01076001
วิศวกรรมคอมพิวเตอร์เบื้องต้น
Introduction to Computer Engineering

Arduino #8

Realtime OS #1

Operating Systems

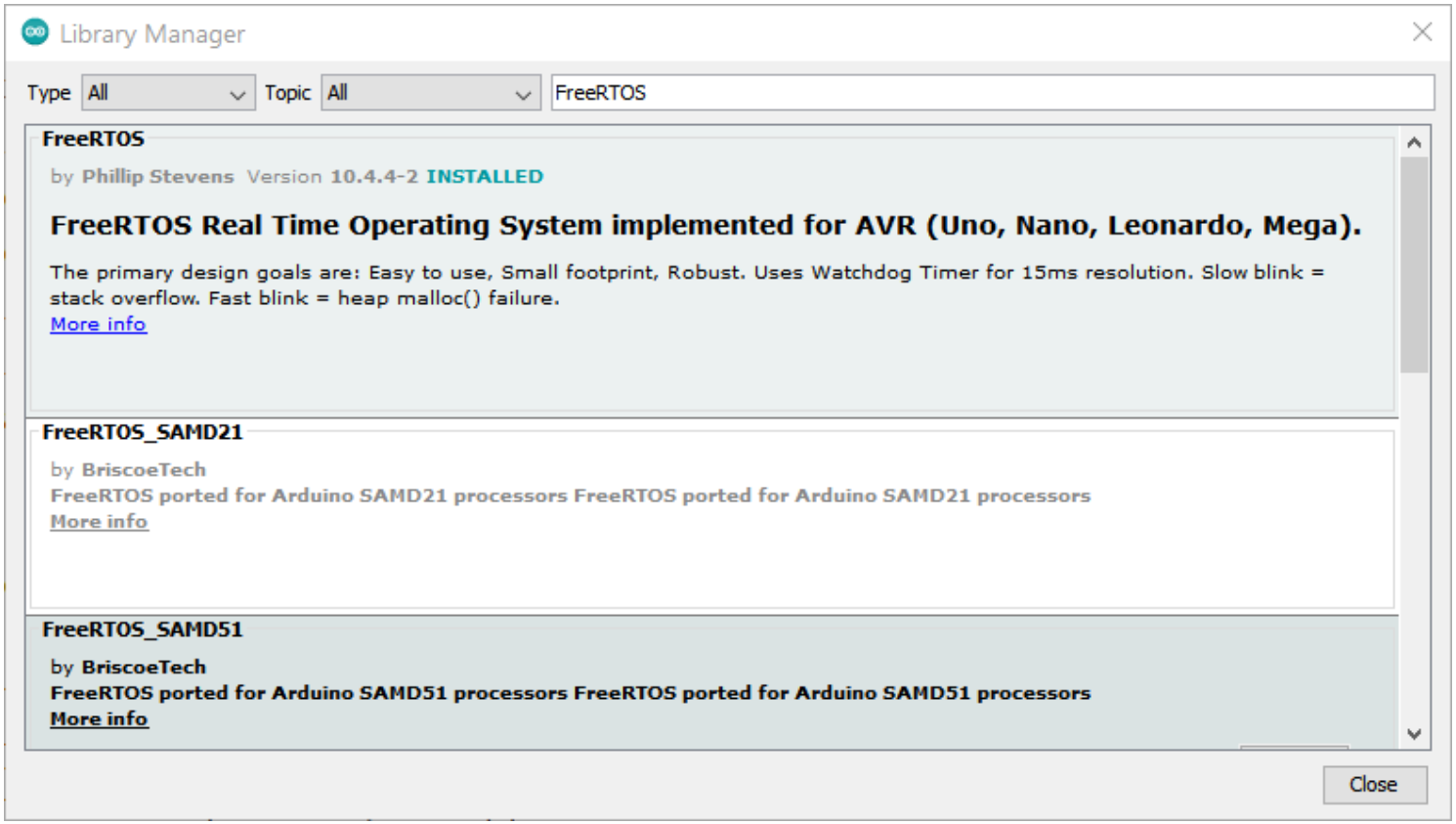
- ที่ผ่านมา เราพัฒนาโปรแกรมและโหลดลงในบอร์ด Arduino ในลักษณะที่เรียกว่า Bare Metal คือ โปรแกรมที่เขียนจะเป็นผู้ครอบครองทรัพยากรของระบบแต่ผู้เดียว แต่ในคอมพิวเตอร์ในปัจจุบัน มักจะใช้งานระบบปฏิบัติการ เพื่อให้การเขียนโปรแกรมสะดวกขึ้น
- โดยเราจะทดลองนำระบบปฏิบัติการขนาดเล็กมาใช้กับ Arduino ชื่อว่า FreeRTOS
- FreeRTOS พัฒนาขึ้นมาโดยบริษัท Real Time Engineer โดย FreeRTOS เป็นระบบปฏิบัติการที่ออกแบบมาสำหรับไมโครคอนโทรลเลอร์หรือไมโครโพรเซสเซอร์เล็กๆ เพื่อการใช้งานแบบ Multitasking

FreeRTOS

- FreeRTOS เป็น Realtime OS (Realtime แปลเป็นภาษาไทย คือ เวลาจริง แต่ความหมายที่ใช้งานกันทั่วไป คือ ระบบปฏิบัติการที่ต้องมีการตอบสนองต่อ Input ไม่เกินค่าเวลาที่กำหนด ซึ่งระบบปฏิบัติการทั่วไป ไม่สามารถทำได้)
- ในโลกของ Embedded System การตอบสนองที่ไม่เกินเวลานี้มีความสำคัญมาก เพราะหากล่าช้าไปอาจสร้างความเสียหายได้ ลองนึกถึงระบบควบคุมในรถยนต์ ที่เมื่อเหยียบเบรคแล้ว รถยนต์มีการเบรคล่าช้าไป แม้จะเสี้ยววินาทีก็ตาม อะไรจะเกิดขึ้น

