RTOS - Quest

Day - 1

Mon Tue Wed Thu Fri Sat Sun

Date: / /

→ 3 Levels of Software dev. in Embedded:

① Baremetal ② RTOS ③ Embedded Linux

→ Context Switching in RTOS: Task 1 —(Save Execution State)→ Task 2 —(Restore Execution state)→ Task 1 [Stacks, Program needed to save State of Registers]

→ STM32 CubeMx, STM32 Cube Ide Install

→ Thread / Task → main & while loop

→ ① Sequential Execution (Busy wait) ② ~~RTOS~~ ISR (Interrupt-Service-Routine)

③ RTOS: All tasks independent of one another (Scheduling Algo)

→ Generate Report: CubeMX > File > Generate Report

Variable Data Types: TickType_t (Uint16, Uint32), BaseType_t Uint32

c : char
s : short / uint16_t
l : long / int32_t
x : BaseType_t
u : unsigned
p : pointer

(defined within queue.c)
vTask, xQueue, pvTimer
Pointer → void

FreeRTOS: Macro Names

portMAX_DELAY : Located in portable.h

Prefix ↑ macro

taskENTER_CRITICAL(); : task.h
pdTRUE : projdefs.h

FreeRTOSConfig.h
configUSE_PREEMPTION :
errQUEUE_FULL : projdefs.h

Mon Tue Wed Thu Fri Sat Sun Date: / /

Macro	Value
pd TRUE	1
pd FALSE	0
pd PASS	1
pd FAIL	0

DAY-02

STM32

→ Creation of Tasks (Cube IDE)

File → New → STM32 Project

Select Board > Next

Project Name, C, Exec, STM32Cube

Next > Finish

→ Later we will add RTOS Kernel & Time Base

Yes → Open Device Configuration CubeMX Prospective

:) Add middleware > Select timmess for RTOS Kernel Time Base

→ Important Concepts

① There is a API layer on Top of FreeRTOS called CMSIS which wraps around Free RTOS function.

② Task creation, Semaphore creation first we do that from scratch. Once the concepts are learnt then we will create RTOS code (free) using CubeMX. Later we will learn how to apply Free RTOS CMSIS APIs.

Mon Tue Wed Thu Fri Sat Sun Date: / /

Enable FreeRTOS

Middleware > FreeRTOS > CMSIS_V2

Config Parameters (keep Default Config.)

System Clock = Tick_Rate_Hz = 1000

Minimal_Stack_Size = 128 Words

Max_Task_Name_Length = 16 (No. of Characters)

Tasks & Queues

→ Automatically creates Tasks for us, named default Task & we cannot remove it.

→ But we can delete this task after code generation & write tasks from scratch

→ Once middle ware selected all you have to do is to select Time base.

→ System Core

SYS > Timebase Source > TIM1, Debug = Disable

→ Connectivity

USART2 > Mode > Asynchronous

:) Keep the Rest Config Default

→ = Generate Code

Click > Yes open C/C++ prespective

→ C/C++ Prespective > Windows

Prespective > ~~Open~~ Prespective

Reset

WaRQnotes

Mon Tue Wed Thu Fri Sat Sun Date: / /

Increase Font Size

Windows > Preferences > General > Editors > Text Editors > Color & Fonts Clicked > Edit > 14 size > OK > Apply & Close

Understanding Code:

HAL_init(); // Hardware Abstraction Layer
SystemClock_Config(); // Clock for main system
{ // AHB bus // as well as various buses
{ // APB1 bus, APB2 bus // which can be prescaled
↳ Defined in SystemClock_Config function

MX_USART2_UART_Init();
↳ functin implementation in static void MX_USART2_UART_Init(void) function

MX_GPIO_Init(void)
__HAL_RCC_GPIOA_CLK_ENABLE();
↳ This is enabled so that we can use this GPIOA as UART2

Mon Tue Wed Thu Fri Sat Sun Date: / /

OS Kernel Initialize();

↳ CMSIS RTOS

"Standardization of RTOS, that is applied on top of free RTOS to make free RTOS functions look similar to other RTOS functions,,

For example:

defaultTaskHandle = osThreadNew (StartDefaultTask, Null, &defaultTask_attributes);

↳ same in Karl RTOS, to create a Task

→ So first we will use freeRTOS functions later we will use CMSIS RTOS API functions.

→ void StartDefaultTask (void *argument)
; its like main loop with infinite ~~for~~ for loop.

→ So for now delete the
> OS kernel Initilize(); its implementation
> defaultTask & its implementation
> Also remove void StartDefaultTask from Private function Prototypes before

int main (void)

→ So now we are left with only hardware initialization

Mon Tue Wed Thu Fri Sat Sun Date: / /

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)

This is the TIM1 we selected Earlier.

→ Write click on #include "main.h"

→ Click on Open Declaration

→ Now click on cmsis_os.h & open declaration (See FreeRTOS is included here)

→  = Build all

→ Download & Install RealTerm

RealTerm ≠ Putty Terminal

→ We cannot use printf in USART; for printing we need to write PutChar function in C language

Write int __io__ putchar(int ch); befor the int main(void) → to define it

Mon Tue Wed Thu Fri Sat Sun

Date: / /

~~After while loop~~

```
int __io_putchar(int ch)
{
    HAL_UART_Transmit(); → & UART2
}
```

DAY-03

```
int msg[1] = {'H'};

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART2_UART_Init();

  while (1)
  {
      HAL_UART_Transmit(&huart2, (uint8_t*)
                        msg, 1, 0xFFFF);
  }
}

int __io_putchar(int ch)
{ HAL_UART_Transmit(&huart2, (uint8_t*)ch, 1, 0xFFFF);
  return ch; }
```

Mon Tue Wed Thu Fri Sat Sun

Date: / /

To Flash Code: on board

→ Click debug

→ Main

→ C/C++ App : Debug \ I_Task_Creation.elf

→ Project: I_Task_Creation

→ Build Configuration: select Automatically

→ use workspace settings

→ click OK

Realterm:

Port: 16 = \USB SER000

Baud: 115200

open

→ Click ▶▶ play on STM32 Cube IDE

To use printf:

#include <stdio.h>

→ putchar & implementation after while loop

→ inside while(1)

{ printf ("Hello from STM32 \n\r");

}

Mon Tue Wed Thu Fri Sat Sun Date: / /

→ before int main(void) declare putchar

```
int __io_putchar(int ch);
```

→ After while(1) implementation will look like this

```
int __io_putchar(int ch)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1,
                      0xffff);
    return ch;
}
```

→ inside while(1)

```
{
    printf("Hello from STM32 \n\r");
}
```

→ Flash & test on Real Term Terminal

Mon Tue Wed Thu Fri Sat Sun **DAY-4** Date: / /

Free RTOS Task Creation Function:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char* const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPoiority,
                        TaskHandle_t *pxCreatedTask);
```

BaseType_t → Return type

xTaskCreate → Prefix

pcName → TaskName

pvParameters, uxPoiority, pxCreatedTask → arguments

```
void TaskA (void *pvParameteres)
```

Example:

→
```
xTaskCreate( vBlueLedControllerTask,
             "Blue Led Controller",
             100,
             NULL,
             1,
             NULL);
```

→ `vTaskStartScheduler();`

→ ~~while (1)~~ `void vBlueLedControllerTask(void *pvParams)`
`{printf("vBlueTask Running...\n\r");};`

Mon Tue Wed Thu Fri Sat Sun Date: / /

Important Concepts to learn:

→ Seamaphores & Mutexes are RTOS tools that solve the priority conflict issue among different tasks trying to access CPU; while the scheduler doesnot give enough time to each task.

→ Each Task runs for a particular Time Quanta.

→ v TaskStartScheduler();

↳ Runs the tasks according to their set priority. Lower the priority number, higher the priority for instance 1 priority is the highest priority of a task.

→ Another debuging tool of RTOS task priority is task profiler.

Task Profiler: Global variables that are placed in each task & theis increment tells us weather the task was executed or not.

→ Once, we see that the Global variables of all tasks are executed at the same pace we know all are running correctly with same rate.

DAY-05: RTOS Quest

→ Creating Tasks from One Tasks

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();

    xTaskCreate(vBlueLedControllerTask,
Task Name ← "Blue Led Controller",
                100,  ──#──→ Stack Size
Task Parameter ← NULL,  // (void *)blue_led,
Task Priority ← 1,
Variable to hold Task Handle ← NULL); // Allows us to Configure Tasks at Runtime.
    vTaskStartScheduler();
    while (1) { }
}

void vBlueLedControllerTask(void *pvPara
{ (Task creation of 2 tasks here completes)
    while (1)
    {
        BlueTaskProfiler++;
    };
}
```

Chapter-04: Video-04 (Task_Parameter

```
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /* Enable Clock access to port */
  /* Reset pins */
  /* Configure Pins */
  /* Initilize pins */

}

  __HAL_RCC_GPIOD_CLK_ENA

  HAL_GPIO_WritePin(GPIOD,
GPIOD_PIN_13|GPIOD_PIN_12, GPIO_PIN_RE

  GPIO_InitStruct.Pin = GPIO_PIN_12|
    GPIO_PIN_13|GPIO_PIN_14|
    GPIO_PIN_15;

  GPEO_InitStruct.Mode = GPIO_MODEOUT
  GPIO_InitStruct.Pull = GPIO_MODEO
  GPIO_InitStruct.speed = GPIO_NOPU
  GPIO_InitStruct.Speed = GPIO_SPEE
```

k_Parameters

Init(void)
truct;

Enable */
<_ENABLE();

to post */

();
CLK_ENABLE

GPIOD,
PIN_RESET

PIN_12|
_14|

GPIO_MODE_OUTPUT_PP
_MODE_OUTPUT_PP;
_NOPULL;
IO_SPEED_FREQ_LOW;

/* Initialize pins */

HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

---

int main (void)

HAL_Init();
System_Clock_Config();
MX_GPIO_Init();
MX_USART2_UART_Init();
// Now declare Tasks Here

//Above int main we are going to
#define same Macros.

#define GREEN GPIO_PIN_12
#define ORANGE GPIO_PIN_13
#define RED GPIO_PIN_14
#define BLUE GPIO_PIN_15

void System_Clock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
int __io_putchar(int ch);

① void vBlueLedControllerTask (void *pvParameters);

② void vRedLedControllerTask (void *pvParameters);

③ void vGreenLedControllerTask (void *pvParameters);

④ void vOrangeLedControllerTask (void *pvParameters);

→ typedef uint32_t TaskProfiler;

→ TaskProfiler BlueTaskProfiler, RedTaskProfiler, GreenTaskProfiler, OrangeTaskProfiler;

→ HAL_GPIO_TogglePin(GPIOB, (uint16_t) pv Parameters);

void vBlueLed(void *pvParameters)

```
{
    while(1)
    {
        // Green Task Profiler-1
        HAL_GPIO_TogglePin(GPIOB, GREEN);
    };
}
// Similar Tasks
```

→ We can do it by creating only one function:

```
:) const uint16_t *blueled =
        (uint16_t *) BLUE;

:) const uint16_t *redled =
        (uint16_t *) RED;

:) const uint16_t *green_led =
        (uint16_t *) GREEN;

:) const uint16_t *orange =
        (uint16_t *) ORANGE;
```

Video: 05 Coding Understand
Task Priorities:

Important Concepts:

→ Task with the same priority execute for the same time either 1ms or 10ms based on our selected tick rate or based on our time quanta system.

