TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI

---------&&&&&---------

LẬP TRÌNH NHÚNG

PIC 16F877A

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI

---------&&&&&---------

LẬP TRÌNH NHÚNG

CHƯƠNG 5:

RTOS CHO PIC16F877A

Hệ thống thời gian thực

- là hệ thống đó phải thực hiện các chức năng của mình trong một khoảng thời gian xác định và nhỏ nhất có thể chấp nhận được. Khi đáp ứng được yêu cầu này, hệ thống đó có thể gọi là hệ thống thời gian thực

Hệ thống thời gian thực

Gần như tất cả các hệ thống dựa trên vi điều khiển đều thực hiện nhiều hơn một tác vụ. Ví dụ hệ thống giám sát nhiệt độ tạo ra ba tác vụ lặp lại trong một thời gian ngắn đó là:

Tác vụ 1: Đọc nhiệt độ

Tác vụ 2: Định dạng nhiệt độ

Tác vụ 3: Hiển thị nhiệt độ

Hệ thống phức tạp với độ phức tạp cao thì sẽ có nhiều tác vụ phức tạp. Trong một hệ thống đa nhiệm,nhiều tác vụ đòi hỏi thời gian CPU, và do chỉ có một CPU, nên ta cần thiết có một số cách thức phối hợp và tổ chức để mỗi tác vụ có đủ thời gian CPU mà nó cần. Trong thực tế, mỗi tác vụ chiếm số lượng thời gian là rất ngắn, do đó dường như có vẻ như là tất cả các tác vụ được thực hiện song song và đồng thời.

Hầu như tất cả các hệ thống dựa trên vi điều khiển hoạt động trong thời gian thực. **Một hệ thống thời gian thực** là một hệ thống đáp ứng thời gian nghĩa là hệ thống đó có thể đáp ứng với môi trường hoạt động của nó trong thời gian ngắn nhất có thể.

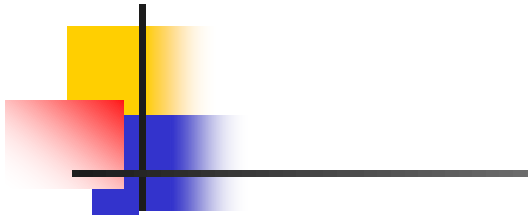
Thời gian thực không nhất thiết có nghĩa là các vi điều khiển sẽ hoạt động ở tốc độ cao. Thời gian đáp ứng nhanh chóng những gì là quan trọng trong một hệ thống thời gian thực, mặc dù tốc độ cao điều này có thể giúp ích được nhiều. Ví dụ, một hệ thống thời gian thực dựa trên vi điều khiển kết nối với các công tắc điều khiển ở bên ngoài được kỳ vọng sẽ đáp ứng ngay lập tức khi một công tắc được kích hoạt hoặc khi một số sự kiện khác xảy ra.

Hệ điều hành thời gian thực

Một hệ điều hành thời gian thực (RTOS) là một đoạn mã (thường được gọi là các hạt nhân hay **kernel**) điều khiển phân công tác vụ khi vi điều khiển được hoạt động trong một môi trường đa tác vụ. Quyết định RTOS , ví dụ, trong đó xác định nhiệm vụ để chạy tiếp theo, làm thế nào để phối hợp ưu tiên công việc, và làm thế nào để truyền dữ liệu và các thông điệp giữa các tác vụ.

Chương này tìm hiểu các nguyên tắc cơ bản của hệ thống nhúng đa nhiệm và đưa ra ví dụ của một RTOS được sử dụng trong các dự án đơn giản. Mã đa nhiệm và RTOS là những chủ đề phức tạp và rộng,và chương này mô tả các khái niệm ngắn gọn liên quan đến những công cụ này . Bạn đọc quan tâm có thể tham khảo nhiều sách và giấy tờ có sẵn về hệ điều hành, hệ thống đa tác vụ, và RTOS.

Có một số hệ thống RTOS thương mại có sẵn cho vi điều khiển PIC. Tại thời điểm viết bài, ngôn ngữ mikroC không hỗ trợ cung cấp các hàm xây dựng sẵn cho RTOS. Hai hệ thống phổ biến cấp cao RTOS cho vi điều khiển PIC là Salvo (www.pumpkin.com) có thể được sử dụng với trình biên dịch Hi-Tech PIC C, và trình biên dịch CCS (Customer Computer Service) với các hàm hỗ trợ RTOS được xây dựng sẵn để sử dụng. Trong chương này, các ví dụ RTOS được viết dựa trên trình biên dịch CCS (www.ccsinfo.com) , một trong những trình biên dịch phổ biến PIC C phát triển cho loạt các vi điều khiển PIC16 và PIC18 .



Support for the following RTOS:

- FreeRTOS
- OpenRTOS
- µC/OS-III, µC/OS-II
- ThreadX
- embOS

- Mục đích
  - Trừu tượng hóa phần cứng
  - Cung cấp cơ chế phân chia nhỏ công việc
  - Code chương trình trong sáng và dễ hiểu hơn
- So sánh với hệ điều hành thông thường
  - Giống
    - Đa tác vụ
    - Khả năng quản lý tài nguyên
    - Cung cấp một số service cho ứng dụng
    - Trừu tượng hóa phần cứng
  - Khác
    - Khả năng thích ứng với nhiều nền tảng phần cứng
    - Khả năng thực thi trên ROM hoặc RAM, sử dụng ít bộ nhớ
    - Chính sách scheduling phù hợp với hệ thống real-time
    - Tính ổn định cao hơn cho các ứng dụng nhúng
    - Khả năng co giãn để phù hợp ứng dụng
    - Tốc độ nhanh hơn

❖ **Trình phục vụ RTOS**

Trình phục vụ RTOS là tiện ích được cung cấp bởi các kernel có thể giúp các nhà phát triển tạo ra các tác vụ thời gian thực hiệu quả. Ví dụ, một công việc có thể sử dụng các trình phục vụ thời gian để có được ngày và thời gian hiện tại. Một số các trình phục vụ này là:

- Trình phục vụ ngắt
- Trình phục vụ thời gian
- Trình phục vụ quản lý thiết bị
- Trình phục vụ quản lý bộ nhớ
- Trình phục vụ đầu vào-đầu ra

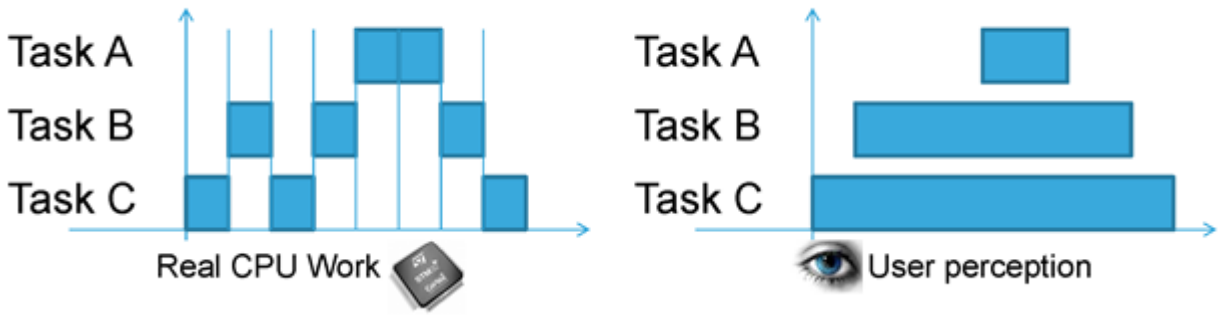
Khi nào bạn cần sử dụng RTOS ?

Các ứng dụng không cần dùng RTOS

- Ứng dụng đơn (ứng dụng chỉ có 1 chức năng)
- Ứng dụng có vòng lặp đơn giản
- Ứng dụng <32kB
- Nếu ứng dụng của bạn mà kích thước chương trình lớn dần và độ phức tạp tăng lên thì RTOS sẽ rất hữu dụng trong trường hợp này, lúc đó RTOS sẽ chia các ứng dụng phức tạp thành các phần nhỏ hơn và dễ quản lý hơn.
- Tại sao lại phải dùng RTOS ?
- Chia sẻ tài nguyên một cách đơn giản: cung cấp cơ chế để phân chia các yêu cầu về bộ nhớ và ngoại vi của MCU
- Dễ debug và phát triển: Mọi người trong nhóm có thể làm việc một cách độc lập, các lập trình viên thì có thể tránh được các tương tác với ngắt, timer, với phần cứng (cái này mình không khuyến khích lắm vì hiểu được phần cứng vẫn sẽ tốt hơn nhiều)
- Tăng tính linh động và dễ dàng bảo trì: thông qua API của RTOS,…

Các khái niệm cơ bản

- Kernel



- Kernel sẽ có nhiệm vụ quản lý nhiều task cùng chạy 1 lúc, mỗi task thường chạy mất vài ms. Tại lúc kết thúc task thường:
- Lưu trạng thái task
- Thanh ghi CPU sẽ load trạng thái của task tiếp theo
- Task tiếp theo cần khoảng vài ms để thực hiện
- Vì CPU thực hiện tác vụ rất nhanh nên dưới góc nhìn người dùng thì hầu như các task là được thực hiện 1 cách liên tục

