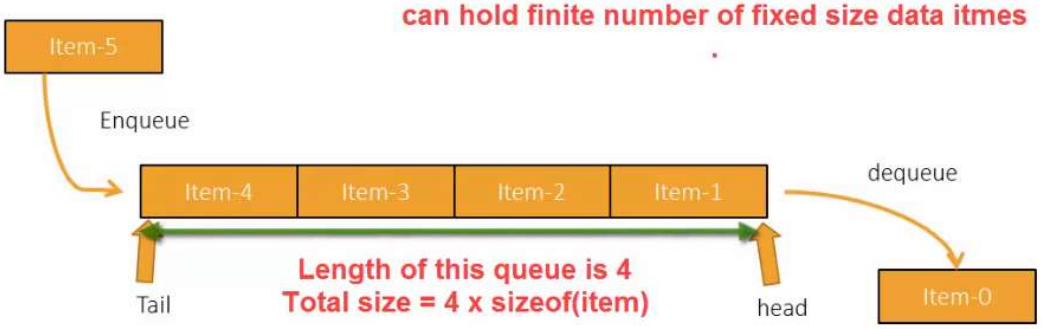
Free RTOS Part II

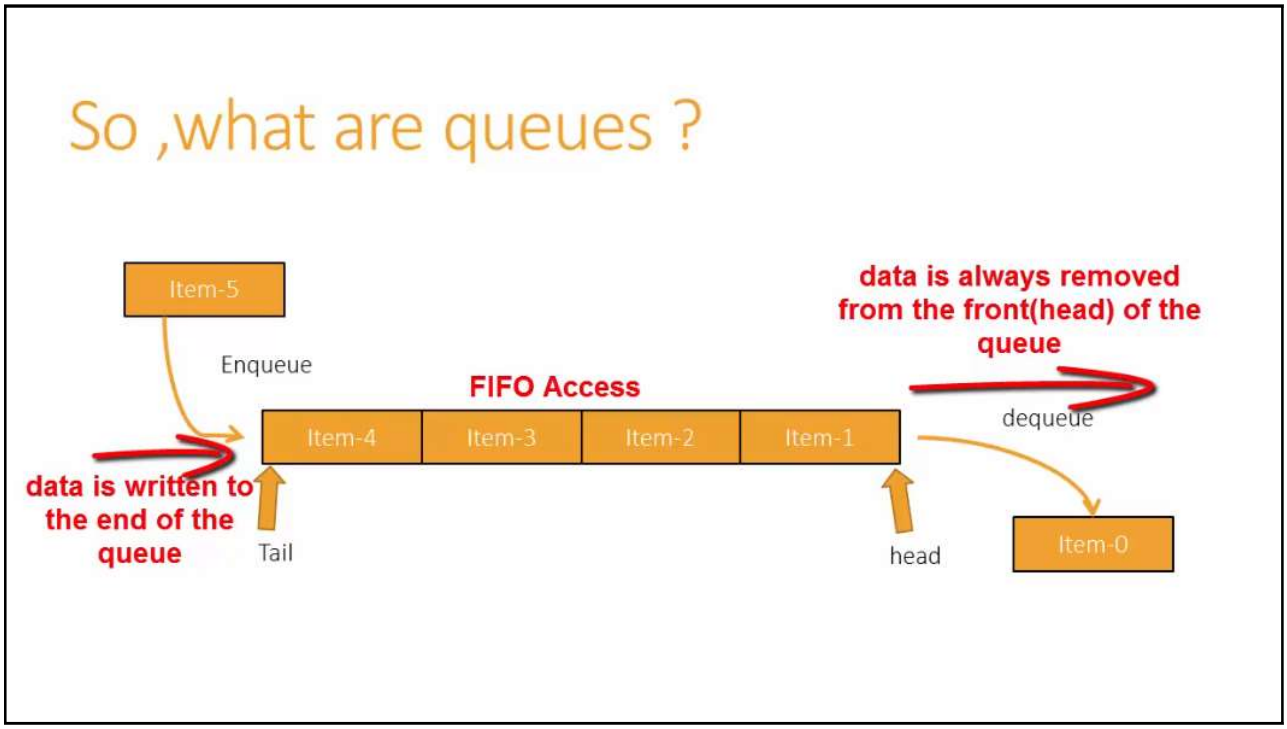
1

So ,what are queues ?

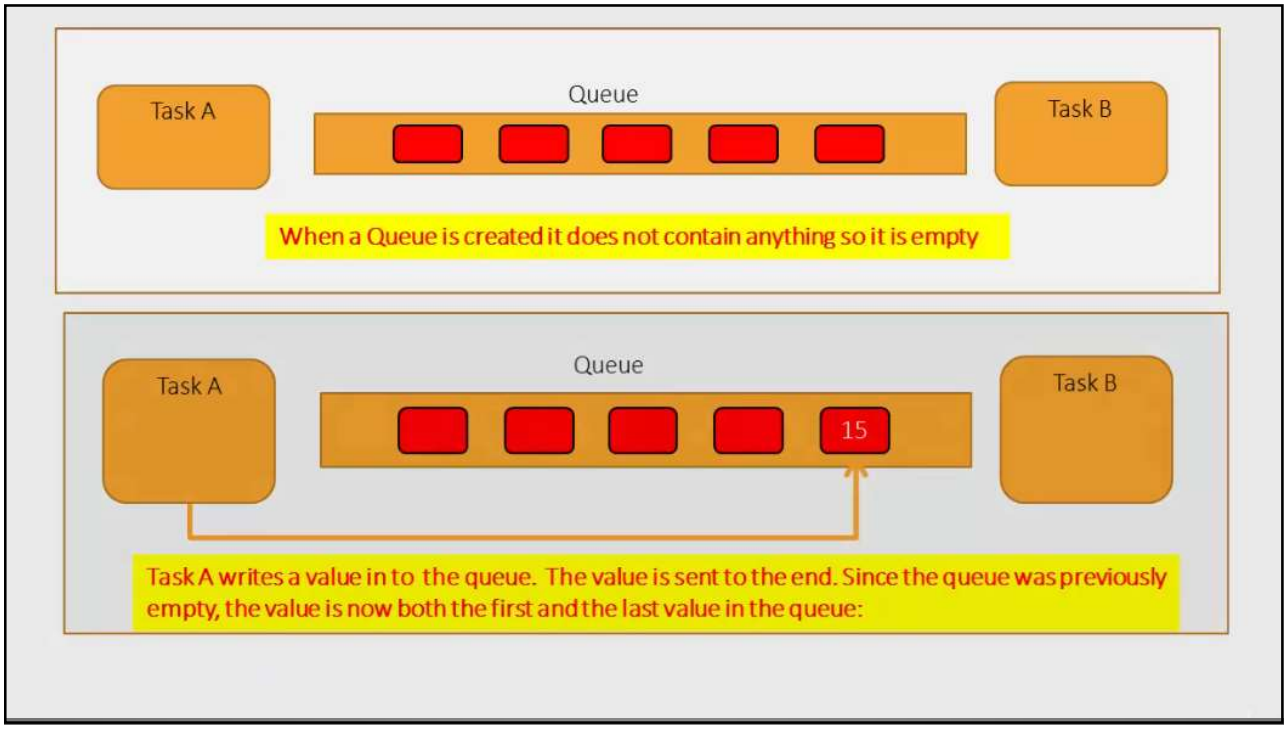
A Queue is nothing but a data structure which can hold finite number of fixed size data itmes



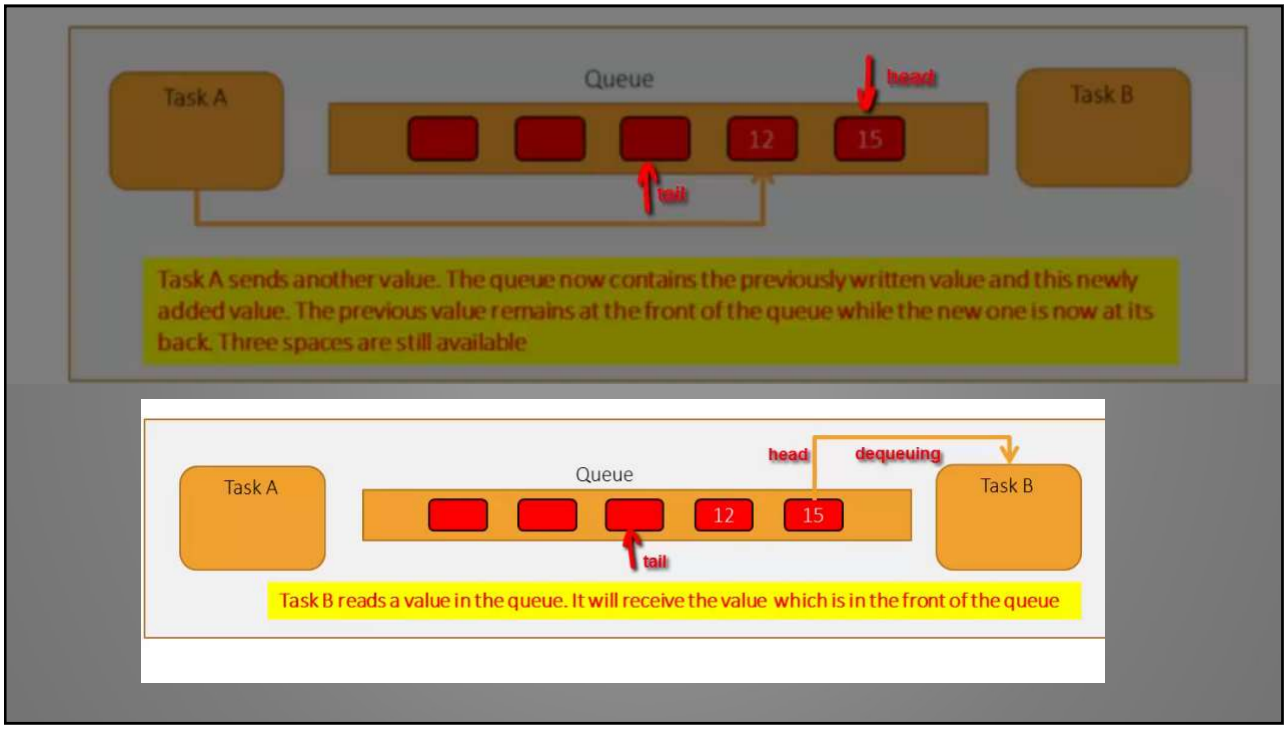
2


So ,what are queues ?
Item-5
Enqueue
FIFO Access
data is always removed from the front(head) of the queue
dequeue
Item-4
Item-3
Item-2
Item-1
data is written to the end of the queue
Tail
head
Item-0

3


Task A
Queue
Task B
When a Queue is created it does not contain anything so it is empty
Task A
Queue
Task B
15
Task A writes a value in to the queue. The value is sent to the end. Since the queue was previously empty, the value is now both the first and the last value in the queue:

4



5

FreeRTOS API to Create a Queue

This handle is then used to do lots of queue operations like writing , reading, seeking,deleting, etc