→ If Quanta = 10ms first task executes of 10ms, then the scheduler stops it & moves to 2nd task. (Round Robin Scheduling) for Tasks with same Priority

→ Let's change priority of a task, ~~smaller~~ larger the no, Higer the priority:

```
xTaskCreate( vGreenLedControllerTask,
             "Green Led Controller",
             100,
             NULL,    // Task Parame
             2,       // Task Prios
             NULL);
```

→ "A lower priority task cannot interrupt/overtake a higher priority task" Rule for Scheduler.

→ That is why priorities are defined at Runtime. If Priorities are defined initially, it means some tasks are never going to be executed.

Video: 06: How to Change Task Priority in Runtime.

→ Using Handler we can change the priority of a task.

```
void vRedLedControllerTask(void *pvParameters)
{ int i;
  while(1)
  {
    RedTaskPriority++;
    for(i=0; i<300000; i++) {}
    vTaskPrioritySet(green_handle, 3);
  }
}
```

→ // Before int main (void)

TaskHandle_t blue_handle,
red_handle,
green_handle,

→ Inside int main (void)

xTaskCreate(vBlueLedController,
"Blue Led Controller"
100,
NULL,
2,
&blue_handle);

// Changing Own Priority.
void vBlueLedControllerTask(void
*pvParameters)
{
while (1)
{
BlueTaskPriority++;
vTaskPrioritySet (NULL, 1);
};

Video: 07 (Coding Reading Priority of a Task:)

```
//Before Main(void)

uint32_t   green_priority;

// After while(1)

void vRelodedControllerTask(void
                            *pvParameters)

while(1)
{
    ReadTaskProfiler++;
    green_priority = uxTaskPriority
                     Get(green_hand);
};
}
```

Video:08 : Suspending a Task

```
//After while(1)
uint32_t suspend_monitor;
void vRedLedControllerTask(void
                           *pvParameters)
{
  int j;
  while(1)
  {
    RedTaskProfiler++;
    for(j=0; j<100000; j++){}
    suspend_monitor++;
    if(suspend_monitor >= 50)
    {
      vTaskSuspend(BlueHandle);
    }
  };
}
```

NULL to self suspend

Video :09: Resuming a Suspendid Task:

```
// Before int main (void)

uint32_t suspend_monitor;
uint32_t resume_monitor;
bool is_suspended = false;

// After while loop

→ void vRedLedControllerTask(void *pvParameters)
  {
     int i;
    while(1)
    {
        RedTaskProfiler++;
        for (i=0; i<100000; i++){ }

     if (suspend_monitor >= 50)
     {
          is_suspended = true;
          suspend_monitor = 0;
          vTaskSuspend (NULL);
     }
    }
  }
```

```
#include <stdbool.h>

void vSecondControllerTask( void
                            *pvParameters )
{
    int i;

    while( 1 )
    {
        SecondTaskProfiler++;
        for( i=0; i<100000; i++ ){

        if( is_suspended )
        {
            resume_monitor++;

            if( resume_monitor >= 30
            {
                vTaskResume( xHandle
                resume_monitor = 0;
                is_suspended = false;
            }
        }
    }
}
```

Video-10: Terminating a Task:

Important Concepts:

→ By default functions like deleting a task are not enabled by default by the kernal & we need to go to the configuration files.

STEPS:

→ Click on cs msis-os.h
→ otherway is to click on src > freertos.c
→ Click on FreeRTOS.h & click open Declaration
→ Scroll down & reach click on FreeRTSConfig.h > open Declaration
→ Inside FreeRTOSConfig.h > Scroll down to #define configENABLE_FPU 0
#define configENABLE_MPU 0

#define configSUPPORT_STATIC_ALLOCATION 1
#define configSUPPORT_DYNAMICALLOCATION 1

#define configUSE_IDLE_HOOK 0
#define config_TICK_HOOK 0

#define configUSE_MUTEXES 1

OTHER TASKS:

#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1

#define INCLUDE_vTaskCleanUpResources 0

→ // Before int main(void)

uint32_t green_priority;

uint32_t execution monitor;
uint32_t resume_monitor;
bool is_killed = false;

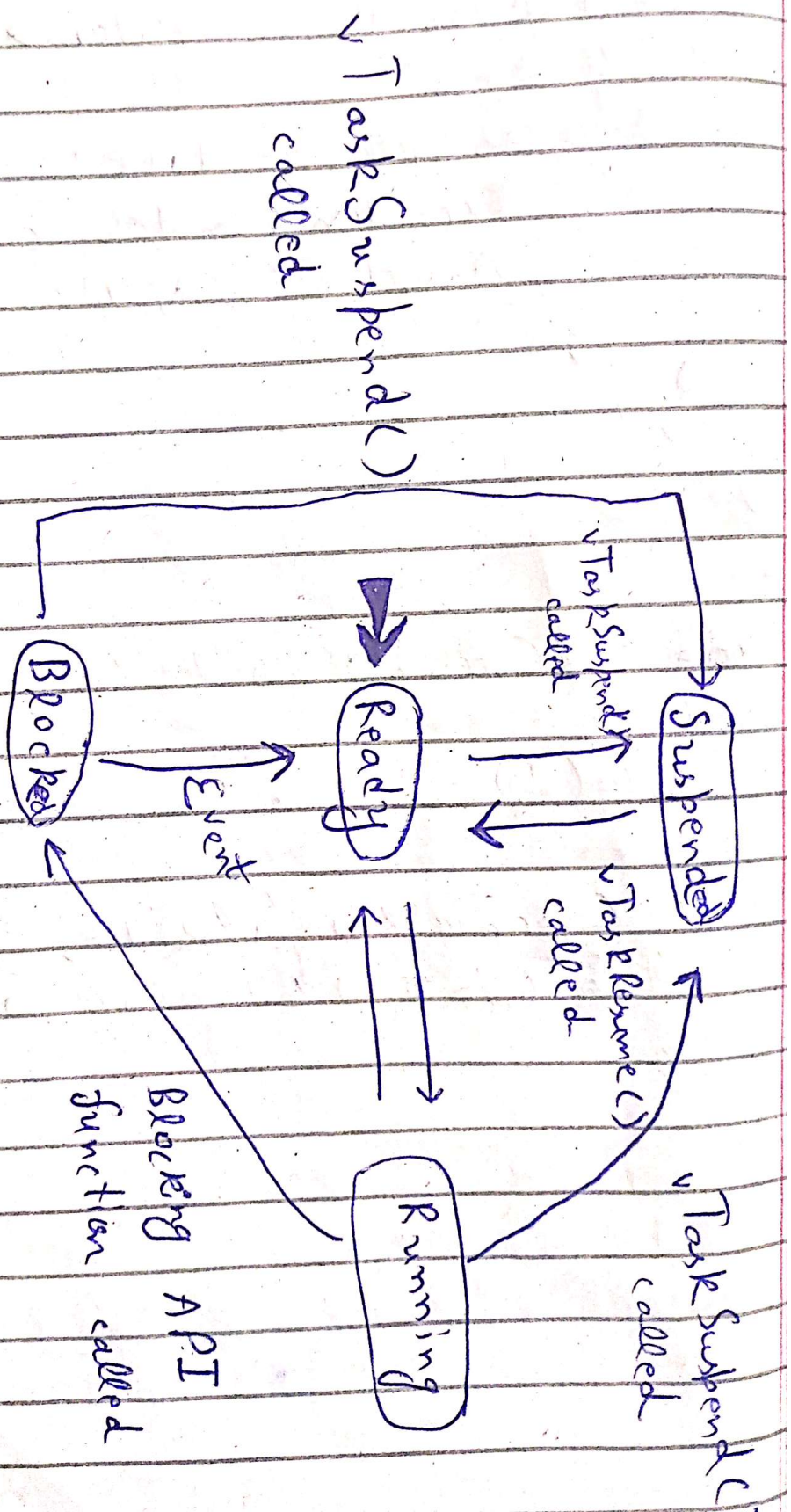
→ // inside vRedLedControllerTask

void vRedLedControllerTask(void *pvParameters)
{ int i;
while(1)
{ RedTaskProfiler++;
for(i=0; i<100 000; i++) {}

```
        execution_monitor++;

        if(execution_monitor >= 50)
        {
            is_killed = true;
            Execution_monitor = 0;
            vTaskDelete(NULL);
        }
    };
}
```

```
void vGreenLedControllerTask(void *pvParameters)
{int i;
 while(1)
    {
        GreenTaskPro++;
        for(i=0; i<100000; i++) {}
    }
}
```

Video: 11 : States Of A Task



Ready State:

→ Waiting for the thread scheduler / thread Switches to execute it.
→ Waiting for their turn.

Running State:

→ ~~Time~~ Task Executing for a Time Quanta like 10ms.

Block State:

→ Task waiting for a particular event to occour. or waiting for a resource token.
→ Event could be a Semaphore to occour or time delay. (sleep)

→ Semaphore: Its a resource for example a key, & when it available it ~~event~~ could be event to change the task state from Block state to Ready State.

Suspend State:

Don't know the difference btw Block/Suspend.

Video: 12

Using the Blocked State to create delays:

Causes of Blocked state:

- Time Related event.
- Synchronized event.

→ Go to blocked state for a particular time period.

→ A synchronization event could be the task waiting for a signal from another task.

→ We can use blocked states to make delays.

vTaskDelay()

INCLUDE_vTaskDelay()
// in FreeRTOSConfig.h we need to set this function to 1 to enable it.

vTicksToDelay(TickType_t xTicksToDelay)

The no. of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into Ready state.

pdMS_TO_TICKS()

Convert a time specified in milliseconds into a time specified in Ticks.

vTaskDelay(pdMS_TO_TICKS(100))

Video-13: Coding Blocking a Task (Precise Delay)

→ // Before int main(void)

```
const TickType_t _50ms =
        pdMS_TO_TICKS(50);
```

→ //Now in one of the task:

```
void vRedLedControllerTask(void
                        *pvParamet
{
   while(1)
   {
        RedTaskProfiler++;
        vTaskDelay(_50ms);
   };
}
```

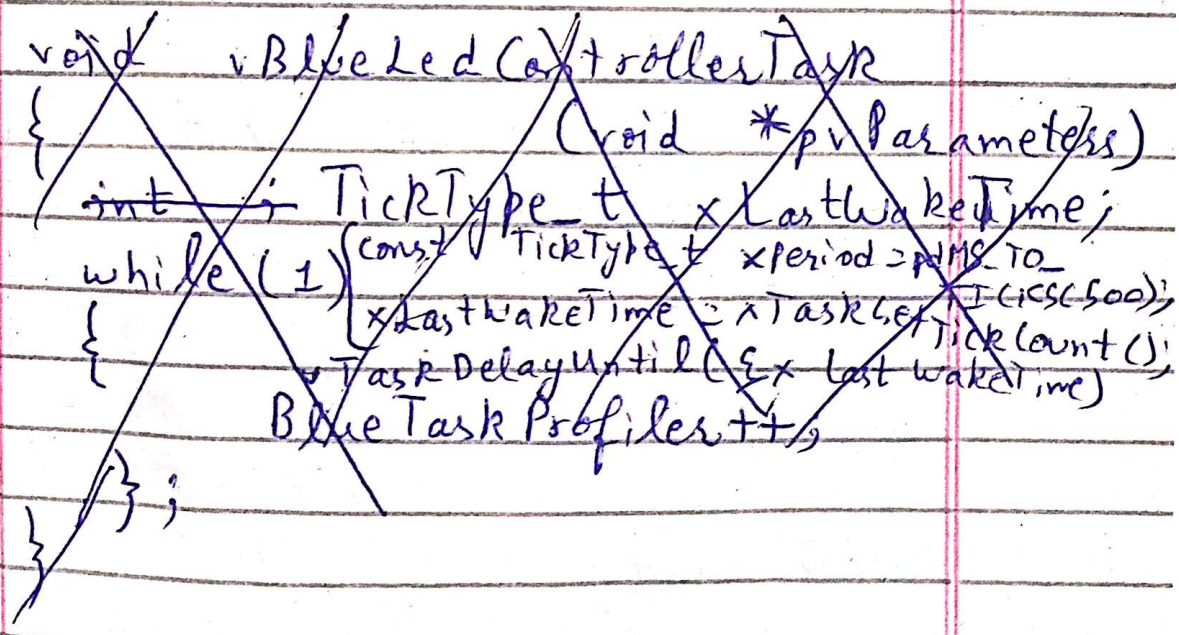
Video : 14 : Executing Tasks Periodically

" Tasks execute based on the Time Quanta that is in the configuration of the RTOS Kernel,,

Task Delay Until function

→ Delay until this amount of time.
→ Periodic execution of a task.