Các khái niệm cơ bản

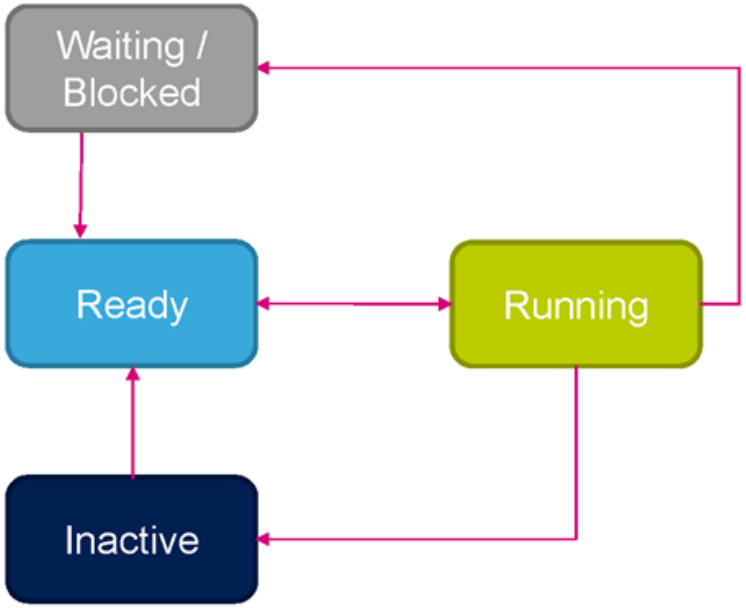
- **Task state**
- Một task trong RTOS thường có các trạng thái như sau

RUNNING: đang thực thi

READY: sẵn sàng để thực hiện

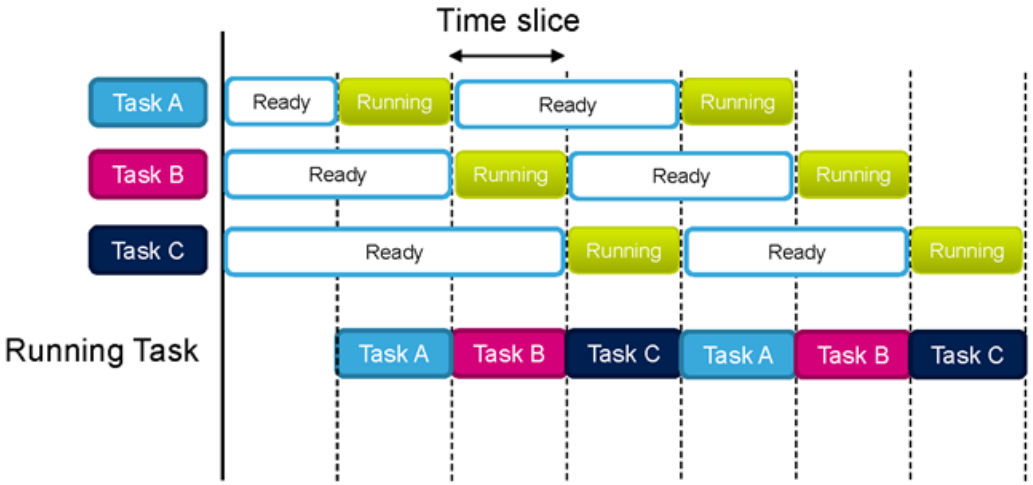
WAITING: chờ sự kiện

INACTIVE: không được kích hoạt



Các khái niệm cơ bản

- **Scheduler**
- Đây là 1 thành phần của kernel quyết định task nào được thực thi. Có một số luật cho scheduling như:
- **Cooperative**: giống với lập trình thông thường, mỗi task chỉ có thể thực thi khi task đang chạy dừng lại, nhược điểm của nó là task này có thể dùng hết tất cả tài nguyên của CPU
- **Round-robin**: mỗi task được thực hiện trong thời gian định trước (time slice) và không có ưu tiên.

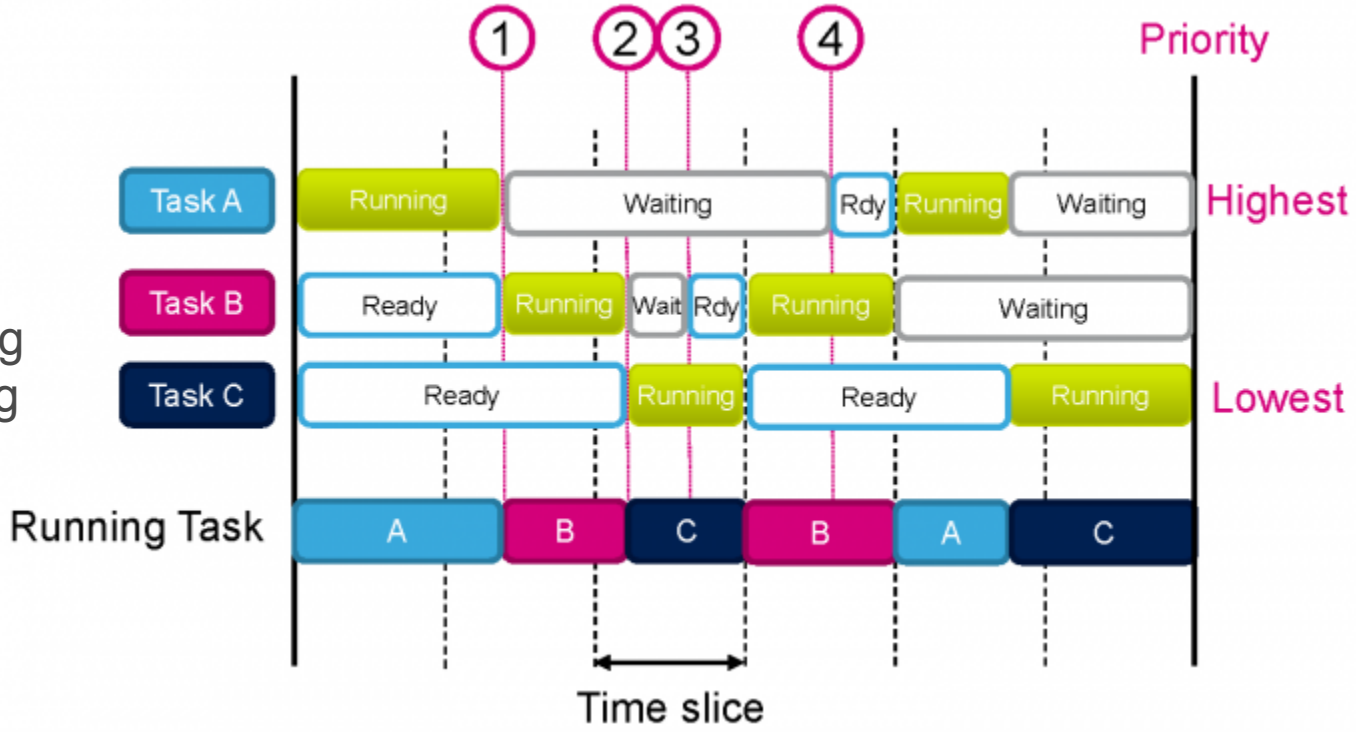


Các khái niệm cơ bản

Scheduler

Priority base: Task được phân quyền cao nhất sẽ được thực hiện trước, nếu các task có cùng quyền như nhau thì sẽ giống với round-robin, các task có mức ưu tiên thấp hơn sẽ được thực hiện cho đến cuối time slice

1. Task A chờ event
2. Task B chờ event
3. Task B event sẵn sàng
4. Task A event sẵn sàng



Các khái niệm cơ bản

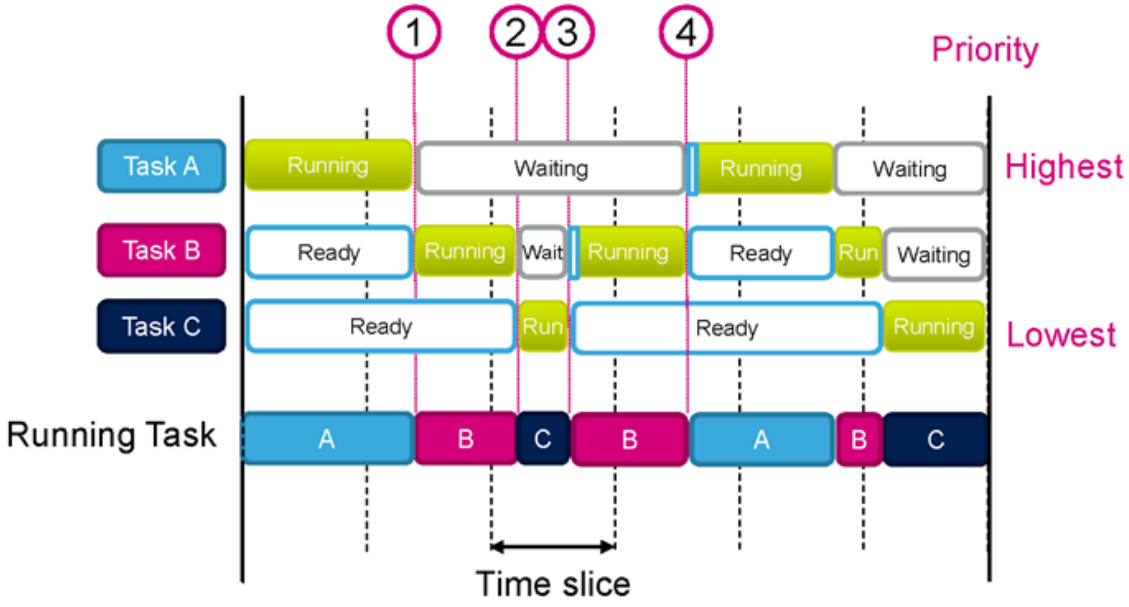
Scheduler

Priority base: Task được phân quyền cao nhất sẽ được thực hiện trước, nếu các task có cùng quyền như nhau thì sẽ giống với round-robin, các task có mức ưu tiên thấp hơn sẽ được thực hiện cho đến cuối time slice

Các khái niệm cơ bản

- **Priority-based pre-emptive**: Các task có mức ưu tiên cao nhất luôn nhường các task có mức ưu tiên thấp hơn thực thi trước.

