RTOS Terminologies And Building Blocks

Time-Driven Vs. Event Driven

Time-Driven Vs. Event Driven

Time-driven programming is a computer programming paradigm, where the control flow of the computer program is driven by a clock and is often used in Real-time computing.

Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs or threads.

Time-Driven Vs. Event Driven

- Time-driven
- Non blocking
- No prioritization handling
- Time delays will cause problems
- Cannot meet real-time requirements

```c
unsigned long int tickCounter = 0;

/* Timer ISR to increment tickCounter every 1ms */
ISR(TIMER)
{
    tickCounter++;
}

void main(void)
{
    while(1)
    {
        /* Check that 100ms passed */
        if((tickCounter % 100) == 0)
        {
            doSomething_1();
        }

        /* Check that 500ms passed */
        if((tickCounter % 500) == 0)
        {
            doSomething_2();
        }
    }
}
```

Time-Driven Vs. Event Driven

Time-driven

Non blocking

No prioritization handling

Time delays will cause problems

Cannot meet real-time requirements

```
unsigned long int tickCounter = 0;

ISR(Event1)
{
    EVENT_1 = TRUE;
}

ISR(Event2)
{
    EVENT_2 = TRUE;
}

void main(void)
{
    while(1)
    {
        /* Check that 100ms passed */
        if(EVENT_1 == TRUE)
        {
            EVENT_1 == FALSE
            doSomething_1();
        }

        /* Check that 500ms passed */
        if(EVENT_2 == TRUE)
        {
            EVENT_2 = FALSE;
            doSomething_2();
        }
    }
}
```

Time-Driven Vs. Event Driven

```c
void main(void)
{
    int data = 0;

    while(1)
    {
        /* Check user input */
        if(getUserInput(&data) == TRUE)
        {
            /* Do a certain action */
            doSomething(data);
        }

        /* Delay 1 second */
        delay(1000);
    }
}
```

```c
void main(void)
{
    int data = 0;

    while(1)
    {
        /* Check user input */
        while(getUserInput(&data) == FALSE);

        /* Do a certain action*/
        doSomething(data);
    }
}
```

Can you guess the characteristics of the above examples ? ..

Time-Driven Vs. Event Driven

- Event-driven
- Non blocking
- No prioritization handling
- Time delays will cause problems
- Cannot meet real-time requirements

```
unsigned long int tickCounter = 0;

ISR(Event1)
{
    EVENT_1 = TRUE;
}

ISR(Event2)
{
    EVENT_2 = TRUE;
}

void main(void)
{
    while(1)
    {
        /* Check that 100ms passed */
        if(EVENT_1 == TRUE)
        {
            EVENT_1 == FALSE
            doSomething_1();
        }

        /* Check that 500ms passed */
        if(EVENT_2 == TRUE)
        {
            EVENT_2 = FALSE;
            doSomething_2();
        }
    }
}
```

Time-Driven Vs. Event Driven

```
{
    int data = 0;

    while(1)
    {
        /* Check user input */
        if(getUserInput(&data) == TRUE)
        {
            /* Do a certain action */
            doSomething(data);
        }

        /* Delay 1 second */
        delay(1000);
    }
}
```

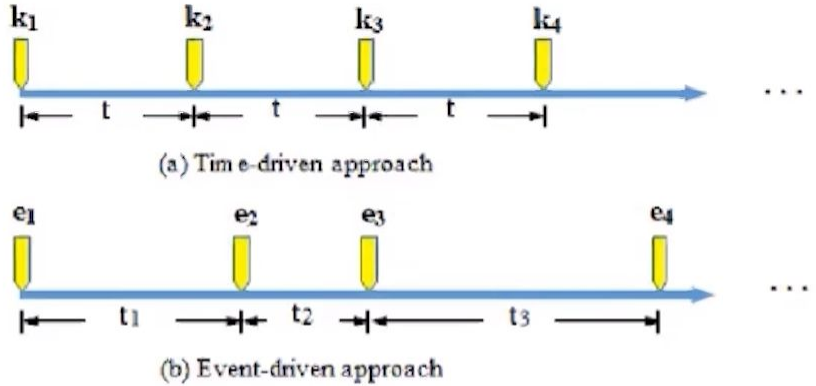
```
void main(void)
{
    int data = 0;

    while(1)
    {
        /* Check user input */
        while(getUserInput(&data) == FALSE);

        /* Do a certain action*/
        doSomething(data);
    }
}
```