// Inside a Task



```
void vBlueLedControllerTask
        (*pvParameters)
{
    TickType_t xLastWakeTime;
    const TickType_t xPeriod =
        pdMS_TO_TICKS(500);

    xLastWakeTime= xTaskGetTick
                   Count();

    while(1)
    {
        vTaskDelayUntil(&xLastWak
            xPeriod);

        BlueTaskProfiler++;

    };

}
```

Non RTOS way

// Otherway to do this is to use Hardware interrupt. for Lets say 50ms, 100ms exe. ~~Configure~~ the time to create

Video: 15 : Idle - Task

// Can be used for cleanup low priority code.

Important Concepts:

→ There must be at least one task in the Running state at any given moment. Because of this, the Idle task automatically moves to Running state when there is no task there.

→ The Idle task is created automatically when we start the scheduler.

→ The Idle task has the lowest priority (0), this ensures that it never prevents a higher priority application task from entering the Running state.

→ We can add application specific functionality directly into the idle task through the use of an idle hook (or idle callback) function.

→ This function is called automatically by the idle task once per iteration of the idle task loop

Enable

config USE_IDLE_HOOK()

in FreeRTOSConfig.h

To Consume less CPU for Idle Task, we can set following parameter to 1:

→ configIDLE_SHOULD_YIELD is used to prevent idle task from consuming CPU time.

Video: 16 (Idle Tasks)

```
TaskProfiler IdleTaskProfiler;
// After whil where task
// definitions are present.

void vApplicationIdleHook(void)
{
    IdleTaskProfiler++;
}

// we see our IdleTask
// is not running.

// Before int main(void)

const TickType_t _250ms =
    pdMS_TO_TICKS(250);
```

```
void vBlueLedControllerTask
        (void *pvParameters)
{
    while(1)
    {
        BlueTaskProfiler++;
        vTaskDelay(250ms);
    };
}
```

→ We need to Enable

```
//define configUSE_IDLE_HOOK
```

// in

SRC > freertos.c > freeRTOS.h

> FreeRTOSConfig.h

FUN-FACT

Video: 17 (THE TICK HOOK)

// It should be very Precise otherwise RTOS won't be Real time

→ A tick hook (or tick callback) is a function that is called by the kernel during each tick interrupt.

→ Tick hook functions execute within the context of the tick interrupt, & so must be kept very short, must use only a moderate amount of stack space, & must not call any FreeRTOS API functions that do not end with 'FromISR()'

for instance:

→ we can't call semaphore give, since it doesn't end with "FromISR()"

→ It is recommend not to use Tick Hook, at any cost unless there is no way out.

Enable

configUSE_TICK_HOOK()

in FreeRTOSConfig.h

SRC > freertoc.c > FreeRTOS.h
> FreeRTOSconfig.h

CHAPTER: 05 : DAY-05

Video-01: Understand Queues & Queuesets

Important Concepts:

→ Queues are used as F buffers, where data is inserted at the back removed from the front.

→ Queues can hold a fin number of fixed data items.

→ The maximum number items a queue can is called the length.

→ For Passing Data we h 2 options.

① Pass by Value:
"Simply insert data into the queue"

② Pass by reference:

- Involves using queues to transfer pointers to the data, rather than copy the data itself into & out of the queue byte by byte.
- Preferred choice for passing large queue data.
- Saves memory.

Queues : Blocking

→ Blocking on Queue Reads

When a task attempts to read from a queue, it can optionally specify a "block time". This is the time the task will be kept in the Blocked state to wait for the data to be available from

the queue, if the queue is empty. As soon as data becomes available the task is automatically moved to the ready state.

Blocking on Queue Writes

Task is placed in blocked state if a queue is full, as soon as space becomes available in the queue task is moved to ready state.

Queues: Commonly used as APIs,

xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

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

Parameters:

① Handle of the Queue.

② Pointer to the buffer, we want to send or the variable's adress we want to send to the Queue.

③ The Block time, like how much time we want to wait for this Data Sending into Queue.

× QueueRecieve()

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

① Handle of the Queue.

② Where we want to store recieved data.

③ The amount of BlockTime we want to wait for this.

xQueueCreate()

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

① The length of the Queue, which tells us how many items will the queue hold.

② Second argument is the size of the items.

Queuesets:

"Queuesets allow a task to recieve data from more than one queue without the task polling each queue in turn to determine which, if any, contains data.

Enable

configUSE_QUEUE_SETS()

// in freeRTOSConfig.h

SRC > freertos.c > freeRTOS.h
> freeRTOSConfig.h

Important concepts.

→ When we use QueueSets the task will automatically receieve a notification that one of the queues in the queue set has new data.

→ Otherwise task has to continuslly check each of the Queue for new data.

→ This way the task ~~resumes~~ does not use the task polling each queue to determine which one contains the new data.

x QueueCreateSet()

QueueSetHandle_t xQueueCreateSet
(const UBaseType_t uxEventQueue
Length);

→ Creating a QueueSet itself is called xQueueCreateSet.

→ Takes the length of the Queue.

→ ~~The length of the queue set~~ The length of the queue set basically dictate how many queues the queueset can hold.

xQueueAddToSet()

BaseType_t xQueueAddToSet
(QueueSetMemberHandle_t
xQueueOrSemaphore,
QueueSetHandle_t xQueueSet
);

→ This allows us to add a queue to a QueueSet & we can also add semaphores to a Queueset as well.

→ Second argument is the handle of the QueueSet.

VIDEO : 02 : CHAP-05

Working with Queues

→ Queues are used to send data from one task to another or from a task to a Interrupt Service Routine or from ISR to a task.

→ Message Boxes → [Transmit piece of info.]

SRC > freertos.c

#include "queue.h"

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init
                    (void);


int __io_putchar(int ch);

void SenderTask(void *pvParameters);
void ReceiverTask(void *pvParameters);

TaskHandle_t sender_handle,
             receiver_handle;

QueueHandle_t yearQueue;

int main(void)
{
```

```
HAL_Init();
SystemClock_Config();
MX_GPIO_Init();
MX_USART2_UART_Init();

yearQueue = xQueueCreate(5, sizeof(int32_t));

xTaskCreate(SenderTask,
            "Sender Task",
            100,
            NULL,
            1,
            &senderTaskHandle);

// same pattern for xTaskCreate(Receiver)

vTaskStartScheduler();

---------------------

// After while(1){} }
// implement the tasks.
```

Sender Task

```
void SenderTask(void *pvParameters)
{
    int32_t value_to_send = 2050;
    BaseType_t qStatus;

    while(1)
    {
        qStatus = xQueueSend
                  (yearQueue,
                  &value_to_send,
                  0);  ---> Time out

        if(qStatus != pdPASS)
        {
            printf("Error: Data could
                    not be sent...\r\n");
        }

        // Delay
        for (int i=0; i<100000; i++){}
    }
}
```

Reciever Task

```
void ReceiverTask(void *pvParameters)
{
    int32_t value_received;
    const TickType_t wait_time=
        pdMS_TO_TICKS(100);

    BaseType_t qStatus;

while(1)
{
    qStatus = xQueueReceive
            (year_Queue,
            &value_received,
            wait_time);

    if (qStatus == pdPASS)
    {
        printf ("The value
                received id: %ld
                ....\r\n",
                value_recieved);
    }
    else
    {
```

```
        printf ("The value is not
                received. \r\n");
    }
  }
}
```

// There will be a warning so write %ld %s & write #include "queue.h" in cmsis_os.h instead of free_rtos.c.

CHAPTER: 05 (Video-03)

Before Printf Uart correctly/e

// ~~After~~ int __io_putchar(int

{

```
int uart2_write(int ch)
{
    while(!(USART2->SR & 0x0080)){}

    USART2->DR = (ch & 0xFF);

    return ch;
}

int __io_putchar(int ch)
{
  //HAL_UART_Transmit(&huart2,
  //        (uint8_t *)&ch, 1,
  //          0xFFFF);

    uart2_write(ch);
    return ch;
}
```

CHAPTER: 05 (, Video: 04).

Synchronizing Tasks with Queues

Problem statement:

→ We are going to create 2 sending tasks of the same priority (1) & a receiver task of a higher priority (Priority 2)

→ We did this by assigning same priorities for sending tasks & higher priority of receiving task.

→ Copied Task declaration, implementation creation all other remained same.

Video:05: Chapter 5

Sending More Complex data with Queues:

Problem statement:

→ we will have data prepared from 2 sensors to send into Queue.

→ We will have 3 tasks humidity task, pressure task, receiver task, Thus, also 3 task handles

```
// Before int main(void)

typedef enum
{
    humidity_sensor,
    pressure_sensor
} DataSource_t;

// Define structure type to be passed to the queue
```

```
typedef struct
{
    uint8_t ucValue;
    DataSource_t sDataSource;

} Data_t;

// Declare two "Data_t" variables
that will be passed to the
queue

static const Data_t xStructsToSend[2]
        =
{
    { 77, humidity_sensor}, //humidity
    { 63, pressure_sensor} // poessure


};

// Define 3 task handles & Queue

TaskHandler_t hum_task_handle,
              press_task_handle,
              receiver_handle;
```

```
QueueHandle_t xQueue;
void SenderTask(void *pvParameters);
void ReciverTask(void *pvParameters);
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();

    // Create queue to hold a
    // maximum of 3 structures
                                  max. of 3 structures
    xQueue = xQueueCreate(3, sizeof
                                  (Data_t));
                  Defined our own data type
    // Create a receiver task
    with a priority of 1

    xTaskCreate(ReceiverTask,
                "Receiver Task",
                100,
                NULL,
    Priority ← 1,
                &receiver_handle);
```

// Create sender task with priority of 2