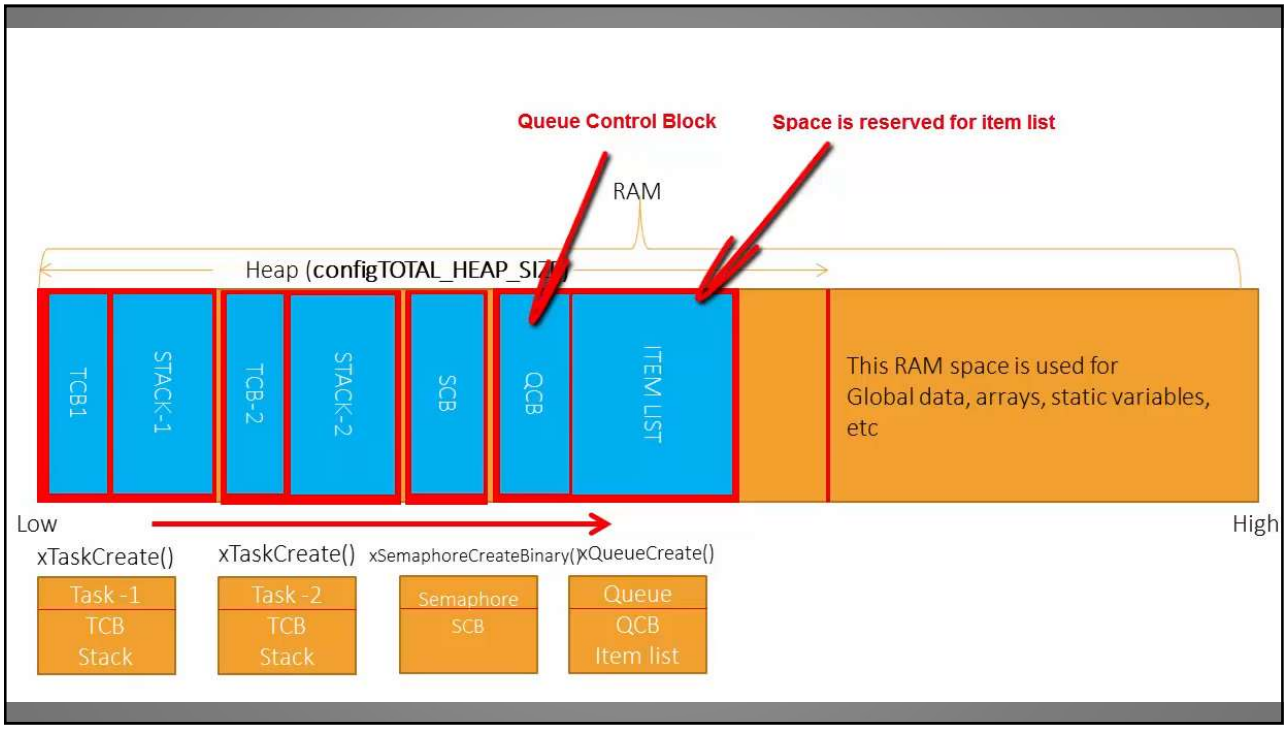
mention here, How many items this queue should hold (i.e. length)

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize );
```

if the creation is successful the API returns the reference(a Pointer) to the created queue otherwise NULL.

here you should mention, what's the size of single item in bytes.

6



7

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

    /* Create a queue capable of containing 10 unsigned long values.
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    if( xQueue1 == NULL )
    {
        /* Queue was not created and must not be used. */
    }

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These are to be queued by pointers as they are
    relatively large structures. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    if( xQueue2 == NULL )
    {
        /* Queue was not created and must not be used. */
    }

    /* ... Rest of task code. */
 }
```

10 chunks

4 bytes

Total Size = 40 bytes

- xQueue1 က QCB နဲ့ ITEM LISTS များကို pointer အနေနဲ့ လှမ်းသုံး။
- memory မလုံလောက်ပါက NULL return ပြန်မှာပါ။

8

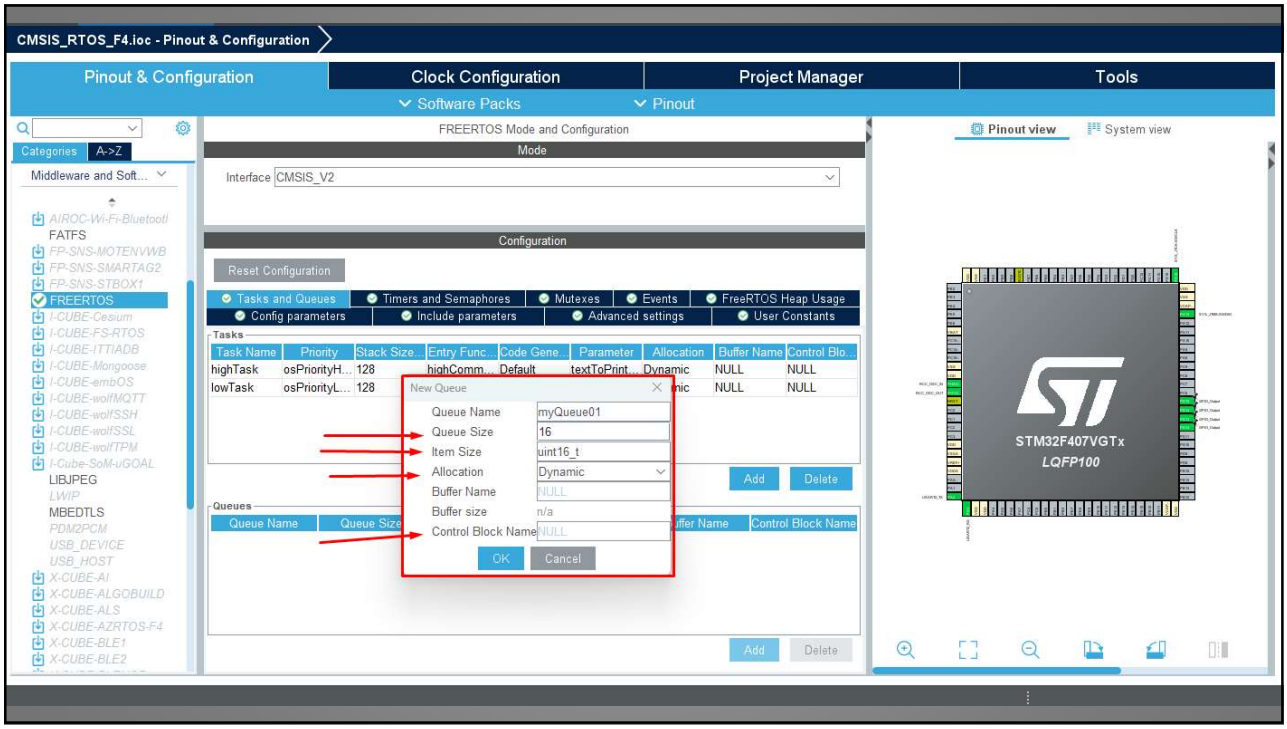
```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};
```

```
/* Create a queue capable of containing 10 pointers to AMessage
structures.  These are to be queued by pointers as they are
relatively large structures. */
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

if( xQueue2 == NULL )
{
    /* Queue was not created and must not be used. */
}

/* ... Rest of task code. */
}
```

Here, this queue will hold 10 different struct pointers instead of struct itself.

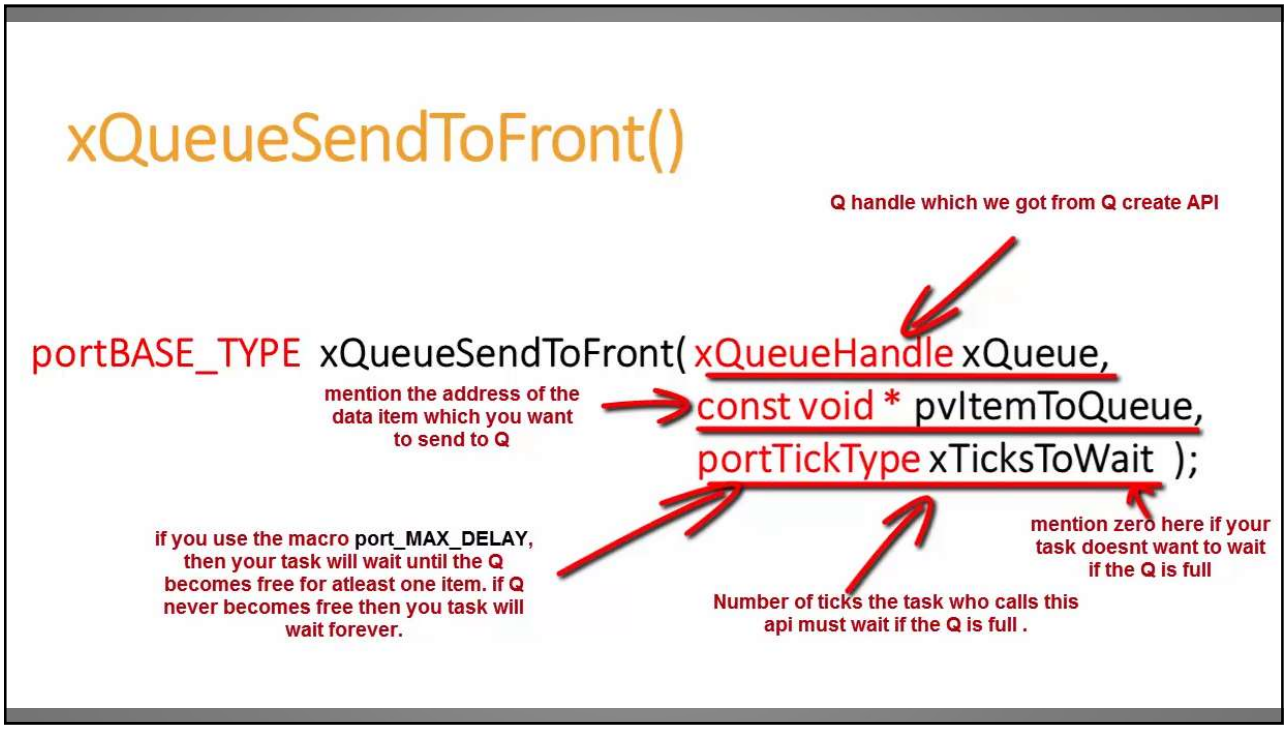
- QCB အတွက်လည်း heap ထဲက size အနဲငယ်ယူသုံးသေးတာကိုသတိပြုရန်
- AMessage Structure အသုံးပြုပြီး Queue တစ်ခုဆောက်မှာပါ။
- ပုံမှန် Structure တစ်ခုလုံးရဲ့ size က 21 bytes ဖြစ်မယ်။ Queue ဆောက်ထားတာက 10 chunks ဆိုရင် = 210 bytes ဖြစ်ပါတယ်။
- Space ကများလွန်းတဲ့အတွက်၊ Amessage Structure ကို pointer အနေနဲ့ ပြန်သုံးလိုက်ခြင်းဖြစ် Struct ရဲ့ size က 4 bytes နဲ့ 10 chunks ဆိုရင် 40 bytes ပဲပြန်ရမှာပါ။

9



10

xQueueSendToFront()



11

```
unsigned long ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These should be passed by pointer as they contain a lot of
    data. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    /* ... */

    if( xQueue1 != 0 )
    {
        /* Send an unsigned long.  Wait for 10 ticks for space to become
        available if necessary. */
        if( xQueueSendToFront( xQueue1,
                               ( void * ) &ulVar,
                               ( TickType_t ) 10 ) != pdPASS )
        {
            /* Failed to post the message, even after 10 ticks. */
        }
    }
```

12

xQueueSendToBack()

```
portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait );
```

This API will post the data item to the back(tail) of the Q

13

xQueueReceive()

This API will Read a data item from the Q. The data item which is being read will be removed from the Q.