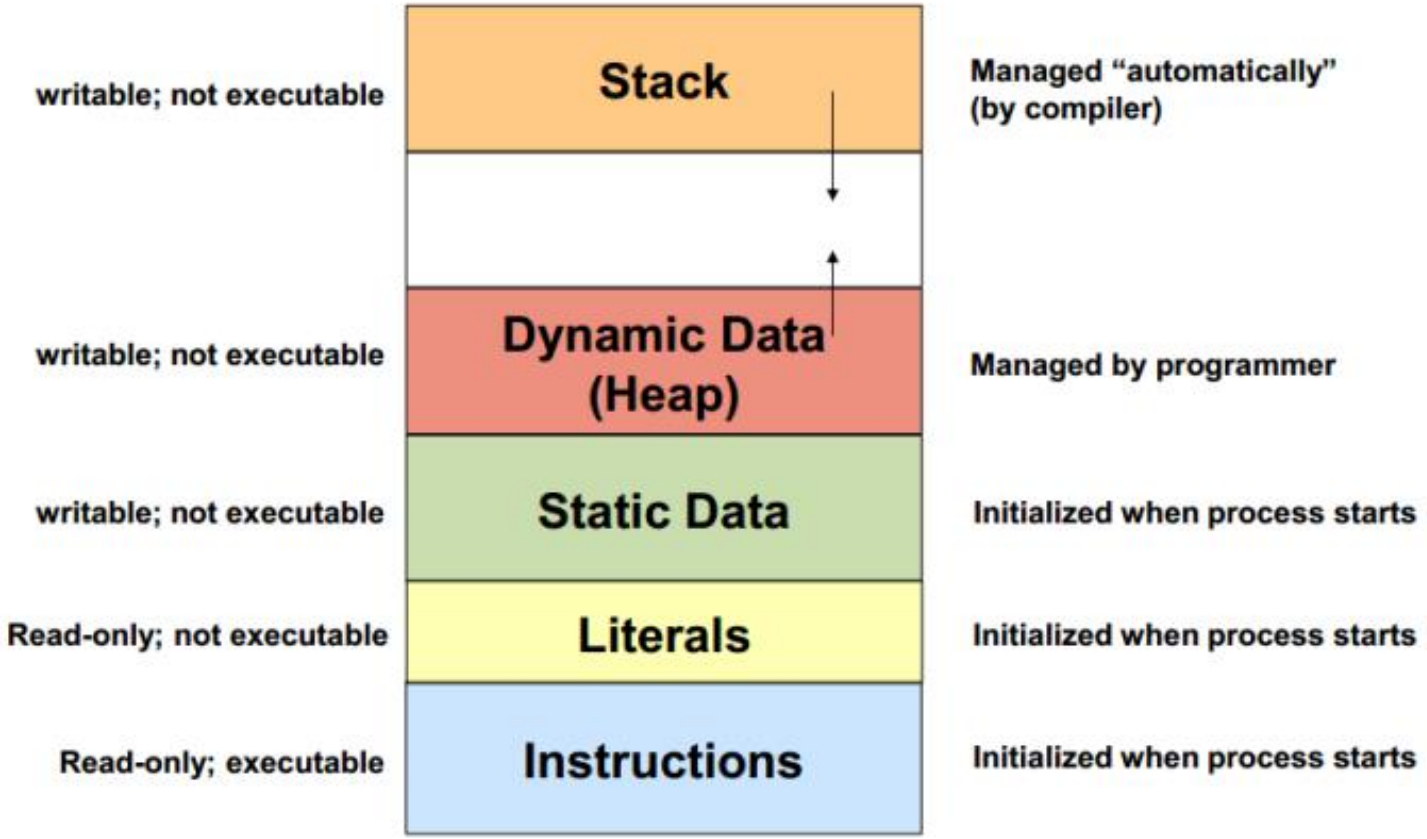
FreeRTOS

- การติดตั้ง FreeRTOS ใน Arduino จะไม่เหมือนกับการในเครื่อง PC ที่เป็นการติดตั้ง OS ลงไปก่อน แต่จะเป็นการโหลด OS พร้อมโปรแกรมลงไป ติดตั้ง Lib ตามรูป



Task Structure

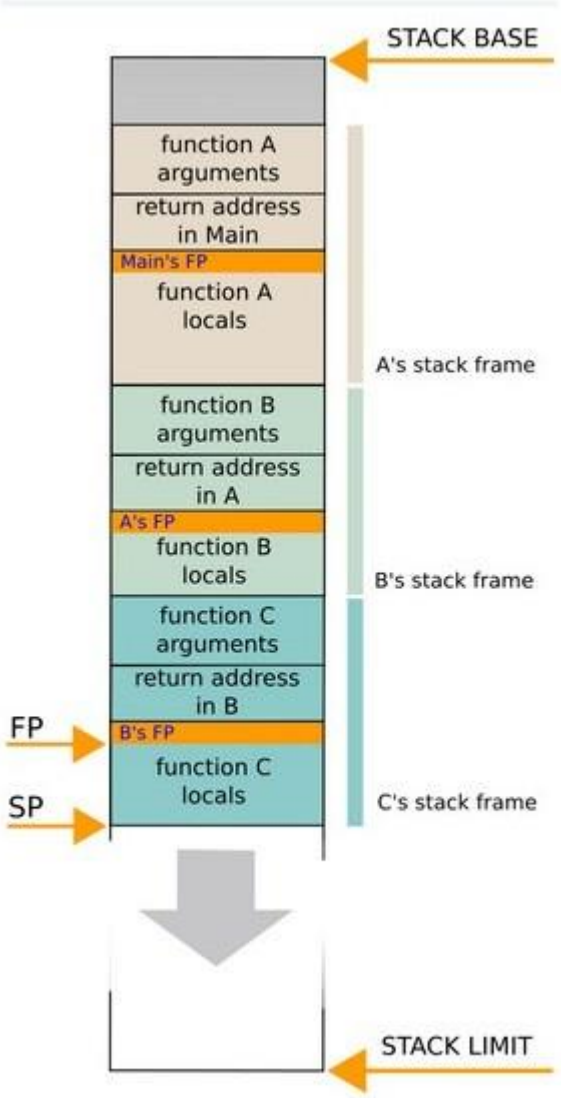
- โครงสร้างของ Task



Task Structure

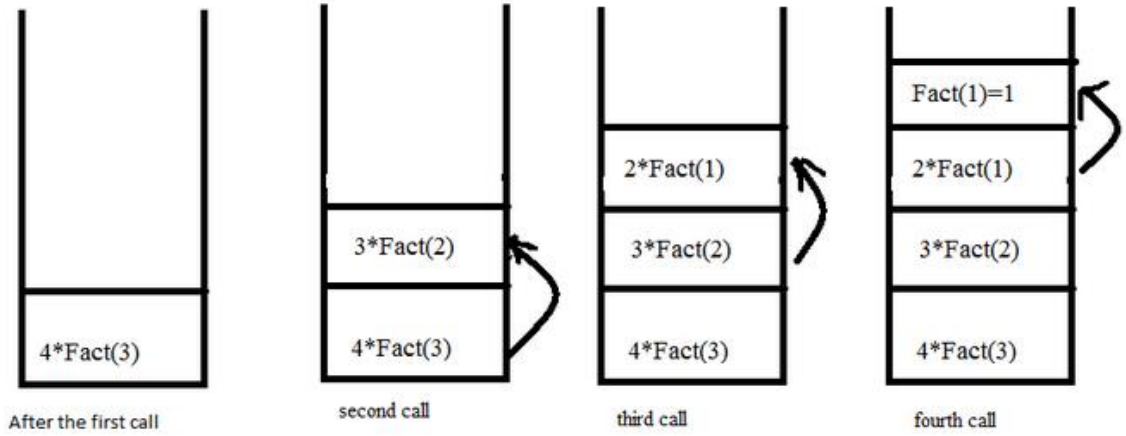
- โครงสร้างของ Task
  - Instruction เป็นส่วนของโปรแกรมที่เขียนขึ้น และ แปลเป็นภาษาเครื่องโดย Compiler
  - Literals เป็นส่วนข้อมูลที่มีลักษณะเป็นค่าคงที่ คือ อ่านอย่างเดียว
  - Static Data เป็นข้อมูลที่เป็นตัวแปรชนิด Global โดยมีการกำหนดล่วงหน้าแล้วว่าต้องมีตำแหน่งที่เก็บโปรแกรม
  - Heap เป็นข้อมูลที่สร้างขึ้นภายหลังที่โปรแกรมเริ่มทำงาน เช่น Objects ต่างๆ
  - Stack เป็นข้อมูลประเภท Local และเก็บตำแหน่งที่ Return กลับ

FreeRTOS

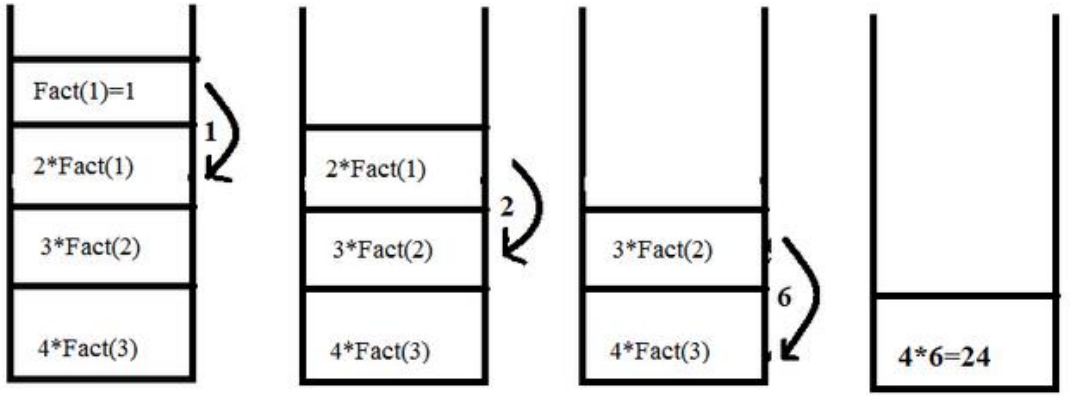



```
long factorial(int n) {
    if (n>=1)
        return n* factorial(n-1);
    else
        return 1;
}
```