```
xTaskCreate( SenderTask,
             "Humidity Sensor Task",
             100,
             (void *) &(xStructsToSend[0]),
             2,
             &hum_task_handle);

// Create task to send humidity with priority 2

xTaskCreate( SenderTask,
             "Pressure Sender Task",
             100,
             (void *) &(xStructsToSend[1]),
             2,
             &press_task_handle);

vTaskStartScheduler();

// After while(1) { }
```

~~Enter the Blocked~~

Sender Task for Both Hum
Pa

```
void SenderTask(void *pvParameters)
{
BaseType_t qStatus;
// Enter the blocked state for
200ms for space to become
available in the queue each
time the queue is full
const TickType_t wait_time =
        pdMS_TO_TICKS(200);
while(1)
{
qStatus = xQueueSend
          (xQueue, pvParameters,
           wait_time);
if(qStatus != pdPASS)
        {
        }
        for(int i=0; i<100000; i++)
    }
}
```

Receiver Task

```
void ReceiverTask( void *pvParameters )
{
    Data_t xReceivedStructure;
    BaseType_t qStatus;
    while(1)
    {
        qStatus = xQueueReceive( xQueue,
                  &xReceivedStructure, 0);

        if( qStatus == pdPASS )
        {
            if(xReceivedStructure.sDataSource ==
                humidity_sensor)
            { printf("Humidity sensor value=%d \n\r", xReceived
                                                    Structure.
                                                    ucValue);
            }
        }
        else { printf("Pressure sensor value
                = %d \n\r",
                xReceivedStructure.
                ucValue);
        }
    }
    else { // }
}
```

Chapter 05: Video - 06

Working With Queuesets

```
//Before int main(void)
//declare 2 queues:

static QueueHandle_t

xQueue1 = NULL, xQueue2 = Null;

//Declare a queue set

static QueueSetHandle_t = NULL;
// void Declare all tasks here;
//Before going further
//we need to enable QueueSetH
//in FreeRTOSConfig.h

#define configUSE_QUEUE_SETS

// inside int main(void)
// Create 2 queues, each
// sends a character pointer
```

```
// Create a queue set to hold 2 queues
// Each queue holds 1 element
xQueueSet = xQueueCreateSet( 1 * 2 );

xQueue1 = xQueueCreate( 1, sizeof(char*) );
xQueue2 = xQueueCreate( 1, sizeof(char*) );
printf("System initializing...\n");
// Add the 2 queues to
// queue set:

xQueueAddToSet( xQueue1, xQueueSet );

xQueueAddToSet( xQueue2, xQueueSet );

// Create 2 sender tasks, with the
same priority of 1:

xTaskCreate( vSenderTask1, "Sender1", 100,
             NULL, 1, NULL );

xTaskCreate( vSenderTask2, "Sender2", 100,
             NULL, 1, NULL );

xTaskCreate( vReceiverTask, "Receiver", 100,
             NULL, 2, NULL );

vTaskStartScheduler();
```

// Sender Task 1

```
while (1)
    {
    }
}
```

```
//After while (1)
void vSenderTask1(void *pvParameters)
{
    const TickType_t xBlockTime
        = pdMS_TO_TICKS(100);

    const char * const msg =
        "Message from vSenderTask1\r\n";

    while(1)
    {
        //Block for 100ms
        vTaskDelay(xBlockTime);

        //Send the string "msg"
        // to xQueue1

        xQueueSend(xQueue1,&msg,0
    }
}
```

//SenderTask2

```
void vSenderTask2( void *pvParameters )
{
    const TickType_t xBlockTime =
    pdMS_TO_TICKS(200);

    const char * const msg =
    "Message from vSenderTask2\r\n";

    while(1)
    {
        //Block for 200ms
        vTaskDelay (xBlockTime);

        //Send the string "msg" to xQueue1
        xQueueSend (xQueue2, &msg, 0);

    }
}
```

```
//ReceiverTask

void vReceiverTask( void *pvParameters )
{
  QueueHandle_t xQueueThatContainsData;
  char *pcReceivedString;

  while(1)
  {
    xQueueThatContainsData =
    ( QueueHandle_t )
    xQueueSelectFromSet(
      xQueueSet, portMAX_DELAY );

    xQueueReceive( xQueueThat
                   ContainsData,
                   &pcReceivedString,
                   "%s",
    printf( pcReceivedString );

  }
```

Chapter: 06 (Video - 01)

Seamaphores:

A seamaphore is a signal or a key sent btw tasks or btw tasks & interrupts. It does not carry any data.

Types:

→ Binary Sequence:

The Semaphore can assume two values: '1' or '0', to indicate weather there is a signal or not.

→ Counting Semaphore:

A semaphore with an associate counter which can be incremented or decremented.

The counter indicates the number of keys available to access a particular resource.

Mutex:
Stands for mutual exclusion. allows multiple tasks to access a sigle shared resource but only one at a time.

[Binary Semaphore]: Commonly used APIs

→ xSemaphore CreateBinary()

SemaphoreHandle_t
xSemaphoreCreateBinary
(void);

→ xSemaphoreGive()

BaseType_t xSemaphoreGive
(xSemaphoreHandle_t
xSemaphore);

→ xSemaphoreGiveFromISR()

```
BaseType_t xSemaphoreGiveFromISR(
    SemaphoreHandle_t xSemaphore,
    BaseType_t *pxHigherPriorityTaskWoken);
```

```
x SemaphoreTake()
```

```
BaseType_t xSemaphoreTake(
    SemaphoreHandle_t xSemaphore,
    TickType_t xTicksToWait);
```

Counting Semaphores : Enabling

Enable

~~confuse~~

configUSE_COUNTING_SEMAPHORES

in FreeRTOSConfig.h

Uses: Counting Semaphores:

① Counting Events.
② Resource Management.

Counting Semaphores: Commonly used APIs

xSemaphoreCreateCounting( )

SemaphoreHandle_t

xSemaphoreCreateCounting(

UBaseType_t uxMaxCount,
UBaseType_t uxInitialCo

);

MUTEX: Enabling

Enable

configUSE_MUTEXES

// (in FreeRTOSConfig.h)

MUTEX: Commonly used APIs

* SemaphoreCreateMutex()

SemaphoreHandle_t
    xSemaphoreCreateMutex(void);

→ Issues we face if we don't use MUTEXES & SEMAPHORES PROPERLY:

PRIORITY INVERSION:

This occours when a higher priority task waiting for a lower priority task inherently assumes the priority of the lower priority task.

PRIORITY INHERITANCE:

Temporarily raising the priority of the resource holder to the priority of the highest priority task waiting for the resource.

Deadlock:

→ A deadlock occours when two tasks cannot proceed because they are both waiting for a resource that is held by the other.

Gatekeeper Task

A gatekeeper task is a task that has sole ownership of a resource.

Only the gatekeeper task is allowed to access the resource directly. Any other task needing to access the resource can do so only indirectly by using the services of the gatekeeper.

Chapter: 06 Video: 02

// ~~Add~~ Open cmsis_os.h declaration
// Inside cmsis_os.h add the line

```
#include "semphr.h"
```

// Inside int main(void), Create three tasks:

```
xTaskCreate(RedLedControllerTask,
            "Red LED Task", 100, NULL,
            1, NULL);

xTaskCreate(yellowLedControllerTask,
            "Yellow LED Task", 100, NULL,
            1, NULL);

xTaskCreate(blueLedControllerTask,
            "Blue LED Task", 100, NULL,
            1, NULL);
```

// Before int main(void) we need to declare TaskProfiles, Tasks & create Semaphore (Binary)

```
SemaphoreHandle_t xBinarySemaphore;

typedef int TaskProfiler;

TaskProfiler RedLEDProfiler;
TaskProfiler YellowLEDProfiler;
TaskProfiler BlueLED Profiler;

//Inside int main(void)
 after Task creation we
 start our scheduler

vTaskStartSchedular();

//Now after while(1) we
 need to implement our
 task functions but before
 that we declare all
  tasks above int main(void)

void redLedControllerTask(void *pvParam

void blueLedControllerTask(void *pvParame

void yellowLedControllerTask(void *pvParamet
```

// Inside Int main(void), we need to create semaphore binary before task creation

```
xBinarySemaphore = xSemaphoreCreateBinary();

// Implementation Tasks
// First Task after while(1)

void redLedControllerTask(void *pvParameters)
{
→ xSemaphoreGive(xBinarySemaphore);
  while(1)
  {
→ xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
    RedLEDProfiler++;
    printf("This is REDTask \r\n");
→ xSemaphoreGive(xBinarySemaphore);
    vTaskDelay(1);
  }
}
```

```
// Third Task

void yellowLedControllerTask
        (void *pvParameters)

{
  while(1)
    {

    xSemaphoreTake(xBinarySemaphor
            portMAX_DELAY);

    YellowLEDProfiler++;

    printf("This is YELLOW TASK

    xSemaphoreGive(xBinarySemaphor
    vTaskDelay(1);

    }

}
```

```
// 2nd Task

void blueLedControllerTask(void *pvParameters)
{
    while (1)
    {

  xSemaphoreTake(xBinarySemaphore,
      portMAX_DELAY);

   BlueLEDProfiler++;

  printf("This is Blue Task \r\n");

  xSemaphoreGive(xBinarySemaphore);

   vTaskDelay(1);

  }

}
```

CHAPTER:07 VIDEO-01

DEVELOPING SOME DRIVER PART-1

→ // Goal: Develop Peripheral Driver

① Right click on the Src file, click on New > Source file "uart.c" > Finish

② Now Right click on the Inc file, click on New > File & name it "uart.h"

③ Right click on the Src file, click on New > File & name it "adc.c"

④ Right click on the Inc folder, click on New > File & name it as "adc.h"

Inside uart.h

```
#ifndef __UART_H
#define __UART_H
void USART2_UART_TX_Init(void);
void USART2_UART_RX_Init(void);
char USART2_read(void);
#endif
```

Inside adc.h

```
#ifndef INC_ADC_H_
#define INC_ADC_H_
void adc_init(void);
uint32_t read_analog_sensor(void);
#endif
```

Inside uart.c

```
#include "stm32f4xx_hal.h

int __io_putchar(int ch);
UART_HandleTypeDef huart2;
void USART2_UART_TX_Ini
{
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLEN
  huart2.Init.StopBits = UART_STOPBITS_
  huart2.Init.Parity = UART_PARITY_NONE
  huart2.Init.Mode = UART_MODE_TX
  huart2.Init.HwFlowCtl = UART_HWCONTR
  huart2.Init.OverSampling = UART_OVERSAMPLIN

  if (HAL_UART_Init(&huart2) != HAL_
      { }
}  // Add another UART_Rx here.

int USART2_write(int ch) {
  while(!(USART2->SR & 0x00
    USART2->DR = (ch & 0xF
  return ch;
}
```

```
//Read a charachter from USART2
char USART2_read(void) {

while (!(USART2->SR & 0x0020));
{}
return USART2->DR; }

int __io_putchar(int ch)
{

// HAL_UART_Transmit(&huart2,
//      (uint8_t *)&ch, 1, 0xFF);

  USART2_write(ch);

} return ch;
```

```
void USART2_UART_RX_Init(void)
{

huart2.Instance ------
huar2.Init ~ ~~
~ ~ ~~~~~
. ~ ~ ~~~
huart2~Init.Mode = UART_MODE_RX;
. ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
. ~~ ~~.