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
                            const void * pvBuffer,
                            portTickType xTicksToWait );
```

the pointer which you supply here will hold the data item which is being read.

returns TRUE if read is successfull
returns QUEUE_EMPTY if Q is Empty even after xTicksToWait

if the Q is empty, then mention here for how many number of ticks your task wishes to block or wait !

14

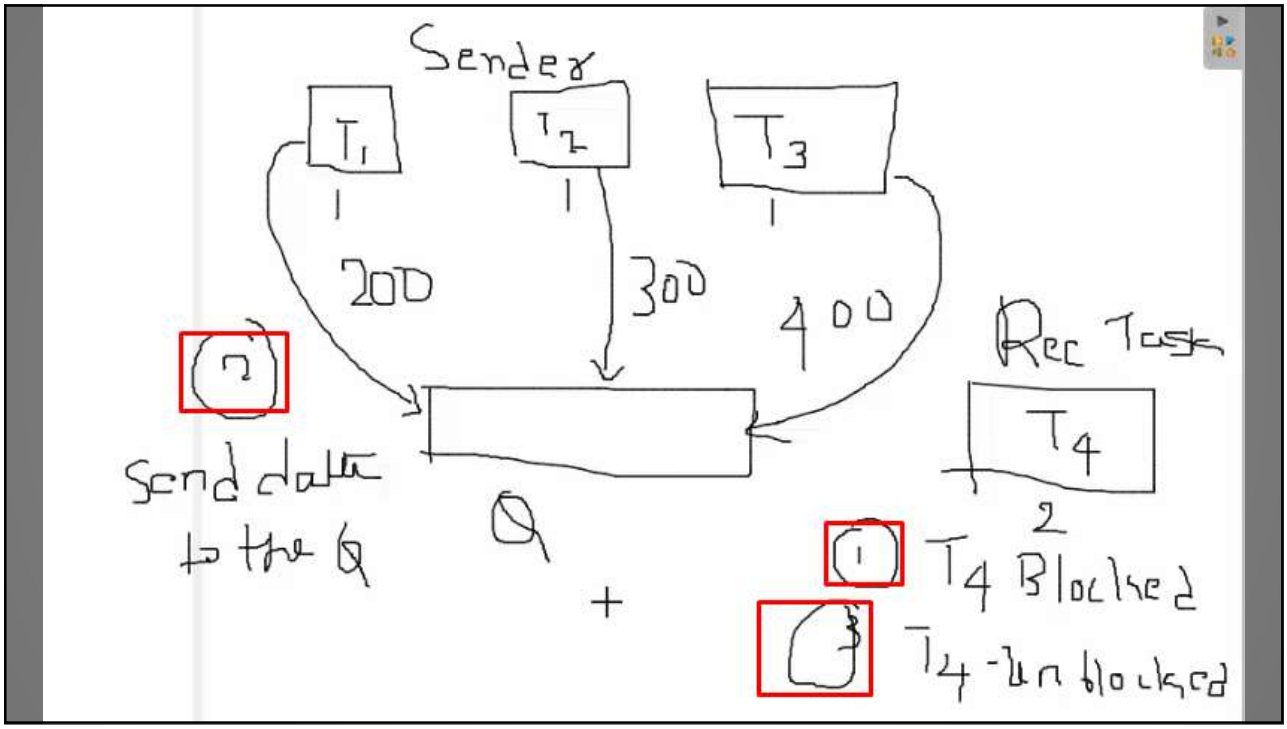
xQueuePeek()

In this case, the data item being read is not removed from the Queue

```
portBASE_TYPE xQueuePeek(xQueueHandle xQueue,
                         const void * pvBuffer,
                         portTickType xTicksToWait );
```

Receive က Queue ထဲ data ကို ထုတ်ယူသွားတာဖြစ်တယ်။
Peek က တော့ Queue ထဲက data ကို ထုတ်မယူသွားပဲ copy ကူးသွားမှာဖြစ်တယ်။

15



16

```c
/* USER CODE END Header_StartTask01 */
void SenderStartFunc(void *argument){
    long sendVal;
    osStatus_t  xStatus;

    sendVal = (long) argument;
    printf("Started put to Queue..\n");
    for(;;){
        xStatus = osMessageQueuePut(myQueue01Handle, &sendVal, 0,0);
        if (xStatus != osOK){
            printf(" Couldn't send to queue01.\r\n");
        }
        taskYIELD(); //same as portYIELD() in CMSIS-API
        BLUE_LED_TOGGLE();
        //osDelay(1);
    }
}
```

osMessageQueuePut()

osMessageQueueGet()

```c
void readQueueInit(void *argument)
{
  /* USER CODE BEGIN readQueueInit */
    long recVal;
    osStatus_t xStatus;
    const portTickType xTicksToWait = 100/portTICK_RATE_MS; //100 ms
  /* Infinite loop */
  for(;;)
  {
      printf("About to read value from Queue 01 \n");
      xStatus = osMessageQueueGet(myQueue01Handle, &recVal, NULL, xTicksToWait);
      if (xStatus == osOK){
          printf ("Received Value : %ld \n", recVal);
          GREEN_LED_TOGGLE();

      }
      else {
          printf ("Not Received ! \n");
          RED_LED_ON();
      }
    //osDelay(1);
  }
  /* USER CODE END readQueueInit */
}
```

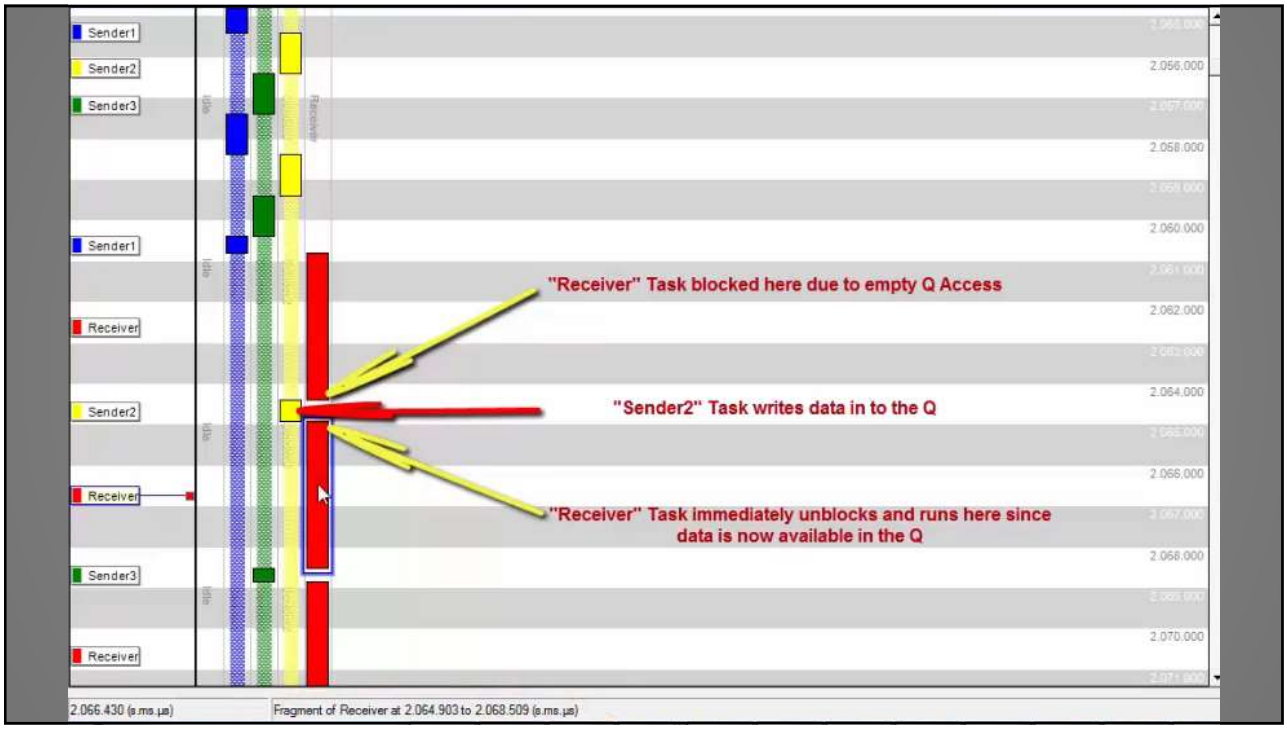
17

```c
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for myTask02 */
osThreadId_t myTask02Handle;
const osThreadAttr_t myTask02_attributes = {
  .name = "myTask02",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for myTask03 */
osThreadId_t myTask03Handle;
const osThreadAttr_t myTask03_attributes = {
  .name = "myTask03",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for readQueue */
osThreadId_t readQueueHandle;
const osThreadAttr_t readQueue_attributes = {
  .name = "readQueue",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityHigh,
};
/* Definitions for myQueue01 */
osMessageQueueId_t myQueue01Handle;
```

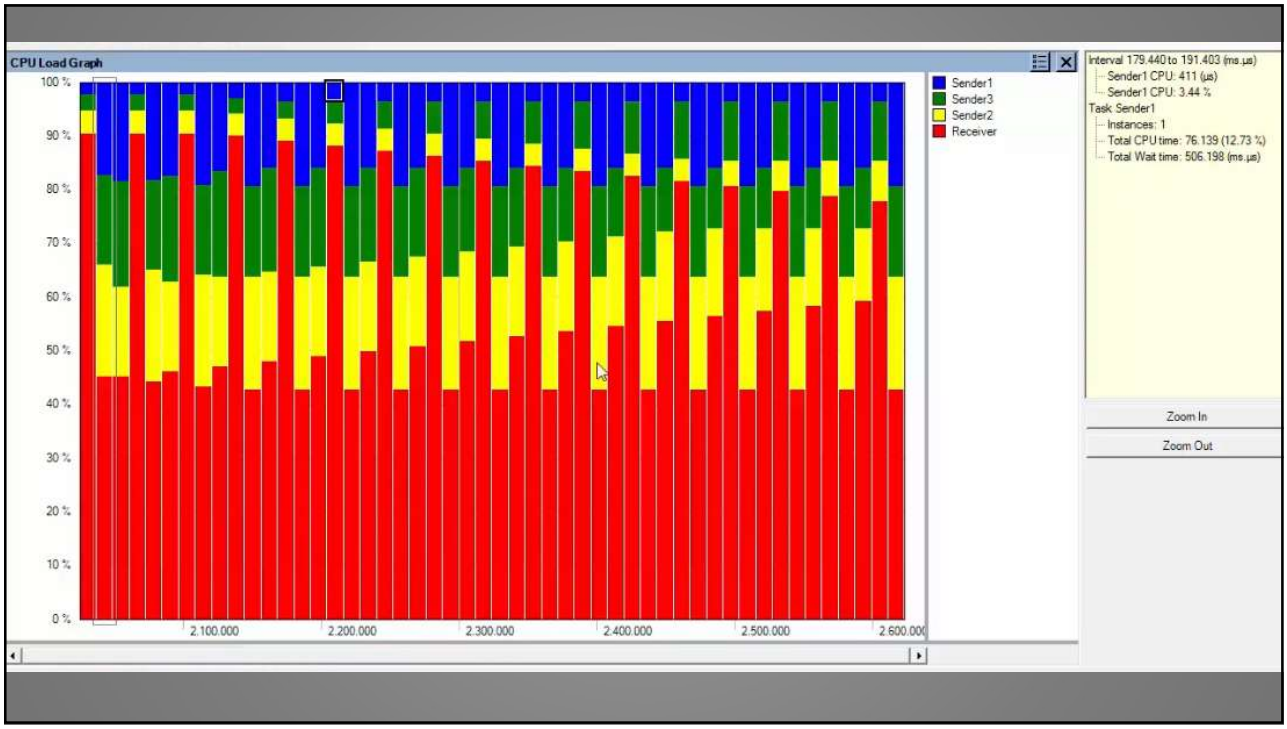
```
SWV ITM Data Console - ITM Port: 0
osKernel Initialized....
About to read value from Queue 01
Started put to Queue..
Received Value : 200
About to read value from Queue 01
Started put to Queue..
Received Value : 300
About to read value from Queue 01
Started put to Queue..
Received Value : 400
About to read value from Queue 01
Received Value : 200
About to read value from Queue 01
Received Value : 300
About to read value from Queue 01
Received Value : 400
About to read value from Queue 01
Received Value : 200
About to read value from Queue 01
Received Value : 300
About to read value from Queue 01
```

