RTOS solution

RTOS solution

VxWorks

ecOS

ChibiOS

embOS Segger

FreeRTOS

µcOS

Zephyr OS

ARM mbedOS

ThreadX

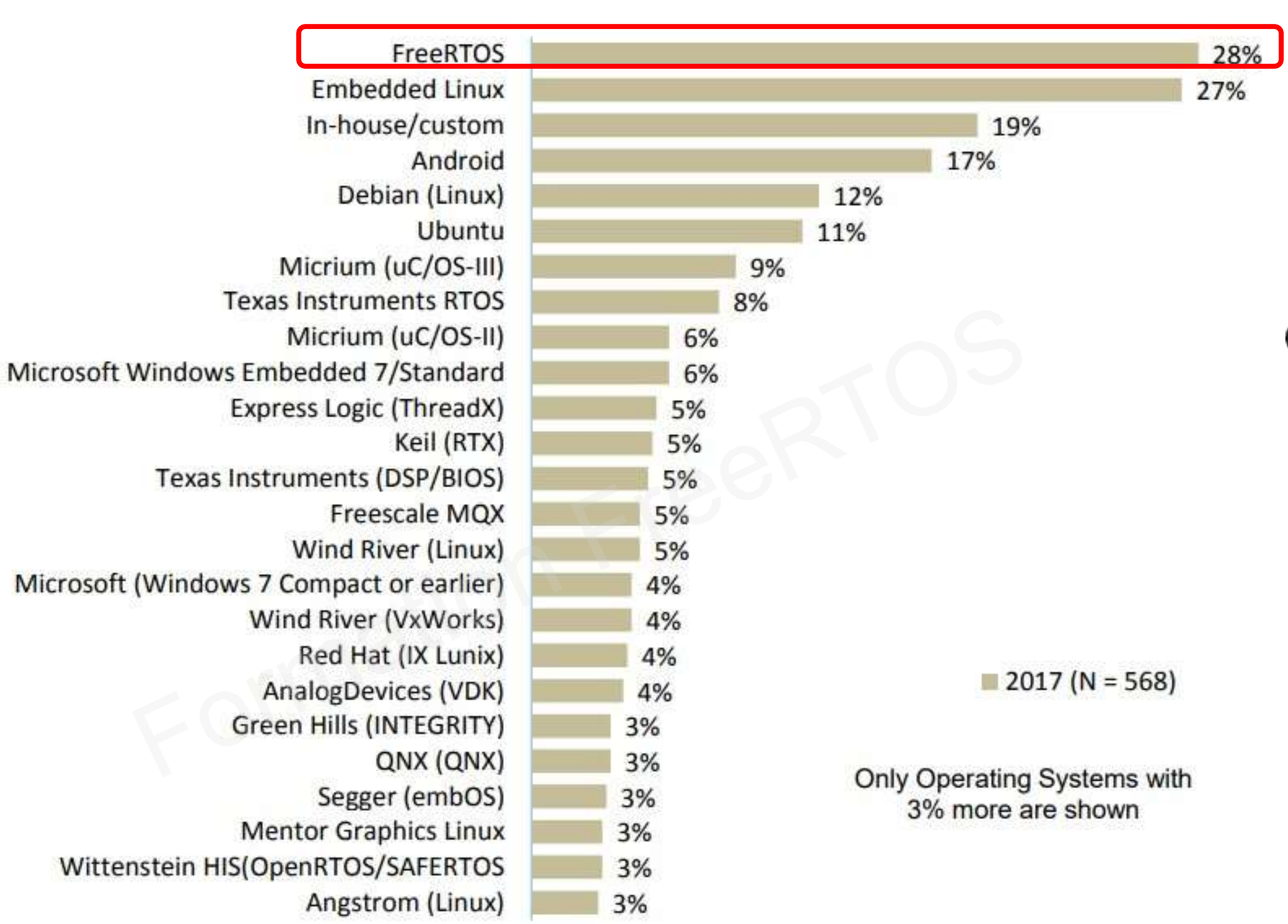
RTOS benchmarks

COMPARISON OF VARIOUS REAL TIME OPERATING SYSTEMS

RTOS	License	Scheduling Algorithm	Platforms	Memory Allocation
VxWorks	Proprietary	Preemptive and Round Robin Scheduling	ARM, IA-32, Intel 64, MIPS, PowerPC, SH-4, StrongARM, xScale	Best Fit Algorithm
QNX	Proprietary	Priority-Preemptive Scheduling	IA-32, MIPS, PowerPC, SH-4, ARM, StrongARM, XScale	Strict Memory Protection by Memory Management Unit
eCos	Modified GNU GPL	Bitmap Scheduler and Multiple-Priority, Queue-Based Scheduler	ARM-XScale-Cortex-M, 680x0-ColdFire, fr30, FR-V, IA-32, MIPS, MN10300, OpenRISC, PowerPC, SPARC, SuperH	Memory Pool Based Dynamic Memory Allocation
RTLinux	GNU GPL	FIFO, Earliest Deadline First Scheduler	Alpha, ARC, ARM, AVR32, Blackfin, C6x, ETRAX CRIS, M32R, m68k, META, Microblaze, MIPS, MN103, Nios II, OpenRISC, SPARC, x86	Uses Regular Linux Memory Management Provisions. No Real Time Allocation
WinCE	Proprietary	Priority-Based Time-Slice Algorithm	ARM, MIPS, SH4 and x86 Architectures	Large Memory Mapped File Support
FreeRTOS	Modified GPL License	Priority Based Round Robin Scheduling	ARM (ARM7, ARM9, Cortex-M3, Cortex-M4, Cortex-A), Atmel AVR, AVR32, HCS12, MicroBlaze, Cortus (APS1, APS3, APS3R, APS5, FPF3, FPS6, FPS8), MSP430, PIC, Renesas H8/S, SuperH, RX, x86, 8052, Coldfire, V850, 78K0R, Fujitsu MB91460 series, Fujitsu MB96340 series, Nios II, Cortex-R4, TMS570, RM4x	Primitive Allocate and Free Algorithms with Memory Coalescence.

Please select ALL of the operating systems you are considering using in the next 12 months.

ASPENCORE

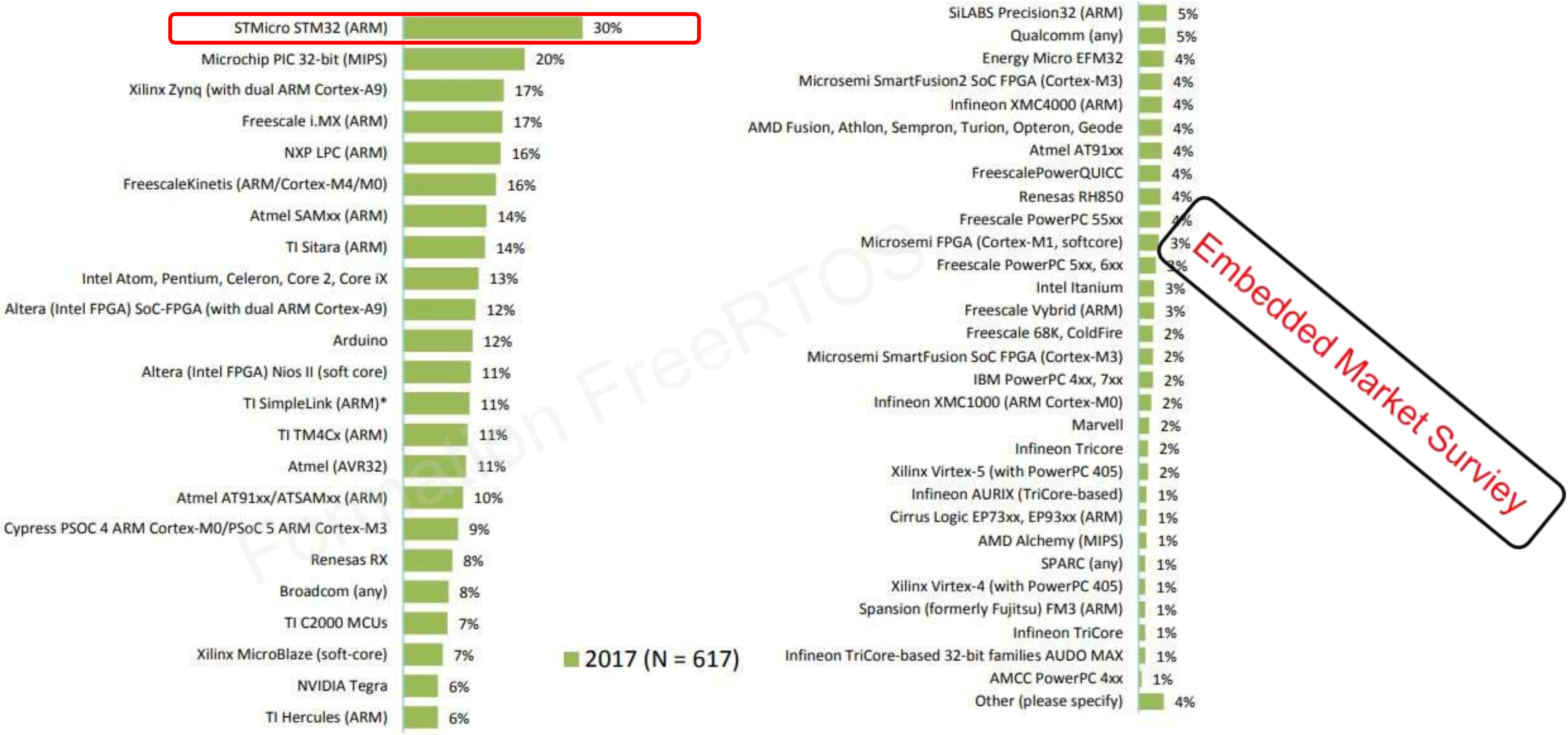


Embedded Market Surviey

Base: Those who are considering an operating system in any project in the next 12 months

Which of the following 32-bit chip families would you consider for your next embedded project?

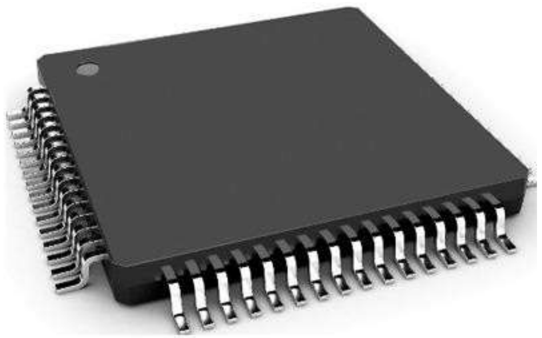
ASPENCORE



What is RTOS

Bare-Metal Application

- Legacy MCU application (Bare Metal)



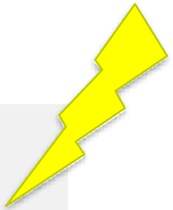
periodic

Background

```
int main()
{
 Init()

 While(1)
  {
   ADC_Read();         3ms
   SPI_Read();         2ms
   LCD_WritePacket();  1ms
   USB_Packet();       5ms
  }

}
```

Thread mode

Foreground



```
void USB_ISR()
{
  Read_PAcket()
  ClearFlag()
}
```

Handler mode

OS Application vs Baremetal

- RTOS Application

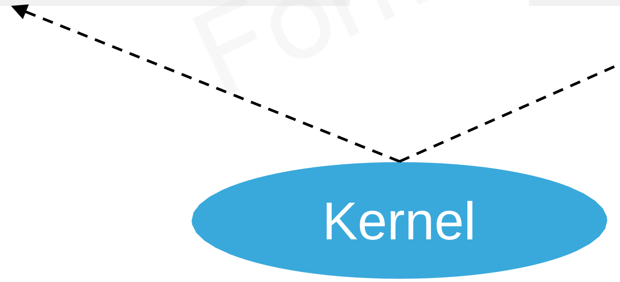
Kernel Task

```
void ADC_Task(void *p)
{
 Init()
  While(1)
  {
    ADC_Read();
    sleep(1ms)
}
}
```

Thread mode

```
void USB_Task(void *p)
{
 Init()
  While(1)
  {
    wait for signal form ISR
    USB_Packet();
}
}
```

Thread mode



Background loop

```
int main()
{
 Init()

 While(1)
  {
    ADC_Read();
    SPI_Read();
    LCD_WritePacket();
    USB_Packet();
  }

}
```

Thread mode

OS application

- RTOS Application

Kernel Task

```
void ADC_Task(void *p)
{
 Init()
  While(1)
  {
    ADC_Read();
    sleep(1ms)
}
}
```

Thread mode

```
void USB_Task(void *p)
{
 Init()
  While(1)
  {
   wait for signal form ISR();
    USB_Packet();
}
}
```

Thread mode

Foreground

```
void USB_ISR()
{
  ClearFlag();
    signal usb task
}
```

Handler mode

Kernel

- RTOS: Real Time operating system

Real time

OS

Real-time systems have been defined as: "those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced";
J. Stankovic, "Misconceptions About Real-Time Computing," ***IEEE Computer,*** 21(10), October 1988.c

Multi-tasking (Scheduler) + Services:
Inter Task communication
task synchronization
Managing resources

What is the real time in RTOS

- Target Detection
- Calculate location
- Lunch rocket

Ci:computation time

Task(Ji)

Arrival time (release time ri)