- **Can you guess the characteristics of the above examples ? ..**

Time-Driven Vs. Event Driven

Sometimes when using time-driven architecture CPU spends most of its time doing nothing useful or executing IDLE task. Using Event-driven architecture optimizes the use of CPU time by triggering an execution only when an event is available to process.

Depending on the application requirements any of these paradigms is used and many times both are used in the same application.



RTOS Terminologies And Building Blocks

RTOS Components And Terminologies

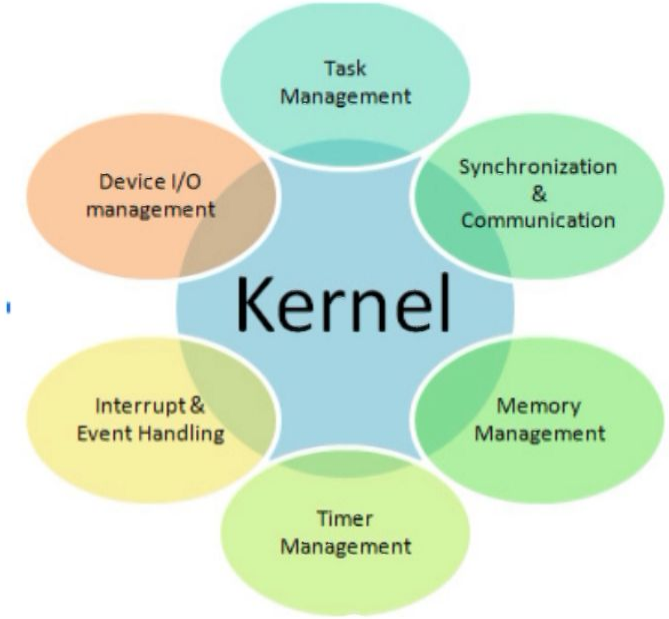
RTOS Components And Terminologies

Important Terminologies:

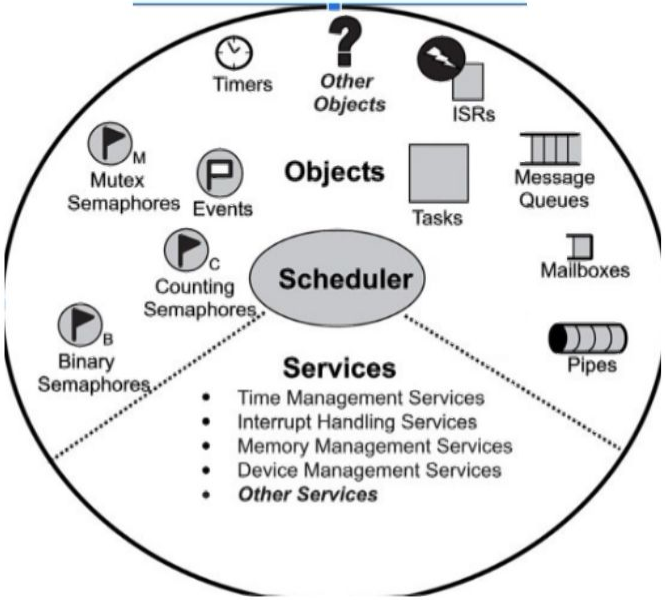
- Interrupt service routine
- Scheduling
- Dispatcher
- Context switching
- Task
- Idle task
- Priority
- Preemption

RTOS Components And Terminologies

RTOS Components:



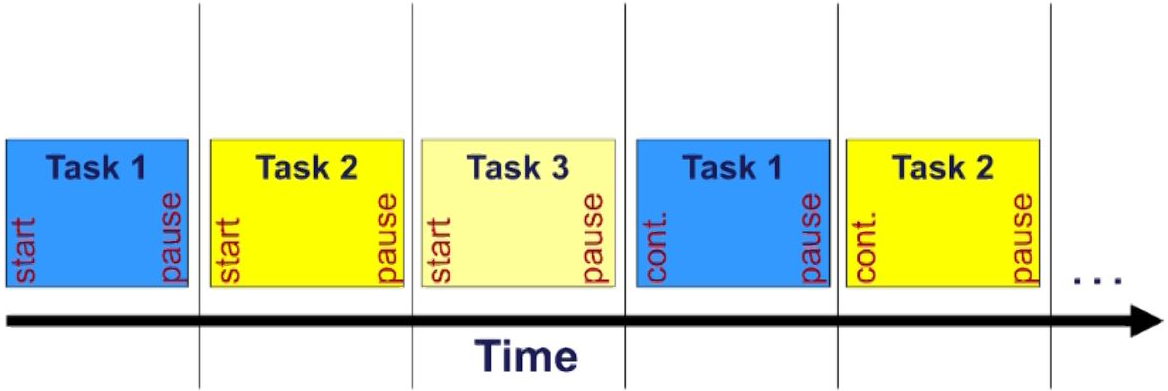
RTOS Components And Terminologies



Schedulers And SysTick

Schedulers And SysTick

Scheduling is the process or algorithm which decides which task or thread should be accessed and run at what time by the system resources.



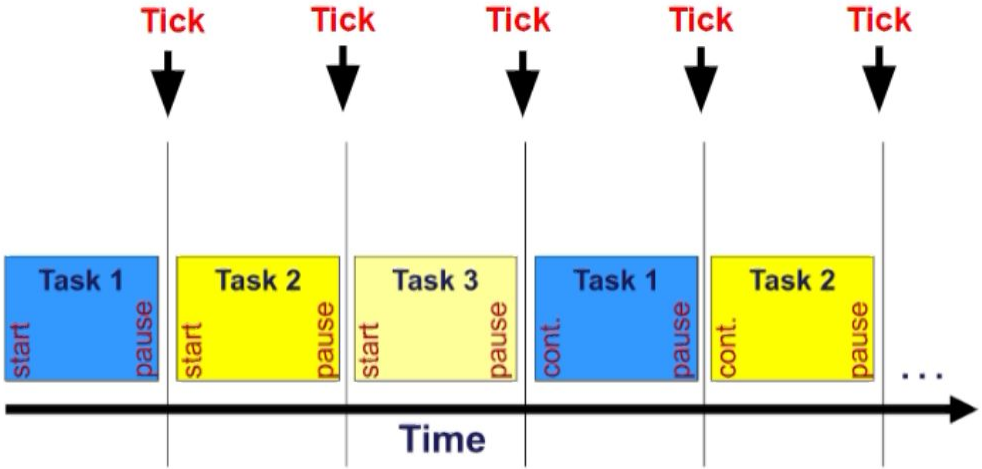
Schedulers And SysTick

Famous Scheduling algorithms:

- **Fixed priority**
- **Earliest Deadline First.**
- **Round-robin.**
- **Rate-monotonic.**