```
File download complete
Time elapsed during download operation: 00:00:00.815
```

18

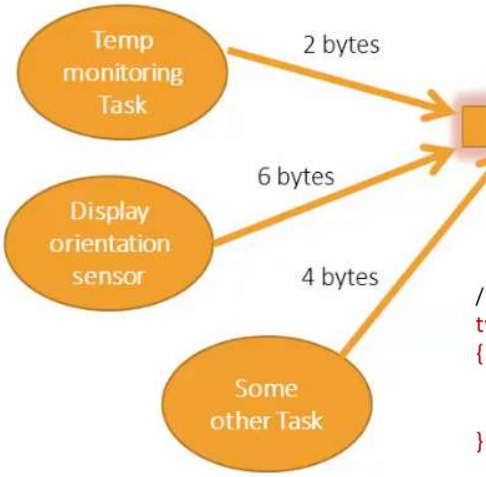

Sender1
Sender2
Sender3
Sender1
Receiver
Sender2
Receiver
Sender3
Receiver
"Receiver" Task blocked here due to empty Q Access
"Sender2" Task writes data in to the Q
"Receiver" Task immediately unblocks and runs here since data is now available in the Q
2.056.000
2.058.000
2.060.000
2.062.000
2.064.000
2.066.000
2.068.000
2.070.000
2.066.430 (s.ms.µs)
Fragment of Receiver at 2.064.903 to 2.068.509 (s.ms.µs)

19


CPU Load Graph
100 %
90 %
80 %
70 %
60 %
50 %
40 %
30 %
20 %
10 %
0 %
2.100.000
2.200.000
2.300.000
2.400.000
2.500.000
2.600.000
Sender1
Sender3
Sender2
Receiver
Interval 179.440 to 191.403 (ms.µs)
Sender1 CPU: 411 (µs)
Sender1 CPU: 3.44 %
Task: Sender1
Instances: 1
Total CPU time: 76.139 (12.73 %)
Total Wait time: 506.198 (ms.µs)
Zoom In
Zoom Out

20

Scenario of Task receiving data from multiple sources on a single queue