When function call happens previous variables gets stored in stack

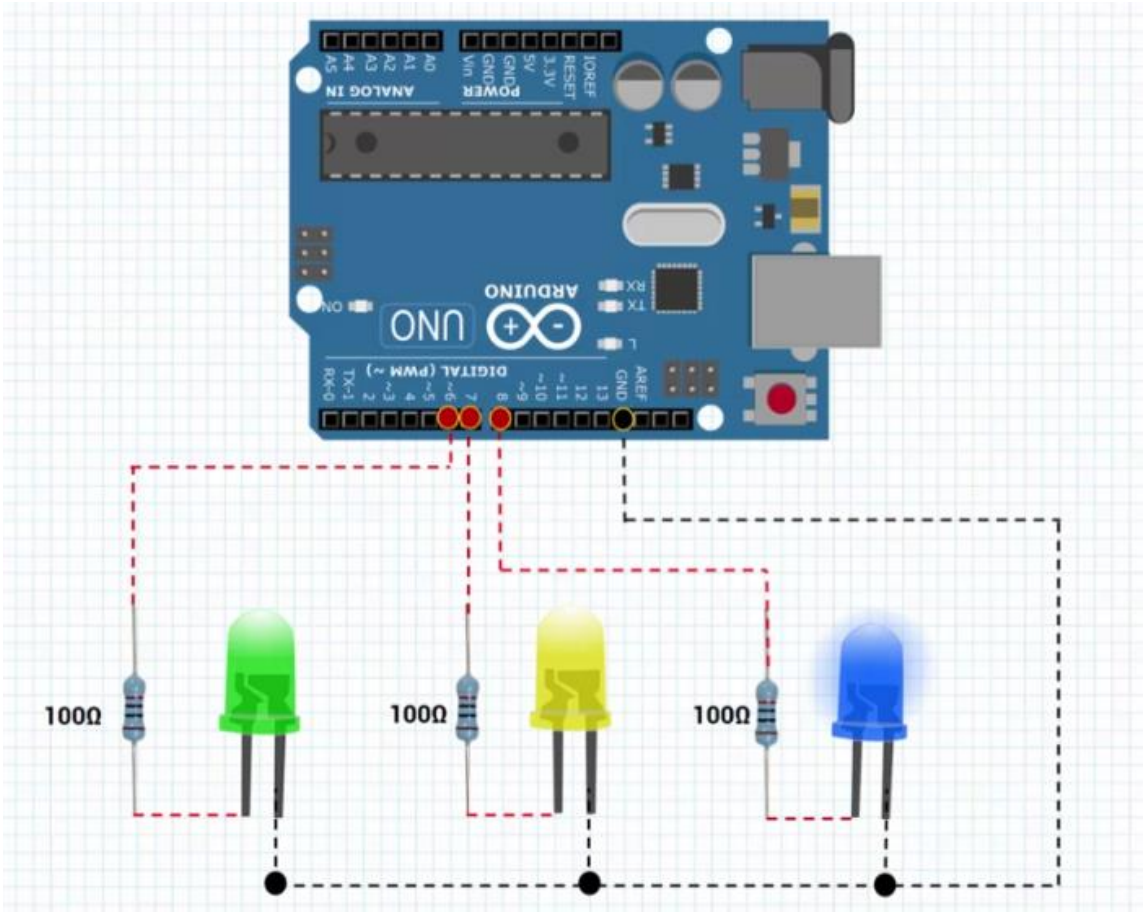


Returning values from base case to caller function



FreeRTOS

- ต่อ LED ตามรูป (ต่อขาอื่นก็ได้)



FreeRTOS : xTaskCreate

- ใน FreeRTOS จะเรียกแต่ละงานที่ทำว่า Task เช่น หากจะทำให้ LED แต่ละดวงกระพริบ จะเรียกแต่ละงานว่า Task เนื่องจากมี LED จำนวน 3 ดวง จึงเป็น 3 Task
- ฟังก์ชันหลักที่เราจะใช้มีชื่อว่า xTaskCreate ซึ่งมีรูปแบบการใช้ คือ

```
BaseType_t xTaskCreate(     TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            configSTACK_DEPTH_TYPE usStackDepth,
                            void *pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t *pxCreatedTask
                          );
```

FreeRTOS : xTaskCreate

- xTaskCreate มีพารามิเตอร์ ดังนี้
  - ตัวที่ 1 ตำแหน่งฟังก์ชันที่จะเรียกขึ้นมาทำงานเป็น Task หนึ่งในระบบ
  - ตัวที่ 2 เป็นชื่อเรียกของ Task จะเห็นว่ามีลักษณะเป็นข้อความ
  - ตัวที่ 3 เป็นขนาดของ Stack ที่จะจองให้ Task นั้นใช้งาน
  - ตัวที่ 4 เป็นพารามิเตอร์ที่จะส่งเข้าไปใน Task
  - ตัวที่ 5 เป็น Priority หรือลำดับความสำคัญของงาน ซึ่งสามารถจะกำหนดให้แต่ละงานมีลำดับความสำคัญไม่เท่ากันก็ได้
  - ตัวที่ 6 เป็น Task Handle จะกล่าวถึงรายละเอียดภายหลัง

FreeRTOS : Example 1

```
#include <Arduino_FreeRTOS.h>

#define RED     6
#define YELLOW  7
#define BLUE    8

void setup() {
  xTaskCreate(redLedControllerTask, "RED LED Task", 128, NULL, 1, NULL);
  xTaskCreate(blueLedControllerTask, "BLUE LED Task", 128, NULL, 1, NULL);
  xTaskCreate(yellowLedControllerTask, "YELLOW LED Task", 128, NULL, 1, NULL);
}

void redLedControllerTask(void *pvParameters)
{
  pinMode(RED, OUTPUT);

  while (1)
  {
    delay(500);
    digitalWrite(RED, digitalRead(RED) ^ 1);

  }
}
```