1. Task A chờ event
2. Task B chờ event
3. Task B event sẵn sàng
4. Task A event sẵn sàng



Các khái niệm cơ bản

- **Task**
- Một task là một chương trình, chương trình này chạy liên tục trong vòng lặp vô tận và không bao giờ dừng lại. Với mỗi task thì có niềm tin duy nhất là chỉ mình nó đang chạy và có thể sử dụng hết nguồn tài nguyên sẵn có của bộ xử lý (mặc dù là thực tế thì nó vẫn phải chia sẻ nguồn tài nguyên này với các task khác).
- Một chương trình thường sẽ có nhiều task khác nhau.
- Kernel sẽ quản lý việc chuyển đổi giữa các task, nó sẽ lưu lại ngữ cảnh của task sắp bị hủy và khôi phục lại ngữ cảnh của task tiếp theo bằng cách:
  - Kiểm tra thời gian thực thi đã được định nghĩa trước (time slice được tạo ra bởi ngắt systick)
  - Khi có các sự kiện unblocking một task có quyền cao hơn xảy ra (signal, queue, semaphore,…)
  - Khi task gọi hàm Yield() để ép Kernel chuyển sang các task khác mà không phải chờ cho hết time slice
  - Khi khởi động thì kernel sẽ tạo ra một task mặc định gọi là Idle Task.
  - Để tạo một task thì cần phải khai báo hàm định nghĩa task, sau đó tạo task và cấp phát bộ nhớ, phần này mình sẽ nói sau.

Các khái niệm cơ bản

Kết nối Inter-task & Chia sẻ tài nguyên

- Các task cần phải kết nối và trao đổi dữ liệu với nhau để có thể chia sẻ tài nguyên, có một số khái niệm cần lưu ý
  - Với Inter-task Communication:
  - Signal Events – Đồng bộ các task
  - Message queue – Trao đổi tin nhắn giữa các task trong hoạt động giống như FIFO
  - Mail queue – Trao đổi dữ liệu giữa các task sử dụng hằng đợi của khối bộ nhớ
- Với Resource Sharing
  - Semaphores – Truy xuất tài nguyên liên tục từ các task khác nhau
  - Mutex – Đồng bộ hóa truy cập tài nguyên sử dụng Mutual Exclusion

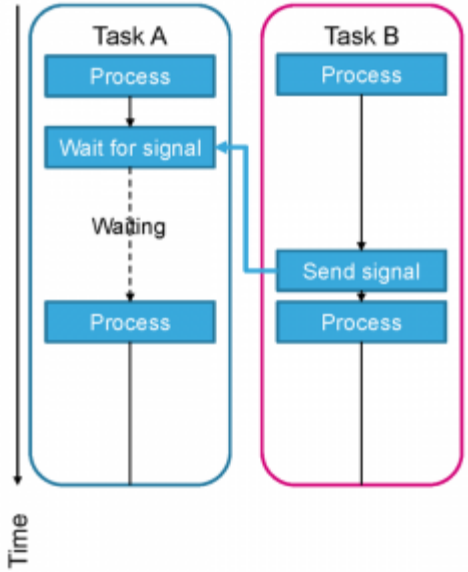
Các khái niệm cơ bản

Signal event

- Signal event được dùng để đồng bộ các task, ví dụ như bắt task phải thực thi tại một sự kiện nào đó được định sẵn

Mỗi task có thể được gán tối đa là 32 signal event

Ưu điểm của nó là thực hiện nhanh, sử dụng ít RAM hơn so với semaphore và message queue nhưng có nhược điểm lại chỉ được dùng khi một task nhận được signal.



Các khái niệm cơ bản

Message queue

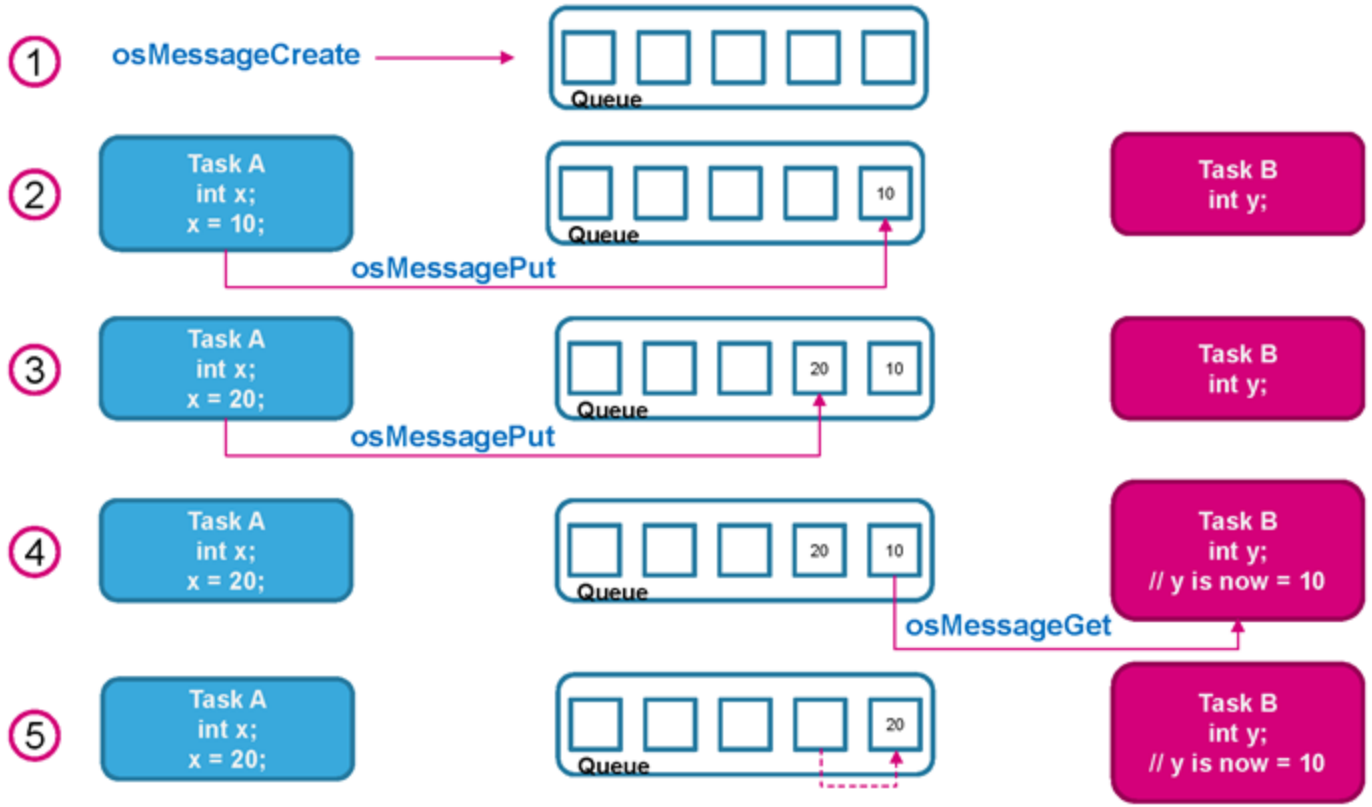
- Message queue là cơ chế cho phép các task có thể kết nối với nhau, nó là một FIFO buffer được định nghĩa bởi độ dài (số phần tử mà buffer có thể lưu trữ) và kích thước dữ liệu (kích thước của các thành phần trong buffer). Một ứng dụng tiêu biểu là buffer cho Serial I/O, buffer cho lệnh được gửi tới task

- Task có thể ghi vào hằng đợi (queue)
  - Task sẽ bị khóa (block) khi gửi dữ liệu tới một message queue đầy đủ
  - Task sẽ hết bị khóa (unblock) khi bộ nhớ trong message queue trống
  - Trường hợp nhiều task mà bị block thì task với mức ưu tiên cao nhất sẽ được unblock trước
- Task có thể đọc từ hằng đợi (queue)
  - Task sẽ bị block nếu message queue trống
  - Task sẽ được unblock nếu có dữ liệu trong message queue.
  - Tương tự ghi thì task được unblock dựa trên mức độ ưu tiên

Các khái niệm cơ bản

Message queue

VD:



Các khái niệm cơ bản

Mail queue

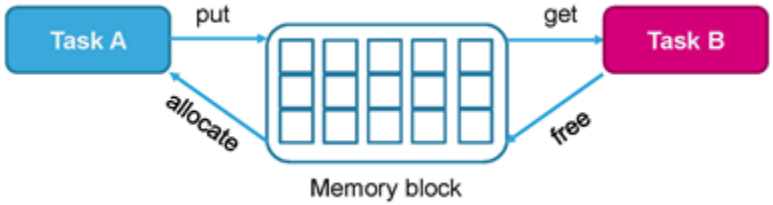
- Giống như message queue nhưng dữ liệu sẽ được truyền dưới dạng khối(memory block) thay vì dạng đơn. Mỗi memory block thì cần phải cấp phát trước khi đưa dữ liệu vào và giải phóng sau khi đưa dữ liệu ra.