```
/* Define the generic structure used by different tasks to send their data to queue. */
typedef struct
{
    unsigned char code;
    unsigned char data[MAX_LEN];
} xData;
```

21

Structure နှင့် Pointer သုံးပြီး စမ်းသပ်ရန် ကျန်ရှိ

22

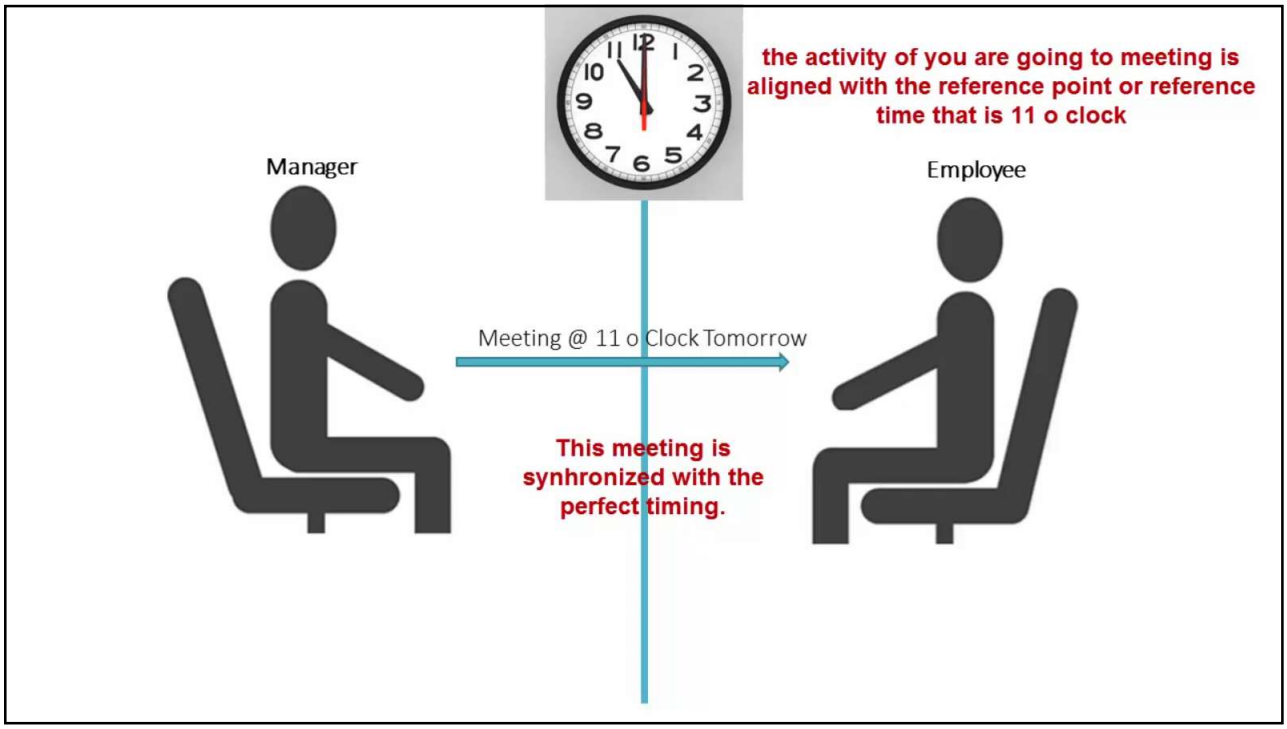
Synchronization in Computing

23

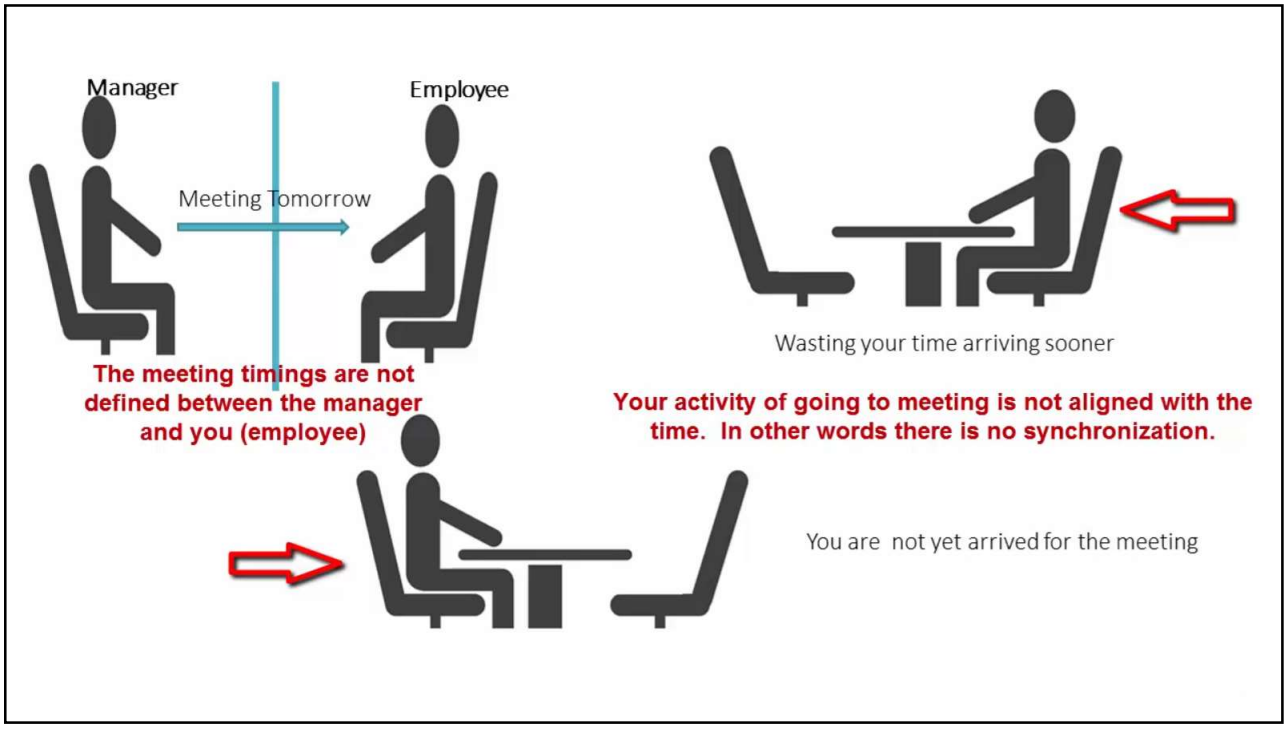
A semaphore is a kernel object or you can say kernel service, that one or more threads of execution can acquire or release for the purpose of *synchronization* or *mutual exclusion*.

Before understanding semaphore, let's first explore what is synchronization and mutual exclusion.

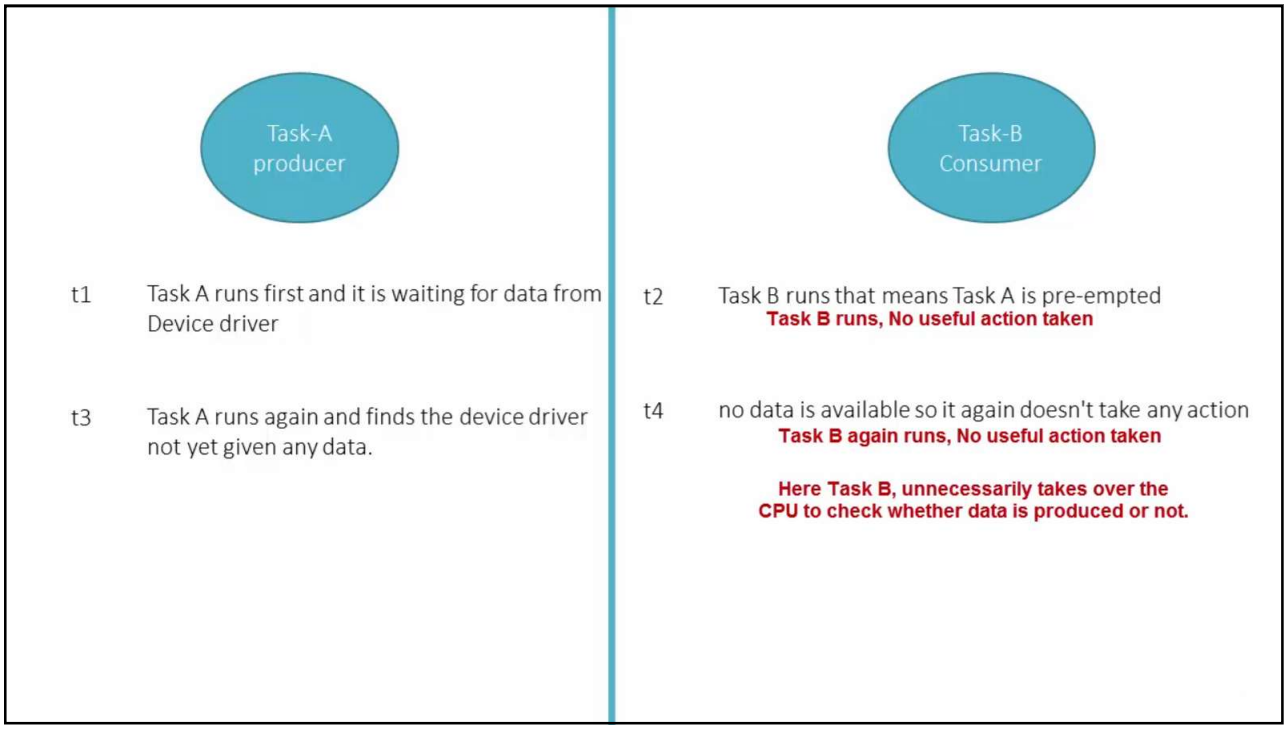
24


the activity of you are going to meeting is aligned with the reference point or reference time that is 11 o clock
Manager
Employee
Meeting @ 11 o Clock Tomorrow
This meeting is synhronized with the perfect timing.

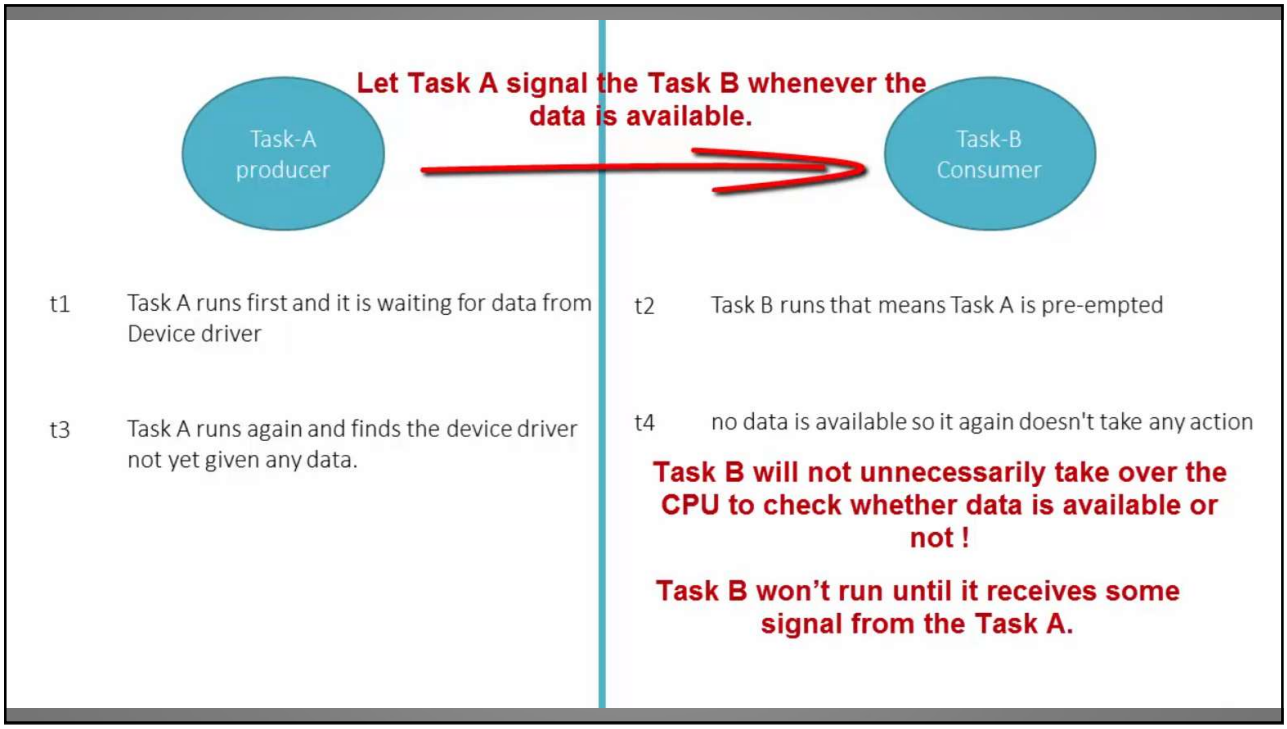
25


Manager
Employee
Meeting Tomorrow
The meeting timings are not defined between the manager and you (employee)
Wasting your time arriving sooner
Your activity of going to meeting is not aligned with the time. In other words there is no synchronization.
You are not yet arrived for the meeting

26


Task-A
producer
Task-B
Consumer
t1
Task A runs first and it is waiting for data from Device driver
t2
Task B runs that means Task A is pre-empted
Task B runs, No useful action taken
t3
Task A runs again and finds the device driver not yet given any data.
t4
no data is available so it again doesn't take any action
Task B again runs, No useful action taken
Here Task B, unnecessarily takes over the CPU to check whether data is produced or not.

27


Let Task A signal the Task B whenever the data is available.
Task-A
producer
Task-B
Consumer
t1
Task A runs first and it is waiting for data from Device driver
t2
Task B runs that means Task A is pre-empted
t3
Task A runs again and finds the device driver not yet given any data.
t4
no data is available so it again doesn't take any action
Task B will not unnecessarily take over the CPU to check whether data is available or not !
Task B won't run until it receives some signal from the Task A.

28

Kernel Objects which can be used for Synchronization

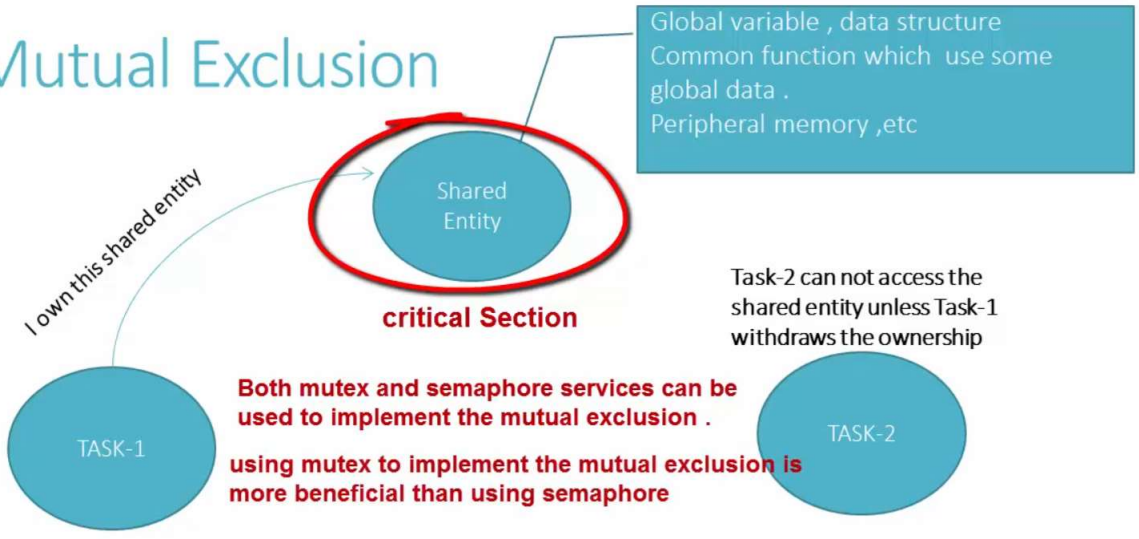
Events (or Event Flags)
Semaphores (Counting and binary)
Queues and Message Queues
Pipes
Mailboxes
Signals (UNIX like signals)
Mutex

In this section, let's explore how we can implement synchronization using semaphore.

FreeRTOS Supports Semaphores, Queues and Mutex

29

Mutual Exclusion



30

Concluding points

Synchronization is nothing but aligning number of Tasks to achieve a desired behaviour.

Where as mutual exclusion is avoiding a task to execute the critical section which is already owned by another task for execution.

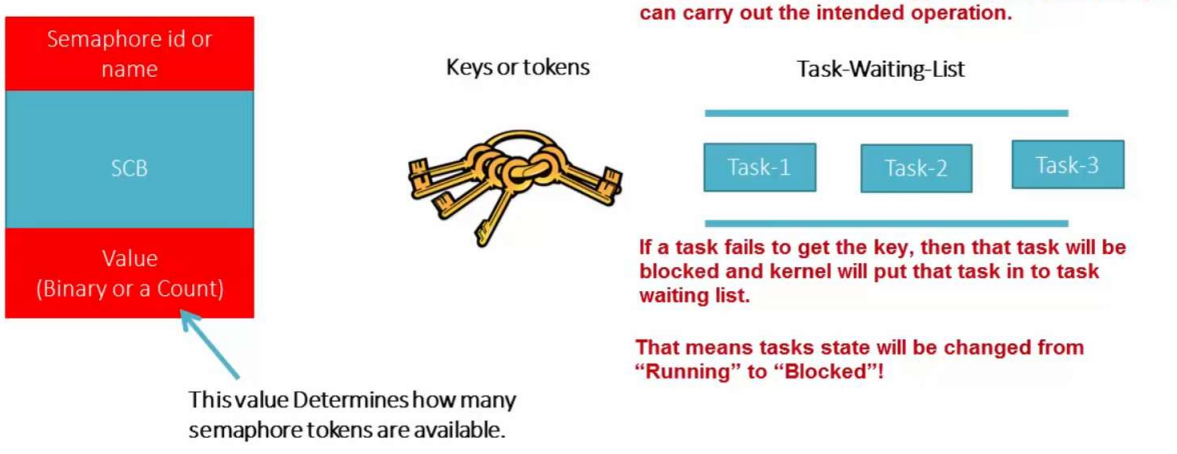
Typically Semaphores are used to implement the synchronization between tasks and between tasks and interrupts.

Mutex are the best choice to implement the mutual exclusion. That is protecting access of a shared item.

Semaphores also can be used to implement the mutual exclusion but it will introduce some serious design issues which we will see later.

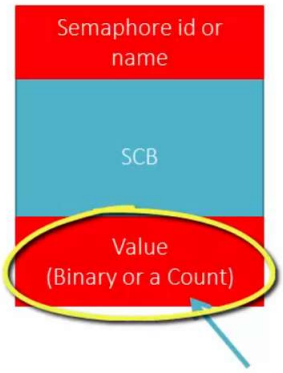
31

Creating a Semaphore



32

Creating a Semaphore



A single semaphore can be acquired, finite number of times by the tasks depending upon, how you first initialize the semaphore.

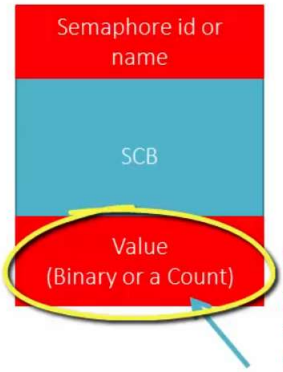
Keys or tokens

Task-Waiting-List

This value Determines how many semaphore tokens are available.

33

Creating a Semaphore



A single semaphore can be acquired, finite number of times by the tasks depending upon, how you first initialize the semaphore.

Keys or tokens

Task-Waiting-List

If value=4, then 4 semaphore keys or tokens are currently available!

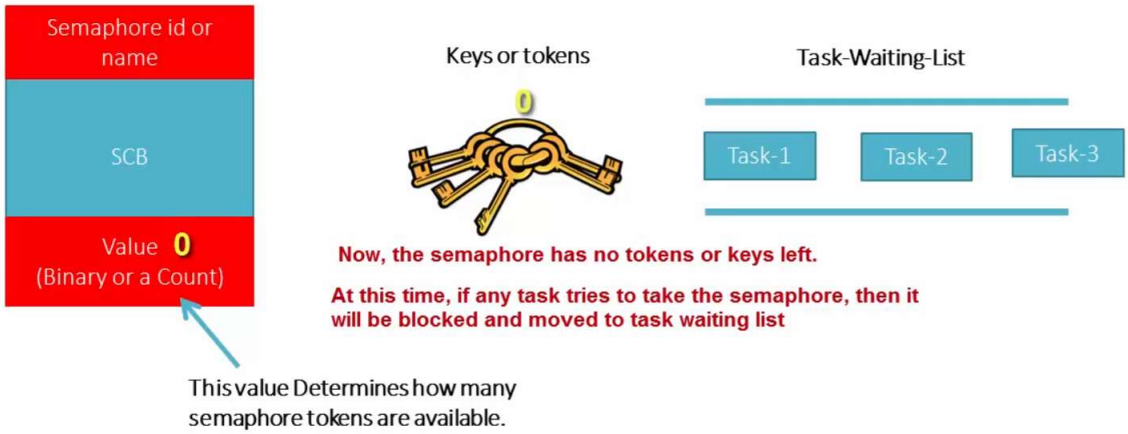
if a task tries to acquire the key for the 5th time, then it will be blocked.

This value decreases when the key is acquired and it increases when the semaphore token or key is given back.

This value Determines how many semaphore tokens are available.

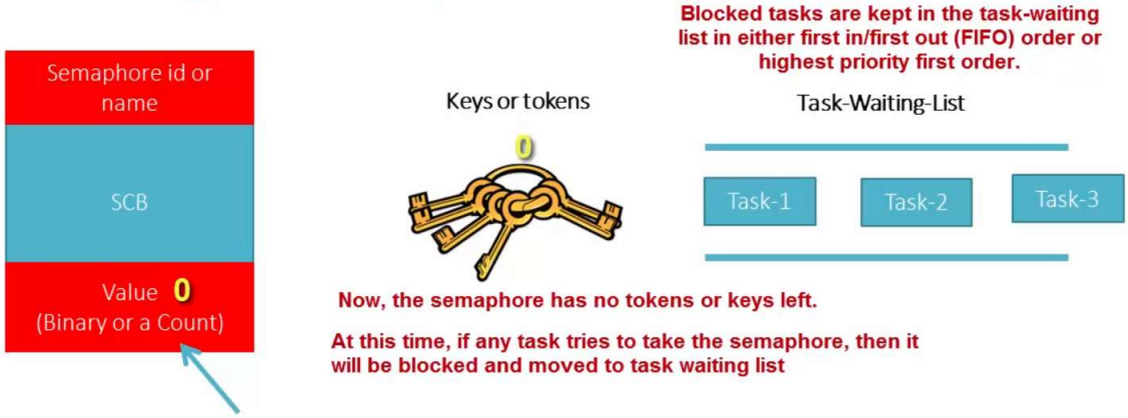
34

Creating a Semaphore



35

Creating a Semaphore



36

Two Types of Semaphore

- Binary
- Counting

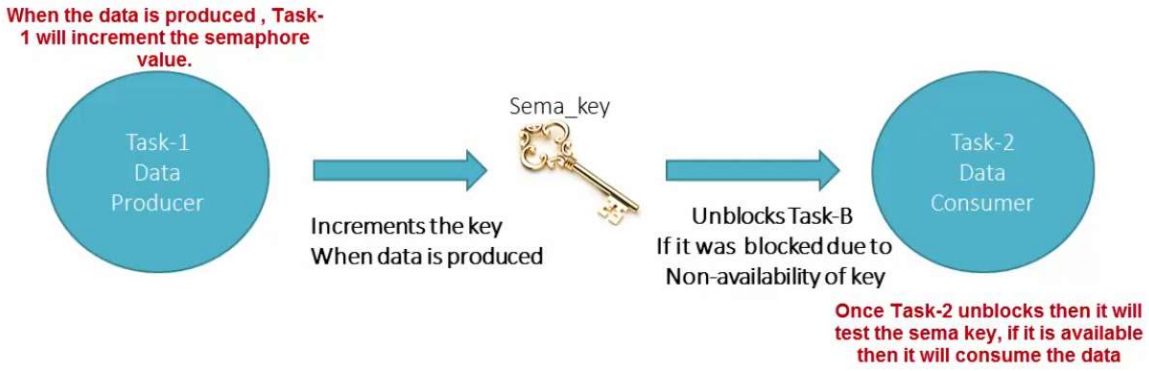
37

The main use of binary semaphore or counting semaphore is synchronization. The synchronization can be between tasks or between a task and an interrupt.

Lets first see how binary semaphore can be used for synchronization between 2 tasks.

38

Binary sema to Synchronize between Tasks



39

