở khóa"). Nó thường được sử dụng để loại trừ lẫn nhau, trong đó một tác vụ phải có được semaphore trước khi truy cập vào tài nguyên được chia sẻ.
- Một tác vụ có thể lấy một đèn hiệu nhị phân và giải phóng nó khi nó được thực hiện bằng cách sử dụng các hàm xSemaphoreTakevà xSemaphoreGivetương ứng.
- Các ngữ nghĩa nhị phân thường được sử dụng để bảo vệ các phần mã quan trọng nhằm đảm bảo rằng chỉ một tác vụ có thể truy cập tài nguyên tại một thời điểm.
- Trong FreeRTOS, khi bạn khởi tạo một binary semaphore bằng hàm xSemaphoreCreateBinary(), giá trị mặc định của semaphore đó là 0.

• Counting Semaphore

- Một semaphore đếm có thể có giá trị lớn hơn một và được sử dụng để kiểm soát quyền truy cập vào tài nguyên nơi có thể có sẵn nhiều phiên bản của tài nguyên.
- Các tác vụ có thể lấy và đưa ra các ẩn dụ đếm bằng cách sử dụng hàm xSemaphoreTakevà xSemaphoreGive.
- Các ngữ nghĩa đếm thường được sử dụng cho các tình huống trong đó nhiều tác vụ có thể truy cập tài nguyên đồng thời nhưng cần có giới hạn về số lượng truy cập đồng thời.

b. Function

➢ Binary Semaphore

• xSemaphoreCreateBinary(): hàm khởi tạo

- Hàm này phân bổ bộ nhớ cho Semaphore Bimary . **Số đếm ban đầu của Semaphore Binary được đặt thành 0.**
- Kiểu trả về SemaphoreHandle_t
  VD: BinSemaxHandle=xSemaphoreCreateBinary();

• xSemaphoreGive(): đặt gt của Bin semaphore lên 1 khi nó đang là 0, tham số truyền vào là Handle của Semaphore Bin
  VD: xSemaphoreGive(BinSemaxHandle)

• xSemaphoreTake()
  BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait)
  xSemaphorelà phần xử lý của đèn hiệu nhị phân mà bạn muốn lấy. Điều khiển này thường thu được khi bạn tạo semaphore bằng cách sử dụng hàm như xSemaphoreCreateBinary().
  xTicksToWaitlà giá trị hết thời gian chỉ định khoảng thời gian tác vụ sẽ đợi để đèn tín hiệu khả dụng nếu nó hiện đang được thực hiện bởi một tác vụ khác. Nó được chỉ định trong dấu tích FreeRTOS. Các giá trị phổ biến là portMAX_DELAY(chờ vô thời hạn), 0(không đợi và quay lại ngay lập tức) hoặc một số tích tắc cụ thể để chờ.

Giá trị trả về của xSemaphoreTakethuộc loại BaseType_t, thường được xác định là **pdTRUE**(1) nếu semaphore được lấy thành công hoặc **pdFALSE**(0) nếu không thể lấy nó *trong khoảng thời gian chờ đã chỉ định.*

- Hàm này Thu lại giá trị trong Semaphore, Task chứa hàm khi k có semaphore sẽ bị block
  VD: xSemaphoreTake(xSemaphore, osWaitForever)

- uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore): Lấy ra giá trị của semaphore ở 1 thời điểm
  VD: uint8_t count = (uint8_t)uxSemaphoreGetCount(xSemaphore);

➢ Counting semaphore

- xSemaphoreCreateCounting(): Hàm khởi tạo, hàm này có hai tham số: số lượng tối đa được phép cho semaphore và giá trị đếm ban đầu.
  VD: SemaphoreHandle_t xSemaphore;
  xSemaphore = xSemaphoreCreateCounting(maxCount, initialValue);
- xSemaphoreTake(): giảm giá trị của semaphore đi 1, **Task chứa hàm khi k có semaphore sẽ bị block**
  VD: xSemaphoreTake(xSemaphore, osWaitForever)
- xSemaphoreGive(): Tăng giá trị Semaphore lên 1. **Khi Samaphore đầy, nếu tiếp tục gọi hàm xSemaphoreGive() thì chương trình sẽ trực tiếp nhảy qua hàm này và thực hiện các hàm khác, giá trị semaphore k tăng lên được nữa và Task chứa hàm xSemaphoreGive() không bị block.**
  VD: xSemaphoreGive(SemaxHandle);
- uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore): Lấy ra giá trị của semaphore ở 1 thời điểm . Kiểu trả về là **UBaseType_t** (là một kiểu dữ liệu cơ bản được sử dụng trong FreeRTOS và các hệ thống nhúng khác để đại diện cho các giá trị không âm)
  VD: uint8_t count = (uint8_t)uxSemaphoreGetCount(xSemaphore);