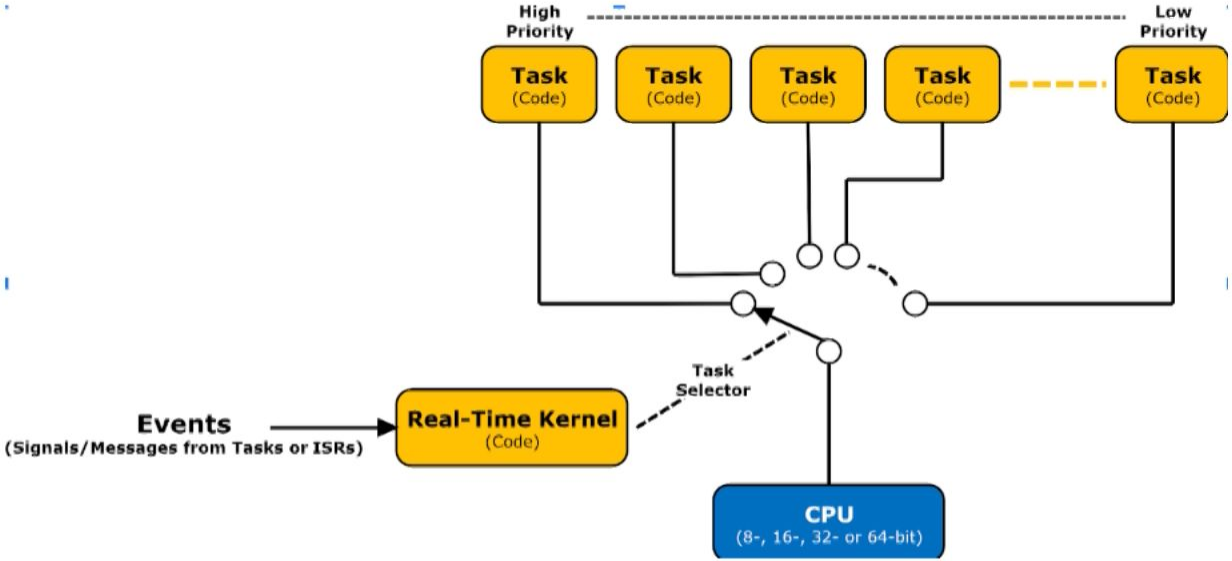
Schedulers And SysTick

The system tick is the time unit that OS timers and delays are based on. The system tick is a scheduling event. It causes the scheduler to run and may cause a context switch.



Schedulers And SysTick

- General Overview



Task Structure

Task Structure

A task is a set of program instructions that are loaded in memory. They are normally implemented as an infinite loop, and must never attempt to return or exit from their implementing function. Tasks can however delete themselves. A task should have the following structure:

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }

    /* Tasks must not attempt to return from their implementing
    function or otherwise exit.  In newer FreeRTOS port
    attempting to do so will result in an configASSERT() being
    called if it is defined.  If it is necessary for a task to
    exit then have the task call vTaskDelete( NULL ) to ensure
    its exit is clean. */
    vTaskDelete( NULL );
}
```

Task Structure

Any task is defined by the following parameters:

- **Periodicity: how often task should be scheduled (Ex. Every 100ms).**
- **Priority: task priority of execution in relation to other tasks in the system.**
- **Execution Time: time the task take to run to its completion since it was scheduled.**
- **Deadline: the time bound or limit that the task execution should not exceed.**

Task Types

Task Types

Tasks are also classified into the following types:

- **Periodic: a task arrives on fixed intervals and must meet it's deadline.**
- **Aperiodic: a task arrives unexpectedly and can miss it's deadline.**
- **Sporadic: a task arrives unexpectedly and must meet it's deadline.**

Task Types

Task types can also be extended depending on the application usage, For example in embedded systems they are often classified as:

- **Initialization Task: a task scheduled once or more to initialize system related hardware and peripherals.**
- **Cyclic Task: a task scheduled on fixed intervals based on the task's periodicity.**
- **Event-Based Task: a task scheduled when an event occurs.**
- **Interrupt-Based Task: a task scheduled when an interrupt occurs.**

Task States

RTOS Terminologies And Building Blocks

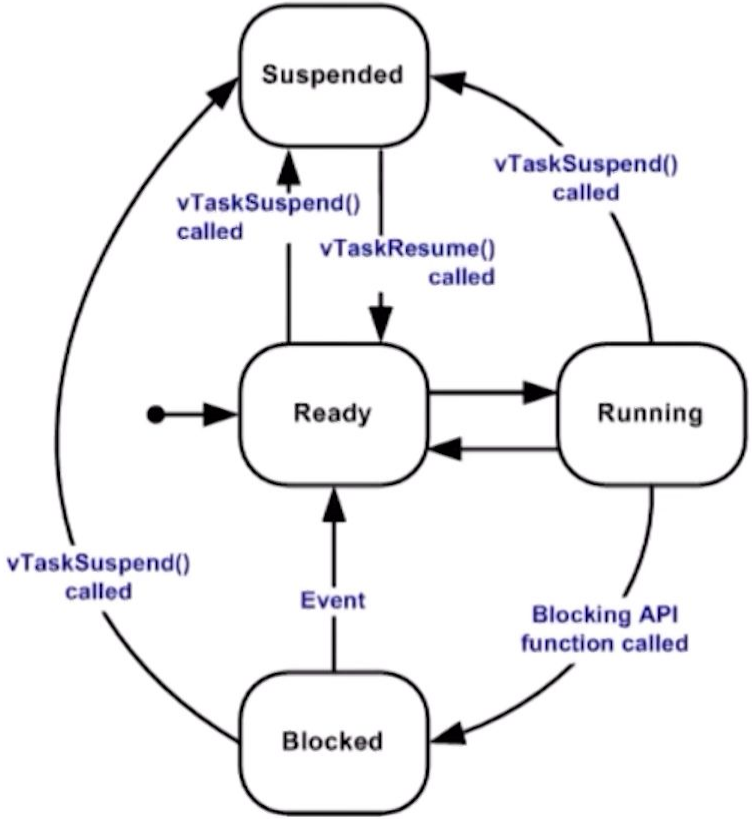
Task States

A task can exist in one of the following states:

- **Ready: Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state).**
- **Running: When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor.**
- **Suspended: tasks in the Suspended state cannot be selected to enter the Running state or any other state.**
- **Blocked: A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event.**

Task States

- Task State Transition



Task TCB

RTOS Terminologies And Building Blocks

Task TCB

A task control block (TCB) is a data structure used by kernels to maintain information about a task. Each task requires its own TCB and the user assigns the TCB in user memory space (RAM). It includes mainly the following data:

- Program counter
- Stack pointer
- Task ID
- Task priority
- Task name
- CPU registers
- Etc..

Task TCB

- **Example on TCB Block In FreeRTOS:**

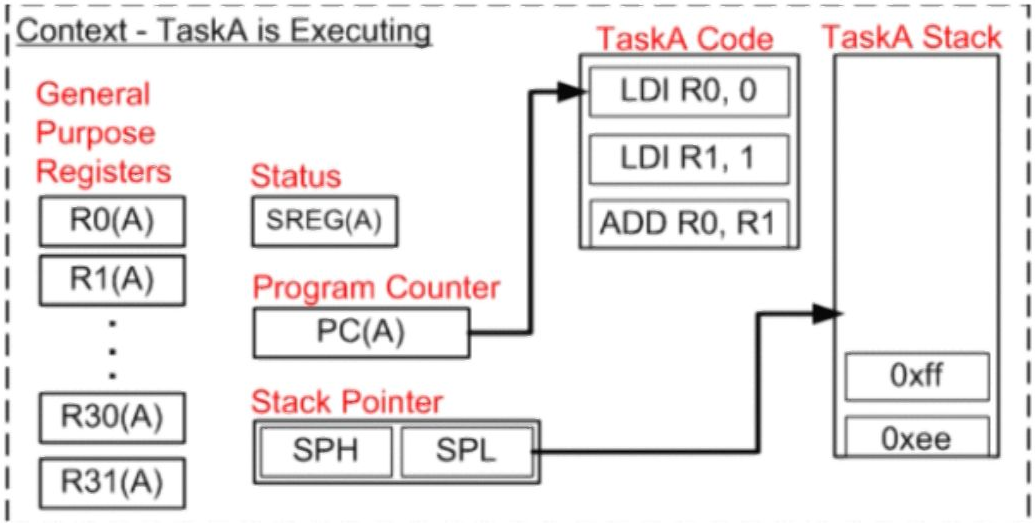
```
/*
 * Task control block.  A task control block (TCB) is allocated for each task,
 * and stores task state information, including a pointer to the task's context
 * (the task's run time environment, including register values)
 */
typedef struct tskTaskControlBlock
{
    volatile StackType_t    *pxTopOfStack;
    ListItem_t              xStateListItem;
    ListItem_t              xEventListItem;
    UBaseType_t             uxPriority;
    StackType_t             *pxStack;
    char                    pcTaskName[ configMAX_TASK_NAME_LEN ];
    StackType_t             *pxEndOfStack;
    UBaseType_t             uxTCBNumber;
    UBaseType_t             uxTaskNumber;
    UBaseType_t             uxBasePriority;
    UBaseType_t             uxMutexesHeld;