}
```

① Inside main.c

→ Remove #ifded FULL_ASSE
→ Remove void assert_failed(
→ Remove #endif /* USE_FULL

```
#include <stdio.h>

#include "main.h"

#include "cmsis_os.h"

#include "uart.h"

#include "exti.h"

#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int __io_putchar(int ch);

uint8_t btn_state;
uin8_32_t sensorvalue;

// In the int main(void)
```

```
HAL_Init();
System_Clock_Config();
MX_GPIO_Init();
USART2_UART_TX_Init();
gpio_init();
adc_init();

while(1)
{
    btn_state = read_digital_sensor();
    sensor_value = read_analog_sensor();

}
```

~~// Rest of Code after extic~~
~~file Under Sensor Testing~~
video -
4000+ ADC full Read

Inside "adc.c"

```
#include "stm32f4xx_hal.h"

// AHB = Advanced High Perfor
//       Bus

// APB = Advanced Peripheral
```

→ // So, we need to find ADC1 & See to what bus its connected to So in our case ~~it's~~ ADC1 is connected to APB2 100MHz. This means we need to find the register in APB2 range to enable clock access to ADC1.

→ // To find the register we need a seperate manual known as reference manual.

STM32F411 Reference manual

→ // Ctrl + F → ahb1 Find Reg Name.

31 zeros

OB 0000 ---- 1

(Hex-dec)
7 zeros

```
void adc_init(void) {

    RCC -> AHB1ENR |= 1;   // 0X0000000...
(module)
                  OR
Recommended -> //  (1U<<0);
                   Shift 1 to Position 0

    RCC -> APB2ENR |= (1U<<8);

    GPIOA -> MODER |= 0XC;
               // Set PA1 as analog
               // 1100

// config of ADC1:  // "|" sets only the bit we want to set
                    //  all remaining remain same.
    ADC1 -> CR2 = 0;   // SW trigger (software trigger)
    ADC1 -> SQR3 = 1;  // Conversion Sequence starts at ch 1
    ADC1 -> SQR1 = 0;  // Conv. Seq. Length 1
    ADC1 -> CR2 |= 1;  // Enable ADC 1

}
```

Embeded systems baremetal Programming course.

```
uint32_t read_analog_sensor(v
{
    ADC1->CR2 |= (1U<<30);
        //start conversion

    while(!(ADC1->SR & 2))
        { } //wait for
            conversion to
            complete.

    ~~ADC1->DR~~

    return ADC1->DR;
        //Return
        results

}
```

VIDEO: 02 : CHAPTER: 07 : DEVELOPING DRIVERS

→ GPIO External Interrupt Drives

1) Write Click on [Src] file, click on New, Click on Source file name it as "exti.c"

2) Write click on [Inc] file, click on New, Click on source file name it as "exti.h"

Inside exti.K file:

→ // Remember in Nucleo board push button is connected to PC-13 (Port C, Pin 13)

```
#include "stm32f4xx_hal.h"

void p13_interrupt_init(void)
{
// Enable GPIOC clock
RCC -> AHB1ENR |=4;

// Enable SYSCFG clock
RCC -> APB2ENR |= 0x4000;
```

// Input Interrupt Mode for Push Bu

→ // Configure PC13 for push bu
interrupt

GPIOC → MODER &= ~0x0C0000

→ // clear port selection for
// EXTI 13

SYSCFG → EXTICR[3] &=
~0x00F0;

→ // Select port C for EXTI3
SYSCFG → EXTICR[3] |=
0x0020;

→ // unmask EXTI13
EXTI → IMR |= 0x2000;

→ //Select falling edge trigge
EXTI → FTSR |= 0x2000;

→ NVIC_SetPriority(EXTI15_10_
6);

```
NVIC_EnableIRQ(EXTI15_10_IRQn);
}
```

// In exti.h file:

```
#ifndef INC_EXTI_H_
#define INC_EXTI_H_

void p13_interrupt_init(void);
void gpio_init(void);
#endif
```

→ uint8_t read_digital_sensor(void);

```
void gpio_init(void)
{
    // Enable GPIOC clock
    RCC -> AHB1ENR |= 4;
}
```

CHAPTER : 07 Testing the Drivers

```
uint8_t   read_digital_sensor(void)
{
    if(GPIOC->IDR & 0x2000)
    {
        return 1;
    }
    else{
        return 0;
    }
}
```

CHAPTER : 08 VIDEO - 01

Semaphore Mutex

```
// Above int main(void)

uint8_t btn_state;
uint32_t sensor_value;

SemaphoreHandle_t xSerialSemaphore;

void analog_sensor_task(void *pvParameters);
void digital_sensor_task(void *pvParameters);
```

```
// Inside main(void)

gpio_init();
adc_init();

printf("System Initializing...\n\r");

xSerialSemaphore = xSemaphoreMutex();
```

```
xTaskCreate(
        digital_sensor_task,
        "ButtonRead",
        128,
        NULL,
        2,
        NULL);

xTaskCreate(
        analog_sensor_task,
        "SensorRead",
        128,
        NULL,
        1,
        NULL);

vTaskStartScheduler();
```

```
//After while(1)

void digital_sensor_task(void *pvParameters)
{
    gpio_init();
    while(1)
    {
        btn_state = read_digital_sensor();
//If we can obtain or Take the Serial Semaphore
//If the semaphore is not available,
wait 5 ticks of the scheduler to see
                                    if it becomes
                                    free.
    }
}

void analog_sensor_task(void
                        *pvParameters)
{
    adc_init();
    while(1)
    {

        sensor_value=read_analog_sensor();
if (xSemaphoreTake(xSerialSemaphore,
    (TickType_t) 5 ) == pdTRUE)
    { printf("The sensor data is:
        %d \r\n", sensor_value);
    xSemaphoreGive(xSerialSemaphore);
    vTaskDelay(1); }}
```

```
// Inside the void digital_sensor_task while(1) after btn_state:

if(xSemaphoreTake(xSerialSemaphore, (TickType_t) 5) == pdTRUE)
{
printf("The button state is: %d \r\n", btn_state);
}
vTaskDelay(1);
}
}
```

// Inside CMSIS_OS.h

#include "semphr.h"

Before int main (void);

uint8_t btn1state;

uint32_t sensor_value;

CHAPTER: 08: VIDEO-02

COUNTING SEMAPHORES

// In previous Lab replace
x Serial Semaphore = x Semaphore
CreateMutex();

x Serial Semaphore = x Semaphore
CreateCounting
(1, 0);
Total No. of count ↑ ↑ Initial no. of count

// In Mutex we didn't had to Give Semaphore Signal but in Counting Semaphore we do need to give it. So inside int main (void) after vTaskStartScheduler();

x SemaphoreGive(x SerialSemaphore);

// If in above (1,1) we already give semaphore signal then we would not be required to explicitly Give a semaphore.

```
xSerialSemaphore = xSemaphoreCreate
                   Counting(1, 1);
```

↑ we seted initial Count to 1.

CHAPTER: 09 IMPLEMENTING GATEKEEPER TASK


```
// Inside Int main(void), we have three tasks:
int main(void)
{
 HAL_Init();
 SystemClock_Config();
 MX_GPIO_Init();
 USART2_UART_TX_Init();
printf("System Initializing....\n\r");                    NULL,
xTaskCreate(digital_sensor_task, "ButtonRead", 128, 1, NULL
xTaskCreate(analog_sensor_task, "SensorRead", 128, NULL, 1, NUL
xTaskCreate(lcd_task, "GateKeeper", 128, NULL, 0, NULL);
xPrintQueue = xQueueCreate(2, sizeof(int32_t));
  → // Creates a Queue with [Queue Handler] named xPrintQueue.

 vTaskStartScheduler();

 while(1) {
          }

}

void digital_sensor_task(void *pvParameters)
 {
   gpio_init();
   while(1)
    {
      btn_state = read_digital_sensor();
      xQueueSendToBack(xPrintQueue, &btn_state, 0);
      vTaskDelay(10);

    }
 }
```

```
void analog_sensor_task( void *pvParameters )
{
    adc_init();
    while(1)
    {
        sensor_value = read_analog_sensor();
        xQueueSendToBack( xPointQueue, &sensor_value, 0 );
        vTaskDelay(10);
    }
}

int value_to_print;
void lcd_task( void *pvParameters )
{
    while(1)
    {
        // Wait for a message to arrive.
        xQueueReceive( xPointQueue, &value_to_print,
                       portMAX_DELAY );
        printf("New value: %d \n\r", value_to_print);
    }
}
```

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"
#include "usart.h"
#include "gpio.h"
#include "adc.h"
```

```
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
int __io_putchar(int ch);
uint8_t btn_state;
int sensor_value;
QueueHandle_t xPointQueue;
void analog_sensor_task( void *pvParameters );
void digital_sensor_task( void *pvParameters );
```

```
void lcd_task( void
    *pvParameters );

QueueHandle_t
    xPointQueue;
```

Software Timers: Introduction [CH-10:Video:01]

→ Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency.

→ The function executed by the software timer is called the software timer's callback function.

They are implemented under the control of the RTOS kernel They donot require hardware support & are not related to hardware timers.

able: configUSE_TIMERS 1 in FreeRTOSConfig.h

ypes:

Auto-Reload Timer: Once started, it will re-start itself each time it expires, resulting in periodic execution of its callback function.

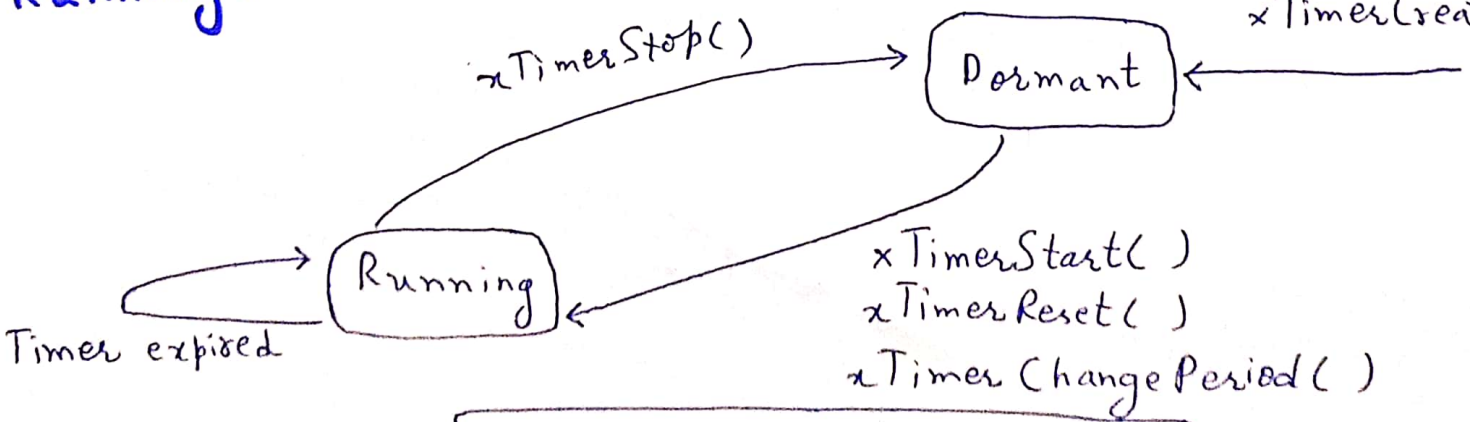
One-shot Timer: Once started, it will execute its callback function once only. It can be restarted manually, but will not restart itself.