FreeRTOS : Example 1

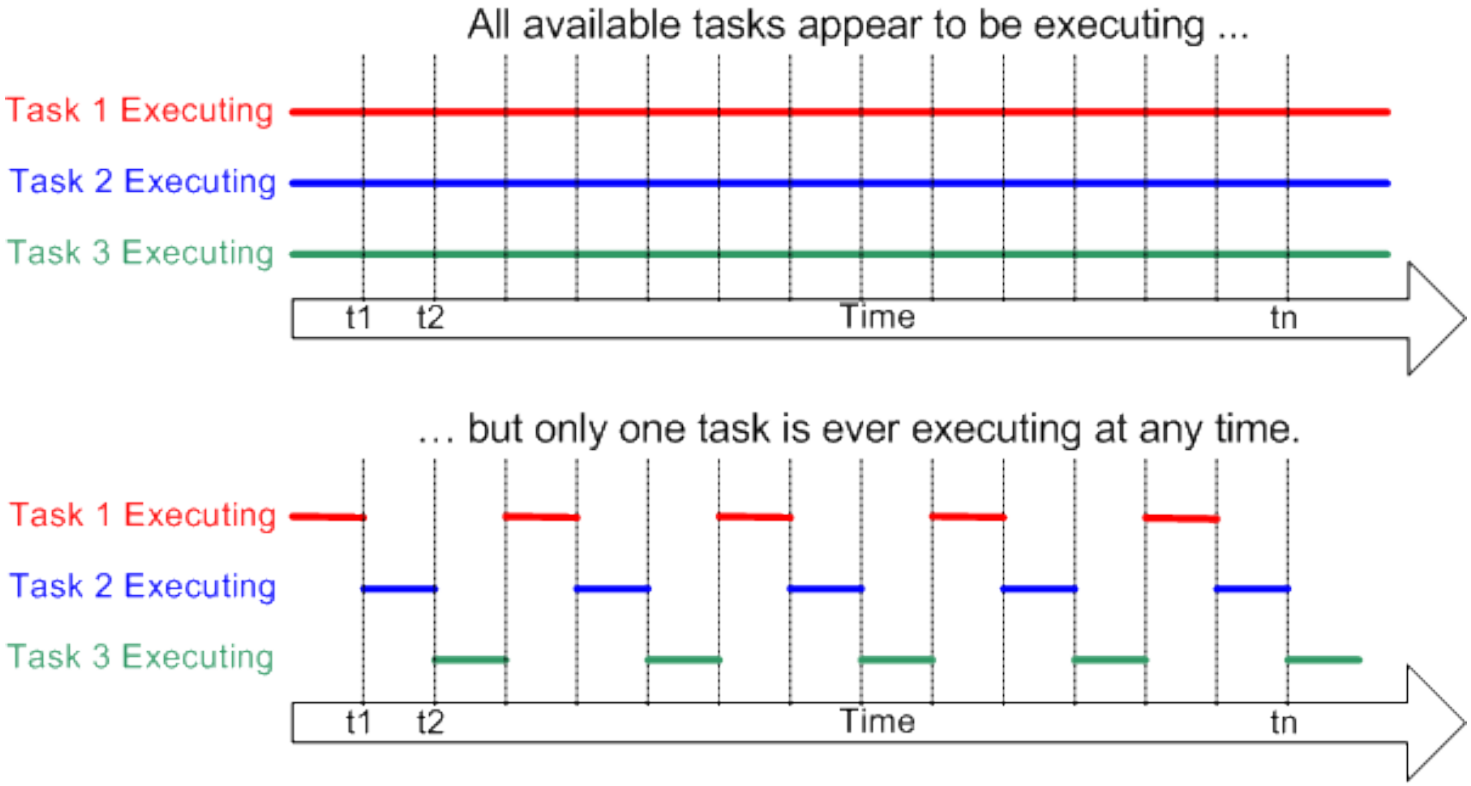
```
void blueLedControllerTask(void *pvParameters)
{
  pinMode(BLUE, OUTPUT);

  while (1)
  {
    delay(500);
    digitalWrite(BLUE, digitalRead(BLUE) ^ 1);

  }
}

void yellowLedControllerTask(void *pvParameters)
{
  pinMode(YELLOW, OUTPUT);

  while (1)
  {
    delay(500);
    digitalWrite(YELLOW, digitalRead(YELLOW) ^ 1);

  }
}
void loop() {}
```

FreeRTOS : Example 1

- จากโปรแกรม จะมีการสร้าง Task ขึ้นมา 3 Task คือ redLedControllerTask, blueLedControllerTask และ yellowLedControllerTask แต่ละ Task จะทำงานเสมือนกับเป็นคนละโปรแกรมที่ทำงานไปพร้อมๆ กัน จะเห็นว่ามีการใส่ pinMode ใน Task ด้วย
- พารามิเตอร์ที่ 3 เป็นขนาดของ Stack ซึ่งจองให้แต่ละ Task ใช้งาน Stack ได้ 128 words
- พารามิเตอร์ที่ 4 กรณีนี้ไม่มีพารามิเตอร์ จึงกำหนดเป็น NULL
- พารามิเตอร์ที่ 5 เป็น Priority หรือลำดับความสำคัญของงาน โปรแกรมนี้จะกำหนดให้แต่ละงานมีลำดับความสำคัญเท่ากัน คือเป็น 1
- โปรแกรมนี้เมื่อรัน จะเป็นการสั่งให้ไฟ LED แต่ละดวงกระพริบ ซึ่งสามารถเปลี่ยนค่า delay ให้กระพริบด้วยความเร็วต่างกันก็ได้

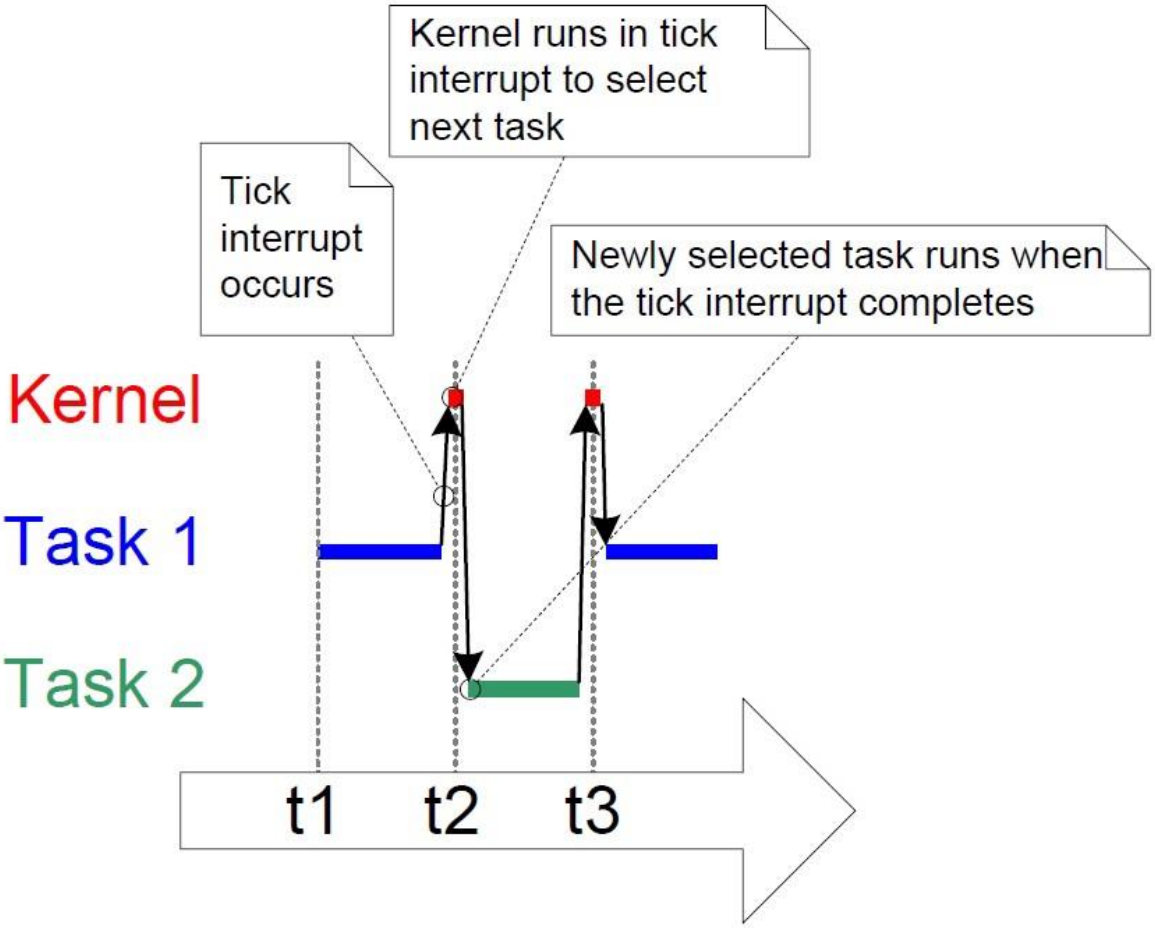
FreeRTOS : Task Scheduler

- โปรแกรมนี้จะทำงานตามรูป คือ แต่ละ Taskจะได้รับเวลาจาก CPU สลับกันไป แต่จะดูเหมือนว่าแต่ละ Task ทำงานอย่างต่อเนื่อง (Time Slicing)