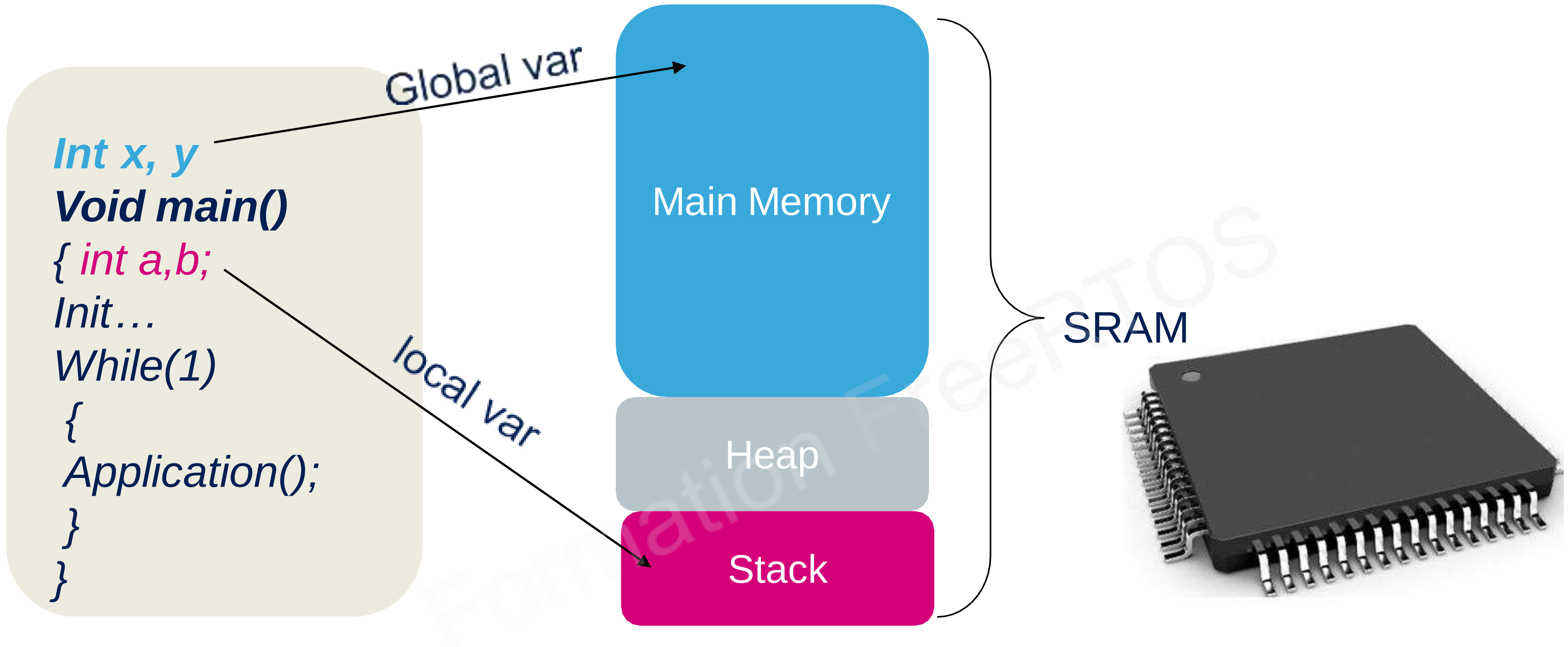
Si
Start time

fi
Finish time

Deadline (di)

What is the real time in RTOS

- All behavior in RTOS should be determinitic.
- Some calculation/decision have deadlines
  - A late answer is a wrong answer !!
  - Real time does not necessarily mean « real fast »
- When deadline are involved, it is real time.
- 3 kind of deadline
  - Hard deadline : a missing cause serious dangerous and lead to total failure.
  - Firm deadline : a missing make the value of the computation usless, but doesn't cause a serious damage.
  - Soft deadline : missing a deadline doesn't causes a serious damage

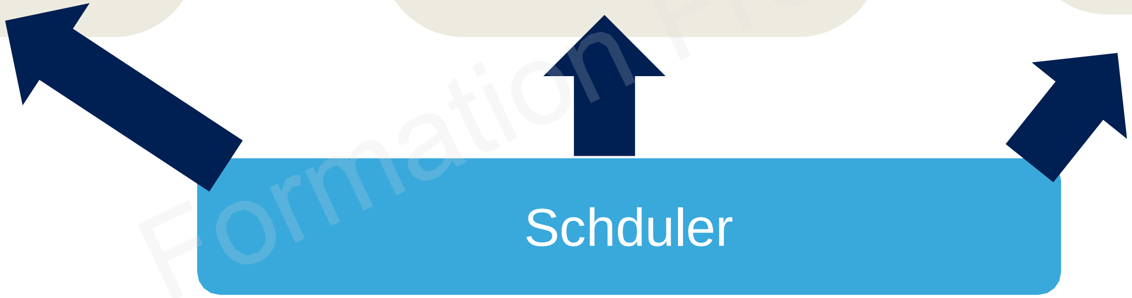


Scatter file

```
Void Task1()
{
Init…
 While(1)
 {
 Application();
 }
}
```

```
Void Task2()
{
Init…
 While(1)
 {
 Application2();
 }
}
```

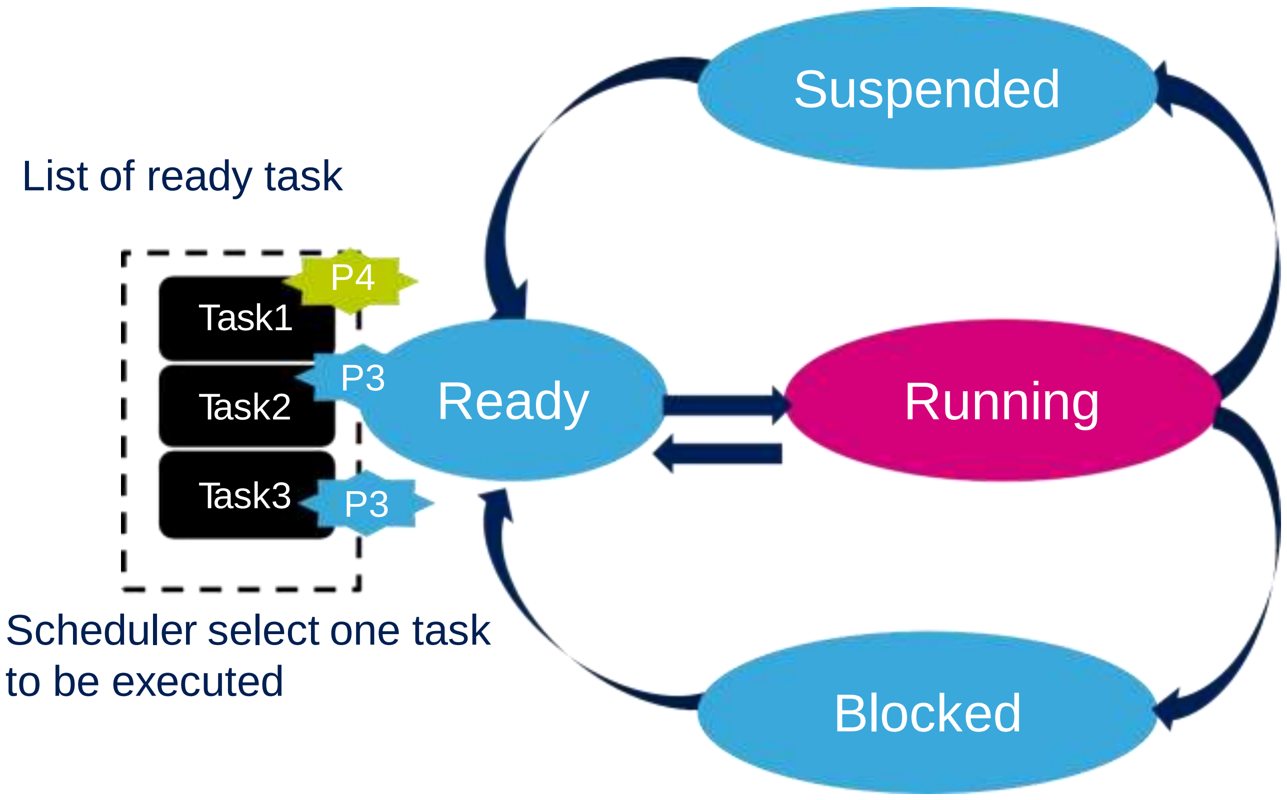
```
Void Task3()
{
Init…
 While(1)
 {
 Application3();
 }
}
```




```
Void main()
 {
  create Task..
Star
Scheduler()
}
```

Task Lifetime


```
Void main()
{

TaskCreate(Task1,P4)
TaskCreate(Task2,P3)
TaskCreate(Task3,P3)

RunScheduler()
}
```

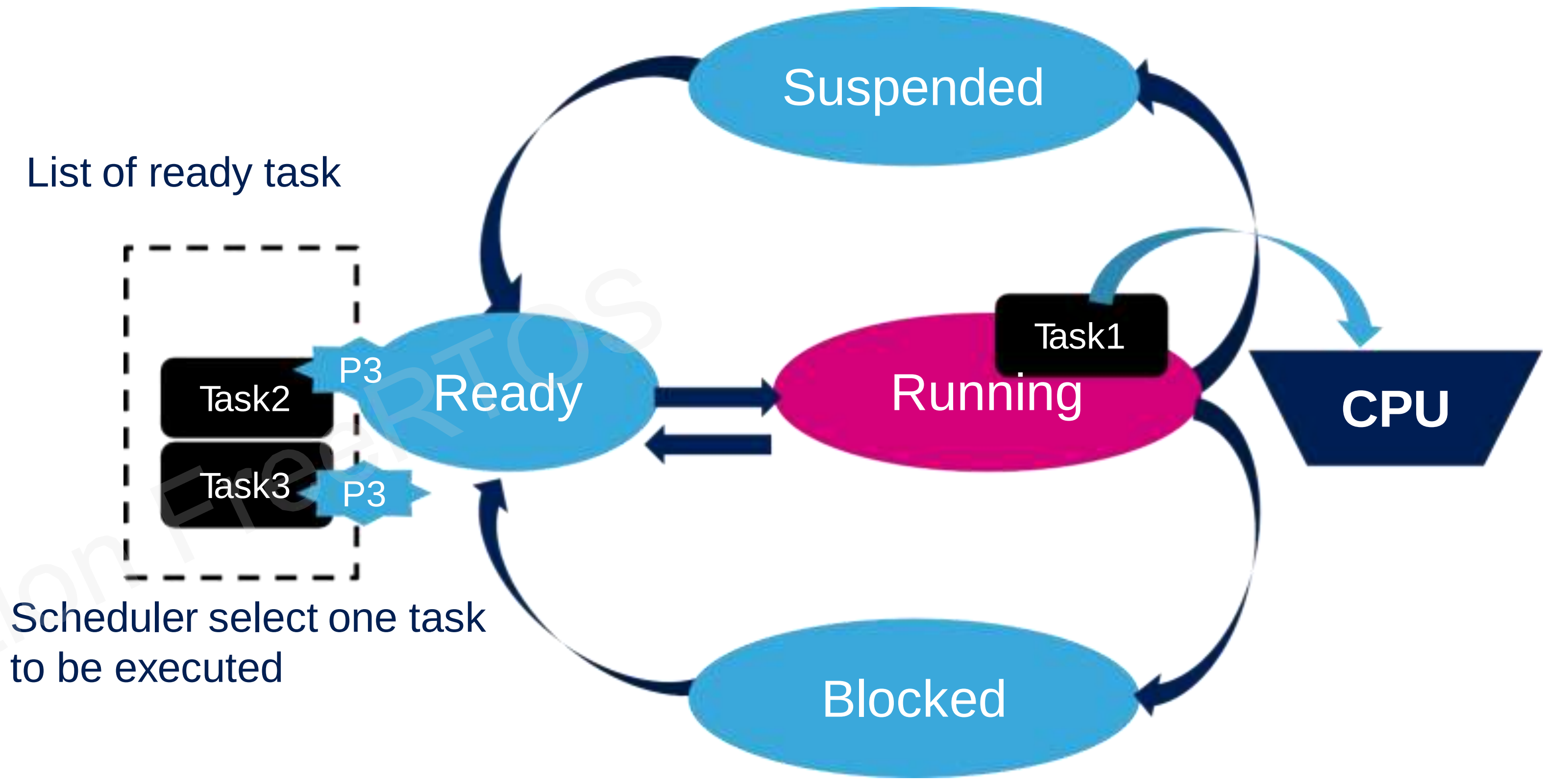


Task Lifetime

Void main()
{

TaskCreate(Task1,P4)
TaskCreate(Task2,P3)
TaskCreate(Task3,P3)

RunScheduler()
}

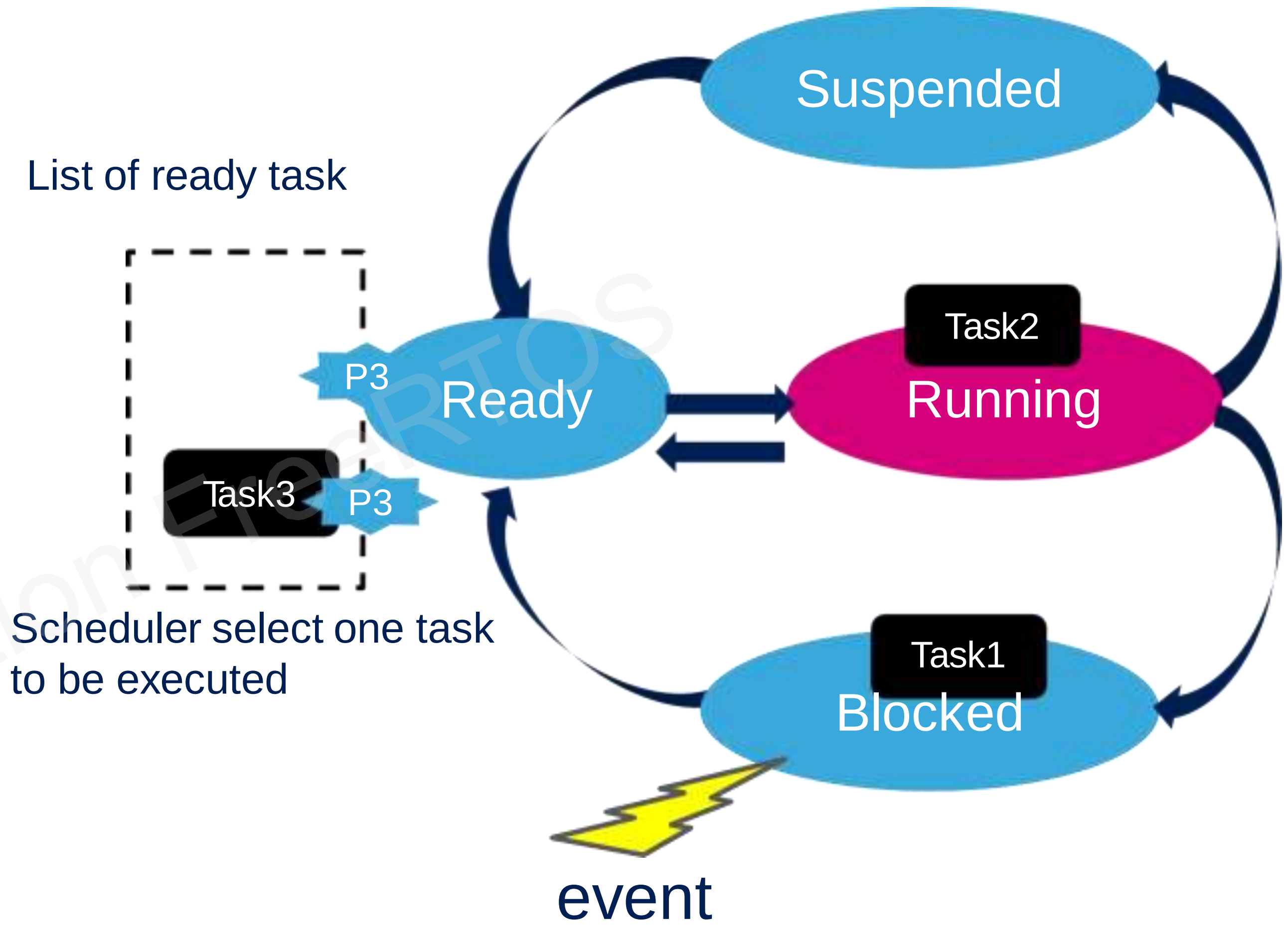


Task Lifetime

Void main()
{

TaskCreate(Task1,P4)
TaskCreate(Task2,P3)
TaskCreate(Task3,P3)