Period: The time btw the software timer being started, & the software timer's callback function executing.

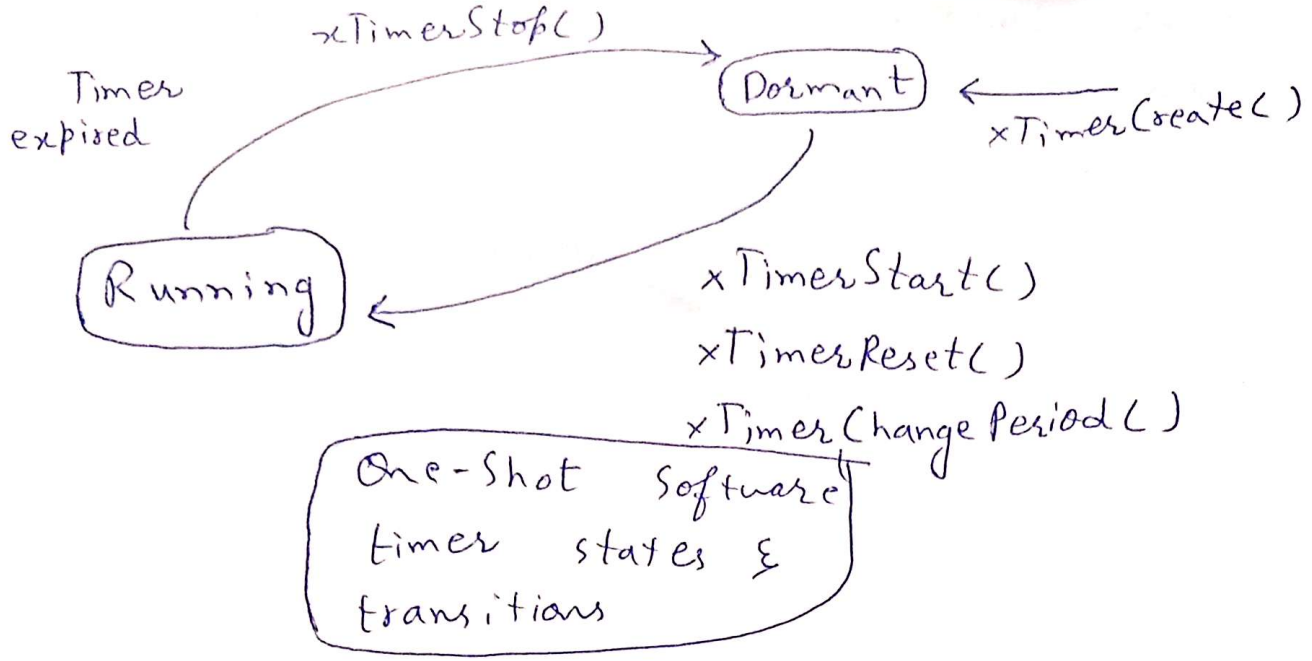
STATES:

→ **Dormant:** Callback functions does not execute

→ **Running:** Executes callback function.



Auto-Reload Software timer states & transitions



Commonly Used APIs:

xTimerCreate()

```
TimerHandle_t xTimerCreate(
    const char *const pcTimerName,        <- Timer Name
    TickType_t xTimerPeriodInTicks,       <- Period
    UBaseType_t uxAutoReload,             <- True=AutoReload, False=OneShot
    void *pvTimerID,                      <- Give Timer ID
    TimerCallbackFunction_t               <- Callback func: Timer should execute
        pxCallbackFunction
    );
```

xTimerStart()

```
BaseType_t xTimerStart( TimerHandle_t xTimer,        <- Timer Handler
    TickType_t xTicksToWait                          <- How much time we need to wait for the function to execute
    );
```

Timer Id

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID);

void *pvTimerGetTimerID
    ( TimerHandle_t xTimer);
```

CHAPTER: 10 (VIDEO-02)

Working with Software Timers

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"  ← #include "timers.h"
#include "usart.h"
#include "exti.h"
#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int __io_putchar(int ch);

// The periods assigned to the
one-shot & auto-reload timers
respectively //

#define mainONE_SHOT_TIMER_PERIOD
        (pdMS_TO_TICKS(4000UL))

#define mainAUTO_RELOAD_TIMER_PERIOD
        (pdMS_TO_TICKS(500UL))

TimerHandle_t xAutoReloadTimer,
              xOneShotTimer;
```

status of Timer, wheater started/or not..

```
BaseType_t  xTimer1Started,
            xTimer2Started;

int main(void)

{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    USART2_UART_TX_Init();

    printf("System Initializing...\n\r");

    xOneShotTimer = xTimerCreate
                    ("OneShot",
    mainONE_SHOT_TIMER_PERIOD,
    pdFALSE,
    0,
    prvOneShotTimerCallback);

xAutoReloadTimer = xTimerCreate
                    ("AutoReload",
    mainAUTO_RELOAD_TIMER_PERIO
    pdTRUE,
    0,
    prvAutoReloadTimerCallback);
```

```
xTimer1Started = xTimerStart( xOneShotTimer, 0 );
xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );
vTaskStartScheduler( );

while (1) {
        }
}

void prvOneShotTimerCallback
        ( TimerHandle_t xTimer )
{
    static TickType_t xTimeNow;

    // Obtain the current tick count.

    xTimeNow = xTaskGetTickCount( );

    printf ( "One-shot timer
        callback executing :
        %d \n\r",
        (int) xTimeNow );

}
```

```
void prvAutoReloadTimerCallback
    ( TimerHandle_t xTimer )
{
    static TickType_t xTimeNow;
    // Obtain the current tick
       count.
    xTimeNow = xTaskGetTickCount();
    printf ("Auto-reload timer
          callback executing
          : %d \n\r",
          (int) xTimeNow);
}
```

```
//Above int main(void), define
the two methods

void prvOneShotTimerCallback
    ( TimerHandle_t xTimer );

void prvAutoReloadTimerCallback
    ( TimerHandle_t xTimer );
```

CHAP: 10 (VIDEO-03)

STOPPING THE AUTO-RELOAD TIMER DURRING RUNTIME:

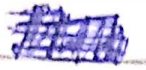
// In Previous Video, we learnt how to do a infite Timer and run timer at a specific time. So in this video we will learn how to stop a autoReload timer after a said time.

```
uint32_t timeout_count = 0;
const uint8_t STOP_MARK = 10;

void prvAutoReloadTimerCallback
  ( TimerHandle_t xTimer )
{
static TickType_t xTimeNow;

  // Obtain the current Tick count

xTimeNow = xTaskGetTickCount();
```

```
printf ("Auto-Reload timer callback
        executing : %d \n\r",
    (int) xTimeNow);

timeout_count ++;

if (timeout_count == STOP_MARK)
{
    printf ("STOP MARK
            Reached %d \n\r",
            (int) xTimeNow);

    xTimerStop (xAutoReloadTimer, 0);

}

}
```

CHAPTER : 11 (VIDEO-01)

RECEIVING DATA FROM THE UART USING POLLING:



// Before int main(void), make the definitions:

```
#define STACK_SIZE 128

static QueueHandle_t uart2_BytesRecei
                    =NULL;

void polledUartReceive (void *pvParameters);
void HandlerTask (void *pvParameters);

int main (void)

{  HAL_Init();
   SystemClock_Config();
   MX_GPIO_Init();
   ~~USART2_UART_TX_Init();~~
~~// gpio_init();~~
~~// adc_init();~~
```

```
xTaskCreate (polledUartReceive,
             "polledUartRx",
             STACK_SIZE,
             NULL,
             tskIDLE_PRIORITY +2,
             NULL);

xTaskCreate (HandlerTask,
             "uartPoint Task",
             STACK_SIZE,
             NULL,
             tskIDLE_PRIORITY. +3,
             NULL);

uart2_BytesReceived = xQueueCreate
                      (10, sizeof(char));

vTaskStartScheduler();

while(1)
{//btn_state = read_digital_sensor();
 //sensor_value = read_analog_sensor();
}
}
```

```
void polledUartReceive
     (void *pvParameters)

{
   uint8_t nextByte;
   USART2_UART_RX_Init();

   while(1)
   {
      nextByte = USART2_read();
      xQueueSend
         (uart2_BytesReceived,
          &nextByte, 0);
   }

}
```

```
char rcvByte

void HandlerTask(void *pvParameters)
{
    while(1)
    {
        xQueueReceive(uart2_BytesReceived,
        &rcvByte, portMAX_DELAY);
    }
}
```

CHAPTER - 11 : VIDEO - 02

CODING INTERACTING WITH INTERRUPT SERVICE ROUTINES: (ISRs)

```
//After while(1), implement the method

void start_rx_interrupt (void)

{
    rxInprogress = 1;
    USART2 -> CR1 |= 0x0020;
    // Enable Rx interupt
    NVIC_SetPriority(USART2_IRQn, 6);

    NVIC_EnableIRQ(USART2_IRQn);


void USAR2_IRQHandler(void)
{

    portBASE_TYPE

    xHigherPriorityTaskWoken = the
        = pdFALSE;
```

```
if (USART2->SR & 0x0020)
{
    uint8_t tempVal = (uint8_t)
        USART2->DR;

if (xNoProgress)
{
    xQueueSendFromISR
    (uart2_ByteReceived,
     &tempVal,
     &xHigherPriorityTaskWoken);
}
}
portYIELD_FROM_ISR(xHigherPriority
    TaskWoken);
}
```

```
char rcvByte;
void HandlerTask(void *pvParameters)
{
    USART2_RX_Init();
    start_rx_interrupt();

    while(1)
    {
xQueueReceive(uart2_BytesReceived,
    &rcvByte, portMAX_DELAY);
    }
}
```

VIDEO:03 : CHAPTER-11

RECEIVING AN ENTIRE DATA PACKET:

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h" → #include "semphr.h"
#include "usart.h"
#include "exti.h"
#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int __io_putchar(int ch);

#define STACK_SIZE 128
#define EXPECTED_PKT_LEN 5
static QueueHandle_t uart2_BytesReceived
                    = NULL;

static int rxInProgress = 0;
static uint16_t rxLen = 0;
static uint8_t* rxBuff = NULL;
static uint16_t rxItr = 0;
```

→ Itterator cloop

→ Pointer to the Buffer, where we need to store

→ rxLength, weather we wish to recieve 5 bytes

```
void HandlerTask(void *pvParameters)
int32_t start_at_interrupt
    (uint8_t* Buffer, uint_fast16_t le

static SemaphoreHandle_t xDone = NULL;


int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
xDone = xSemaphoreCreateBinary();

xTaskCreate(HandlerTask,
            "uartPrintTask",
            STACK_SIZE,
            NULL,
            tskIDLE_PRIORITY+3,
            NULL);

uart2_BytesReceived = xQueueCreate
            (10, sizeof(char));

vTaskStartScheduler();
```

```
while(1)
    { }
}

int32_t  start_rx_interrupt
  (uint8_t * Buffer, uint_fast16_t Len)
{
  if (!rxInProgress && (Buffer != NULL))
  {
    rxInProgress = 1;
    rxLen = Len;
    rxBuff = Buffer;
    rxIto = 0;