Gửi dữ liệu với mail queue

- Cấp phát bộ nhớ từ mail queue cho dữ liệu được đặt trong mail queue
- Lưu dữ liệu cần gửi vào bộ nhớ đã được cấp phát
- Đưa dữ liệu vào mail queue

Nhận dữ liệu trong mail queue bởi task khác

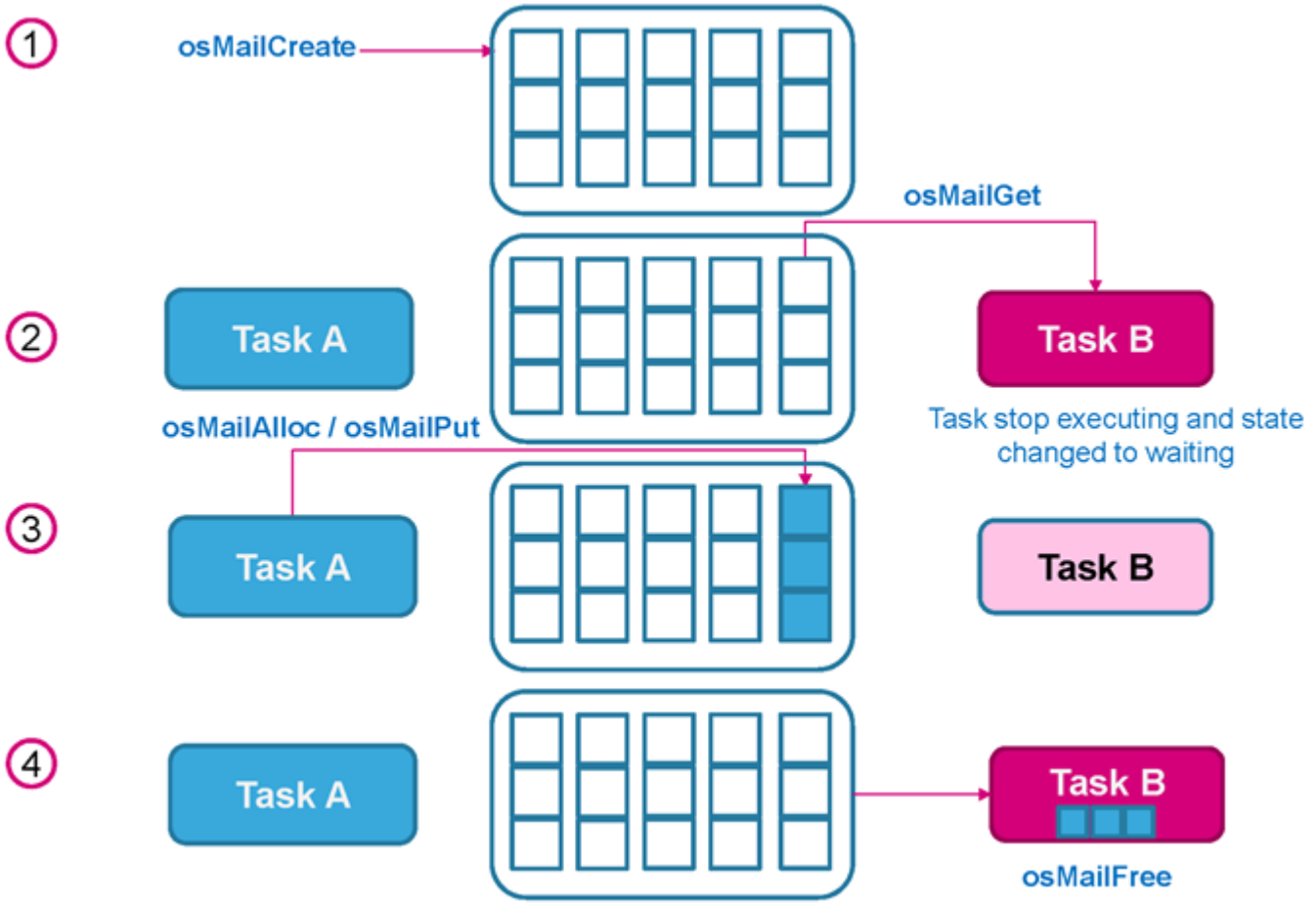
- Lấy dữ liệu từ mail queue, sẽ có một hàm để trả lại cấu trúc/ đối tượng
- Lấy con trỏ chứa dữ liệu
- Giải phóng bộ nhớ sau khi sử dụng dữ liệu



Các khái niệm cơ bản

Mail queue

VD:



Các khái niệm cơ bản

Semaphore

- Được sử dụng để đồng bộ task với các sự kiện khác trong hệ thống. Có 2 loại

Binary semaphore

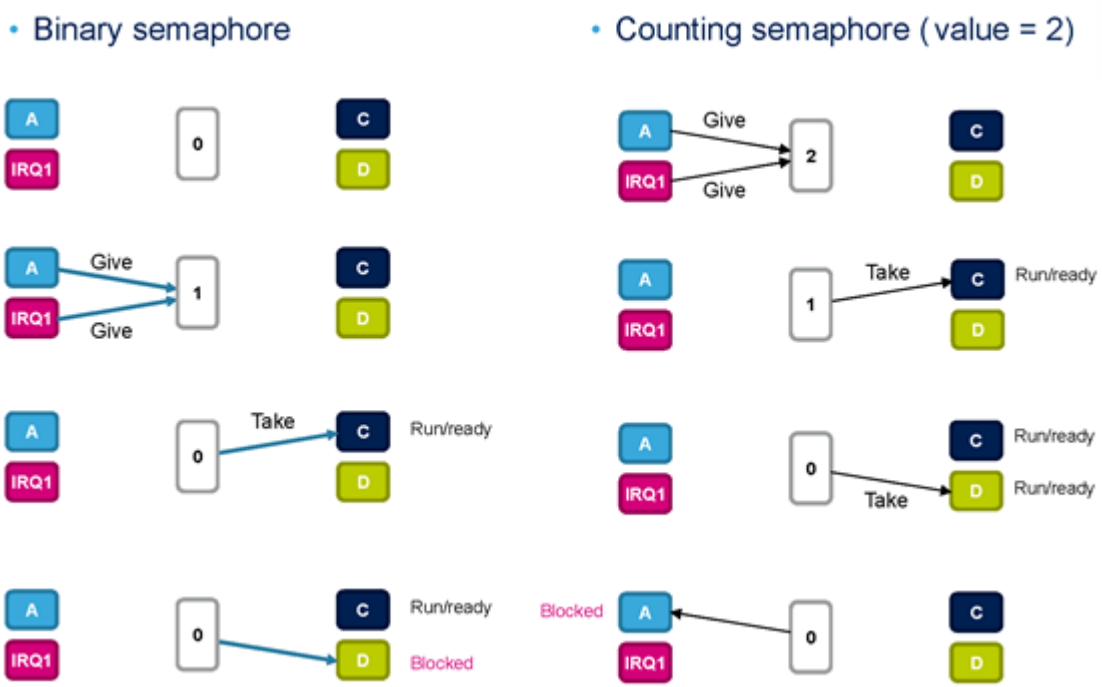
- Trường hợp đặc biệt của counting semaphore
- Có duy nhất 1 token
- Chỉ có 1 hoạt động đồng bộ

Counting semaphore

- Có nhiều token
- Có nhiều hoạt động đồng bộ

Các khái niệm cơ bản

Semaphore



❖ Semaphore

Couting semaphore được dùng để:

Counting event

•Một event handler sẽ ‘give’ semaphore khi có event xảy ra (tăng giá trị đếm semaphore)

•Một task handler sẽ ‘take’ semaphore khi nó thực thi sự kiện (giảm giá trị đếm semaphore)

•Count value là khác nhau giữa số sự kiện xảy ra và số sự kiện được thực thi

•Trong trường hợp counting event thì semaphore được khởi tạo giá trị đếm bằng 0

Resource management

•Count value sẽ chỉ ra số resource sẵn có

•Để điều khiển và kiểm soát được resource của task dựa trên count value của semaphore(giá trị giảm), nếu count value giảm xuống bằng 0 nghĩa là không có resource nào free.

•Khi một task finish với resource thì nó sẽ give semaphore trở lại để tăng count value của semaphore.

•Trong trường hợp resouce management thì count value sẽ bằng với giá trị max của count value khi semaphore được tạo.

Các khái niệm cơ bản

Mutex

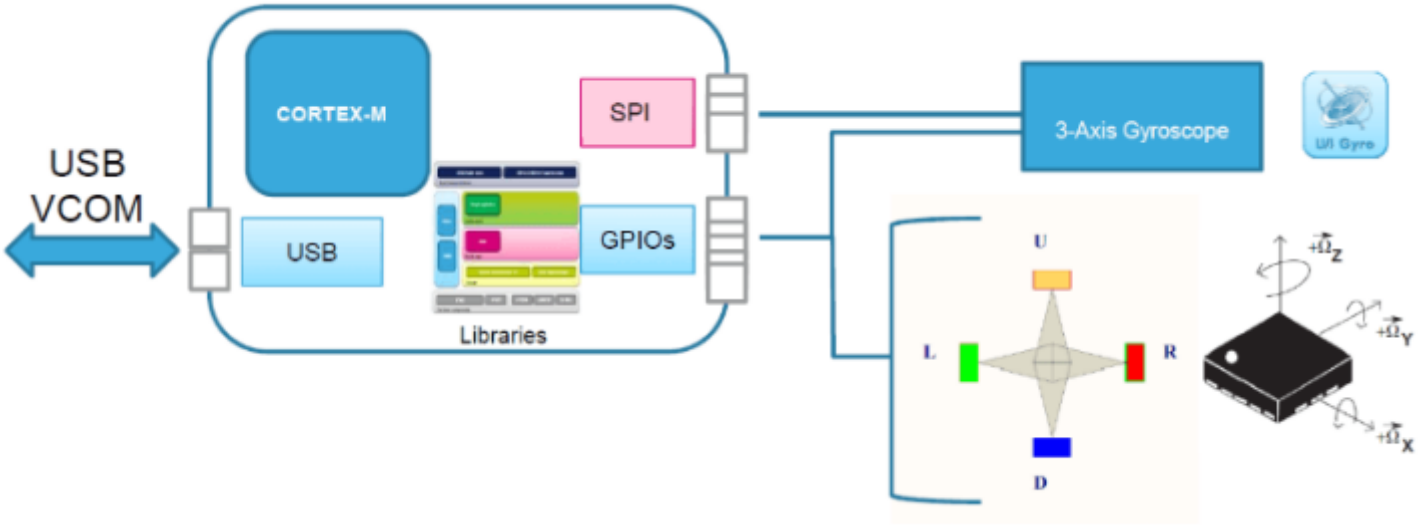
- Sử dụng cho việc loại trừ (mutial exclution), hoạt động như là một token để bảo vệ tài nguyên được chia sẻ. Một task nếu muốn truy cập vào tài nguyên chia sẻ
- Cần yêu cầu (đợi) mutex trước khi truy cập vào tài nguyên chia sẻ
- Đưa ra token khi kết thúc với tài nguyên.
- Tại mỗi một thời điểm thì chỉ có 1 task có được mutex. Những task khác muốn cùng mutex thì phải block cho đến khi task cũ thả mutex ra.