FreeRTOS : Task Switching

- แสดง Task Switching



FreeRTOS

- จะเห็นว่าการเขียนโปรแกรมใน freeRTOS คล้ายกับการเขียนโปรแกรมใน Arduino ตามปกติ แต่สิ่งที่แตกต่าง คือ
  - การ Setup ใดๆ ให้นำมาไว้ใน Task ด้วย
  - ฟังก์ชันที่เขียนต้องทำงานแบบ Infinity Loop คือ ไม่มีวันหลุดจากฟังก์ชันได้
  - ฟังก์ชันต้องมีลักษณะเบ็ดเสร็จในตัวเอง โดยจะทำงานอยู่ในฟังก์ชันนั้นไปตลอด โดยไม่มีการ return ออกจากฟังก์ชัน
- การสิ้นสุดงาน คือ จะให้หยุดทำงาน สามารถทำได้โดยการลบ Task โดยใช้คำสั่ง vTaskDelete()

FreeRTOS

- โปรแกรมที่ผ่านมา เราสามารถเขียนให้สั้นลงได้
- จะเห็นได้ว่า Task ทั้ง 3 ได้แก่ redLedControllerTask, blueLedControllerTask และ yellowLedControllerTask มีลักษณะการทำงานที่คล้ายกัน
- ดังนั้นสามารถปรับปรุงโปรแกรมได้โดยรวมเป็นฟังก์ชันเดียว โดยส่งขาของ Led เข้าไปเป็นพารามิเตอร์

FreeRTOS : Example 2

```
#include <Arduino_FreeRTOS.h>

#define RED     6
#define YELLOW  7
#define BLUE    8

const uint16_t *blueLed = (uint16_t *) BLUE;
const uint16_t *redLed  = (uint16_t *) RED;
const uint16_t *yellowLed = (uint16_t *) YELLOW;

void setup() {
  xTaskCreate(LedControllerTask, "RED LED Task", 128, (void *)redLed, 1, NULL);
  xTaskCreate(LedControllerTask, "BLUE LED Task", 128, (void *)blueLed, 1, NULL);
  xTaskCreate(LedControllerTask, "YELLOW LED Task", 128, (void *)yellowLed, 1, NULL);
}

void LedControllerTask(void *pvParameters) {
  pinMode(RED, OUTPUT);
  pinMode(BLUE, OUTPUT);
  pinMode(YELLOW, OUTPUT);

  while (1) {
    delay(1000);
    digitalWrite(pvParameters, digitalRead(pvParameters) ^ 1);
  }
}
void loop() {}
```

Activity

- ต่อวงจรและโหลดโปรแกรม ในตัวอย่างที่ 1 และตัวอย่างที่ 2 ไปทดสอบการทำงาน

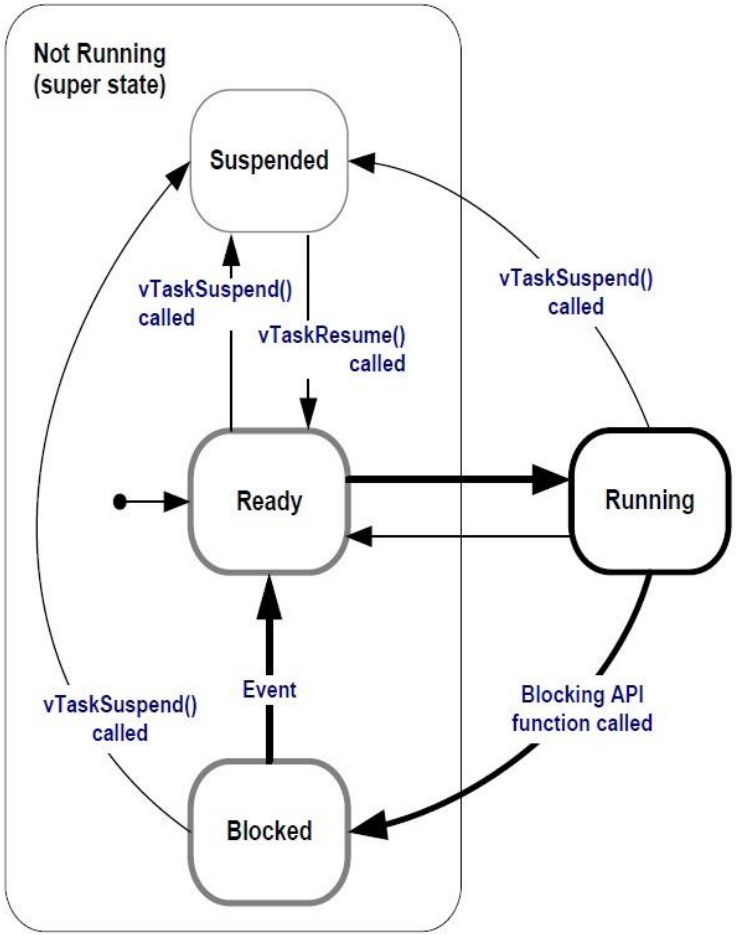
FreeRTOS : Priority

- ให้ลองเปลี่ยนค่าพารามิเตอร์ตัวที่ 5 ของสีแดง ให้เป็น 2 ตามตัวอย่าง

```
xTaskCreate(LedControllerTask, "RED LED Task", 128, (void *)redLed, 2, NULL);
```

- เมื่อรันโปรแกรมนี้ จะพบว่าจากเดิมที่มีไฟกระพริบอยู่ 3 ดวง จะเหลือกระพริบเพียงดวงเดียว คือ ไฟ LED สีแดง ทั้งนี้เนื่องจากคำสั่งข้างต้นมีการเปลี่ยนลำดับความสำคัญ (uxPriority) ของ Task สีแดงเป็น 2 ซึ่งมีความสำคัญสูงกว่าในขณะที่ Task อื่นๆ มีความสำคัญแค่ 1 เท่านั้น ดังนั้น Task อื่นจึงไม่ได้รับ CPU Time เพื่อทำงาน
- ที่เป็นเช่นนี้เนื่องจาก Realtime OS จะให้ความสำคัญกับงานที่มี Priority สูงกว่าซึ่งจะล่าช้าไม่ได้เลย

FreeRTOS : Task Lifecycle



- Task Lifecycle
  - Ready พร้อมทำงาน
  - OS จะเลือก Task ที่ Ready มา 1 Task โดยจะเลือก Task ที่มี Priority สูงที่สุดโดย Task นั้นจะเปลี่ยนเป็น Running แต่ถ้ามีหลาย Task จะวน (round robin)
  - Suspended เกิดจากการเรียก vTaskSuspend() และออกโดย vTaskResume()
  - Blocked เกิดจากการรอการทำงาน เช่น รอการกด Input

FreeRTOS

- สมมติว่าเราต้องการส่งพารามิเตอร์มากกว่า 1 ตัว คือ นอกจากจะส่งขาของ LED แล้ว ยังต้องการส่งค่า delay ไปด้วยเพื่อให้ LED แต่ละดวงกระพริบไม่พร้อมกัน
- เราสามารถส่งพารามิเตอร์ที่เป็น Array ได้ โดยสร้าง Array ขนาด 3 x 2 โดย 3 คือ จำนวน LED และ 2 คือ จำนวนพารามิเตอร์
- พารามิเตอร์ตัวแรก คือ LED pin และตัวที่ 2 คือ delay

```
const uint16_t taskParam[3][2] = {
                                    {BLUE, 500},
                                    {RED, 1000},
                                    {YELLOW, 2000}
                                  };
```