    USART2->CR1 |= 0x0020; // Enable Rx Interupt
Rx InterruptPriority
    NVIC_SetPriority(USART2_IRQn, 6);
    NVIC_EnableIRQ(USART2_IRQn);
    Enable NVIC
    return 0;

  }

  return -1;
}
```

```
void USART2_IRQHandler(void)
{
  portBASE_TYPE xHigherPriorityTaskWoken
      = pdFALSE;

  if (USART2->SR & 0x0020)
  {
      uint8_t tempVal = (uint8_t)
                  USART2->DR;

      if (rxInProgress)
      {
          rxBuff[rxIter++] = tempVal;
          if(rxIter == rxLen)
          {
              rxInProgress = 0;
              xSemaphoreGiveFromISR
              (rxDone, &xHigherPriority
                        TaskWoken);
          }
      }
  }
  portYIELD_FROM_ISR
  (xHigherPriorityTaskWoken);
}
```

uint8_t rxData[EXPECTED_PKT_LEN];
char rxCode[50] = { };

void HandleTask(void *pvParameters)
{
    USART2_UART_RX_Init(); // Initialize rx Uart

    for (int i=0; i< EXPECTED_PKT_LEN; i++)
    {
        rxData[i] = 0; // Initialize Buffer to Store Recived data
    }

    → Timeout Value
    const TickType_t timeout = pdMS_TO_TICKS(8000);

    while(1)
    {
        start_rx_interrupt(rxData, EXPECTED_PKT_LEN);

        if(xSemaphoreTake(rxDone, timeout) == pdPASS)
        {
            if(EXPECTED_PKT_LEN == rxIts)
            {
                sprintf(rxCode, "Received");
            }
            else { sprintf(rxCode, "DataLoss"); }
        }
    }

```
if (xSemaphoreTake (xRxDone, timeout)
        == pdPASS)

{
    if (EXPECTED_PKT_LEN == xLen
    {
        sprintf (rxCode, "received");
    }
    else
    {
        sprintf (rxCode, "Length Mismatch
    }
}
else
{
    sprintf (rxCode, "timeout");
}

}

}
```

CHAPTER: 12 (VIDEO-01)

EVENT-GROUPS: INTRODUCTION

→ Event group allow a task to wait in the Blocked state for a combination of one or more events to occour.

→ Event groups unblock all the tasks that were waiting for the same event, or combination of events, when the evet occours.

USES:

→ Synchoronizing tasks.
→ Broadcasting events to multiple tasks.

Event Groups: Advantages

→ They reduce RAM usage by allowing us to replace multiple binary semaphores with a single event group.

Characteristics:

→ An event group is a set of "event flags"

→ An event flag is a boolean value (1 or 0) used to indicate wheather an event has occoured or not.

→ An event flag is stored in a single bit & the state of all event flags can be stored in a single variable.

→ Event flags are also called "event bits"

Configuration:

→ The number of event bits in an event group is dependent on the:

configUSE_16_BIT_TICKS

→ freeRTOSConfig.h

→ If configUSE_16_BIT_TICKS = 1, then each event group contains 8 useable event bits.

→ If configUSE_16_BIT_TICKS = 0, then each event group contains 24 useable event bits.

CHAPTER: 12 - VIDEO - 2

SETTING & READING EVENTBITS IN AN EVENTGROUP:

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"  → #include "event_groups.h"
#include "uart.h"
#include "exti.h"
#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init();

EventGroupHandle_t xEventGroup;
```

Function Prototypes

```
#define TASK1_BIT (1UL<<0UL)
#define TASK2_BIT (1UL<<1UL)

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  USART2_UART_TX_Init();
```

shift

static void vEventBitSettingTask(void *pvParameters);
static void vEventBitReadingTask(void *pvParameters);

```
printf("System Initializing.....\n\r");
xEventGroup = xEventGroupCreate();
```

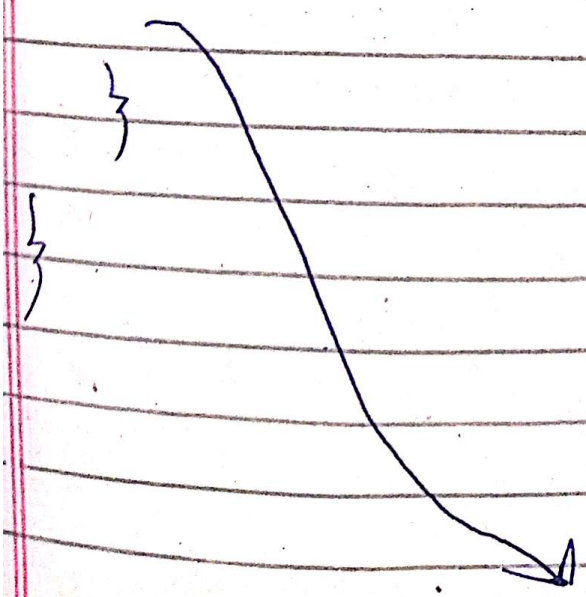
Store Handle Return of Event Group in a Variable

Handle Return

```
xTaskCreate( vEventBitSettingTask,
             "BitSetter",
             100,
             NULL,
             1,
             NULL);

xTaskCreate( vEventBitReadingTask,
             "BitReader",
             100,
             NULL,
             1,
             NULL);

vTaskScheduler();
```

```
while(1)
  { }
}

static void vEventBitSettingTask
      (void *pvParameters)
{
    const TickType_t xDelay500ms
          = pdMS_TO_TICKS (500);

    while(1)
    {
    → vTaskDelay (xDelay500ms);
    → printf ("Bit setting task
       -\t about to set bit 0.
       \r\n");
    → xEventGroupSetBits
       (xEventGroup, TASK1_BIT);


    → vTaskDelay (xDelay500ms);
    → printf ("Bit setting task -\t
       about to set bit 1.\r\n");
    → xEventGroupSetBits(xEventGroup,
          TASK2_BIT);
    }}
```

```
static void vEventBitReadingTask( void *pvParameters )
{
EventBits_t xEventGroupValue;
const EventBits_t xBitsToWaitFor = ( TASK1_BIT | TASK2_BIT );

const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL );

while( 1 )
{
xEventGroupValue = xEventGroupWaitBits(
                        xEventGroup,
                        xBitsToWaitFor,
                        pdTRUE,
                        pdFALSE,
                        portMAX_DELAY );
```



```
}
}
```

```
if( (xEventGroupValue & TASK1_BIT)!
{
  printf("BIT READING TASK:
          EVENT BIT 0 SET -
          TASK 1 \r\n");
}

;f( (xEventGroupValue & TASK2_BIT)
{
  printf("BIT READING TASK:
          EVENT BIT 1 SET -
          TASK 2 \r\n");
}

  vTaskDelay(xDelay200ms);
```

VIDEO - 03 : CHAPTER - 12

EVENT GROUPS WITH MULTIPLE SETTERS:

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h" → include "event_groups.h"
#include "uart.h"
#include "exti.h"
#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

---

→ #define TASK1_BIT (1UL<<0UL)
#define TASK2_BIT (1UL<<1UL)
#define TASK3_BIT (1UL<<2UL)

---

EventGroupHandle_t xEventGroup;
const TickType_t xDelay500ms =
    pdMS_TO_TICKS(500UL);
→ static void InputTask1(void *pvParameteres);
→ static void InputTask2(void *pvParameteres);
→ static void InputTask3(void *pvParameteres);
→ static void OutputTask(void *pvParameteres);
```

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    USART2_UART_TX_Init();

    printf("System Initializing...\n\r");

    xEventGroup = xEventGroupCreate();

→   xTaskCreate(InputTask1, "InputTask1",
                100, NULL, 1, NULL);

→   xTaskCreate(InputTask2, "InputTask2",
                100, NULL, 1, NULL);

→   xTaskCreate(InputTask3, "InputTask3",
                100, NULL, 1, NULL);

→   xTaskCreate(OutputTask, "OutputTask",
                100, NULL, 1, NULL);

    vTaskStartScheduler();
```

```
while (1)
{
}
}
```

```
static void InputTask1 (void *pvParameters)
{
    while(1)
    {
        // Do something

        xEventGroupSetBits (xEventGroup,
                            TASK1_BIT);
    }
}
```

```
static void InputTask2(void *pvParameters)
{
    while (1)
    {
        // Do something

        xEventGroupSetBits (xEventGroup,
                            TASK2_BIT);
    }
}
```

```
static void InputTask3
        (void *pvParameters)
{
    while(1)
    {
        //Do Something

        xEventGroupSetBits
        (xEventGroup, TASK3_BIT);

    }
}
```

```
static void OutputTask(void *pvParameters)
{
const EventBits_t xBitsToWaitFor
    = (TASK1_BIT | TASK2_BIT |
        TASK3_BIT);

EventBits_t xEventGroupValue;

while(1)

{ xEventGroupValue = xEventGroupWaitBits
                (xEventGroup,
                 xBitsToWaitFor,
                 pdTRUE,
                 pdFALSE,
                 portMAX_DELAY);
if((xEventGroupValue & TASK1_BIT) != 0)
{printf("Bit reading task -\t event bit0 was set
        - TASK1 \r\n");}
if((xEventGroupValue & TASK2_BIT) != 0)
{printf("Bit reading task -\t event bit1 was
        set - TASK2 \r\n");}
if((xEventGroupValue & TASK3_BIT) != 0)
{printf("Bit reading task -\t even bit2 was
        set - TASK3 \r\n");}
}
}
```

CHAP:12 VIDEO-03: TASKS USING SYNCHRONIZING EVENT GROUPS

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h" → include "event_groups.h"

#include "uart.h"
#include "exti.h"
#include "adc.h"

#define TASK1_BIT (1UL<<0UL)
#define TASK2_BIT (1UL<<1UL)
#define TASK3_BIT (1UL<<2UL)

EventGroupHandle_t xEventGroup;

EventBits_t uxAllSyncBits =
(TASK1_BIT | TASK2_BIT | TASK3_BIT);

static void Task1(void *pvParameters);
static void Task2(void *pvParameters);
static void Task3(void *pvParameters);
```

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  USART2_UART_TX_Init();

  printf("System Initializing... \n\r");

  xEventGroup = xEventGroupCreate();

  xTaskCreate(Task1, "Task 1", 128, NULL,
              1, NULL);

  xTaskCreate(Task2, "Task 2", 128, NULL,
              1, NULL);

  xTaskCreate(Task3, "Task 3", 128, NULL,
              1, NULL);

  vTaskStartScheduler();

  while(1) { }
}
```

```
static void Task1
  (void *pvParameters)
{
  EventBits_t uxReturn;
  while(1)
  {
    uxReturn = xEventGroupSync
      xEventGroup,
      TASK1_BIT,
      uxAllSyncBits,
      portMAX_DELAY
      );
    if((uxReturn & uxAllSyncBits)
      == uxAllSyncBits)
    {
      //Do something
    }
  }
}
```

```
static void Task2
    (void *pvParameters)
{
  EventBits_t  uxReturn;

  while(1)

  {
    uxReturn = xEventGroupSync(
                xEventGroup,
                TASK2_BIT,
                uxAllSyncBits,
                portMAX_DELAY
               );

  if((uxReturn & uxAllSyncBits)
       == uxAllSyncBits)

    {
        //Do something
    }
}

}
```

```
static void Task3
    (void *pvParameters)
{
    EventBits_t uxReturn;

    while (1)

    { uxReturn = xEventGroupSync(
                xEventGroup,
                TASK3_BIT,
                uxAllSyncBits,
                port_MAX_DELAY
                );

    if ((uxReturn & uxAllSyncBits)
          == uxAllSyncBits)

        {
            // Do Something
            printf("ALL TASKS SET
        } for(int i=0; i<60000;i++)
        }
}
```

CHAPTER: 13 (VIDEO-01)

TASK-NOTIFICATIONS:

Notifications allow tasks to communicate with each other directly without going through communication objects such as queues, semaphores & event groups.