RunScheduler()
}

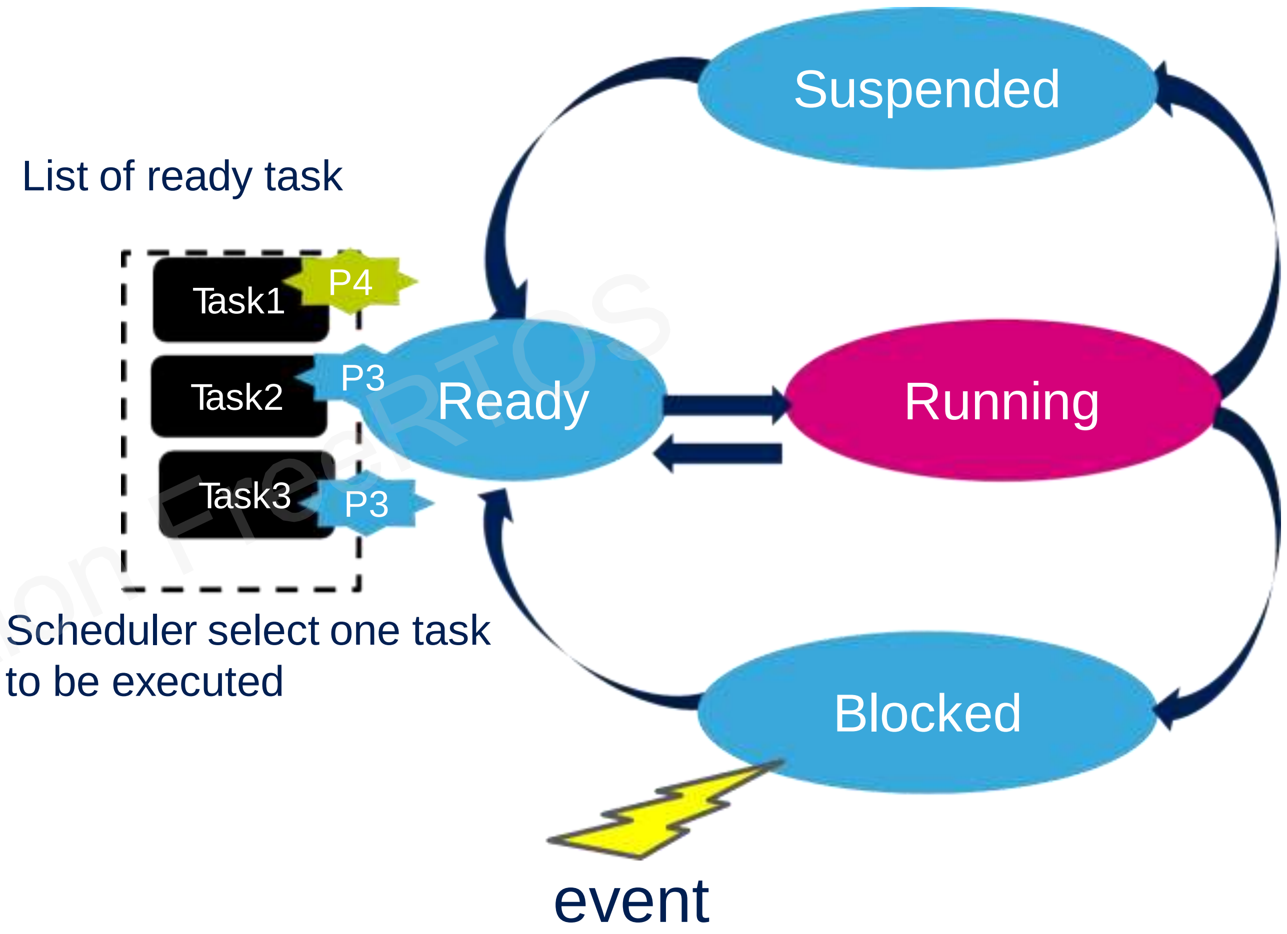


Task Lifetime

Void main()
{

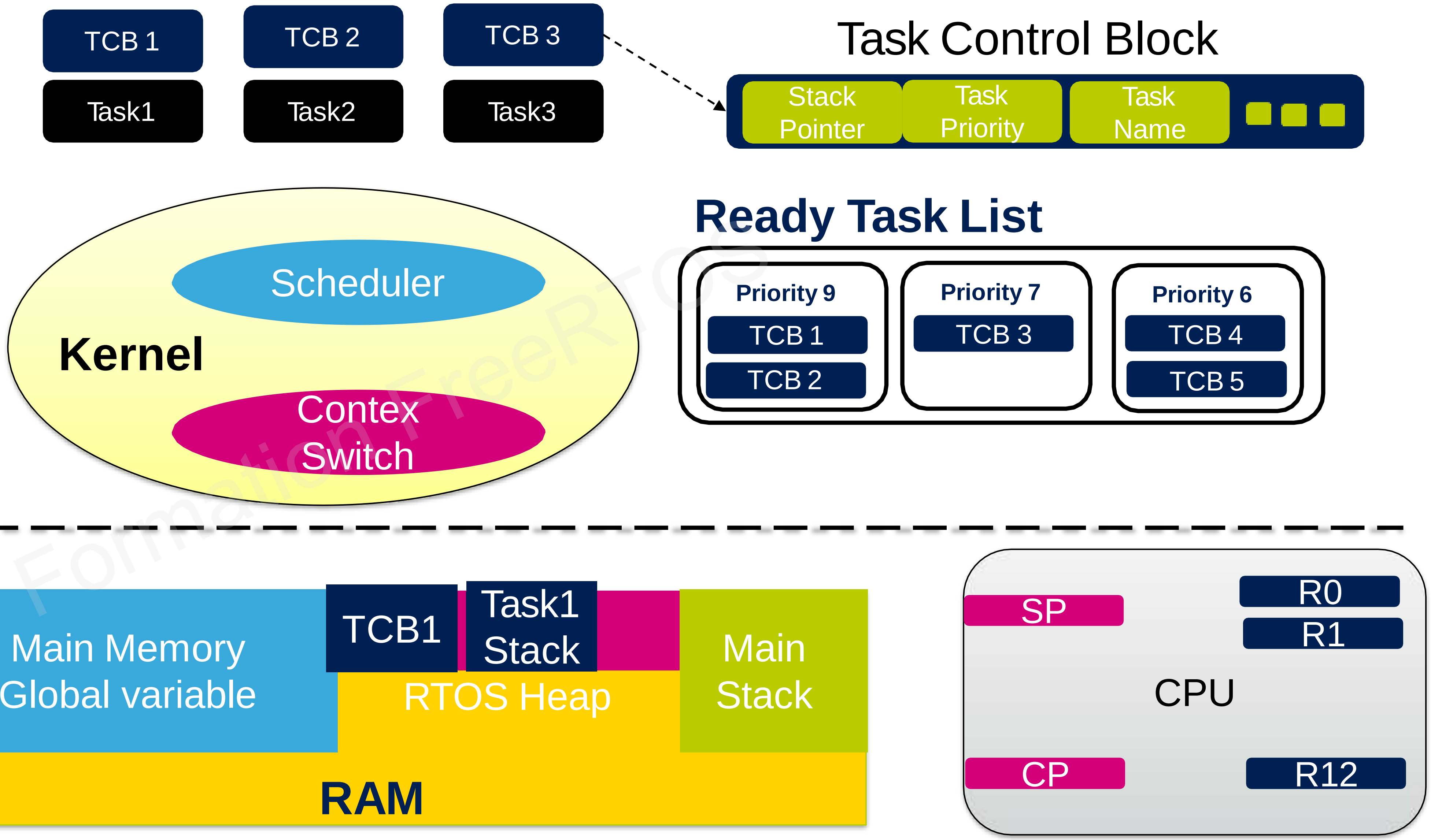
TaskCreate(Task1,P4)
TaskCreate(Task2,P3)
TaskCreate(Task3,P3)

RunScheduler()
}



Task vs Kernel Services

```
Void Task1(void *p)
{
  int a, int b..

  …
  while(1)
  {
   …
    ..
  }
}
```