```
void task1_running(void)
{
      if( TRUE == produce_some_data())
      {
            /* This is a signalling for the task2  to wake up if it is blocked due to non availability of key */
            sema_give(&sema_key); // 'GIVE' operation will increment the semaphore value by 1
      }
}
void task2_running(void)
{
      /* if sema_key is unavailable then task2 will be blocked. */
      /*if sema_key is available, then task2 will take it and sema_key becomes unavailable again
      */
        while( sema_take(&sema_key) )
         {
            //Task will come here only when the sem_take operation is successful.
            /* lets consume the data */
            /* since the sem_key value is zero at this point, the next time when task-2 tries to
      take, it will be blocked. */
         }
}
```

once task-1 gives the sema_key, this task will be unblocked.

40

Implementation

- Button Press
- Long Press, Short Press, Toggle
- Estimate time between press and release for user btn on stmxxx.
- Interrupt occurs whenever pressing the user btn.
- Interrupt handler give sema key and call helper_task().
- Calculate relative (or) absolute time in the helper_task() and print it out SWD Console.

41

Synchronization between Interrupt & Task

The Binary semaphore is very handy and well suited to achieve the synchronization between an interrupt handler execution and the task handler execution.

42

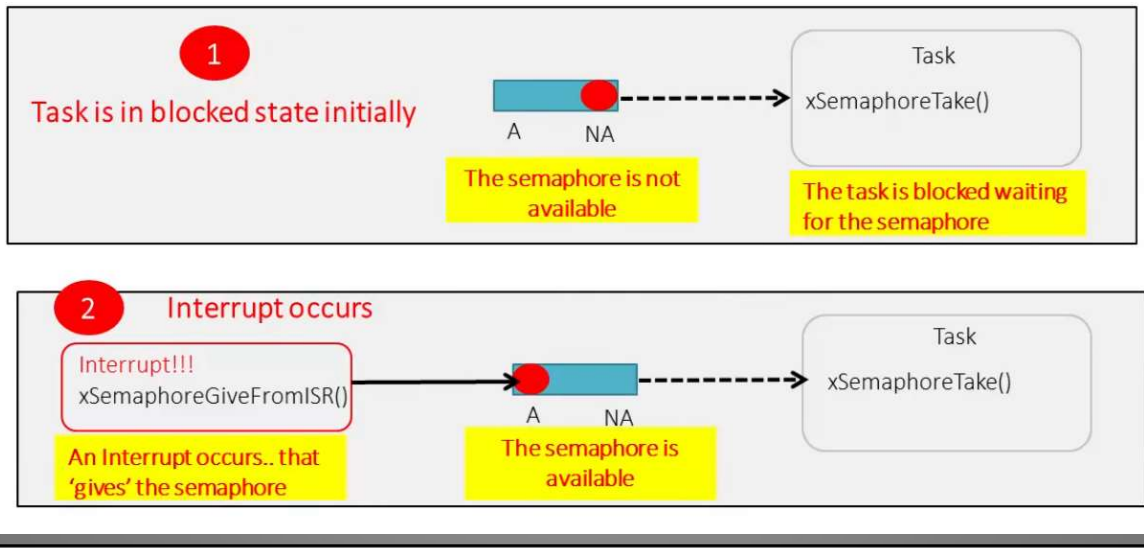
```
void interrupt_handler(void)
{
    do_important_work();  /* this is very short code */
  xSemaphoreGiveFromISR(&sem);      /* Give means release the key */
}/* exit from the interrupt */

/* I am helper task for the interrupt ! I do time consuming work on behalf of interrupt handler*/
void helper_task(void)
{
    /* if taking a key is un-successful then this task will be blocked until key is available */
    while (xSemaphoreTake(&sem ) )  // GET means, trying to take the key
    {
        // it will come here, only if taking a key is successful .
        /* Do time consuming work of the ISR */
    }

}
```

This interrupt handler does only minimal essential work and offloads other work to its helper task below. This interrupt handler signals its helper task using a semaphore.

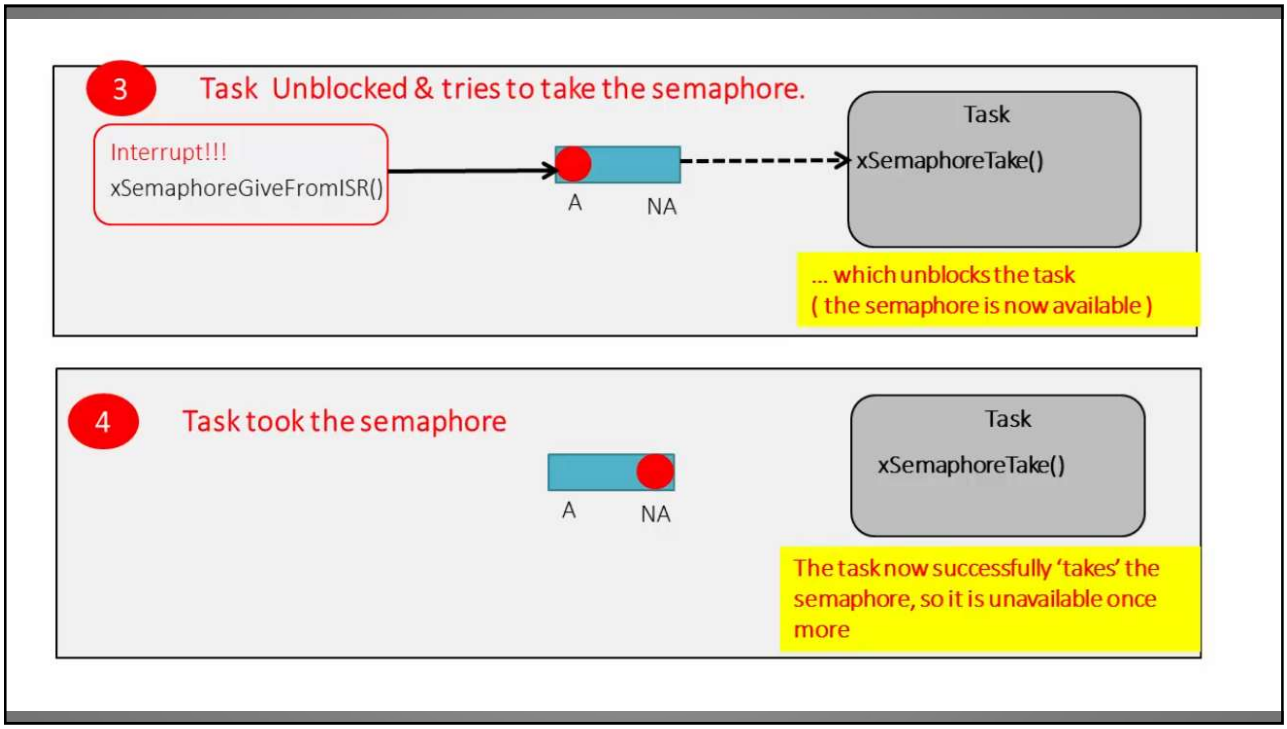
Once this interrupt handler gives the semaphore back, the helper task will get unblocked and starts executing from the point where it was last blocked.

43

Synchronization between an Interrupt and a Task using Binary semaphore



44



45

Interrupt Priority နဲ့ Task Priority ခွဲကွဲပုံ

- Higher-priority tasks preempt lower-priority tasks:

If a higher-priority task becomes ready, the currently running lower-priority task is suspended, and the higher-priority task is executed.

- Round-robin scheduling: If two or more tasks have the same priority, the system can execute them in a round-robin fashion, where each task gets a time slice.

- Idle task: There is always an idle task with the lowest priority that runs when no other tasks are ready.

46

NVIC Priorities (Nested Vectored Interrupt Controller)

- Lower priority numbers represent higher priority levels

an interrupt with priority 0 is the highest, and one with priority 255 is the lowest, depending on the microcontroller's configuration.

- Nested interrupts (high priority နဲ့ interrupt ဝင်လာရင် lower priority နဲ့ handle လုပ်နေတဲ့ handler သည် interrupt အလုပ်ခံရမည်။
- NVIC supports preemption priorities and sub-priorities.
- Preemption priorities define which interrupt can preempt others
- sub-priorities determine the order of execution when two interrupts of the same preemption priority occur simultaneously.

47

Interrupt Priority Grouping in NVIC

- Interrupts can preempt tasks
- Task scheduling resumes after the interrupt service handler
- If a task is critical and should not be interrupted
- Priority grouping is configured using the NVIC_SetPriorityGrouping() function, which decides the balance between preemption and sub-priority bits.

48

Event Latching

- အကယ်၍ interrupt က helper_task အလုပ်လုပ်နေတဲ့အချိန် interrupt ၁၀ ခုဆက်တိုက်ဝင်လာမယ်ဆိုရင် ပြဿနာတက်ပါမယ်။ high frequency interrupt တွေဝင်လာတဲ့အချိန် Binary Semaphore ကိုအသုံးပြုထားမယ်ဆိုရင် Event Latch ဖြစ်ကို ဖြစ်ပါမယ်။
- ရှင်းပြချက်၊ interrupt ဝင်တဲ့အချိန်တိုင်း interrupt handler က semaphore key ကို helper_Task ကို Give လုပ်နေမှာဖြစ်တယ်၊
- အဲ့အချိန် ၁၀ ကြိမ်ဆက်တိုက်ဝင်တဲ့အချိန်မှာ interrupt handler က semaphore key ကို Give လုပ်နေမှာဖြစ်တယ်၊ ဒါမယ့်သုံးထားတာက Binary ဖြစ်တဲ့အတွက် ပေးစရာမရှိတော့ပါဘူး။ အဲ့အတွက် interrupt တစ်ခုကိုပဲ အလုပ်လုပ်ပေးပြီးကျန်တဲ့ ၉ ခုက missed ဖြစ်မှာပါ။

49

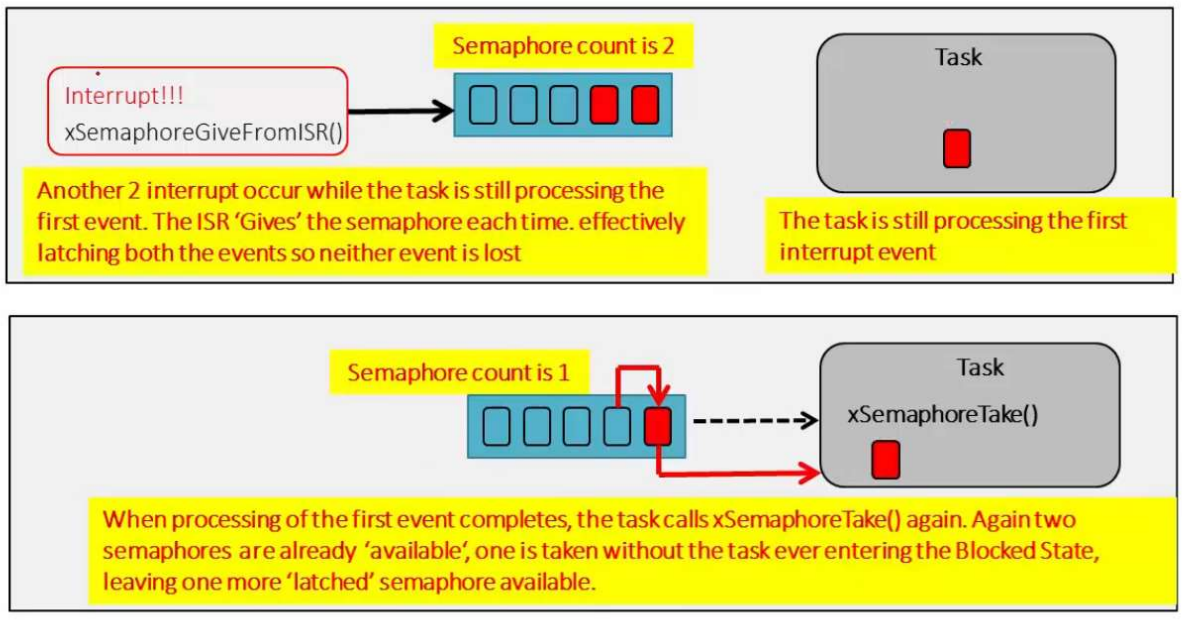
```
အကယ်၍ interrupt က helper_task အလုပ်လုပ်နေတဲ့အချိန် interrupt ၁၀ ချိန်ဆက်တိုက်ဝင်လာမယ်ဆိုရင် ပြဿနာတက်ပါမယ်။
high frequency interrupt တွေဝင်လာတဲ့အချိန် Binary Semaphore ကိုအသုံးပြုထားမယ်ဆိုရင် Event Latch ဖြစ်ကို ဖြစ်ပါမယ်။