Các khái niệm cơ bản

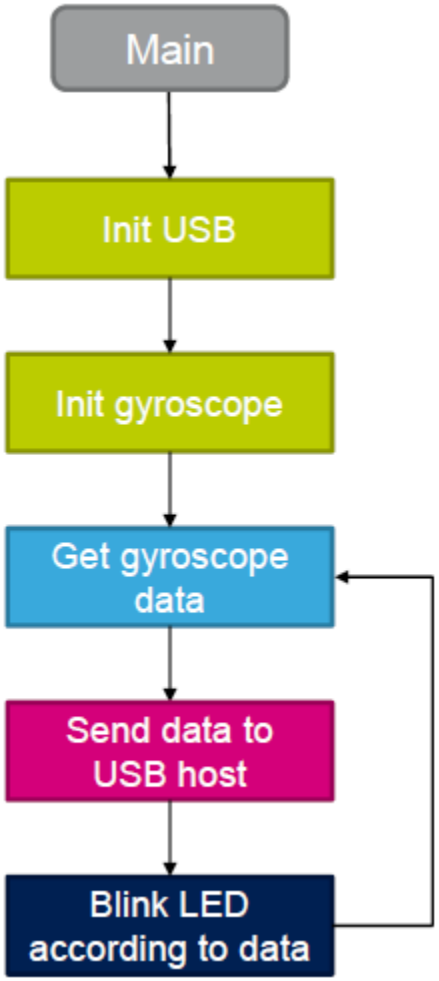
Mutex

- Về cơ bản thì Mutex giống như binary semaphore nhưng được sử dụng cho việc loại trừ chứ không phải đồng bộ. 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, cơ chế này có thể hiểu đơn giản qua ví dụ sau:
  - 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.

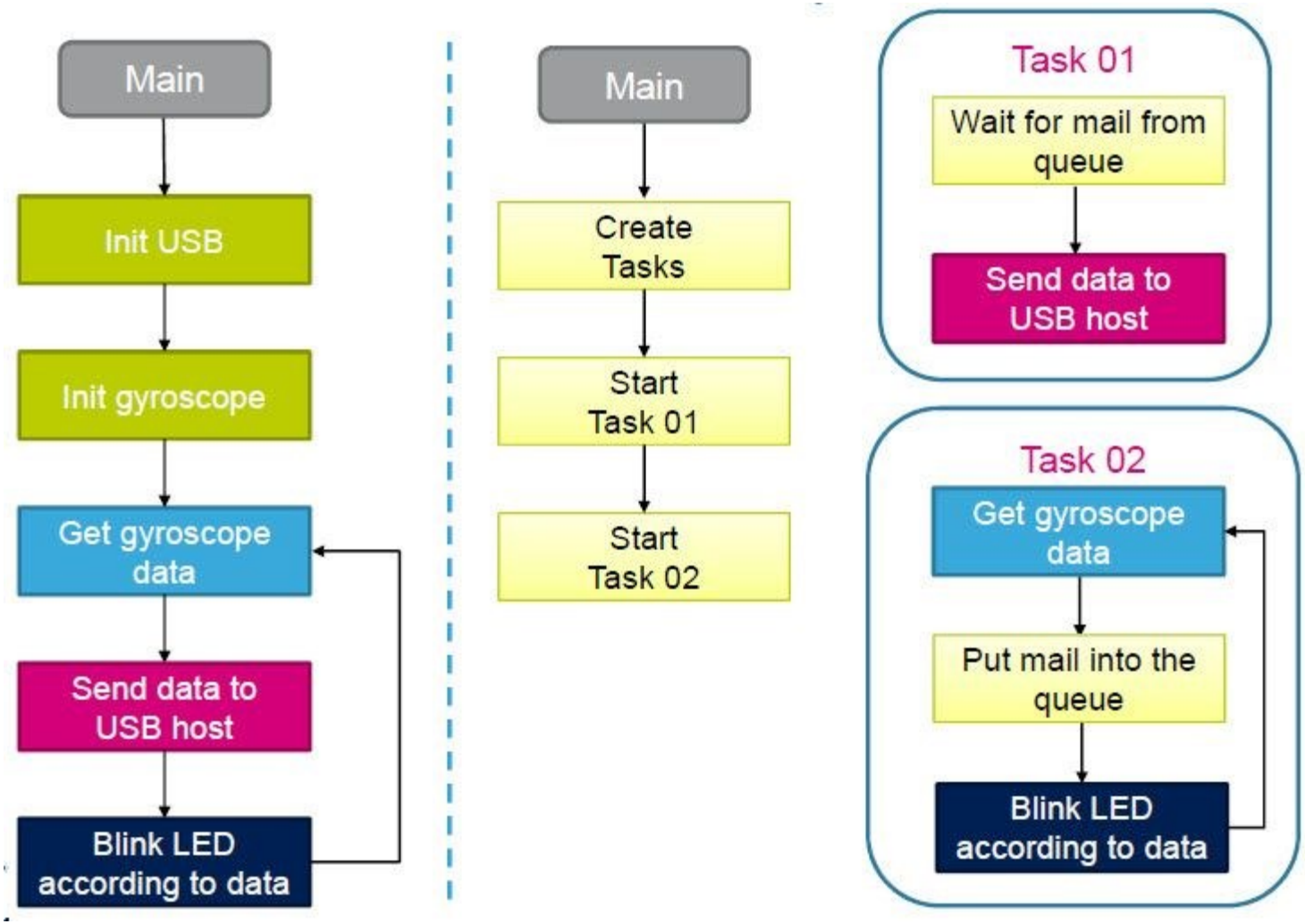
Bài toán minh hoạ



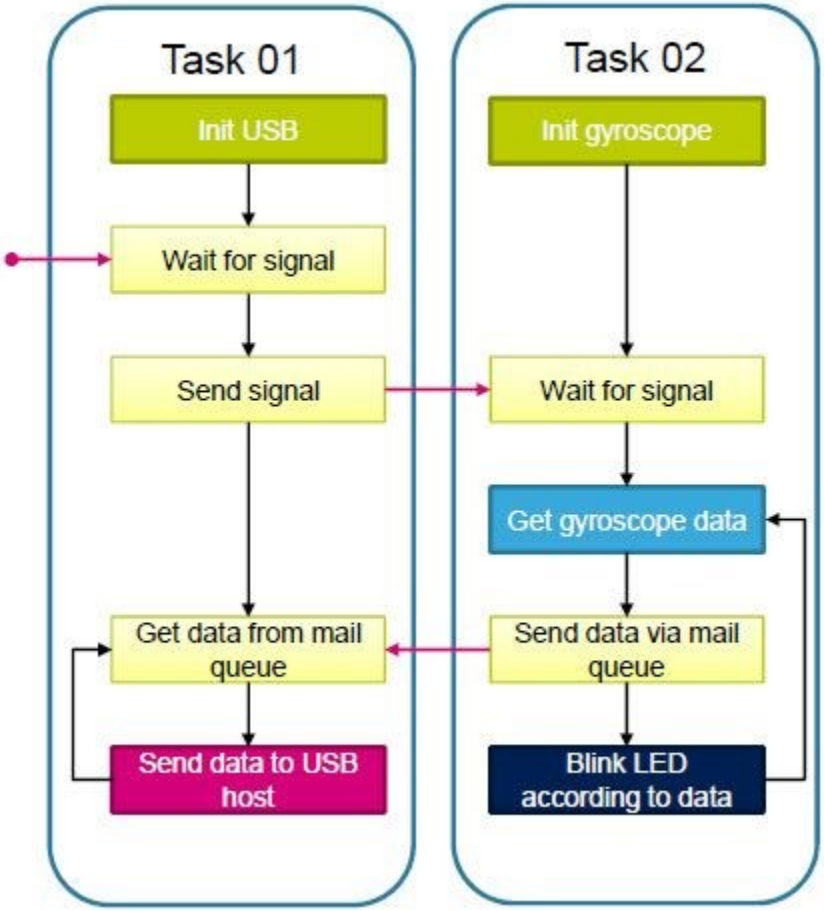
Bài toán minh hoạ



Bài toán minh hoạ



Bài toán minh hoạ



Khung mẫu chương trình CCS-RTOS

```
#include <16F877A.H>
#device *=8 ADC=10
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)
// function declarations
#task(rate=1000ms,max=100ms)
        void The_first_rtos_task ( );
#task(rate=500ms,max=100ms)
        void The_second_rtos_task ( );
#task(rate=100ms,max=100ms)
        void The_third_rtos_task ( );
// more function declarations
```

```
// function implementations
void The_first_rtos_task ( )
{
  // task code
}
void The_second_rtos_task ( )
{
  // task code
}
void The_third_rtos_task ( )
{
  // task code
}
// more function implementations
void main ( )
{
  // initialization code for other resources
  rtos_run ( );
}
```