HARDWARE

Task vs Kernel Services

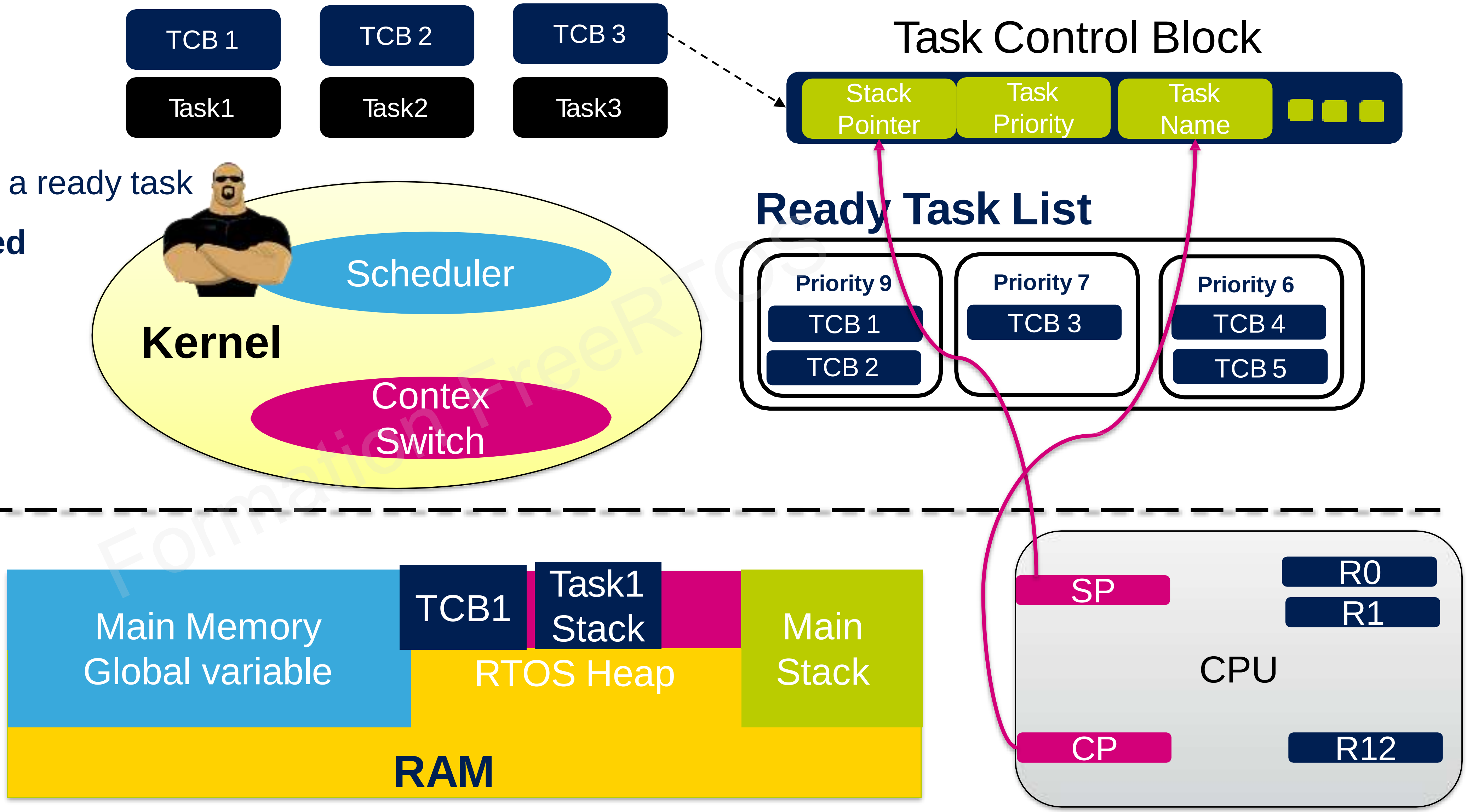
1- Scheduler chose a ready task

1- Task 1 is selected

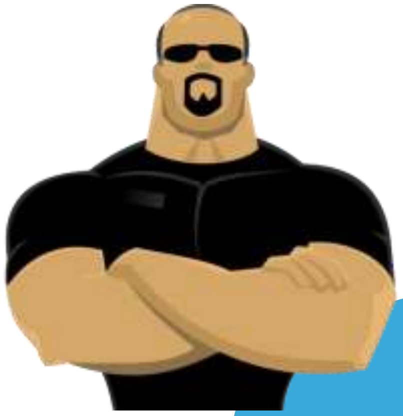
2- Contex Switch

2- Run Task1

HARDWARE



Scheduler



Ready Task List

TASK1	TASK2
TASK3	TASK4
TASK5	TASK6

Scheduler policy

REAL TIME

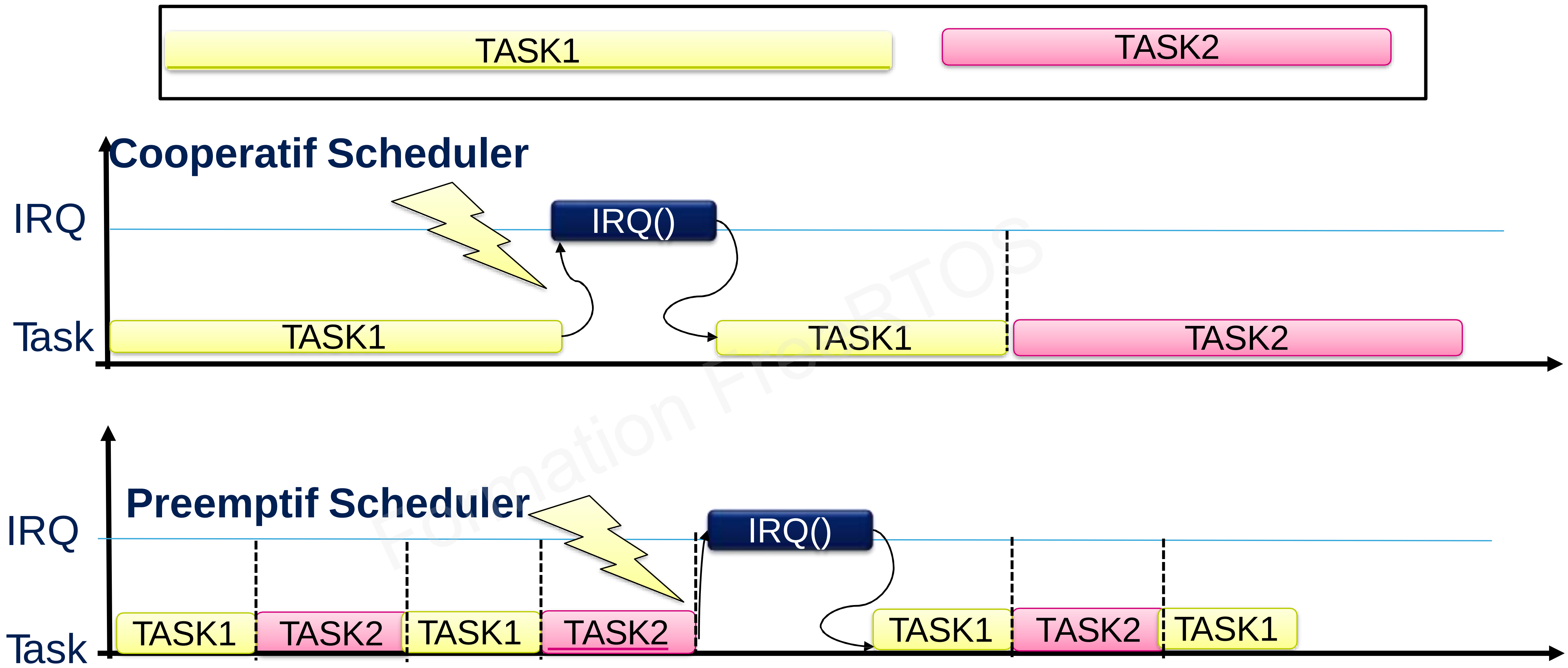
Preemptif Scheduler

Cooperatif scheduler

la tâche peut à tout instant perdre le contrôle du processeur au profit d'une tâche de priorité supérieure.
La tâche qui perd le processeur n'a aucune possibilité de le savoir. C'est le scheduler qui prend cette décision.

la tâche est exécutée jusqu'à ce qu'elle fasse appel à un service du noyau.
Selon la situation présente, le scheduler décide si la tâche doit s poursuivre ou non.

Scheduler Cooperatif vs Preemptif



11.1.2 Context Switch

When the multithreading kernel decides to run a different thread, it simply saves the current thread's context (CPU registers) in the current thread's context storage area (the *thread control block*, or TCB). Once this operation is performed, the new thread's context is restored from its TCB and the CPU resumes execution of the new thread's code. This process is called a context switch. Context switching adds overhead to the application.

What ARM did for RTOS

CPU Cortex M
Thread mode & Handler mode

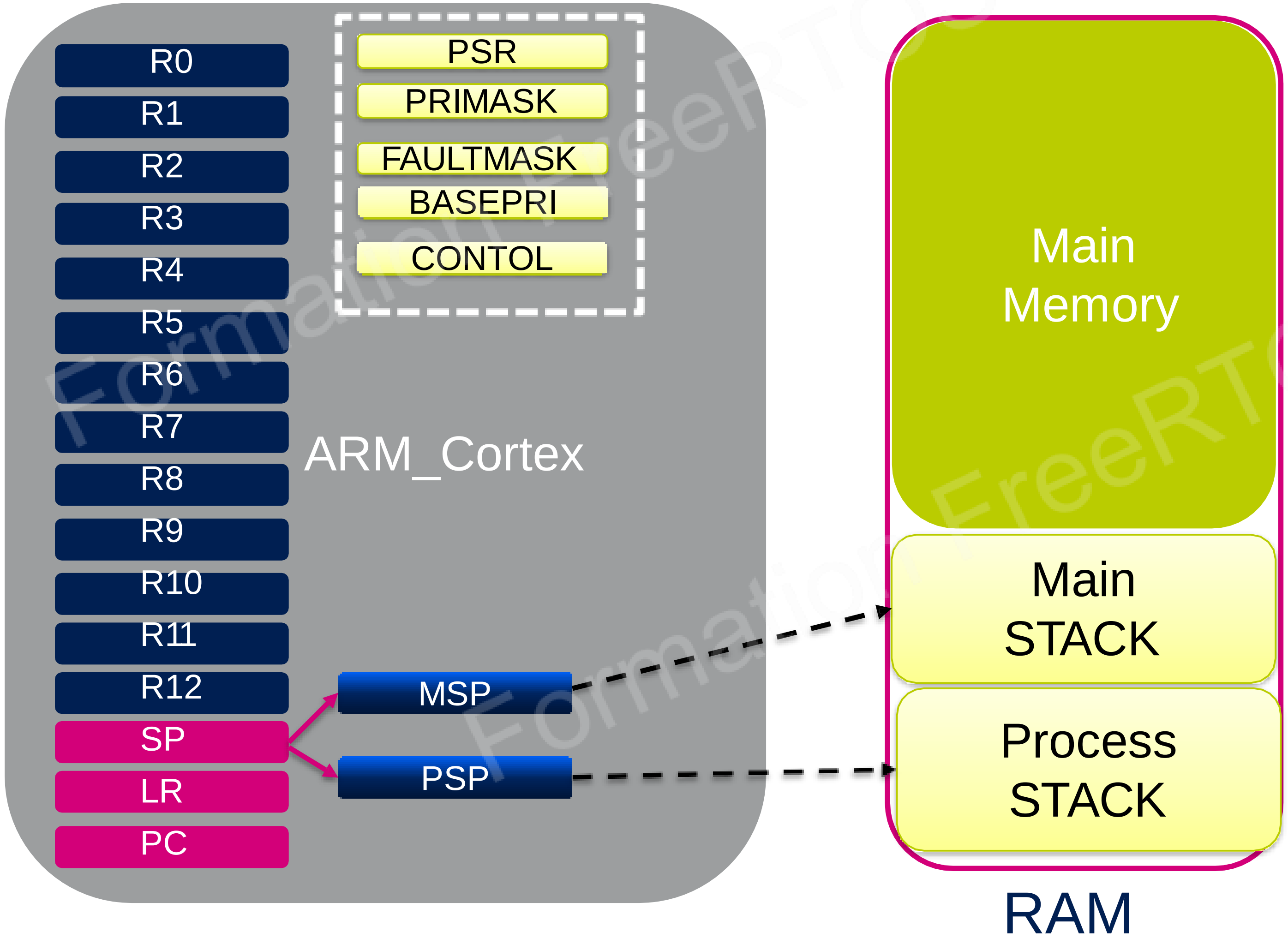
Operating Modes

- The Cortex-M3 supports Privileged and User (non-privileged) execution. Code run as Privileged has full access rights whereas code executed as User has limited access rights.
- The processor supports two operation modes, **Thread mode** and **Handler mode**.

  Thread mode is entered on reset and normally on return from an exception. When in Thread mode, code can be executed as either Privileged or Unprivileged.

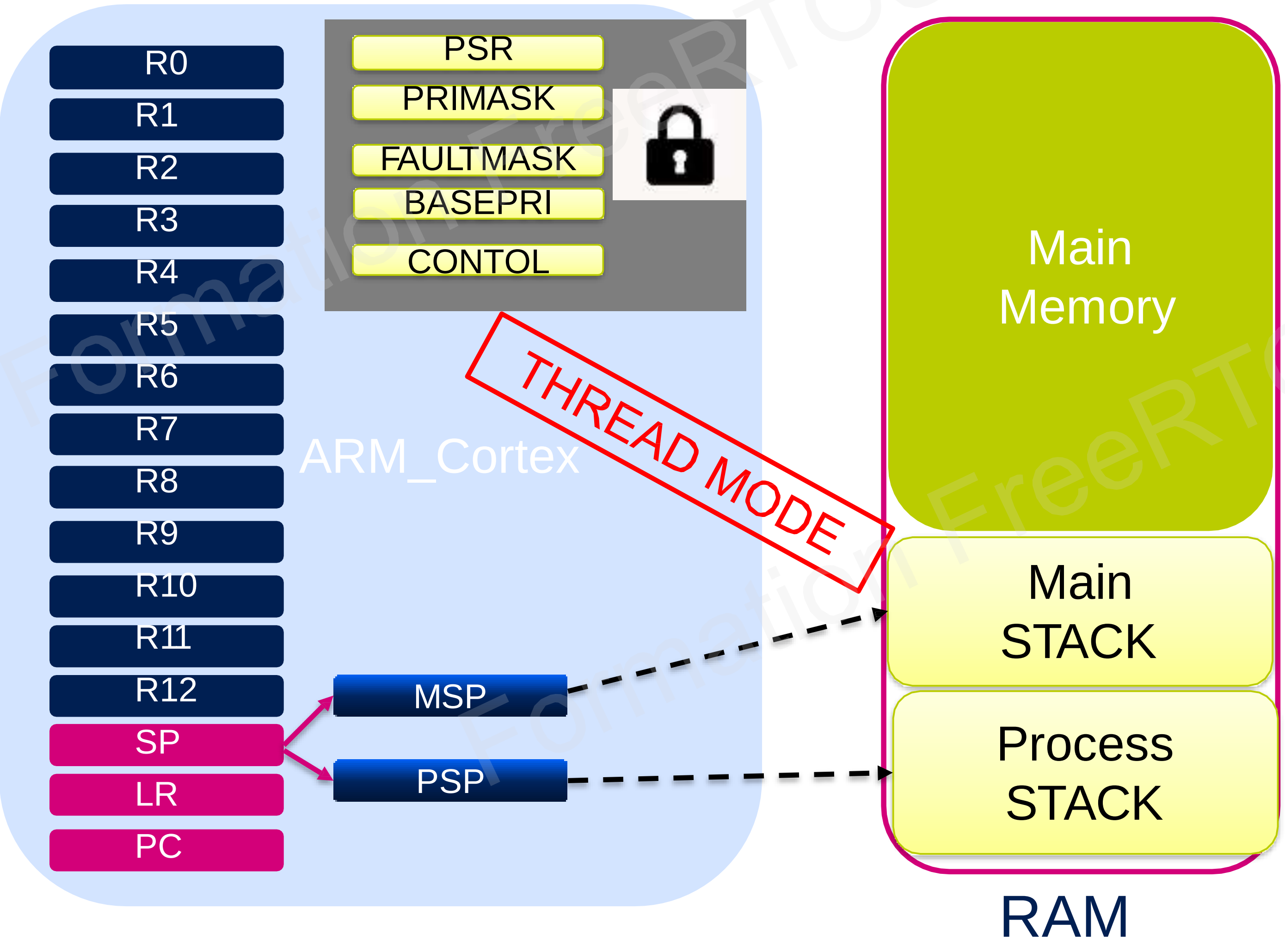
  Handler mode will be entered as a result of an exception. Code in Handler mode is always executed as Privileged, therefore the core will automatically switch to Privileged mode when exceptions occur.
- You can change between Privileged Thread mode and User Thread mode when returning from an exception by modifying the EXC_RETURN value in the link register (R14). You can also change from Privileged Thread to User Thread mode by clearing CONTROL[0] using an MSR instruction. However, you cannot directly change to privileged mode from unprivileged mode without going through an exception, for example an SVC.