FreeRTOS : Example 3

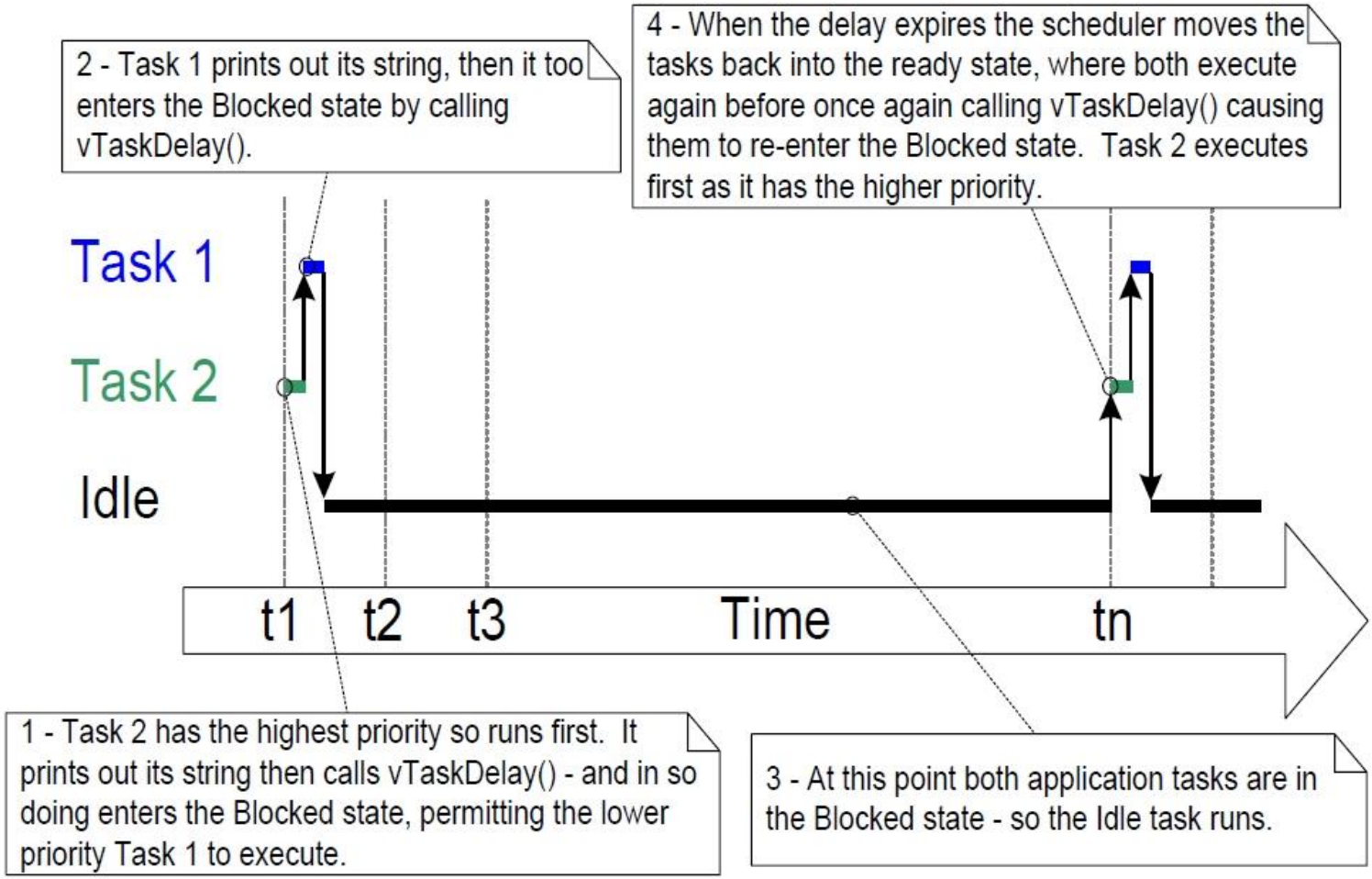
```
#include <Arduino_FreeRTOS.h>

#define RED     6
#define YELLOW  7
#define BLUE    8

const uint16_t *blueLed = (uint16_t *) BLUE;
const uint16_t *redLed  = (uint16_t *) RED;
const uint16_t *yellowLed = (uint16_t) YELLOW;

const uint16_t taskParam[3][2] = { {blueLed, 500},
                                   {redLed, 1000},
                                   {yellowLed, 2000} };
void setup() {
  xTaskCreate(LedControllerTask, "RED LED Task", 128,
              (void *)&taskParam[0], 1, NULL);

  xTaskCreate(LedControllerTask, "BLUE LED Task", 128,
              (void *)&taskParam[1], 1, NULL);

  xTaskCreate(LedControllerTask, "YELLOW LED Task", 128,
              (void *)&taskParam[2], 1, NULL);
}
```

FreeRTOS : Example 3

```
void LedControllerTask(void *pvParameters) {
{
  uint16_t *params = pvParameters;

  int ledPin = params[0];
  int time = params[1];

  pinMode(ledPin, OUTPUT);

  while (1) {
    delay(time);
    digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
  }
}

void loop() {}
```

FreeRTOS

- ในโปรแกรมที่ผ่านๆ มา เราใช้คำสั่ง delay ซึ่งการทำงาน คือ อยู่เฉยๆ เป็นเวลาเท่ากับที่กำหนด เช่น delay(1000) คือ ให้อยู่เฉยๆ เป็นเวลา 1 วินาที
- ในการเขียนโปรแกรมแบบ single task การจะให้อยู่เฉยๆ ไม่มีปัญหาอะไร แต่ในการทำงานแบบ multi-task การให้ CPU อยู่เฉยๆ ก็เสียเวลาในการทำงานอื่น
- ดังนั้นใน FreeRTOS จึงได้สร้างคำสั่ง vTaskDelay ขึ้นมาใช้แทน โดยคำสั่ง vTaskDelay จะทำงานไม่เหมือนกับคำสั่ง delay ปกติ โดยคำสั่ง vTaskDelay จะทำให้ Task นั้นย้ายไปอยู่ในสถานะ Blocked เป็นระยะเวลาที่กำหนด
- ในระหว่างนั้น Scheduler สามารถจะเลือก Task อื่นขึ้นมาทำงานได้ ทำให้แม้จะตั้งค่า Priority ไม่เท่ากัน แต่ทุก Task ก็ยังทำงานอยู่ ดังนั้นให้แก้จาก delay -> vTaskDelay และแก้ priority จะพบว่าไฟทุกดวงยังคงกระพริบทั้งหมด
- หมายเหตุ : พารามิเตอร์ใน vTaskDelay ถ้า 100 = 1 วินาที


2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.
Task 1
Task 2
Idle
t1
t2
t3
Time
tn
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.
3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