} tskTCB;
```

Context Switching

RTOS Terminologies And Building Blocks

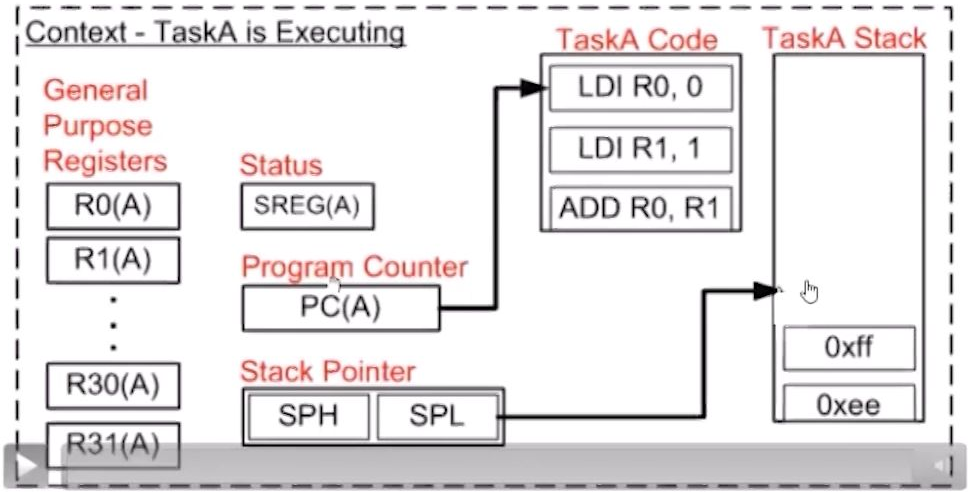
Context Switching

Context switching is the process of saving the context of a task being suspended and restoring the context of a task being resumed .



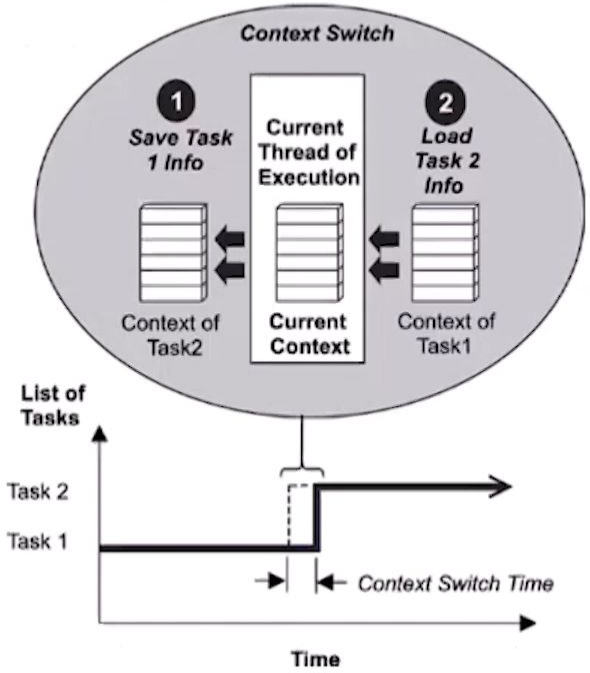
Context Switching

Context switching is the process of saving the context of a task being suspended and restoring the context of a task being resumed .



Context Switching

- General Overview



https://assemblylearningtutorial.blogspot.com/2017/09/80x86-islemcilerinde-context-switch