ARM Cortex M Operating Modes




```
Void main()
{
 Init.........

 while(1)
  {
    GetValue();
    SetDisplay()
  }
}
```

ARM Cortex M Operating Modes

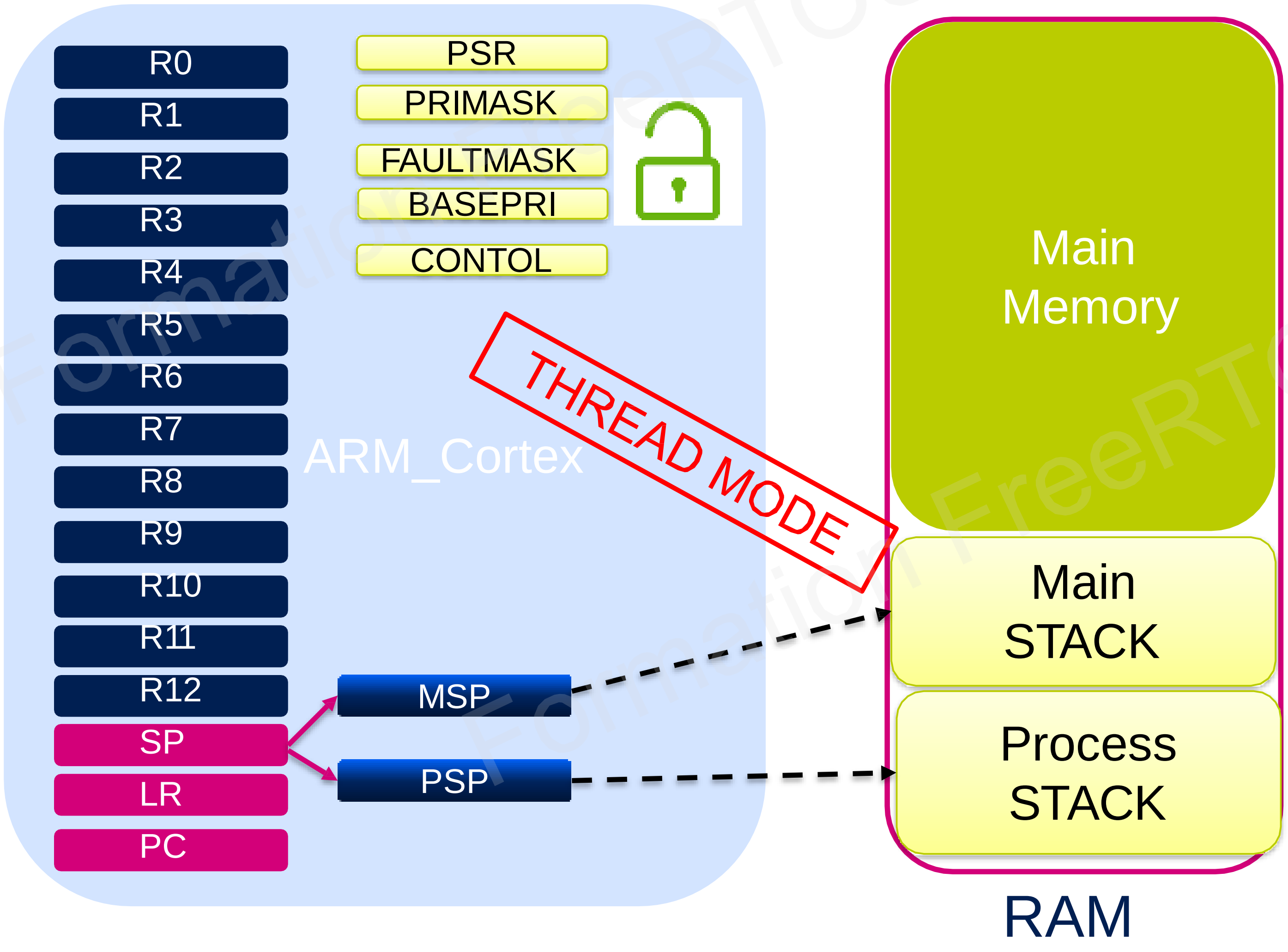



```
Void main()
{
 Init………

 while(1)
  {
    GetValue();
    SetDisplay()
  }
}
```

THREAD MODE

Non Privileged

ARM Cortex M Operating Modes

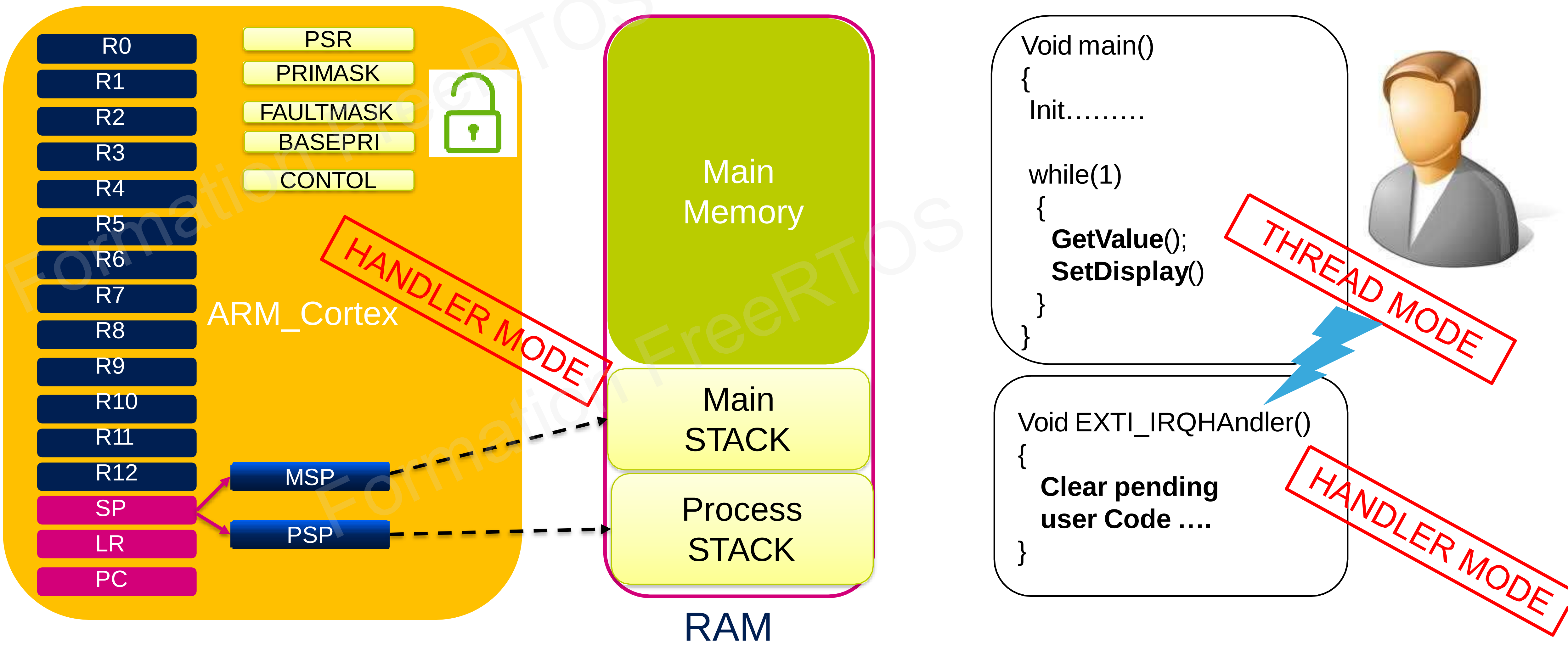



```
Void main()
{
 Init.........

 while(1)
  {
    GetValue();
    SetDisplay()
  }
}
```