FreeRTOS

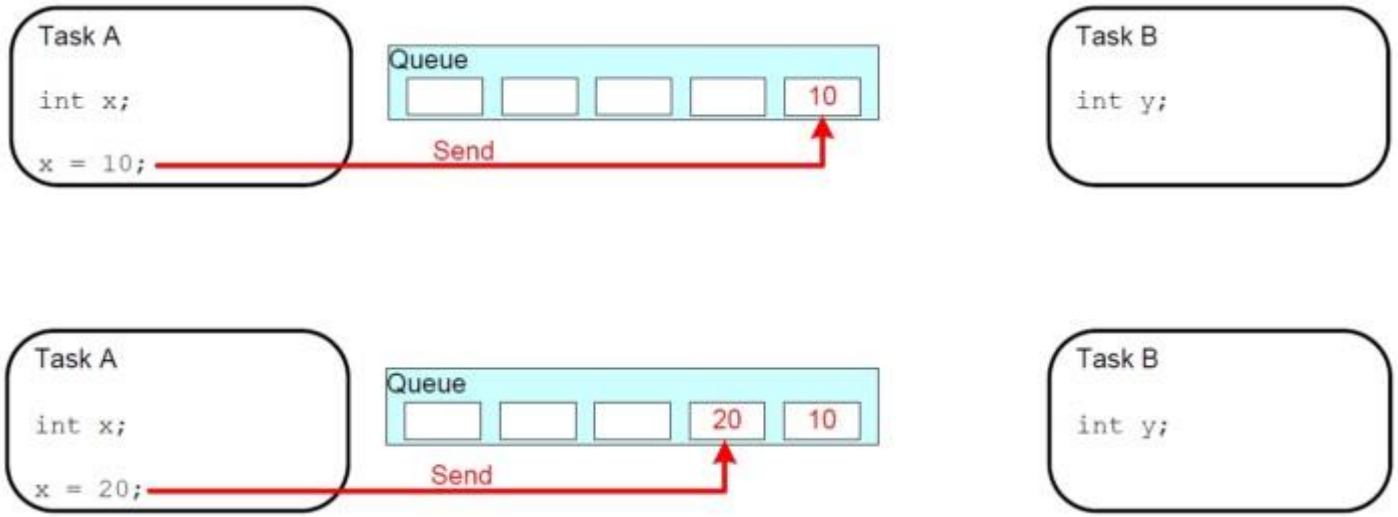
- ที่ผ่านมาโปรแกรมที่ใช้เป็นโปรแกรมง่ายๆ ที่มีเพียง Output เป็นไฟกระพริบ
- ในการ Input จะ ทำในฟังก์ชันเดียวกันก็ได้ เช่น จะเขียนโปรแกรมให้รับสวิตซ์ 3 ตัว ควบคุมไฟ 3 ดวง ก็สามารถทำได้ แต่ก็จะทำให้ฟังก์ชันมีความซับซ้อนมากขึ้น
- ทางเลือกอีกทางเลือกหนึ่ง คือ การแยกฟังก์ชันของแต่ละงานออกจากกัน โดยแยกเป็น 6 Task แต่ละ Task จะรับผิดชอบงานง่ายๆ คือ รับปุ่มแต่ละปุ่ม จำนวน 3 Task และกระพริบไฟแต่ละดวงจำนวน 3 Task โปรแกรมก็จะง่ายขึ้นมาก
- แต่พอแยก Task ออกมา ปัญหาก็เกิดขึ้น คือ จะส่งข้อมูลจาก Task ของสวิตซ์ ไปยัง Task ของ LED ได้อย่างไร

FreeRTOS : IPC

- เราอาจใช้วิธีกำหนดตัวแปรแบบ Global ก็สามารถส่งข้อมูลระหว่าง Task ได้เช่นกัน แต่ปัญหาก็มีอยู่ว่า Task ตัวรับจะรู้ได้ยังไงว่าตัวส่งได้ส่งมาแล้ว และ Task ตัวส่งจะรู้ได้ยังไงว่าข้อมูลที่เอาไปเก็บไว้ที่ตัวแปร มีการอ่านออกไปแล้ว
- ดังนั้นต้องมีกระบวนการเพิ่มเติม ที่สามารถส่งข้อมูลจากงานหนึ่งไปยังอีกงานหนึ่งได้ ซึ่งจะเรียกว่า Inter-Process Communication หรือ Inter-Task Communication

FreeRTOS : Queue

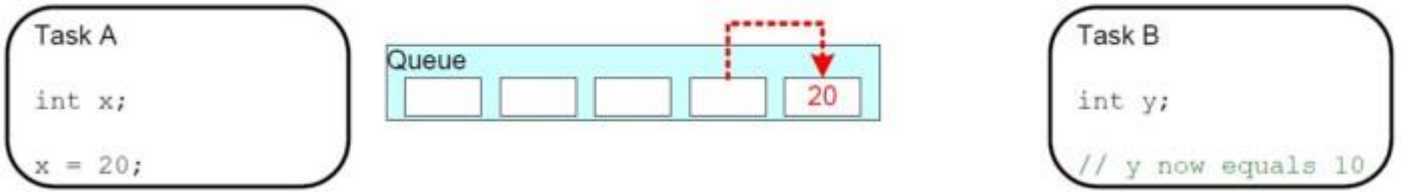
- จากภาพ แสดงการทำงานของคิว โดยเมื่อ Task A จะส่งข้อมูลใน Task B ข้อมูลก็จะไปรอที่ต้นคิว และเมื่อ Task A ส่งไปอีก ข้อมูลก็จะไปรอที่คิวอันดับ 2



FreeRTOS : Queue

- จากนั้นเมื่อ Task B รับข้อมูลไป ก็จะดึงจากส่วนหัวของคิว

- ข้อมูลถัดไป ก็จะมาอยู่ที่ส่วนหัวแทน



FreeRTOS : Queue

- FreeRTOS Quaue
  - คิวจะมีจำนวนจำกัดค่าหนึ่งเสมอ เรียกว่า ความยาวคิว
  - ข้อมูลจะต้องถูกนำออกจากส่วนหัวของคิวเสมอ
  - คิวจะเป็นหน่วยความจำของ FreeRTOS เมื่อสร้างคิวขึ้นมาแล้ว ไม่ต้องจองเพิ่มอีกเมื่อมีการใช้งาน
  - เมื่อส่งข้อมูลเข้าไปในคิว จะเป็นลักษณะของการ copy ข้อมูลทั้งชุด ไม่ใช่แค่การส่ง address แบบ pointer และเมื่อส่งข้อมูลในคิวแล้ว สามารถจะใช้งานตัวแปรที่ส่งเข้าไปในคิวซ้ำได้ทันที (เพราะถูก copy ข้อมูลเข้าไปในคิวแล้ว)
  - เมื่อข้อมูลถูกนำเข้าไปใส่ในคิวจนเต็ม จะใส่เข้าเพิ่มเข้าไปอีกไม่ได้ และในทำนองเดียวกัน เมื่อข้อมูลนำออกจากคิวจนหมด (คิวว่าง) จะไม่สามารถอ่านข้อมูลออกจากคิวได้

FreeRTOS : Queue

- FreeRTOS Quaue
  - Task ผู้ส่ง และ Task ผู้รับ จะแยกกันอย่างเด็ดขาดผ่านคิว ดังนั้นผู้เขียนโปรแกรมไม่ต้องกังวลว่า Task ใดจะทำหน้าที่เป็นผู้รับผิดชอบข้อมูล Task เพียงส่งหรือรับข้อมูลจากคิวเท่านั้น
  - คิว 1 คิว สามารถจะใช้งานได้หลาย Task พร้อมๆ กัน เช่น เราอาจกำหนดให้มี 1 คิว และมี 3 Task ที่รับข้อมูลและนำมาใส่ในคิวเดียวกัน
  - ในขณะที่ Task รับข้อมูลออกจากคิว จะมี Task เดียวที่อ่านได้ในเวลาหนึ่ง เช่น ถ้าคิวมีข้อมูลอยู่ Task แรกจะอ่านข้อมูลได้ ส่วน Task อื่นๆ (ถ้ามี) จะต้องรอ โดยจะอยู่ในสภาวะ block แต่หากคิวไม่มีข้อมูล ทุก Task จะอยู่ในสภาวะ block ทั้งหมด จนกว่าจะมีข้อมูลเข้ามาในคิว Task ที่มี Priority สูงสุดจะเปลี่ยนเป็นสภาวะ Ready ซึ่งจะเข้าไปอ่านข้อมูลในคิวได้