5. Mutex

a. Overview

- Sử dụng cho việc loại trừ lẫn nhau(mutial exclution), nghĩa là nó được dùng để hạn chế quyền truy cập tới một số resource, mutex hoạt động như là một token để bảo vệ share resource, về cách hoạt động này khá giống với lại semaphore
- Mutexes thường được sử dụng trong môi trường đa luồng hoặc đa tác vụ để đảm bảo rằng tại một thời điểm chỉ có một tác vụ có thể truy cập vào phần quan trọng của mã hoặc tài nguyên được chia sẻ.
- Tại mỗi một thời điểm thì chỉ có 1 task duy nhất có được mutex. Khi task này có mutex thì những task khác cũng muốn mutex này đều sẽ bị block cho tới khi task hiện tại release mutex ra.
- Về cơ bản thì Mutex giống như binary semaphore nhưng được sử dụng cho việc loại trừ lẫn nhau chứ không phải đồng bộ.
- Tuy nhiên, đối với Mutex, **chỉ Task nào chứa hàm xSemaphoreGive() mới được thực hiện hàm xSemaphoreTake(), nếu 2 hàm đó ở 2 Task khác nhau thì không thực hiện được.**

- Ngoài ra thì nó bao gồm cơ chế thừa kế mức độ ưu tiên(Priority inheritance mechanism) để giảm thiểu vấn đề đảo ngược ưu tiên.
- Kế thừa ưu tiên là một kỹ thuật đồng bộ hóa được sử dụng để giải quyết vấn đề được gọi là "đảo ngược ưu tiên" trong các hệ điều hành thời gian thực, bao gồm FreeRTOS. Đảo ngược mức độ ưu tiên xảy ra khi một tác vụ có mức độ ưu tiên thấp hơn giữ tài nguyên mà tác vụ có mức độ ưu tiên cao hơn cần, khiến tác vụ có mức độ ưu tiên cao hơn bị trì hoãn.
- VD:
  Task A (low priority) yêu cầu mutex
  Task B (high priority) muốn yêu cầu cùng mutex trên.
  Mức độ ưu tiên của Task A sẽ được đưa tạm về Task B để cho phép Task A được thực thi
  Task A sẽ thả mutex ra, mức độ ưu tiên sẽ được khôi phục lại và cho phép Task B tiếp tục thực thi.

6. Software Timer

a. Overview

- là một cơ chế trong FreeRTOS cho phép bạn tạo và quản lý bộ hẹn giờ trong phần mềm mà không cần dựa vào bộ hẹn giờ phần cứng. Chúng thường được sử dụng để lên lịch các tác vụ định kỳ hoặc để gây delay trong mã của bạn.
- Task Timer Service đảm nhiệm việc check dealine của timer trong list
- Khi timer “wake up” thì hàm callback được gọi tương ứng với Timer đó.
- Ngoài ra còn các Queue để các Task có thể giao tiếp

• Phân loại:

- One Shot timer : Hàm callback chỉ được gọi 1 lần khi Timer wake up
- Auti Reload Timer: tự reload lại sau mỗi lần wake up

b. Functions

• xTimerCreate()

```
TimerHandle_t xTimerCreate(
    const char * const pcTimerName,      // A descriptive name for the timer (for debugging)
    const TickType_t xTimerPeriod,       // The timer's period in ticks
    const UBaseType_t uxAutoReload,      // If pdTRUE, the timer will auto-reload; if pdFALSE, it's one-shot
    void * const pvTimerID,              // A pointer to an ID that can be associated with the timer
    TimerCallbackFunction_t pxCallbackFunction // The callback function to call when the timer expires
);
```