Task - RTOS

```
1. #task(rate=1s,max=20ms,queue=5)
2. void task_name() {
3. // task code
4. }
```

Trình biên dịch PIC C CCS hỗ trợ RTOS

- **rtos_run ()** khởi tạo các hoạt động của RTOS. Tất cả các hoạt động kiểm soát tác vụ được thực thi sau khi gọi hàm này
- **rtos_terminate ()** Chức năng này kết thúc việc thực thi tất cả các tác vụ RTOS. Quyền kiểm soát được trả về chương trình gốc không có RTOS.
- **rtos_enable ()** nhận tên của một tác vụ như là một đối số. Hàm này cho phép một tác vụ thực thi tại tốc độ chỉ định. Tất cả tác vụ được cho phép theo mặc định.
- **rtos_disable()** nhận tên của tác vụ như là một đối số. Hàm này vô hiệu hóa tác vụ để nó không còn được gọi bởi rtos_run() trừ khi nó được cho phép lại bằng hàm rtos_enable ()

Trình biên dịch PIC C CCS hỗ trợ RTOS

- **rtos_ yield()** khi gọi từ bên trong một tác vụ, trả lại điều khiển cho phần tiếp nhận. Tất cả các tác vụ nên gọi hàm này để giải phóng vi xử lý để các tác vụ khác có thể sử dụng thời gian để xử lý tiếp\
- **rtos_msg_send()** nhận tên tác vụ và 1 byte làm đối số. Hàm này gửi byte đến tác vụ chỉ định, nó được đặt trong hàng đợi thông điệp của tác vụ.
- **rtos_msg_read()** đọc byte đặt trong hàng đợi thông điệp của tác vụ
- **rtos_msg_poll ()** trả về true nếu có dữ liệu trong hàng đợi thông điệp của tác vụ. Hàm này nên gọi trước khi đọc 1 byte từ hàm đợi thông điệp của tác vụ
- **rtos_signal ()** nhận một tên cờ hiệu (semaphore) và tăng cờ hiệu.

Trình biên dịch PIC C CCS hỗ trợ RTOS

- **rtos_wait()** nhận một tên semaphore và chờ tài nguyên có liên quan đến cờ hiệu có giá trị. Giá trị cờ hiệu giảm để cho tác vụ có thể chờ tài nguyên
- **rtos_await()** Chức năng này chỉ có thể được sử dụng trong một tác vụ RTOS. Chức năng này chờ biểu thức là đúng trước khi tiếp tục thực hiện phần mã còn lại của của tác vụ RTOS. Chức năng này cho phépcác tác vụ khác thực hiện nhiệm vụ trong khi chờ đợi cho biểu thức là đúng.
- **rtos_overrun ()** nhận một tên nhiệm vụ như một tham số, và hàm trả về true nếu công việc đó đã bị quáthời gian được phân bổ.
- **rtos_stats ()** trả về số liệu thống kê cụ thể về một nhiệm vụ cụ thể. Các số liệu thống kê thời gian chạynhiệm vụ tối thiểu và tối đa và tổng thời gian tác vụ chạy. Tên tác vụ và các loại thống kê được quy định như đối số cho hàm.

Trình biên dịch PIC C CCS hỗ trợ RTOS

- Ngoài các chức năng trên, các lệnh tiền xử lý rtos #use () phải được quy định tại phần đầu của chương trình trước khi gọi bất kỳ chức năng RTOS. Các định dạng của lệnh tiền xử lý này là:

#use rtos(timer = n,minor_cycle=m)

với:

- timer được gán từ 0 đến 4 và xác định bộ định thời được sử dụng bởi RTOS
- minor_cycle là thời gian dài nhất một tác vụ bất kỳ sẽ chạy: s, ms, us, ns.

- Ngoài ra, một tùy chọn statistics (tùy chọn thống kê), khai báo sau tùy chọn minor_cycle, khi đó chương trình biên dịch sẽ theo dõi thời gian tối đa và tối thiểu tác vụ sử dụng tại mỗi lần được gọi và tổng thời gian tác vụ sử dụng

Khai báo tác vụ

#task(rate=n, max=m, queue=p)

- Rate chỉ định chu kỳ mà tác vụ được gọi, đi kèm theo sau là đơn vị s, ms, us, hoặc ns.
- max <=minor cycle: thoi gian thuc thi task
- queue là một tùy chọn, nó quy định số byte để dành cho tác vụ để nhận thông điệp từ tác vụ khác. Giá trị mặt định là 0.
- Ví dụ minh họa dưới đây, một tác vụ gọi my_ticks sau mỗi 20 ms, và quy định không sử dụng quá 100 ms thời gian xử lý, và không sử dụng tùy chọn queue.

```
#task(rate=20ms,max=100ms)
void  my_ticks()
  {.....................}
```

Ví dụ 1 - LEDs

- Ví dụ đơn giản dựa trên RTOS, bốn LED đơn kết nối 4 chân thấp PORTB của vi điều khiển PIC16F877A. Phần mềm bao gồm 4 tác vụ, mỗi tác vụ có nhiệm vụ điều khiển nhấp nháy một đèn led với tốc độ khác nhau.
  - Tác vụ 1 , gọi task_B0, nhấp nháy led kết nối đến port RB0 tại tốc độ 250ms.
  - Tác vụ 2 , gọi task_B1, nhấp nháy led kết nối đến port RB1 tại tốc độ 500ms.
  - Tác vụ 3, gọi task_B2, nhấp nháy led kết nối đến port RB2 mỗi một giây.
  - Tác vụ 4, gọi task_B3, nhấp nháy led kết nối đến port RB3 mỗi 2 giây.

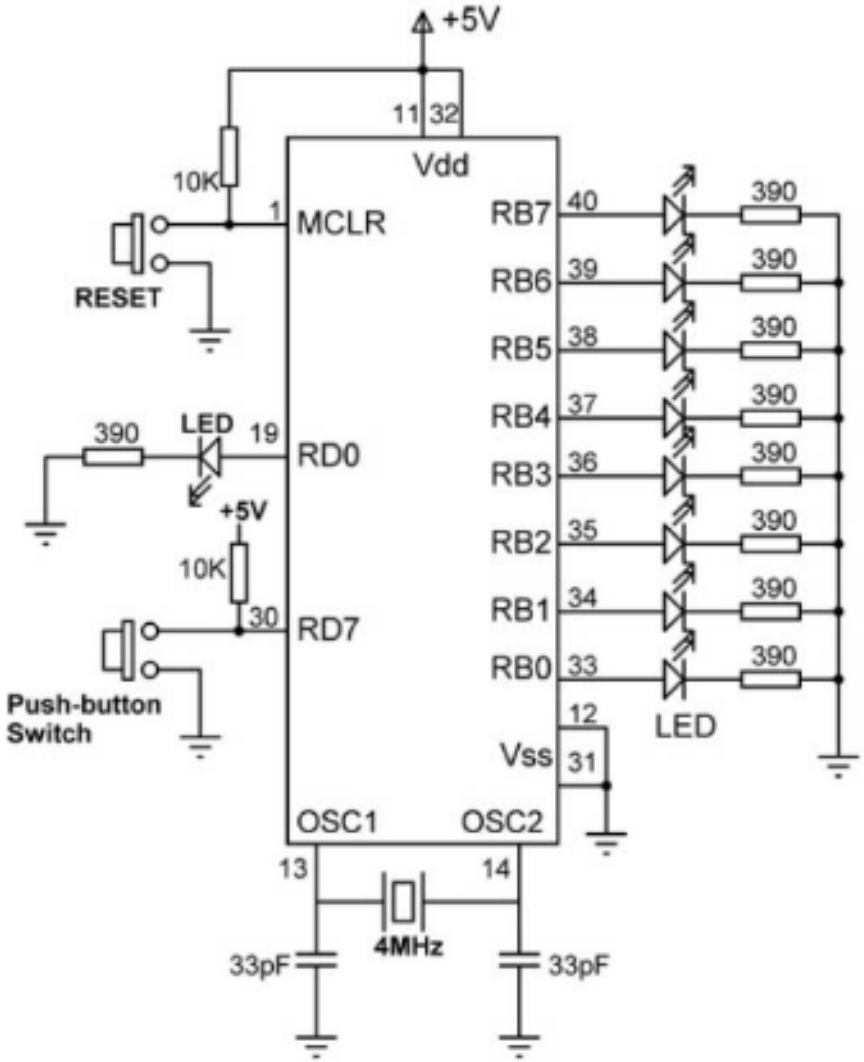
Ví dụ 1 - LEDs

```
#include “16f877A.h"
#use delay (clock=4000000)
// Define which timer to use and minor_cycle
for RTOS
#use rtos(timer=0, minor_cycle=10ms)
// Declare TASK 1 - called every 250ms
#task(rate=250ms, max=10ms)
    void task_B0()
   {
   output_toggle(PIN_B0); // Toggle RB0
   }
// Declare TASK 2 - called every 500ms
#task(rate=500ms, max=10ms)
    void task_B1()
    {
    output_toggle(PIN_B1); // Toggle RB1
    }
```

```
// Declare TASK 3 - called every second
#task(rate=1s, max=10ms)
   void task_B2()
   {
   output_toggle(PIN_B2); // Toggle RB2
    }
// Declare TASK 4 - called every 2 seconds
#task(rate=2s, max=10ms)
    void task_B3()
    {
    output_toggle(PIN_B3); // Toggle RB3
     }
// Start of MAIN program
void main()
{
   set_tris_b(0); // PORTB as outputs
   rtos_run(); // Start RTOS
}
```

Ví dụ 2:

- Task1:Sau mỗi 200 ms nhấp nháy đèn LED kết nối đến chân RD0
- Task*3:* tăng biến từ 0 đến 255 và kiểm trạng thái của nút nhấn. Khi nút nhấn được nhấn giá trị hiện tại của bộ đếm được gửi đến task*2* sử dụng hàng đợi thông điệp.
- Task2: đọc số từ hàng đợi thông điệp và gửi byte nhận đến PORTB. Do đó khi nút nhấn được nhấn thì trạng thái các led kết nối với PORTB thay đổi tương ứng với số ngẫu nhiên.