THREAD MODE

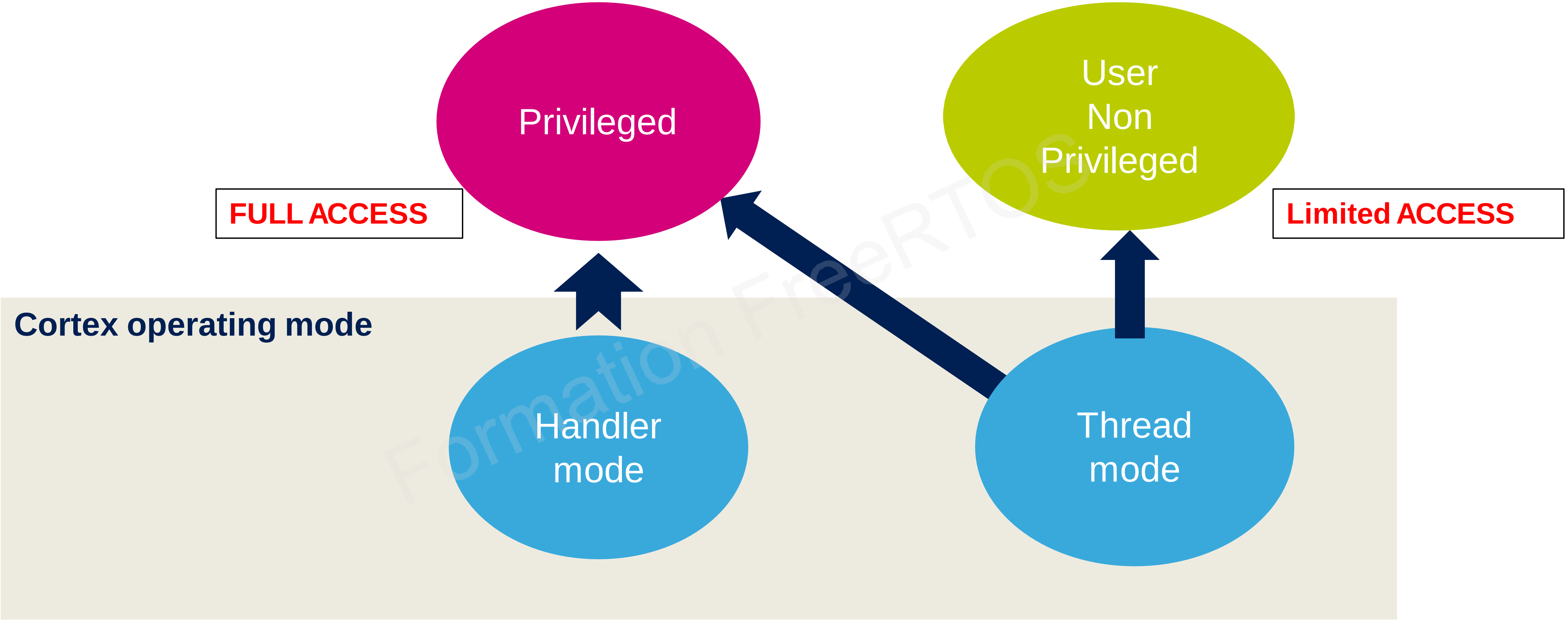
Privileged

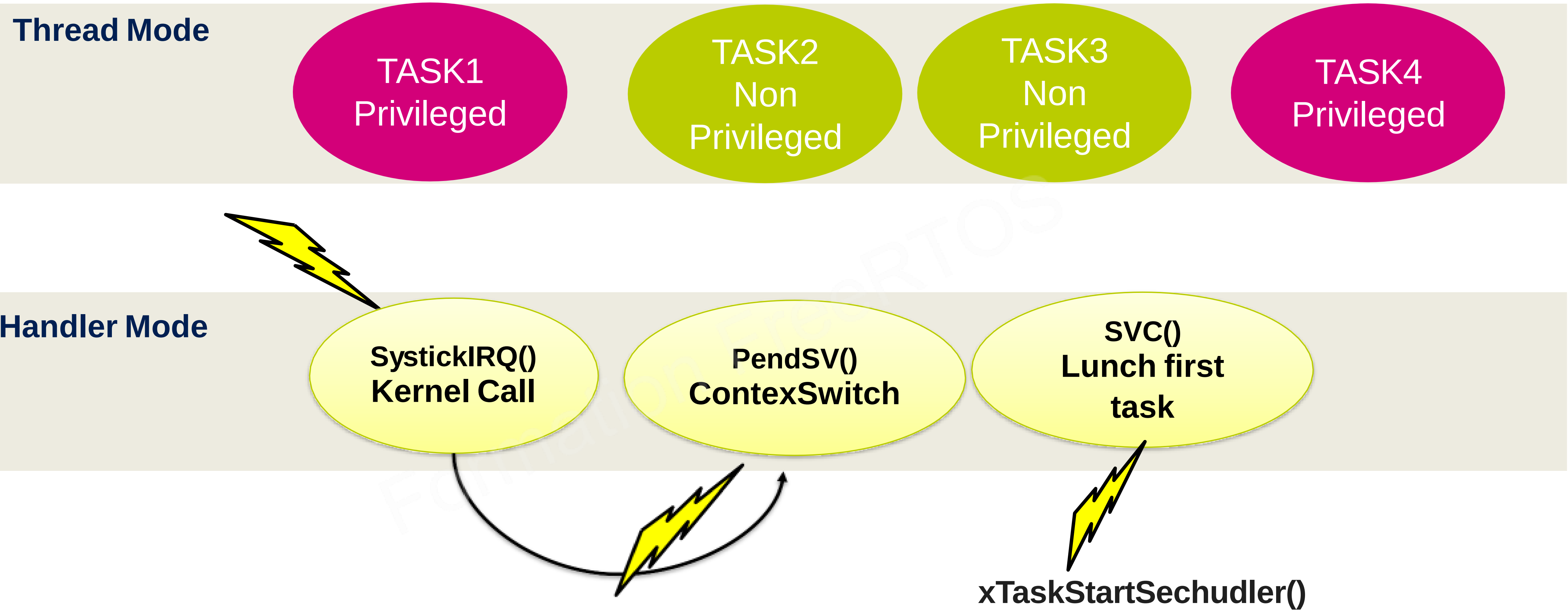
ARM Cortex M Operating Modes



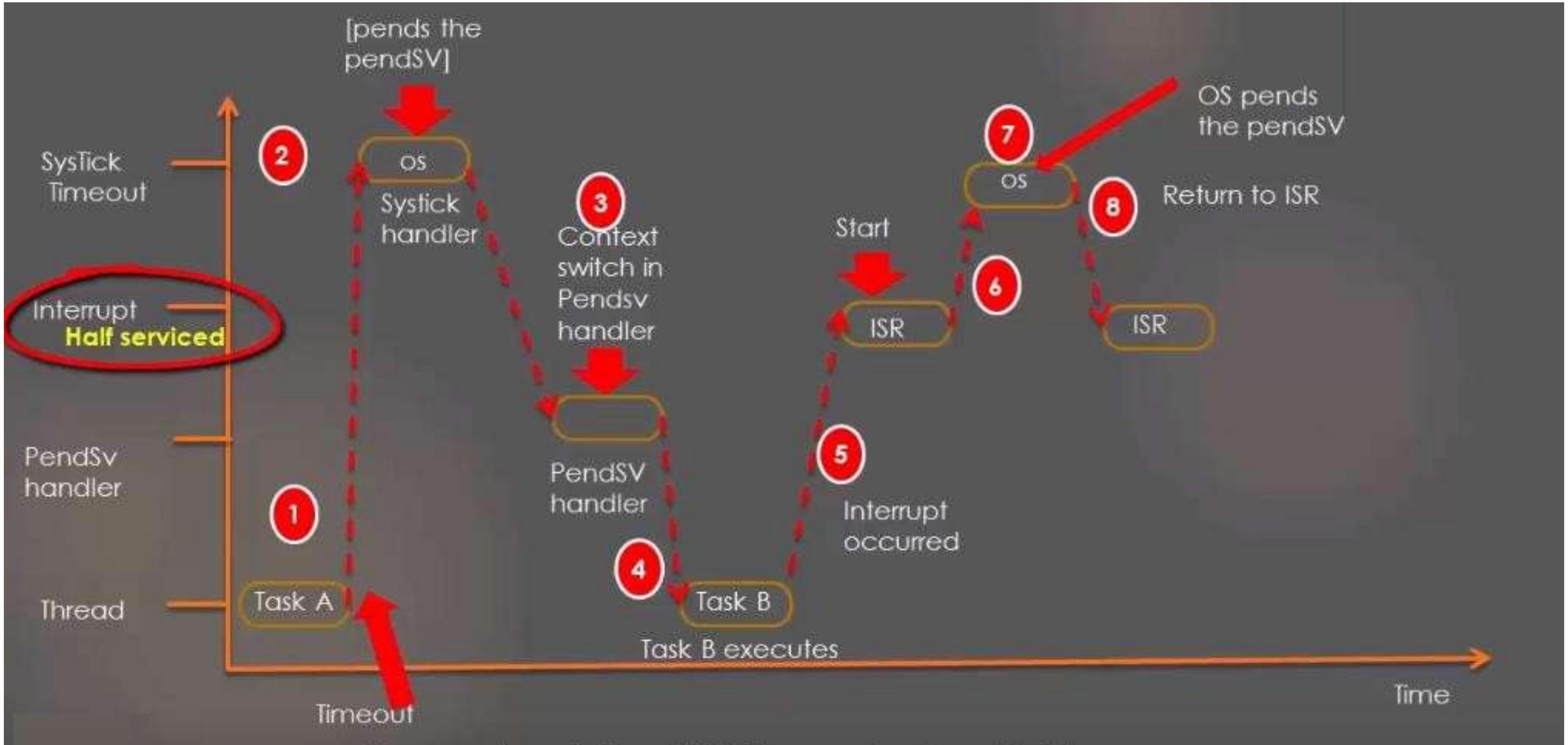
ARM Cortex M Operating Modes

Cortex M Execution mode



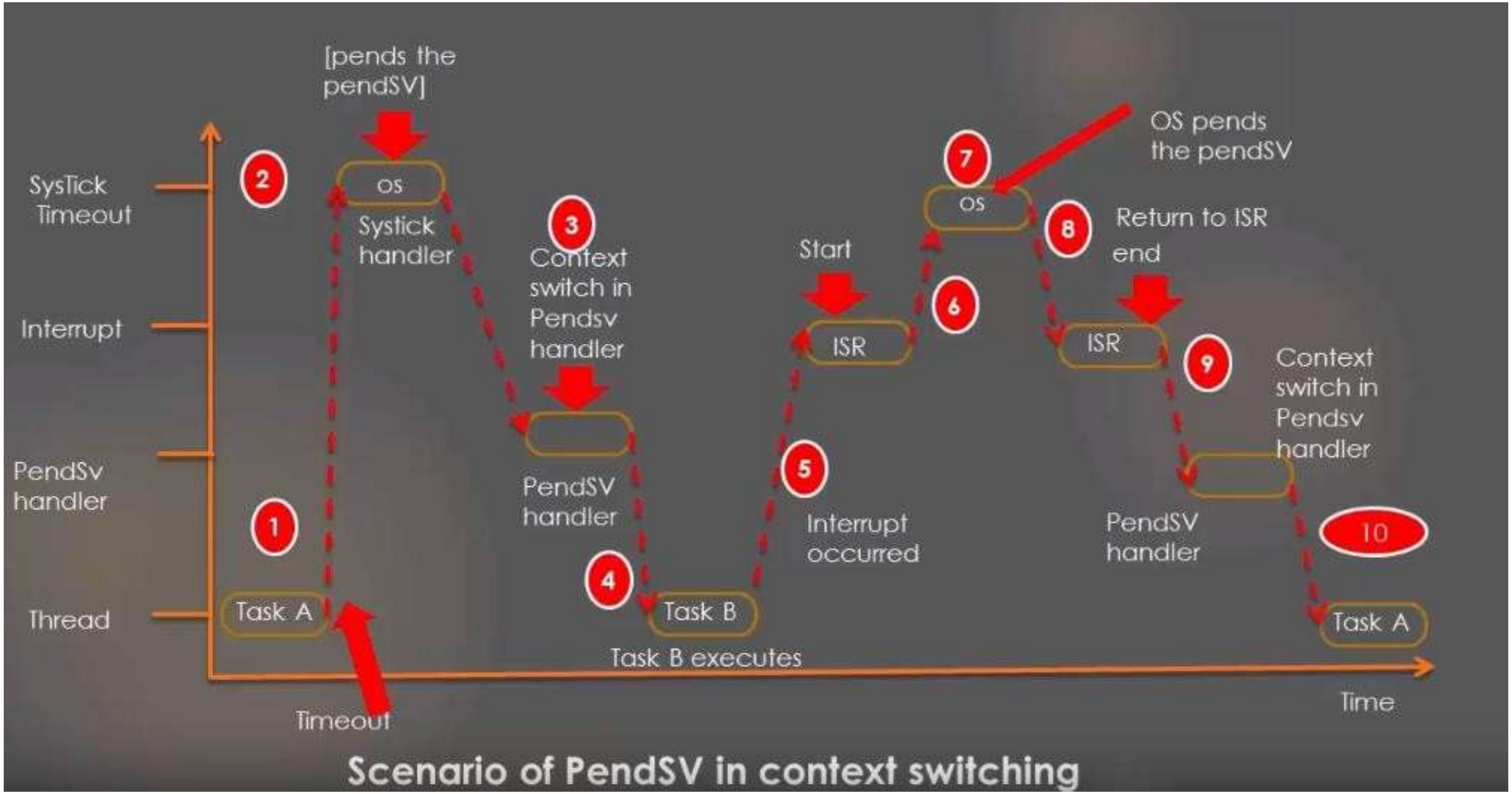

Thread Mode
TASK1 Privileged
TASK2 Non Privileged
TASK3 Non Privileged
TASK4 Privileged
Handler Mode
SystickIRQ() Kernel Call
PendSV() ContexSwitch
SVC() Lunch first task
xTaskStartSechudler()

PendSV context Switch



Scenario of PendSV in context switching

PendSV context Switch



FreeRTOS

FreeRTOS

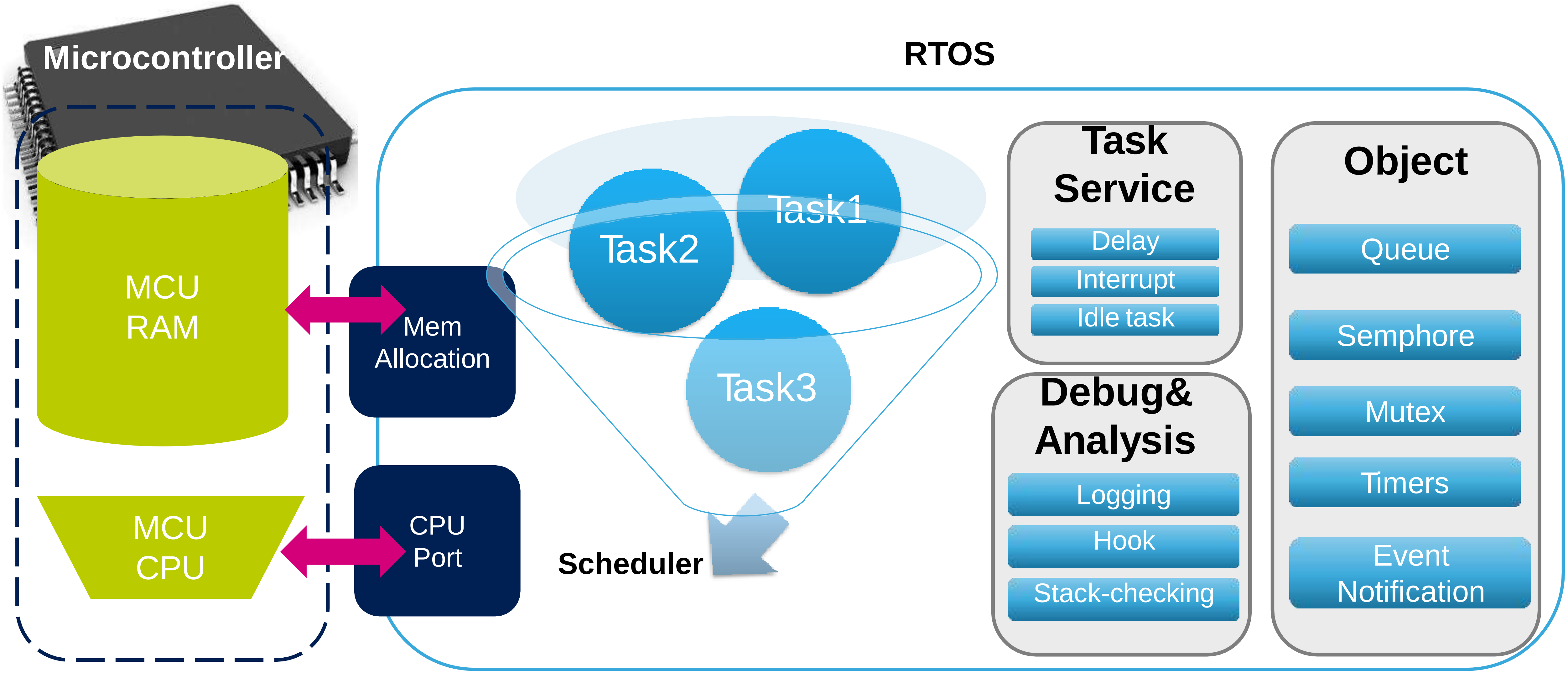
- The FreeRTOS project support 25 official architecture ports, which many more community developed ports.

- The FreeRTOS RT kernel is portabe, open source, royalty free, and very small

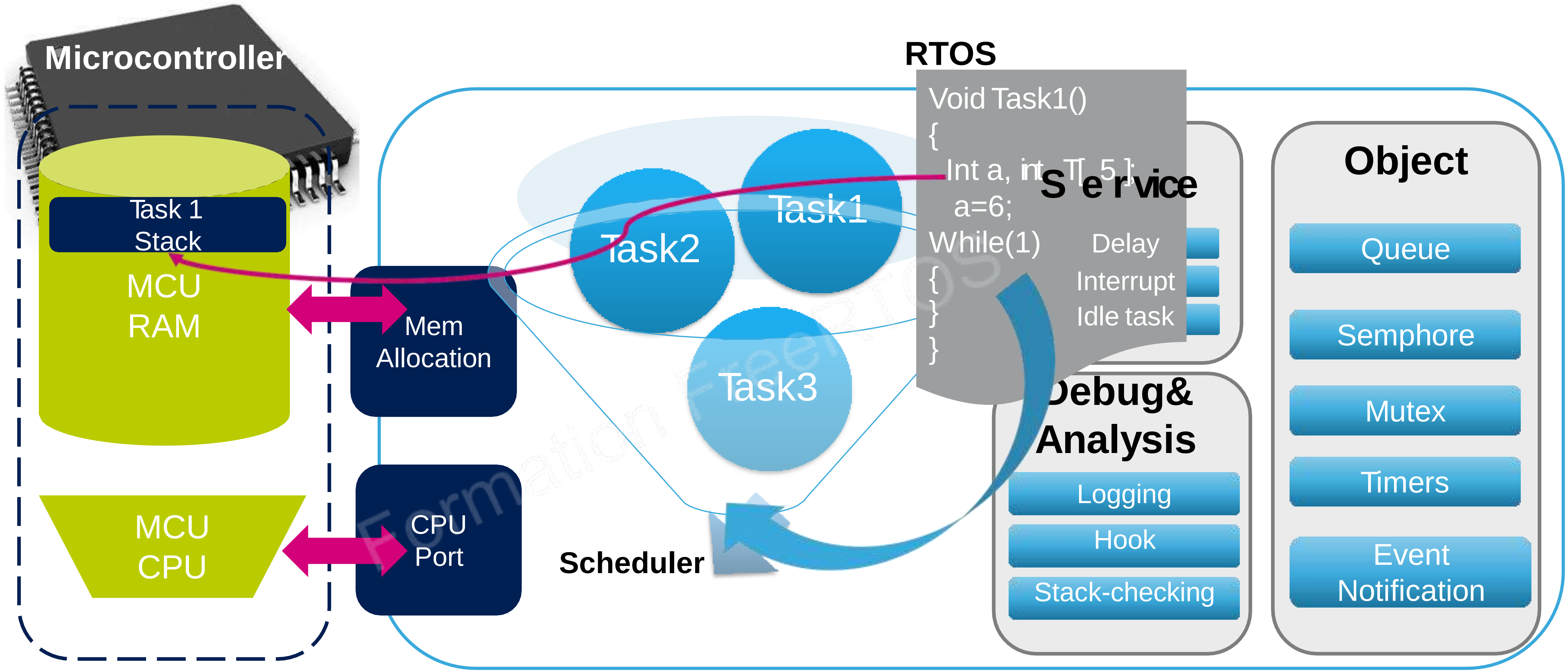
- OpenRTOS is a commercialized version by the sister company High Integrity System

- Richard Barry : I know FreeRTOS has been used in some rockets and other aircraft, but nothing too commercial.