ရှင်းပြချက်၊ interrupt ဝင်တဲ့အချိန်တိုင်း interrupt handler က semaphore key ကို helper_Task ကို Give လုပ်နေမှာဖြစ်တယ်၊

အဲ့အချိန် ၁၀ ကြိမ်ဆက်တိုက်ဝင်တဲ့အချိန်မှာ interrupt handler က semaphore key ကို Give လုပ်နေမှာဖြစ်တယ်၊
ဒါမယ့်သုံးထားတာက Binary ဖြစ်တဲ့အတွက် ပေးစရာမရှိတော့ပါဘူး။ အဲ့အတွက် interrupt တစ်ခုကိုပဲ အလုပ်လုပ်ပေးပြီးကျန်တဲ့ ၉ ခုက missed ဖြစ်မှာပါ။

void interrupt_handler(void)
{
    do_important_work();  /* this is very short code */
   xSemaphoreGiveFromISR(&sem);      /* Give means release the key */

}/* exit from the interrupt */

/* I am helper task for the interrupt ! I do time consuming work on behalf of interrupt handler*/
void helper_task(void)
{
    /* if taking a key is un-successful then this task will be blocked until key is available */
    while (xSemaphoreTake(&sem ) )  // GET means, trying to take the key
    {
        // it will come here, only if taking a key is successful .
        /* Do time consuming work of the ISR */
    }

}
```

50

Concluding Points on Latching Events

1. When the interrupts/events happen relatively slow, the binary semaphore can latch at most only one event
2. If multiple interrupts/events trigger back to back, then the binary semaphore will not able to latch all the events. So some events will be lost .
3. How to solve the above issue ? Welcome to the world of "Counting Semaphore"

51



52

Watch Dog Timer (WDT)

- Timer that runs independent of CPU that can be used to trigger an interrupt or reset the MCU
- What would you use it for?
  - Perform a task at a regular interval
  - Wake up from sleep at a timed interval
  - Reset the MCU if it is stuck in a bad state
- There is an AVR Library for it "#include <avr/wdt.h>" ……. but it isn't really complete :-/

53

Operation of Watch Dog Timer (WDT)

- WDT is started with a timeout period.
- If the system operates correctly, it refreshes the WDT regularly (often from a high-priority task or the main loop).
- If the system hangs (e.g., due to an infinite loop or deadlock), the WDT is not refreshed and expires, triggering a system reset.
- The Watchdog Timer is a **hardware peripheral** on the microcontroller. It operates independently of the RTOS and does not require any task management.
- Ensures system reliability. If the system hangs (e.g., if a critical task is stuck or the CPU gets trapped in the idle thread), the WDT will reset the system to recover from the error.

54

Mutual Exclusion using Binary Semaphore

Access to a resource that is shared either between tasks or between tasks and interrupts needs to be serialized using a some techniques to ensure data consistency.

Usually a common code block which deals with global array , variable or memory address, has the possibility to get corrupted , when many tasks or interrupts are racing around it

55

Mutual Exclusion by Binary semaphore

```
#define  UART_DR            *((unsigned long * ) (0x40000000) )
/* This is a common function which write to UART DR */
int UART_Write( uint32_t len , uint8_t *buffer)
{
    for (uint32_t i=0;i < len ; i++)
        {  get_lock();
        /*  if Data Register is empty write it  */
            while(! is_DR_empty() );
            UART_DR = buffer[i];
        }  release_lock();
}
```

before accessing the critical section get the lock/key first.

56

2 ways we can Implement the mutual exclusion in FreeRTOS

1. Using Binary semaphore APIs
2. Using Mutex APIs

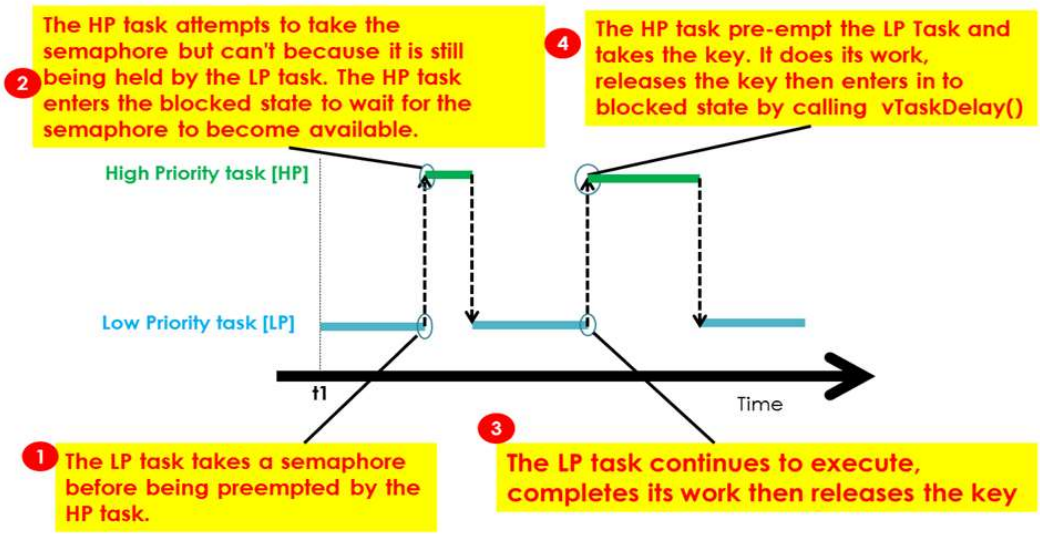
57