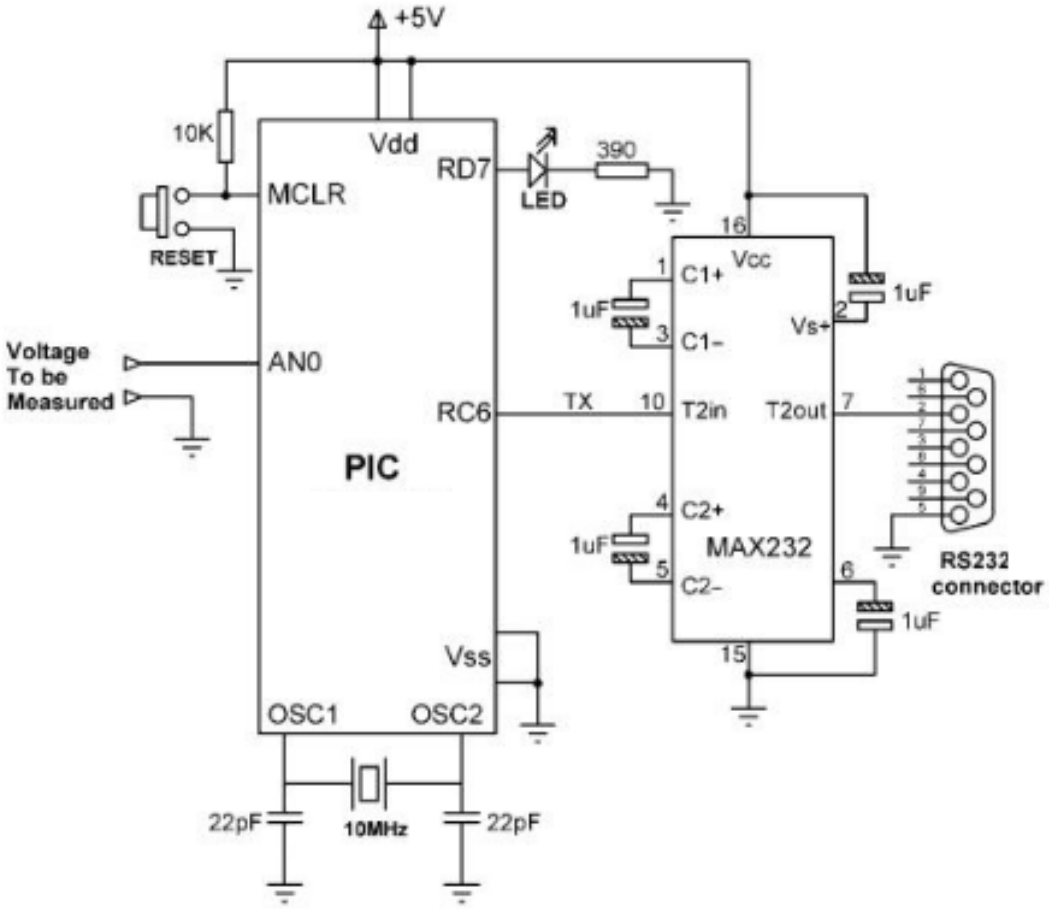
Ví dụ 2

```
#include “16f887A.h"
#use delay (clock=4000000)
int count;
#use rtos(timer=0, minor_cycle=1ms)
// This task flashes the LED on RD0 200ms
#task(rate=200ms, max=1ms)
void task_1()
{output_toggle(PIN_D0);}
// TASK "Display" - called every 10ms
#task(rate=10ms, max=1ms, queue=1)
void task_2()
{
if(rtos_msg_poll() > 0)
// Is there a message ?
{
output_b(rtos_msg_read()); // Send to
PORTB}
}
```

```
// TASK "Generator" - called every ms
#task(rate=1ms, max=1ms)
void task_3()
{
count++; // Increment count
if(input(PIN_D0) == 0) // Switch pressed ?
{rtos_msg_send(task_2,count); // send a
message
}
}
// Start of MAIN program
void main()
{
set_tris_b(0); // Configure PORTB as outputs
set_tris_d(0x10); // RD7=input, RD0=output
rtos_run(); // Start RTOS
}
```

Ví dụ 3: Đo điện áp và xuất đến cổng RS232

- Task *Live* chạy mỗi 20 ms, nhấp nháy đèn led kết nối đến port RD7 của vi điều khiển để báo rằng hệ thống đang hoạt động.
- Task *Get_Voltage* đọc kênh 0 của bộ chuyển đổi A/D, nơi kết nối đến nguồn điện áp cần đo. Giá trị đọc sẽ được định dạng và được lưu trong biến. Tác vụ này chạy sau mỗi 2 giây.
- Task *To_RS232* đọc giá trị điện áp đo được và gửi đến cổng nối tiếp đến máy tính sau mỗi một giây.



Ví dụ 3: Đo điện áp và xuất đến cổng RS232

- Trong chương trình chính PORTD cấu hình như là port xuất, và các chân được xóa về mức 0, PORT A thiết lập như là ngõ nhập, và cấu hình như là ngõ nhập analog, thiết lập xung clock A/D, chọn kênh AN0, khi đó chạy RTOS bằng cách gọi hàm rtos_run().
- Chương trình bao gồm ba tác vụ:
  - Task *Live* chạy mỗi 20 ms, nhấp nháy đèn led kết nối đến port RD7 của vi điều khiển để báo rằng hệ thống đang hoạt động.
  - Task G*et_voltage* đọc điện áp analoge từ kênh 0 (chân RA0 hay AN0) của vi điều khiển. giá trị khi đó chuyển thành miivoltage bởi nhân cho 5000 và chia cho 1024. Điện áp lưu trữ ở biến toàn cục tên Volts.
  - Task *To_RS232* đọc điện áp lưu ở biến Volts và gửi đến port RS232 bằng cách sử dụng lệnh printf . Kết quả được gửi đến máy tính theo định dạng:

    Measuredvoltage = nnnn mV

Ví dụ 3: Đo điện áp và xuất đến cổng RS232

```
#include <18F8520.h>
#device adc=10
#use delay (clock=10000000)
#use rs232(baud=2400,xmit=PIN_C6,rcv=PIN_C7)
unsigned int16 adc_value;
unsigned int32 Volts;
#use rtos(timer=0, minor_cycle=100ms)
// This task flashes the LED on RD7 200ms
#task(rate=200ms, max=1ms)
void Live()
{output_toggle(PIN_D7); // Toggle RD7 LED
}
//TASK "Get_voltage" - called every 10ms
#task(rate=2s, max=100ms)
void Get_voltage()
{
adc_value = read_adc(); // Read A/D value
Volts = (unsigned int32)adc_value*5000;
Volts = Volts / 1024; // Voltage in mV
}
```

```
// TASK "To_RS232"- called every ms
#task(rate=2s, max=100ms)
void To_RS232()
{
printf("Measured Voltage =
%LumV\n\r",Volts); // send to RS232
}
void main()
{
set_tris_d(0); // PORTD all outputs
output_d(0); // Clear PORTD
set_tris_a(0xFF); // PORTA all inputs
setup_adc_ports(ALL_ANALOG); // A/D
ports
setup_adc(ADC_CLOCK_DIV_32); // A/D
clock
set_adc_channel(0); // Select channel 0 (AN0)
delay_us(10);
rtos_run(); // Start RTOS
}
```

Ví dụ 3: Đo điện áp và xuất đến cổng RS232

Sử dụng Semaphore:

- Chương trình ở trên làm việc và hiển thị điện áp đo lường trên màn hình máy tính. Chương trình có thể cải thiện bằng cách sử dụng Semaphore để đồng bộ hiển thị điện áp đo lường với bộ lấy mẫu A/D. Hoạt động của chương trình mới được hoạt động như sau:
- Biến semaphore (sem) được bật bằng 1 tại lúc bắt đầu chương trình
- Task *Get_voltage* giảm biến semaphore (calls rtos_wait) để task To_RS232 bị khóa (biến semaphore sem=0) và không thể gửi dữ liệu đến PC. Khi lấy mẫu A/D mới đã sẵn sàng biến semaphore được tăng lên (calls rtos_signal) và Task To_RS232 có thể tiếp tục. Khi đó Task To_RS232 gửi điện áp đến PC và tăng biến semaphore để chỉ ra rằng nó có quyền truy cập dữ liệu. Task *Get_voltage* có thể thực hiện một lấy mẫu mới. Tiến trình lặp đi lặp lại liên tục.