FreeRTOS architecture

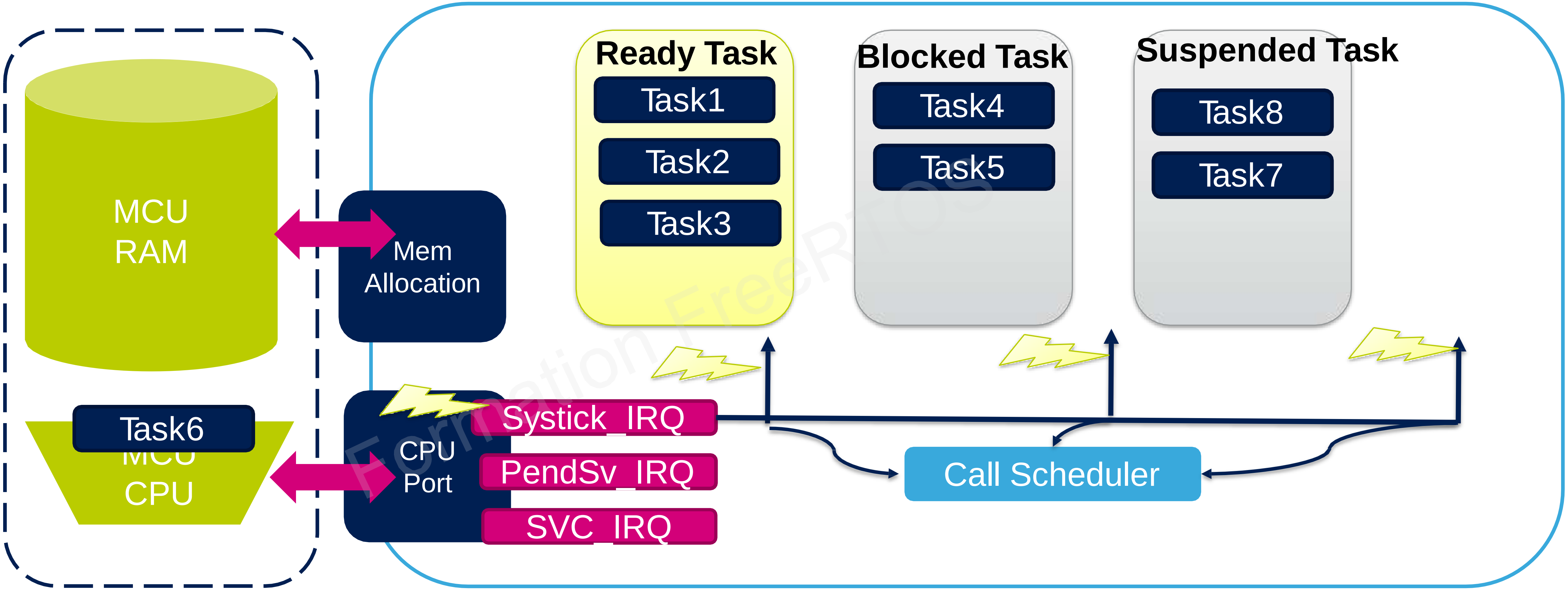


FreeRTOS architecture



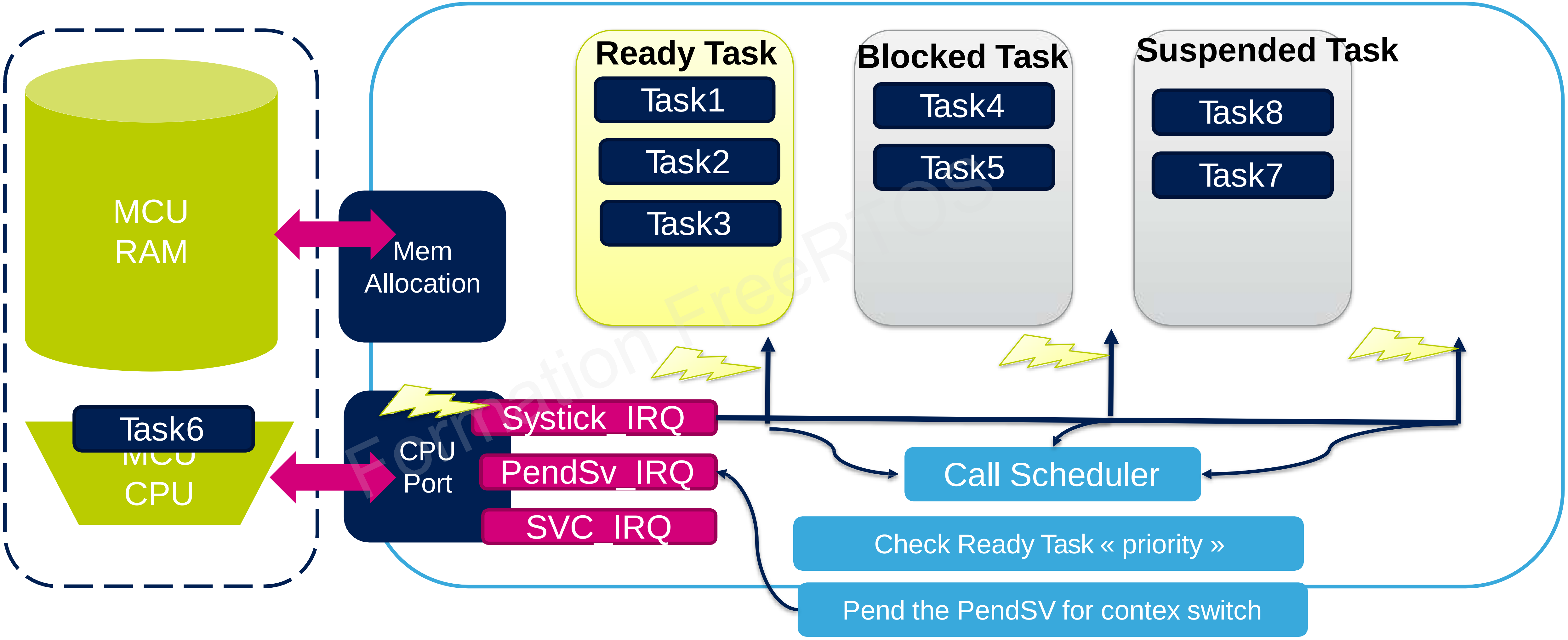
FreeRTOS architecture
How Scheduler works

Task6 is running



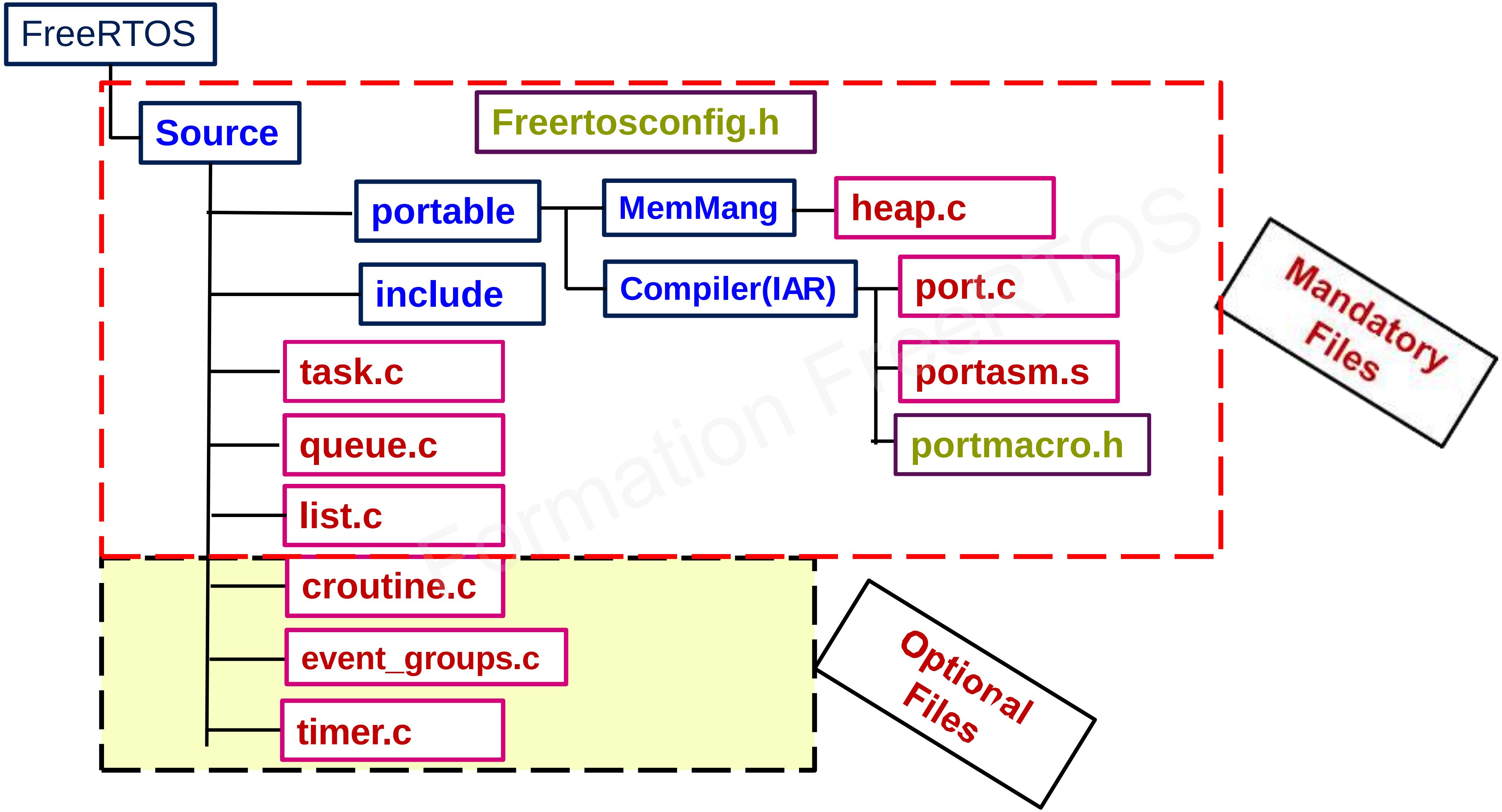
FreeRTOS architecture
How Scheduler works

Task6 is running



FreeRTOS files

Source Files



FreeRTOS files

Source Files

FreeRTOS is supplied as standard C source files that are be built along with all the other C files in your project. The FreeRTOS source files are distributed in a zip file. The RTOS source code organisation page describes the structure of the files in the zip file.

As a minimum, the following source files must be included in your project:

- `FreeRTOS/Source/tasks.c`
- `FreeRTOS/Source/queue.c`
- `FreeRTOS/Source/list.c`
- `FreeRTOS/Source/portable/[compiler]/[architecture]/port.c`.
- `FreeRTOS/Source/portable/MemMang/heap_x.c` where 'x' is 1, 2, 3, 4 or 5.

If the directory that contains the port.c file also contains an assembly language file, then the assembly language file must also be used.

Optional Source Files

If you need software timer functionality, then add `FreeRTOS/Source/timers.c` to your project.

If you need event group functionality, then add `FreeRTOS/Source/event_groups.c` to your project.

If you need steam buffer or message buffer functionality, then add `FreeRTOS/Source/stream_buffer.c` to your project.

If you need co-routine functionality, then add `FreeRTOS/Source/croutine.c` to your project (note co-routines are deprecated and not recommended for new designs).

Header Files

The following directories must be in the compiler's include path (the compiler must be told to search these directories for header files):

- `FreeRTOS/Source/include`
- `FreeRTOS/Source/portable/[compiler]/[architecture]`.
- Whichever directory contains the FreeRTOSConfig.h file to be used - see the Configuration File paragraph below.

Depending on the port, it may also be necessary for the same directories to be in the assembler's include path.

Interrupt Vectors

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

Configuration File

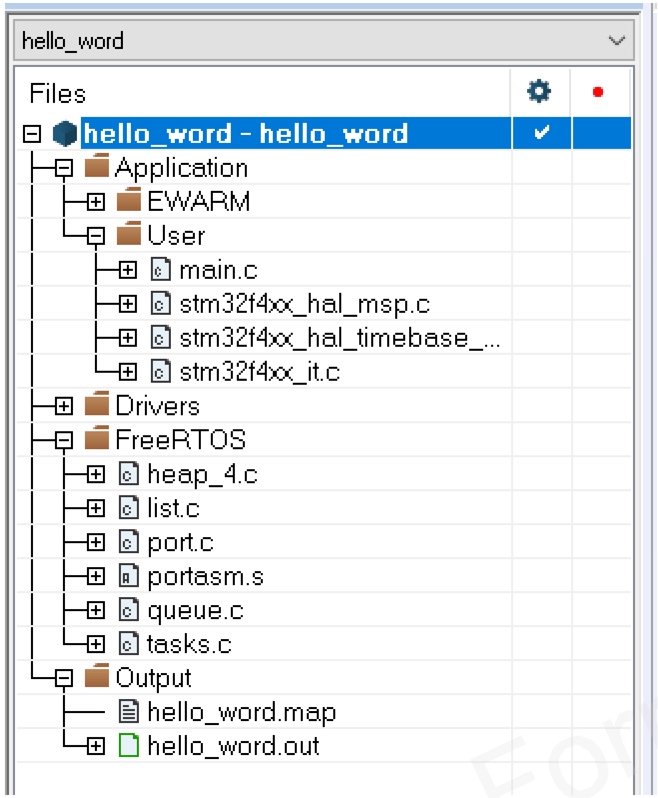
Every project also requires a file called FreeRTOSConfig.h. FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.