- **pcTimerName**: Tên mô tả cho bộ hẹn giờ, được sử dụng cho mục đích gỡ lỗi. Tham số này là một con trỏ tới một chuỗi ký tự không đổi.
- **xTimerPeriod**: Khoảng thời gian tính bằng tích tắc. Bộ hẹn giờ sẽ hết hạn và gọi hàm gọi lại sau số tích tắc này. Bạn có thể chuyển đổi thời gian tính bằng mili giây thành tích tắc bằng pdMS_TO_TICKSmacro.
- **uxAutoReload**: Nếu được đặt thành pdTRUE, bộ hẹn giờ sẽ tự động tải lại sau khi hết hạn, nghĩa là nó sẽ liên tục gọi hàm gọi lại trong khoảng thời gian đã chỉ định. Nếu được đặt thành pdFALSE, bộ hẹn giờ sẽ chạy một lần và chỉ hết hạn một lần.

- **pvTimerID**: Một con trỏ trỏ đến Timer ID

- **pxCallbackFunction**: Chức năng gọi lại được gọi khi hết giờ. Hàm này phải có nguyên mẫu sau:

7. Event group

a. Overview

- Có khả năng như Semaphore, Mutex
- Có thể đồng bộ 1 / nhiều Task
- Tối ưu được bộ nhớ
- Tiết kiệm được tài nguyên

b. Nguyên lý hoạt động

- 1 event group gồm nhiều event flag
- Có tối đa 24 flag trong 1 event group
- Được config qua configUSE_16_BIT_TICKS(1:8; 0:24)

c. Các thông số cần thiết

- Cấu hình Flag trong Event group

- Kiểu dữ liệu của Flag là **EventBits_t :** có 32 bit nhưng chỉ sử dụng 24bits

- xEventGroupCreate() **:** được sử dụng để tạo Event group
  VD: EventGroupHandle_t myEventGroup = xEventGroupCreate();
- xEventGroupSetBits() : được sử dụng để đặt một hoặc nhiều bit sự kiện trong một Event Group. xEventGroupSetBits() thường được sử dụng để báo hiệu sự xuất hiện của một hoặc nhiều sự kiện cho các tác vụ khác đang chờ những sự kiện đó.
  EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet);
  xEventGroup: ID của Event group.
  uxBitsToSet: Một bitmask đại diện cho các bit sự kiện bạn muốn đặt. Mỗi bit được đặt thành 1 trong mask bit này tương ứng với một bit sự kiện sẽ được đặt trong nhóm sự kiện.
  VD: EventBits_t bitsToSet = (1 << 0) | (1 << 1);
  EventBits_t bitsSet = xEventGroupSetBits(myEventGroup, bitsToSet);
- xEventGroupWaitBits(): là hàm API FreeRTOS được sử dụng để đợi cho đến khi các bit sự kiện cụ thể trong nhóm sự kiện được đặt
  EventBits_t xEventGroupWaitBits(
  EventGroupHandle_t xEventGroup,
  const EventBits_t uxBitsToWaitFor,
  BaseType_t xClearOnExit,
  BaseType_t xWaitForAllBits,
  TickType_t xTicksToWait
  );
  xEventGroup: ID của Event group
  uxBitsToWaitFor: Một bitmask đại diện cho các bit sự kiện mà tác vụ đang chờ. Mỗi bit được đặt thành 1 trong mặt nạ bit này tương ứng với một bit sự kiện mà tác vụ đang chờ.

xClearOnExit: Cờ xác định xem các bit sự kiện đang được chờ có được tự động xóa khi tác vụ được mở khóa hay không (tùy chọn, có thể pdTRUEhoặc pdFALSE).
xWaitForAllBits: Cờ xác định xem tác vụ có nên đợi tất cả các bit được chỉ định được thiết lập hay không (pdTRUE) hoặc liệu bất kỳ bit nào được chỉ định được đặt là đủ (pdFALSE).
xTicksToWait: Khoảng thời gian tối đa mà tác vụ phải đợi để thiết lập các bit sự kiện trước khi hết thời gian chờ. Điều này được chỉ định trong các dấu tích FreeRTOS (sử dụng pdMS_TO_TICKS()để chuyển đổi mili giây thành các dấu tích nếu cần) và nó có thể được đặt thành portMAX_DELAYchờ vô thời hạn.
VD : Bit = xEventGroupWaitBits(Group, EVENT0 | EVENT1 | EVENT2, pdTRUE, pdFALSE , osWaitForever);

- Để kiểm tra các Event, ta có thể dùng lệnh VD *: if ((Bit & EVENT1) != 0)*