Ví dụ 3: Đo điện áp và xuất đến cổng RS232

```
#include <18F8520.h>
#device adc=10
#use delay (clock=10000000)
#use rs232(baud=2400,xmit=PIN_C6,rcv=PIN_C7)
unsigned int16 adc_value;
unsigned int32 Volts;
int8 sem;
#use rtos(timer=0, minor_cycle=100ms)
#task(rate=200ms, max=1ms)
void Live()
{output_toggle(PIN_D7); // Toggle RD7 LED
}
#task(rate=2s, max=100ms)
void Get_voltage()
{rtos_wait(sem); // decrement semaphore
adc_value = read_adc(); // Read A/D value
Volts = (unsigned int32)adc_value*5000;
Volts = Volts / 1024; // Voltage in mV
rtos_signal(sem); // increment semaphore
}
#task(rate=2s, max=100ms)
void To_RS232()
{rtos_wait(sem); // Decrement semaphore
printf("Measured Voltage =
%LumV\n\r",Volts); // Send to RS232
rtos_signal(sem); // Increment semaphore
}
void main()
{set_tris_d(0); // PORTD all outputs
output_d(0); // Clear PORTD
set_tris_a(0xFF); // PORTA all inputs
setup_adc_ports(ALL_ANALOG); // A/D
ports
setup_adc(ADC_CLOCK_DIV_32); // A/D
clock
set_adc_channel(0); // Select channel 0 (AN0)
delay_us(10);
sem = 1; // Semaphore is 1
rtos_run(); // Start RTOS
}
```

Câu 1: (3 điểm)

Giả sử Hệ thống điều khiển tự động có yêu cầu ghép nối với vi điều khiển, tín hiệu điều khiển từ 0-12V; đo lường từ 0-5V; giá trị đặt điều khiển 0-5V. Hãy phân tích các yêu cầu phần cứng và vẽ sơ đồ khối phần cứng hệ thống nhúng đảm bảo cho bài toán?

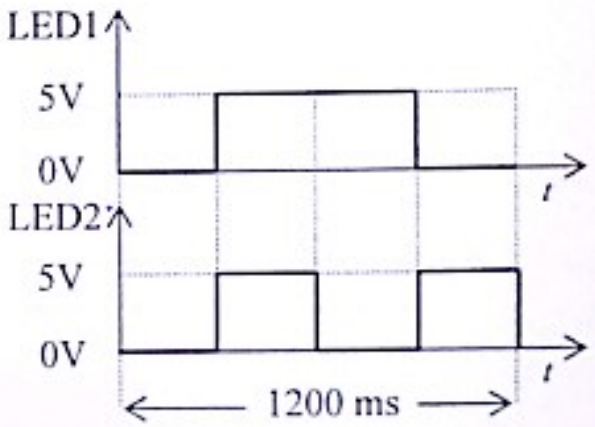
Câu 2: (4 điểm)

Cho sơ đồ điều khiển 02 đèn LED (2.2V, 10mA) như hình vẽ, Anh (Chị) hãy :

a) Thiết kế mạch điều khiển thực hiện nhiệm vụ trên? *(1 điểm)*

b) Tính toán, giải thích các giá trị linh kiện trên mạch? *(1 điểm)*

c) Viết chương trình trên Vi điều khiển thực hiện nhiệm vụ trên? *(2 điểm)*



Câu 3: (3 điểm)

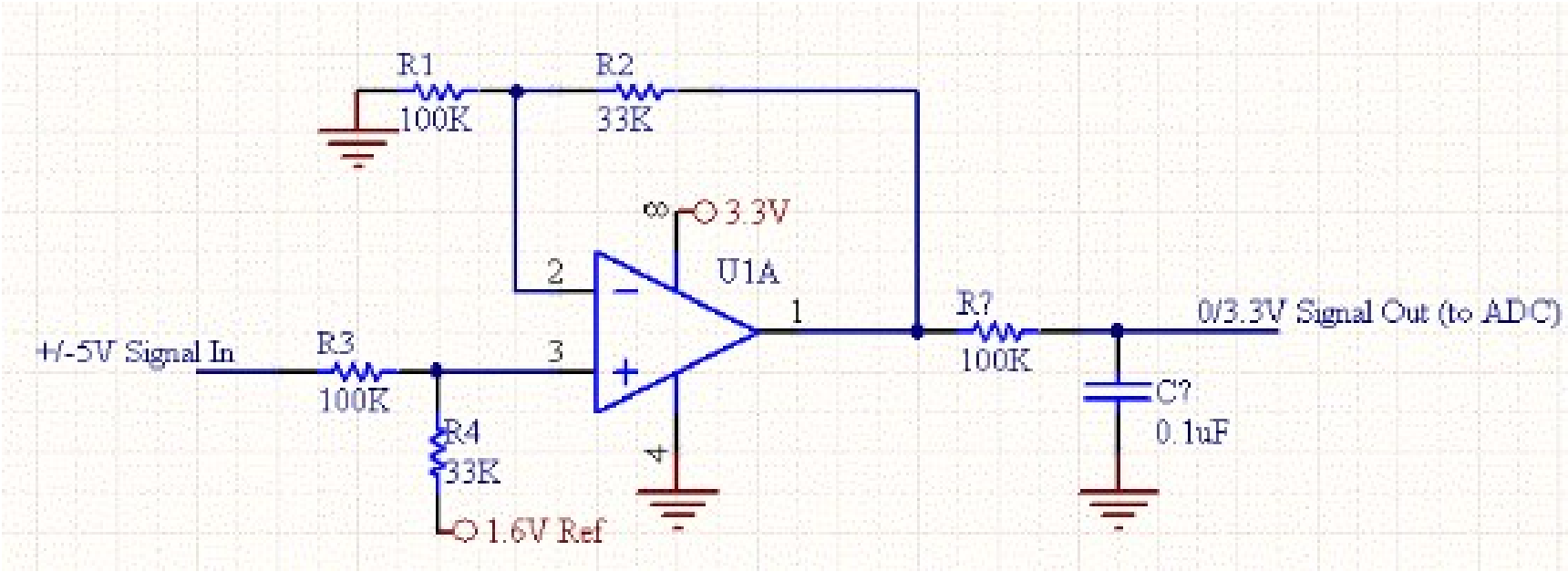
Hãy thiết kế hệ thống điều khiển nhúng trên Vi xử lý bất kỳ, sử dụng cảm biến nhiệt độ LM35, LED 7 thanh, Relay 5V và các linh kiện điện tử cơ bản thực hiện nhiệm vụ sau:

a) Tính toán, giải thích các giá trị linh kiện trên mạch? *(1 điểm)*

b) Hiển thị giá trị nhiệt độ đo được lên màn hình LED 7 thanh (4 số)? *(1 điểm)*

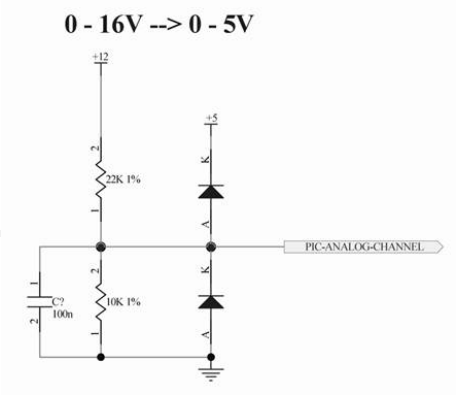
c) Hệ thống đóng 01 Relay khi nhiệt độ nằm trong khoảng từ 85^0C tới 130^0C. *(1 điểm)*

Non-Inverting Op-Amp Level Shifter

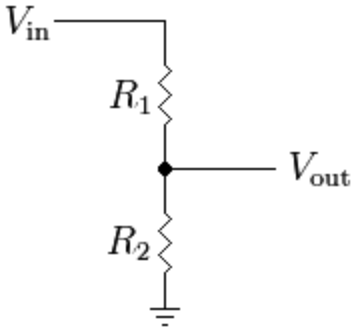


Non-Inverting Op-Amp Level Shifter

- A common engineering task is to convert a positive to negative signal into a range suitable for a single supply ADC. This circuit will convert a +/-5V signal into a 0 to 3.3V signal so that it can be sampled by an ADC on a 3.3V microprocessor.
- **Equations:**
- A = (R4/R1) x (R1+R2)/(R3+R4)
- If R1= R3, and R2=R4, then A= (R4/R1)
- We want to convert a 10Vpp signal to a 3.3V signal so the gain should be 1/3. We can choose R4 to be 33K and R1 to be 100K.
- Now we need to choose the positive offset such that the signal is centered at 1.6V.
- The gain off the offset voltage is:
- A_{offset}= (R2+R1)/R1 x R3/(R3+R4) = R3/R1.
- For the previous resistor values, the gain is 1 since R3=R1, and so we use an offset voltage of 1.6V.



- A simple resistive voltage divider will achieve what you want.
- The formula to calculate the output voltage is:
- So if we assume your input voltage ranges from 0-50V, we need to divide it by 10 to achieve 0-5V. If we also assume we want to load the input voltage with 100kΩ, then the calculations would something like:



- Vout / Vin = R2 / 100kΩ
- 0.1 = R2 / 100kΩ -> R2 = 10kΩ
- R1 = 100kΩ - R2 = 90kΩ
- So R1 = 90kΩ and R2 = 10kΩ
- For an ADC requiring a maximum source impedance, you must make sure the voltage divider impedance is below this level. The impedance at the divider can be calculated as R1||R2.
- For <2.5kΩ, the above won't meet this requirement as 10kΩ||90kΩ = 9kΩ If we use 9kΩ and 1kΩ though, we get 1 / (1/1000 + 1/9000) = 900Ω
- Bear in mind the lower the resistance the higher the wattage rating resistors you need. 50V / 1k = 50mA -> 50mA * 45V = 2.25W across the top resistor (0.25W across the bottom) In these cases it's best to use an opamp buffer in between a high resistance divider and the ADC. Or use a 2kΩ and 18kΩ divider, which is not quite as power hungry as the 1k/9k version.

Cho sơ đồ điều khiển 02 Relay 5V như hình vẽ, Anh (Chị) hãy :

a) Thiết kế mạch điều khiển thực hiện nhiệm vụ trên? *(1 điểm)*

b) Tính toán, giải thích các giá trị linh kiện trên mạch? *(1 điểm)*

c) Viết chương trình trên Vi điều khiển thực hiện nhiệm vụ trên? *(2 điểm)*