FreeRTOS : Queue

- FreeRTOS Quaue
  - แต่หากมี Task ที่มี Priority เท่ากัน Task ที่คอยมานานที่สุด จะเป็น Task ที่ได้เข้าไปในคิว และเพื่อป้องกันไม่ให้ Task ต้องคอยนานเกินไป อาจจะกำหนด block time เอาไว้ได้ โดยหาก Task ต้องรอเกินระยะเวลาที่กำหนดใน block time ก็จะเกิด time out และกลับสู่สภาวะ ready (แต่จะไม่ได้ข้อมูลไป) เพื่อทำงานต่อไป เช่น กลับไปแสดงผลว่าไม่มีข้อมูล เป็นต้น
  - ในขณะที่มีการส่งข้อมูลเข้าไปในคิว (Write Queue) จะมีการ lock คิว เอาไว้ ดังนั้น Task อื่นๆ จะเข้าใช้งานคิวไม่ได้ (ต้องรอในสภาวะ block) ซึ่งแปลว่าในขณะใดขณะหนึ่งจะมีเพียง Task เดียว ที่ส่งข้อมูลคิวได้ กรณีที่มีหลาย Task ถูก block พร้อมกัน เมื่อคิวมีที่ว่าง Task ที่มี Priority สูงสุด จะได้เข้าไปเขียนในคิวก่อน แต่หากมี Task ที่มี Priority เท่ากัน Task ที่คอยมานานที่สุด จะเป็น Task ที่ได้เข้าไปเขียนในคิว และเช่นเดียวกันสามารถกำหนด timeout สำหรับการรอเขียน (กรณีที่คิวเต็ม) ได้ เพื่อไม่ให้ Task รอเขียนข้อมูลนานเกินไป

FreeRTOS : Queue

- การสร้าง Queue จะใช้ฟังก์ชันดังนี้

```
xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize )
```

  - uxQueueLength ใช้กำหนดความยาวของคิวที่จะสร้าง
  - uxItemSize ใช้กำหนดขนาดของแต่ละคิว
  - return value ถ้าเป็น NULL แสดงว่าสร้างคิวไม่สำเร็จ

FreeRTOS : Queue

- ฟังก์ชันฝั่งส่ง

```
xQueueSendToFront(
                    QueueHandle_t xQueue,
                    const void * pvItemToQueue,
                    TickType_t xTicksToWait
                );
```

  - xQueue เป็นตัวแปรของคิวที่จะส่งข้อมูลเข้าไป (Handle)
  - pvItemToQueue พอยเตอร์ที่ชี้ไปยังข้อมูลที่จะส่งเข้าไปในคิว
  - xTicksToWait เป็นระยะเวลาสูงสุดที่จะรอเพื่อส่งข้อมูลเข้าในคิว ถ้ามีค่าเป็น 0 ก็จะไม่รอ แม้จะส่งข้อมูลเข้าในคิวไม่ได้

FreeRTOS : Queue

```
xQueueSendToBack(
                        QueueHandle_t xQueue,
                        const void * pvItemToQueue,
                        TickType_t xTicksToWait );
```

- ฟังก์ชัน xQueueSendToFront จะส่งข้อมูลเข้าไปที่หัวของคิว และ xQueueSendToBack จะส่งข้อมูลเข้าไปที่ส่วนท้ายของคิว เนื่องจากปกติแล้วเราจะส่งข้อมูลเข้าไปที่ส่วนท้ายของคิว ดังนั้น FreeRTOS จึงได้สร้างฟังก์ชัน xQueueSend() ขึ้นมา ซึ่งจะเหมือนกับ xQueueSendToBack
- ค่าที่ส่งกลับจากฟังก์ชัน มี 2 ค่า คือ pdPASS ซึ่งแปลว่าส่งข้อมูลได้สำเร็จ และ errQUEUE_FULL คือ ส่งข้อมูลไม่สำเร็จเนื่องจากคิวเต็ม

FreeRTOS : Queue

- ฟังก์ชันฝั่งรับ

```
xQueueReceive(
              QueueHandle_t xQueue,
              void * const pvBuffer,
              TickType_t xTicksToWait
              );
```

  - xQueue เป็นตัวแปรของคิวที่รับข้อมูลออกมา (Handle)
  - pvBuffer เป็นพอยเตอร์ที่ใช้รับข้อมูลที่ copy ออกมาจากคิว
  - xTicksToWait เป็นระยะเวลาสูงสุดที่จะรอเพื่อรับข้อมูลจากคิว ถ้ามีค่าเป็น 0 ก็จะไม่รอแม้จะส่งไม่มีข้อมูลในคิว
  - ค่าที่ส่งกลับจากฟังก์ชัน มี 2 ค่า คือ pdPASS ซึ่งแปลว่าส่งข้อมูลได้สำเร็จ และ errQUEUE_FULL คือ ส่งข้อมูลไม่สำเร็จเนื่องจากคิวเต็ม

FreeRTOS : Queue

```
#include <Arduino_FreeRTOS.h>
#include "queue.h"

#define RED     6
#define SW1     7

QueueHandle_t ledQueue;

void setup()
{
 Serial.begin(9600);
 ledQueue =  xQueueCreate(5,sizeof(int32_t));

 xTaskCreate(vSenderTask,"Sender Task",100,NULL,1,NULL);
 xTaskCreate(vReceiverTask,"Receiver Task", 100,NULL, 1, NULL);

}
```

FreeRTOS : Queue

```
void vSenderTask(void *pvParameters)
{
  BaseType_t qStatus;
  int32_t valueToSend = 0;
  pinMode(SW1, INPUT);
  while(1)
  {
    valueToSend = digitalRead(SW1);
    qStatus = xQueueSend(ledQueue,&valueToSend,0);
    vTaskDelay(10);
  }
}
```

FreeRTOS : Queue

```
void vReceiverTask(void *pvParameters)
{
  int32_t valueReceived;
  BaseType_t qStatus;
  const TickType_t xTicksToWait = pdMS_TO_TICKS(100);
  pinMode(RED, OUTPUT);
  while(1)
  {
    xQueueReceive(ledQueue,&valueReceived,xTicksToWait);
    Serial.print("Received value  : ");
    Serial.println(valueReceived);
    digitalWrite(RED, valueReceived);
    vTaskDelay(1);
  }
}
void loop(){}
```

FreeRTOS : Queue

- โปรแกรมจะประกอบด้วยส่วน setup() ซึ่งเป็นการสร้างคิวขึ้นมา 1 คิว ชื่อ ledQueue จากนั้นก็สร้าง Task ง่ายๆ ขึ้นมา 2 Task ชื่อ vSenderTask และ vReceiverTask
- vSenderTask จะทำหน้าที่อ่าน Switch ซึ่งต่อเอาไว้ที่ขา 7 และส่งข้อมูลเข้าไปในคิว โดยโปรแกรมกำหนดให้ delay 100 มิลลิวินาที (10 ticks)
- vReceiverTask จะทำหน้าที่รับข้อมูลจากคิว และสั่งให้ LED สว่างหรือดับ ขึ้นกับข้อมูลที่ได้รับมาจากคิว

Assignment #8 : FreeRTOS

- ให้เขียนโปรแกรม โดยใช้ FreeRTOS ส่งข้อมูลผ่าน Queue โดยส่วนของวงจรประกอบด้วย สวิตซ์ 3 ตัว และ LED จำนวน 3 ดวง
- เมื่อกด Sw1 ให้ LED 1 ติดเป็นเวลา 3 วินาที แล้วดับ ถ้ากดซ้ำระหว่างที่ติด ให้นับจากเวลากดไปอีก 3 วินาที (เหมือนต่ออายุ)
- เมื่อกด Sw2 ให้ LED 2 ติดและกระพริบไปเรื่อยๆ จนกว่าจะกด Sw2 อีกครั้ง
- เมื่อกด Sw3 ให้ LED 3 กระพริบจำนวน 3 ครั้ง ครั้งละ 500 ms (ติด 500 ms ดับ 500 ms) ถ้ากดซ้ำไม่มีผล (ต้องให้กระพริบครบ ถึงกดใหม่ได้)

For your attention