```
#define  UART_DR            *((unsigned  long * ) (0x40000000) )
/* This is a common  function  which write to UART DR */
int UART_Write( uint32_t len , uint8_t *buffer)
{
    for (uint32_t i=0;i < len ; i++)
        {
          sema_take_key( bin_sema );
        /*  if Data Register is empty  write it  */
           while(! is_DR_empty() );
            UART_DR = buffer[i];  //Critical section
         sema_give_key( bin_sema );
        }
}
```

binary semaphore ကို အသုံးပြုထားလေ့ရှိတဲ့အတွက် key ကို တခြား task က ယူပြီးသုံးနေရင် လျှို့တဲ့ Task က waiting lists ထဲမှာ blocked state နဲ့ စောင့်နေရမှာဖြစ်လို့ Mutual Exclusion ဖြစ်ပြီးသားဖြစ်ပါတယ်။

bin_sema=1 (available)

58



59

Why people say do not use binary semaphore for mutual exclusion. ??

- carefully look at this timing diagram, there is a chance for priority inversion!
- Task-2 which is higher priority task gets blocked due to non-availability of semaphore and it is waiting for Task-1 to release the semaphore. Right?
- by this design, you made a higher priority task dependent on the lower Priority task. That is a bad.

60

Why people say do not use binary semaphore for mutual exclusion. ??

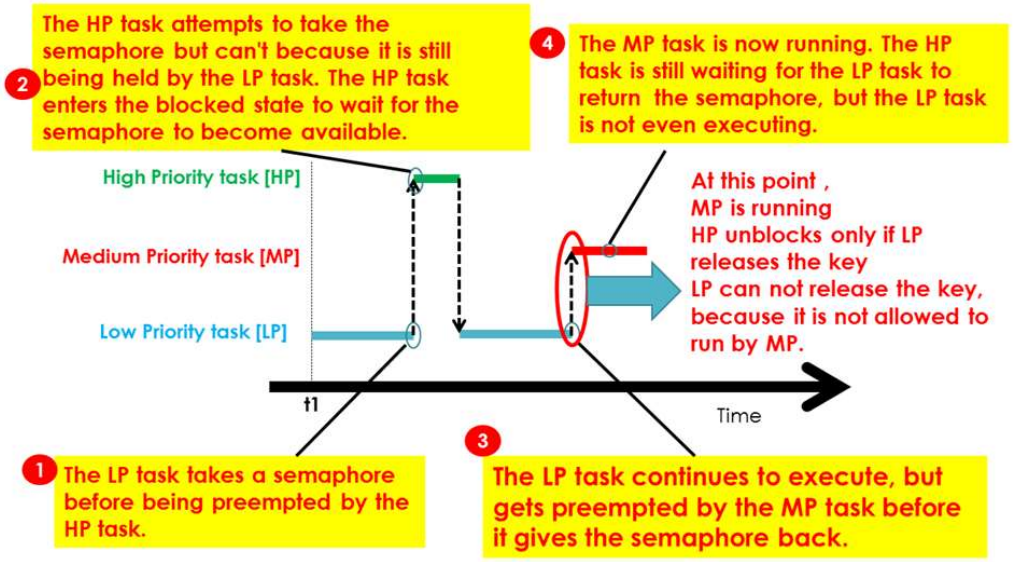
- The low priority task should never decide when the higher priority task must run.
- It's like you made a king dependent on a servant to take some action.
- So until lower priority task releases the semaphore, the higher priority task never runs. Isn't it?
- (Remember this is like your car and ambulance scenario i showed in the very first section of the course. You are blocking the way of ambulance.)

61

Why people say do not use binary semaphore for mutual exclusion. ??

- And lower priority task will release the semaphore only when it gets allowed to run isn't it ?
- So, what if there is another medium task appears and it occupies the processor which will not allow lower Priority task to run?

62



63

ခုနပုံအရ

- higher priority task is blocked and waiting for semaphore to gets released by lower priority task.
- But lower priority task is not allowed to run by the medium priority task until it finishes and exits.
- As a result your lower Priority task may be delayed indefinitely and the higher priority task will remain in blocked state for indefinite time . This is exactly the scenario of Priority inversion.

- until medium task appears everything is OK,

64

Are there any ways to solve or reduce the priority inversion problem ?

- Don’t use binary semaphore for mutual exclusion when your system has many tasks of different Priorities.
- Use MUTEX instead.
- MUTEX has inbuilt intelligence to minimize the priority inversion effect and this is the biggest difference between MUTEX and Binary semaphore.

65

MUTEX reduces the priority inversion effect. But how?

- In RTOS systems like FreeRTOS or CMSIS-RTOS, the **Mutex** often implements **Priority Inheritance** to mitigate the priority inversion problem.
- How **Priority Inheritance** works?

66

Priority Inheritance works

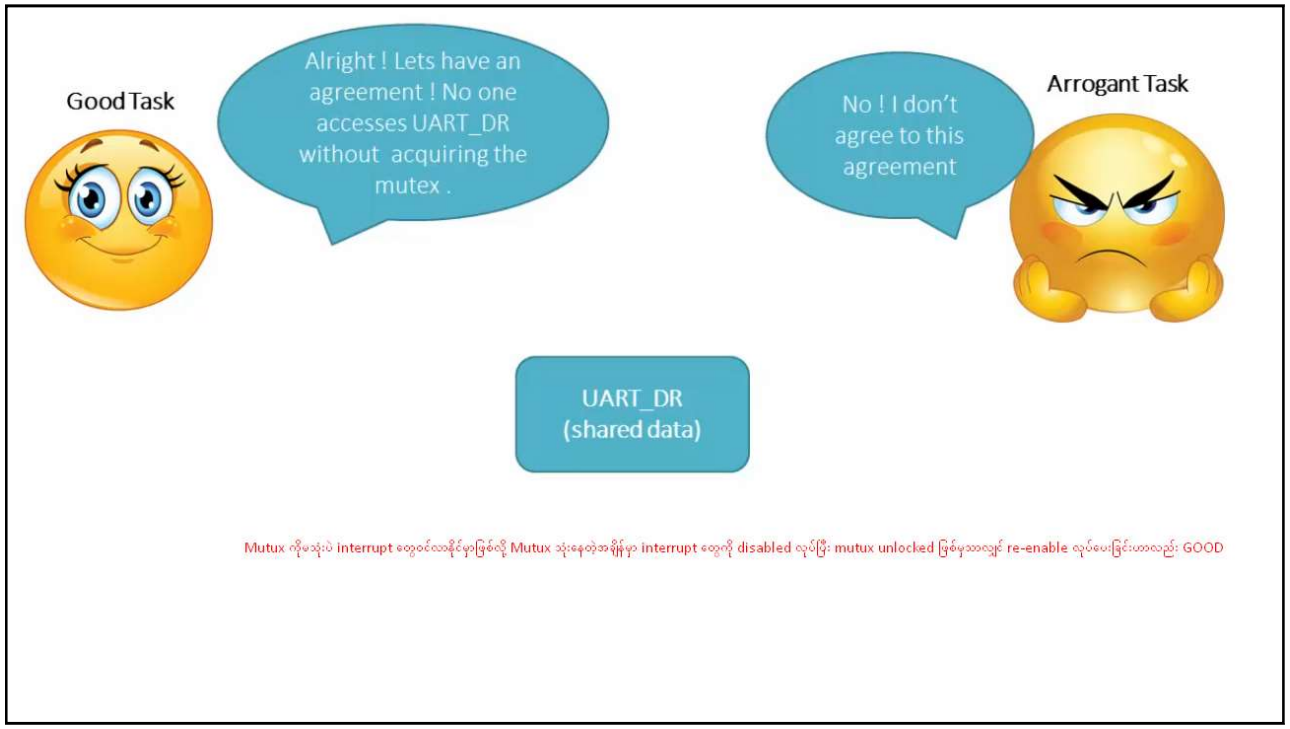
- When a higher-priority task (Task H) tries to acquire a Mutex that is held by a lower-priority task (Task L), **Priority Inheritance** temporarily raises the priority of Task L to that of Task H.
- This prevents an intermediate-priority task (Task M) from preempting Task L.
- Task L continues executing at the boosted priority until it releases the Mutex. Once the Mutex is released, Task L's priority returns to its original lower level.

67

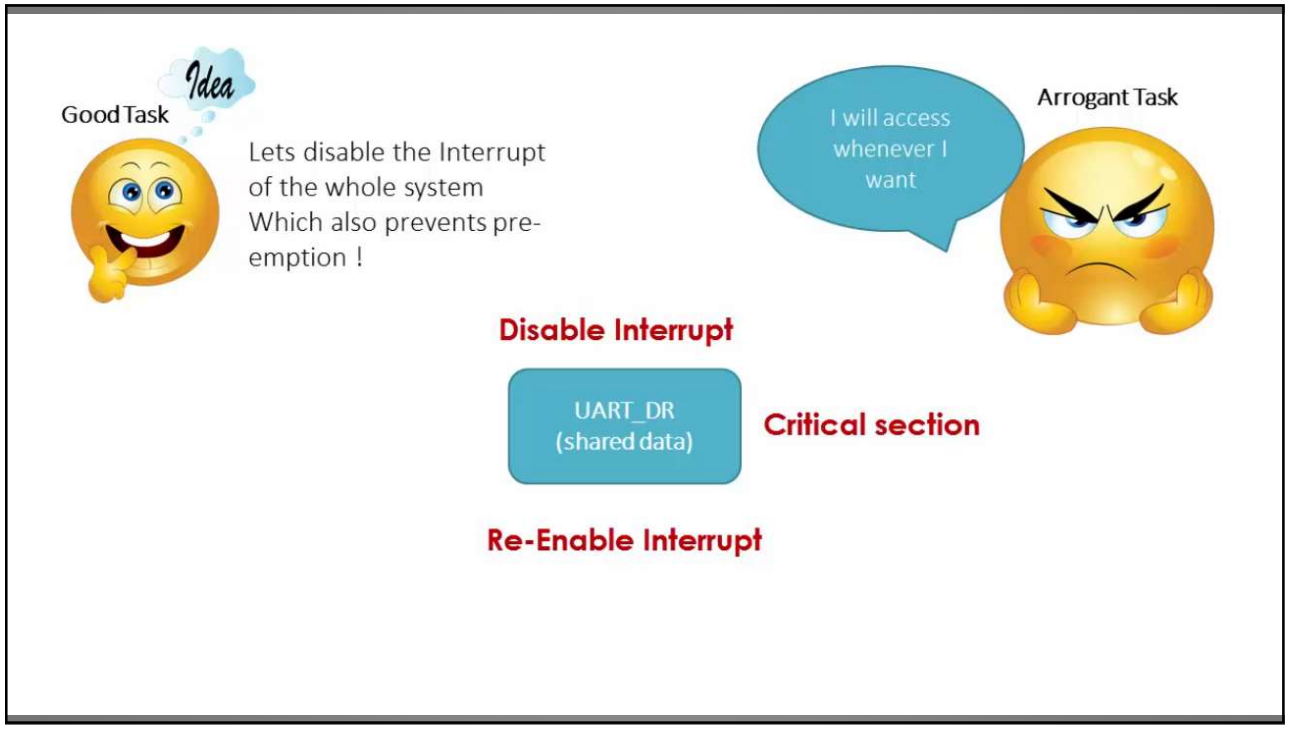
Ways to protect the Critical Section

1. Binary semaphore
2. Mutex
3. Crude way (disabling interrupts of the system, either globally, or up to a specific interrupt priority level)

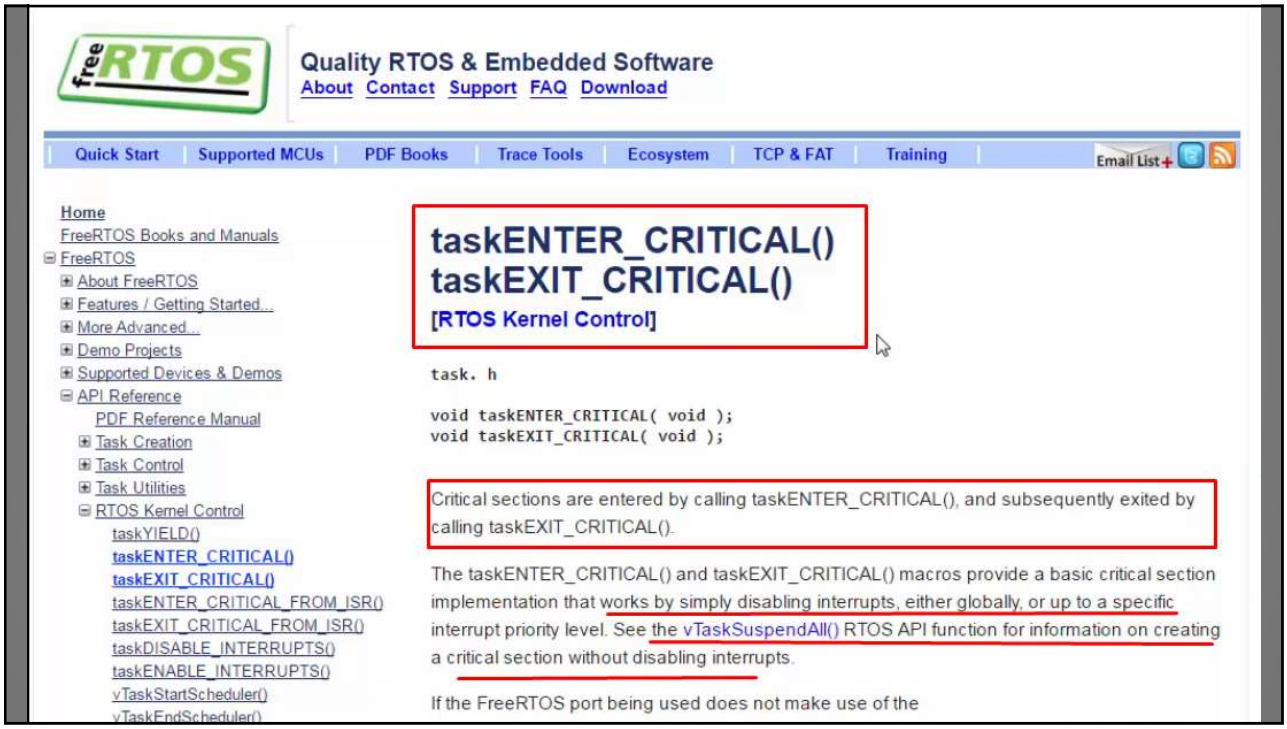
68


Good Task
Alright ! Lets have an agreement ! No one accesses UART_DR without acquiring the mutex .
Arrogant Task
No ! I don't agree to this agreement
UART_DR
(shared data)
Mutux ကိုမသုံးပဲ interrupt တွေဝင်လာနိုင်မှာဖြစ်လို့ Mutux သုံးနေတဲ့အချိန်မှာ interrupt တွေကို disabled လုပ်ပြီး mutux unlocked ဖြစ်မှသာလျှင် re-enable လုပ်ပေးခြင်းဟာလည်း GOOD

69


Idea
Good Task
Lets disable the Interrupt of the whole system Which also prevents pre-emption !
Arrogant Task
I will access whenever I want
Disable Interrupt
UART_DR
(shared data)
Critical section
Re-Enable Interrupt

70



71

- အဲ့ macro တွေဟာ Critical Session နဲ့ interrupt တွေကိုပိတ်နိုင်ဖို့ RTOS System တော်တော်များများက API ထုတ်ပေးထားတာဖြစ်တယ်။
- ဒါမယ့် သုံးမယ်ဆွင်ရင် ကိုယ်သုံးတဲ့ Micro-Controller နဲ့ သေချာတိုက်ဆိုင်စစ်ဆေးပီးမှ သုံးရမှာပါ။
- Architecture Dependent ဖြစ်ပါတယ်။

72