Enable

configUSE_TASK_NOTIFICATIONS
→ in FreeRTOSConfig.h

STATES:

- **Pending:** When a task receives a notification, its notification state is set to pending.
- **Not Pending:** When a task reads its notification value, its notification state is set to not-pending.

Advantages:

→ Faster than using queues, semaphores, & event groups to perform an equivalent operation

→ Require significantly less RAM, then using a queue, semaphore or an event group.

Limitations:

→ Unlike queues, semaphores & event groups, task notifications cannot be used to send events or data from a task to an ISR.

(although they can be used to send events or data from an ISR to a task)

→ Unlike queues, semaphores & event groups, task notifications cannot be sent to multiple tasks.

CHAPTER-13 : VIDEO-02

Working with TASK Notifications

```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"

#include "uart.h"
#include "exti.h"
#include "adc.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int __io_putchar(int ch);

static TaskHandle_t xHandlerTask = NULL;

void HandleTask(void *pvParameters);
```

freeRTOSConfig.h

configUSE_TASK_NOTIFICATIONS 1

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  USART2_UART_TX_Init();
  p13_interrupt_init();
}

xTaskCreate(HandlerTask,
            "HandlerTask",
            STACK_SIZE,
            NULL,
            3,
            &xHandlerTask);

vTaskStartScheduler();

while(1) {}
}
```

```
void HandlerTask( void *pvParameters )
{
    const TickType_t xMaxExpectedBlockTime
            = pdMS_TO_TICKS( 100 );

    while (1)
    {
        if( ulTaskNotifyTake( pdFALSE,
                              xMaxExpectedBlockTime ) != 0 )
        {
            // Do Something...

            printf( "Handler task - Processing event.
                    \r\n" );

        }

        else
        {
            // Do Something...
        }
    }
}
```

```
void EXTI5_10_IRQHandler(void)
{

  BaseType_t xHigherPriorityTaskWoken
      = pdFALSE;

  vTaskNotifyGiveFromISR(xHandlerTask,
      &xHigherPriorityTaskWoken);

  // clear interrupt pending flag

  EXTI->PR = 0x2000;

  // request a context switch

  portYIELD_FROM_ISR
      (xHigherPriorityTaskWoken);
}
```

VIDEO-1 : CHAPTER-14
THE FREE RTOS SCHEDULER

"The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state."

ROUND ROBIN:

The freeRTOS scheduler will ensure tasks that share a priority are selected to enter the Running state in turn, this policy is referred to as Round Robin Scheduling.

The Round Robin scheduling algorithm in FreeRTOS does not gurantee time is shared equally between tasks of equal priority, only that Ready state tasks of equal priority will enter the Running state in turn.

FIXED PRIORITY:

The scheduler does not change the priority assigned to the tasks being scheduled, but also does not prevent the task from changing their own priority or that of other tasks.

PRE-EMPTIVE:

The scheduler will immediately preempt the Running state task if a task that has a priority higher than the Running state task entess the Ready state.

(Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state & into the

Ready state to allow a different task to enter the Running state.

TIME SLICING:

This scheduler shares processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state.

Scheduling algorithms described as using "Time Slicing" will select a new task to enter the Running state, at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task.

"A time slice is equal to the time btw 2 RTOS tick interrupts"

Enable

configUSE_PREEMPTION() 1

configUSE_TIME_SLICING() 1

→ in FreeRTOSConfig.h

static void MX_GPIO_Init(void)

static void MX_USART2_UART_Init(void);

VIDEO-02 : Chap-14

THE SCHEDULER PREEMPTION ONLY:

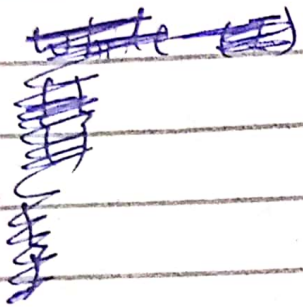
```
#include <stdio.h>
#include "main.h"
#include "cmsis_os.h"
```

// Inside Core > Inc > FreeRTOSConfig.h

```
#define configUSE_PREEMPTION 1
#define configUSE_TIME_SLICING 0
```

```
typedef uint32_t TaskProfiler_t;

TaskProfiler_t OrangeTaskProfiler,
               RedTaskProfiler,
               GreenTaskProfiler,
               BlueTaskProfiler;

void vBlueLedControllerTask(void *pvParameters);
void vRedLedControllerTask(void *pvParameters);
void vOrangeLedControllerTask(void *pvParameters);
void vGreenLedControllerTask(void *pvParameters);

void SystemClock_Config(void);
```

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART2_UART_Init();
```



```
xTaskCreate(vOrangeLedControllerTask,
            "Orange Led Controller",
            100,
            NULL,
            1,
            NULL
            );

xTaskCreate(vBlueLedControllerTask,
            "Blue Led Controller",
            100, NULL,
            2, NULL
            );
```

```
xTaskCreate( vRedLedControllerTask,
             "Red Led Controller",
             100,
             NULL,
             2,
             NULL
             );

xTaskCreate( vGreenLedControllerTask,
   "Green Led Controller", 100,
      NULL, 1,
      NULL
      );

while (1) {}

}
```

```
void vBlueLedControllerTask
   (void *pvParameters)
{
    int i;
    while (1)
    {   Blue_TaskProfiler++;
        for(i=0; i<=50000; i++){}
    }
}
```

```
void vRedled (controllerTask
    (void *pvParameters)
{
  int i;
  while (1)
  {
     Red_TaskProfiler++;
     for (i=0; i<700000; i++){}
  }
}
```

```
void vOrangeled (controllerTask
    (void *pvParameters)
{  int i;
  while (1)
  {
     Orange_TaskProfiler++;
     for (i=0; i<700000; i++){}
  }
}
```

```
void vGreenControllerTask
    (void *pvParameters)
{
  int i;
  while (1)
  {
    Green_TaskProfiler++;
    for (i=0; i<700000; i++) { }
  }
}
```

2.

CHAP-14: VIDEO-02

PSUEDO-TIME-SLICING

→ Everything remains the same as of previous video.

→ We give all 2ed Tasks same priority 1.

→ Disable configUSE_TIME_SLICING

→ Declare a global variable for time, above int main(void)

```
const TickType_t 50ms =
pdMS_TO_TICKS(50);
```

→ In each task after

```
for(i=0; i<700000; i++){}
add a delay
v TaskDelay(_50ms);
```

CHAP-15: VIDEO-01

INTRODUCTION TO CMSIS-RTOS:

→ RTOS APIs define the programming interface (i.e: functions) that programmers use when interacting with the RTOS.

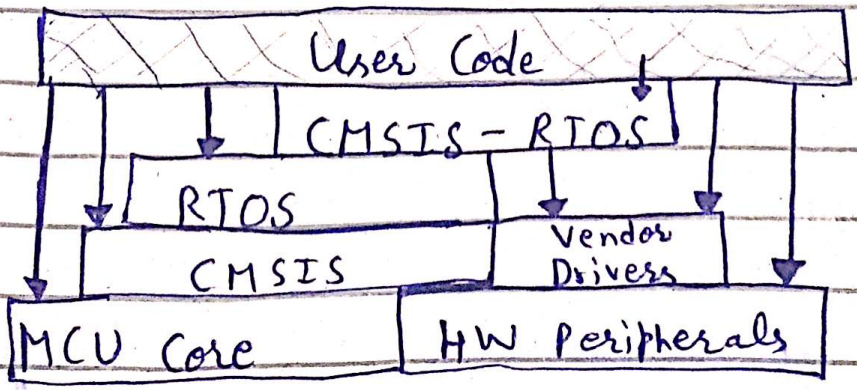
→ RTOS APIs expose all the RTOS functions.

Generic RTOS APIs:

→ They are not tied to a specific RTOS.

→ They are implemented as a wrapper layer above the RTOS APIs.

• CMSIS is Cortex-MCU soft. int. Standard.



There are 2 generic APIs that can be used with FreeRTOS:

→ CMSIS-RTOS:
Cortex Microcontroller Software Interface - RTOS

This is vendor-independent API standard created by ARM.

→ POSIX: Portable Operating System Interface

This API is mostly used general purpose operating systems such as Linux.

Pros:

→ Allows us to write code once & run on multiple MCUs, changing only the RTOS.

→ Allows middleware vendors to write code that interacts with a single API while at the same time supports multiple real-time operating systems.

Cons:

→ Because of its generic nature, unique portions of a particular RTOS may be left out.

Delay Functions:

Free RTOS	CMSIS-RTOS
✓ TaskDelay()	osDelay()
✓ TaskDelayUntill()	osDelayUntil()
✗ TaskGetTickCount()	

Kernel Control:

- ✓ TaskStartScheduler() → KernelStart
- ✗ TaskGetSchedulerState() → KernelUnb
- ✗ TaskResumeAll()
- ✗ TaskGetTickCount() → KernelGet TickCount()

Message Queues:

- ✓ QueueUnregisterQueue() → MessageQueue Delete()
- ✓ QueueDelete()

- ✗ QueueReceive() → MessageQueueGetL
- ✗ QueueCreate() → MessageQueueNew
- ✗ QueueSendToBack() → MessageQueuePutL

Mutexes & Semaphores:

x SemaphoreTake() — os MutexRequest()
x SemaphoreTakeRecursive()

x SemaphoreGive() — os MutexRelease()
x SemaphoreGiveRecursive()

v SemaphoreDelete() — os MutexDelete()
v QueueUnregisterQueue()

x SemaphoreCreateMutex() — os MutexNew()

x SemaphoreGive() — os MutexRelease()
x SemaphoreGiveRecursive()

v SemaphoreDelete() — os SemaphoreDelete()
v QueueUnregisterQueue()

ux SemaphoreGetCount() — os SemaphoreGetCount()

TIMERS:

x TimerDelete() — os TimerDelete()

pc TimerGetName() — os TimerGetName()

x TimerIs TimerActive() os TimerIs Running()

x TimerCreate() os TimerNew()

Consideration for migrations:

→ CMSIS-RTOS task creation functions stack size argum is in bytes while FreeRTOS takes words (4 bytes) as an argument.

→ CMSIS-RTOS functions use structures as inputs.