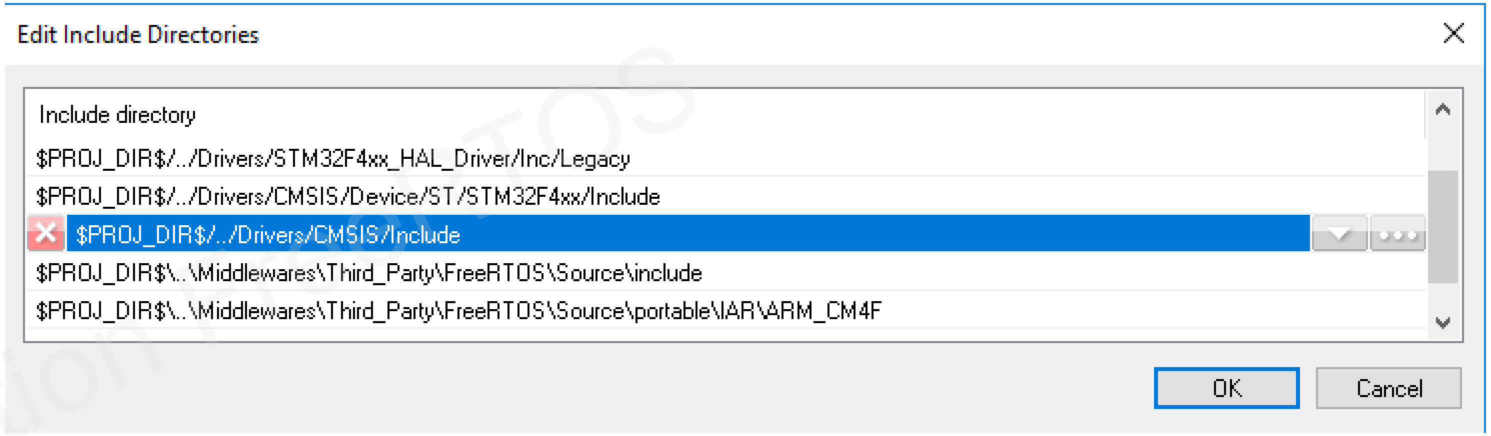
If heap_1, heap_2, heap_4 or heap_5 is included in your project, then the FreeRTOSConfig.h definition configTOTAL_HEAP_SIZE will dimension the FreeRTOS heap. Your application will not link if configTOTAL_HEAP_SIZE is set too high.

The FreeRTOSConfig.h definition configMINIMAL_STACK_SIZE sets the size of the stack used by the idle task. If configMINIMAL_STACK_SIZE is set too low, then the idle task will generate stack overflows. It is advised to copy the configMINIMAL_STACK_SIZE setting from an official FreeRTOS demo provided for the same microcontroller architecture. The FreeRTOS demo projects are stored in sub directories of the `FreeRTOS/Demo` directory. Note that some demo projects are old, so do not contain all the available configuration options.

RTOS LAB « Hello Word »

Hello word project





The interrupt file modication

stm32f4xx_it.c *

```c
* @brief This function handles System service call via SWI instruction.
*/
void SVC_Handler(void)
{
}

/**
* @brief This function handles Debug monitor.
*/
void DebugMon_Handler(void)
{
}

/**
* @brief This function handles Pendable request for system service.
*/
void PendSV_Handler(void)
{
}

/**
* @brief This function handles System tick timer.
*/
void SysTick_Handler(void)
{
 HAL_SYSTICK_IRQHandler();
}
            xPortSysTickHandler();
```

Remove these two functions since its are already declared in <u>portasm.s</u>

Remove this function

add this function to call the RTOS scheduler

Create the first task


```c
/* Includes ---------------
#include "FreeRTOS.h"
#include "task.h"
#include "list.h"
#include "queue.h"
```

```c
#define TASK1_PRIORITY          1
#define TASK1_STACK_SIZE        180 /*The number of words to allocate */
```

```c
/*Task1 prototype */
void Task1(void *p);
```

```c
int main(void)
{
  /* MCU Configuration----------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();

  /*Create Task1 */
  xTaskCreate(Task1,"HelloWord",TASK1_STACK_SIZE,NULL, TASK1_PRIORITY,  NULL);

  vTaskStartScheduler();
  /* should never be here*/
  while (1)
  {
  }
}
```

```c
/*Task 1 program */
void Task1(void *p)
{

  /*Initialization */


  while(1)
  {
    vTaskDelay(200);
    printf("Taks 1\r\n");
  }

}
```

Create the first task


```c
#define TASK1_PRIORITY              1
#define TASK1_STACK_SIZE            180 /*The number of words to allocate */

/*Task1 prototype */
void Task1(void *p);

int main(void)
{
  /* MCU Configuration----------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();

  /*Create Task1 */
  xTaskCreate(Task1,"HelloWord",TASK1_STACK_SIZE,NULL, TASK1_PRIORITY,  NULL);

  /* should never be here*/
  while (1)
  {
  }
}
```

Create a Task1, with a DebugName =''Hello word ''
With a stack size= 180 word
Priority = 1 (low priority)

Create the first task


```c
#define TASK1_PRIORITY              1
#define TASK1_STACK_SIZE            180 /*The number of words to allocate */

/*Task1 prototype */
void Task1(void *p);

int main(void)
{
  /* MCU Configuration--------------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();

  /*Create Task1 */
  xTaskCreate(Task1,"HelloWord",TASK1_STACK_SIZE,NULL, TASK1_PRIORITY,  NULL);

  /* should never be here*/
  while (1)
  {
  }
}
```

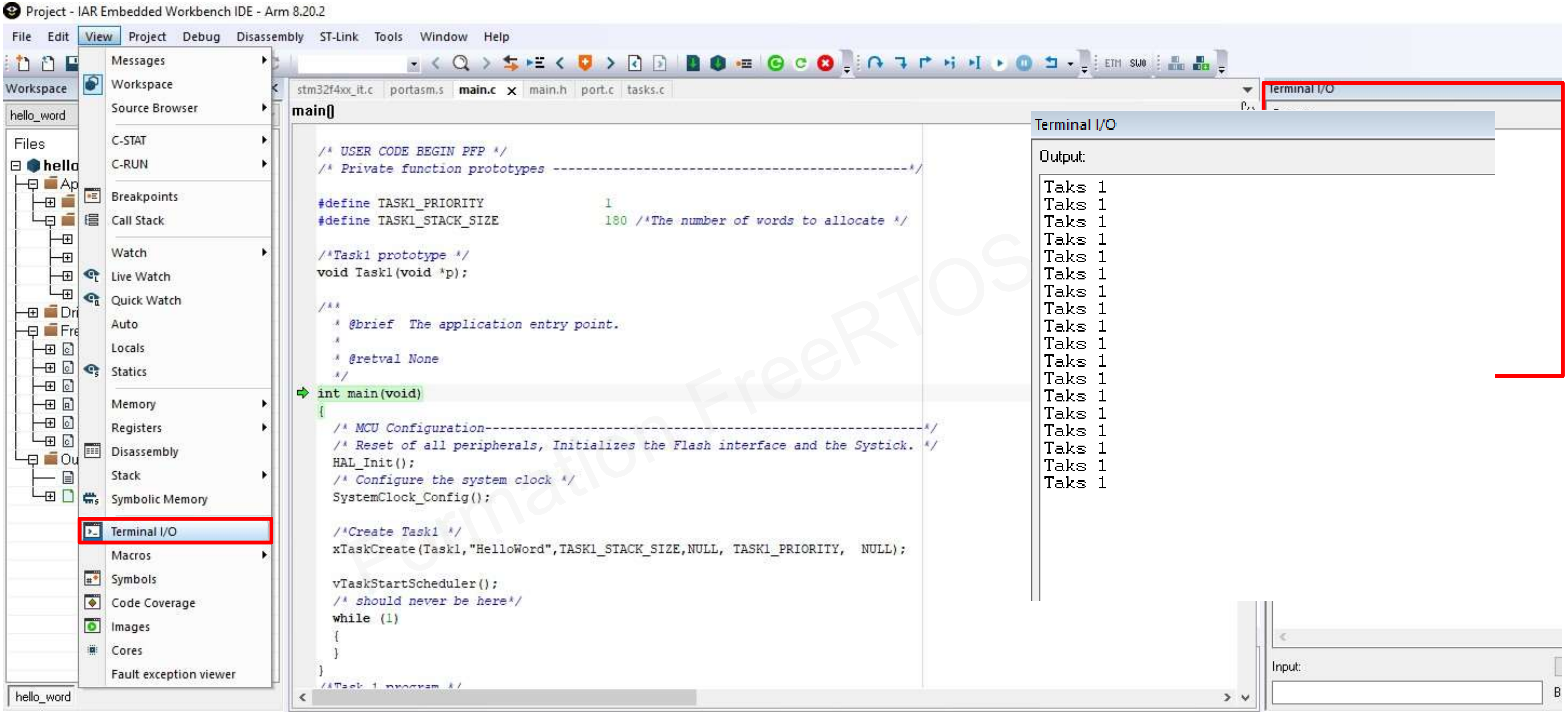
Create a Task1, with a DebugName =''Hello word ''
With a stack size= 180 word
Priority = 1 (low priority)
Without any task parameter
No task Handle

Create the frist task

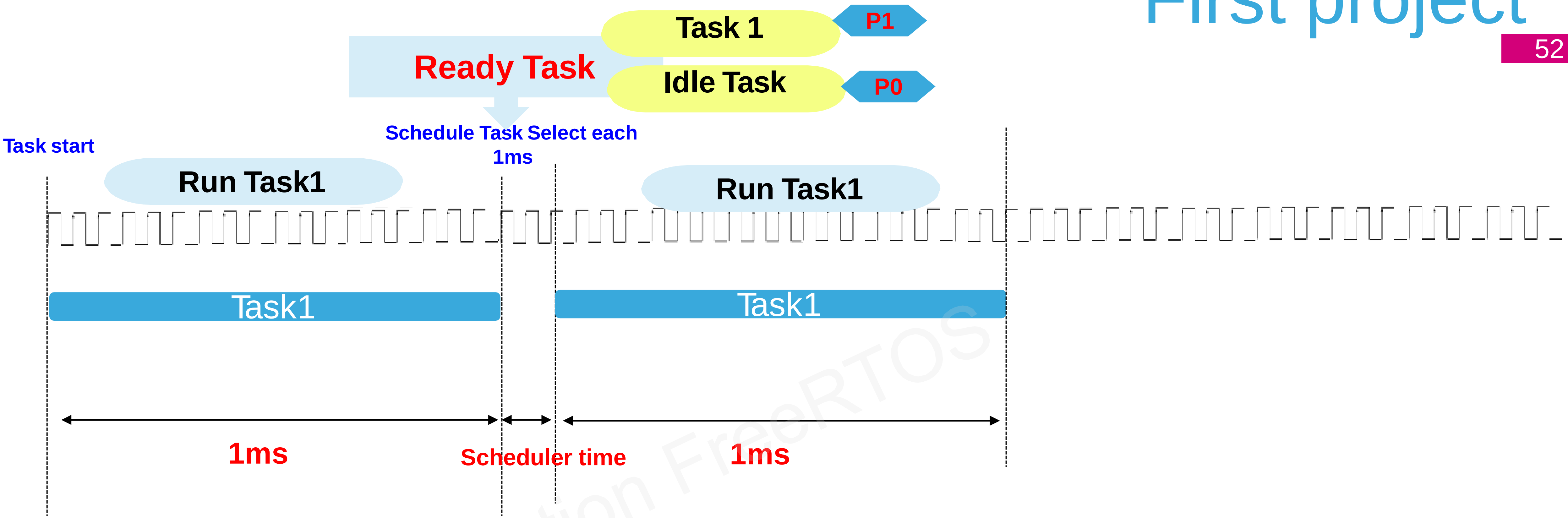

```c
#define TASK1_PRIORITY              1
#define TASK1_STACK_SIZE            180 /*The number of words to allocate */

/*Task1 prototype */
void Task1(void *p);

int main(void)
{
  /* MCU Configuration----------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();

  /*Create Task1 */
  xTaskCreate(Task1,"HelloWord",TASK1_STACK_SIZE,NULL, TASK1_PRIORITY,  NULL);

  vTaskStartScheduler();
  /* should never be here*/
  while (1)
  {
  }
}
```

Run the RTOS scheduler

Debug the first project



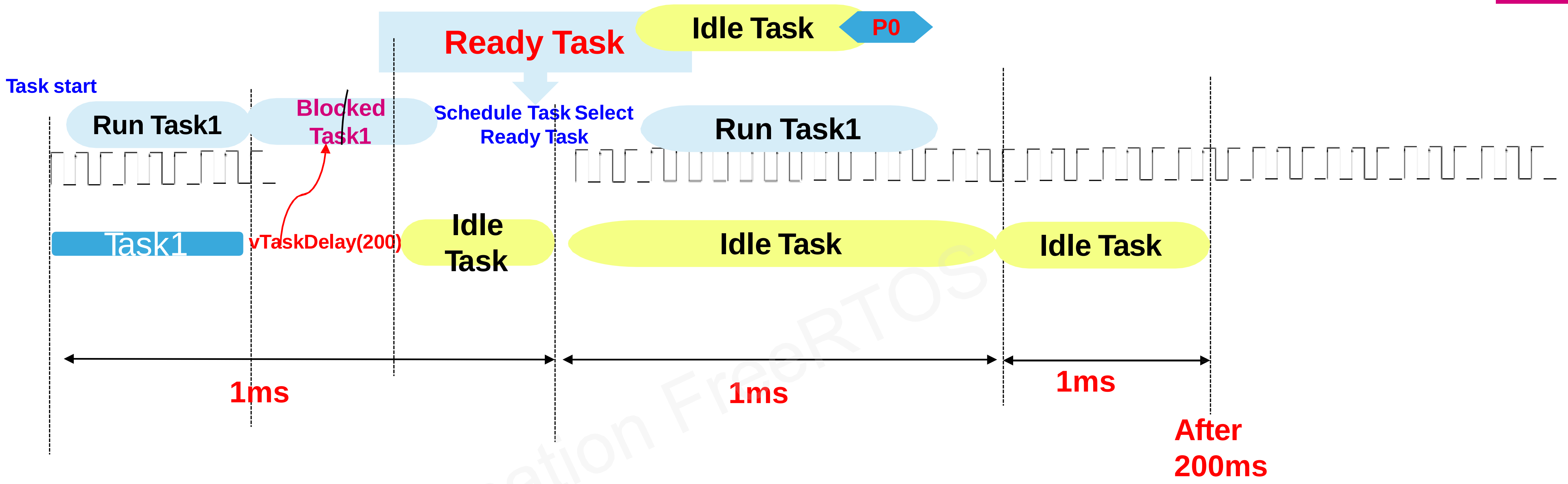
First project



Sysclk=16Mhz **16000 cycle in Run Mode**

Sysclk=160Mhz **160000 cycle in Run Mode**

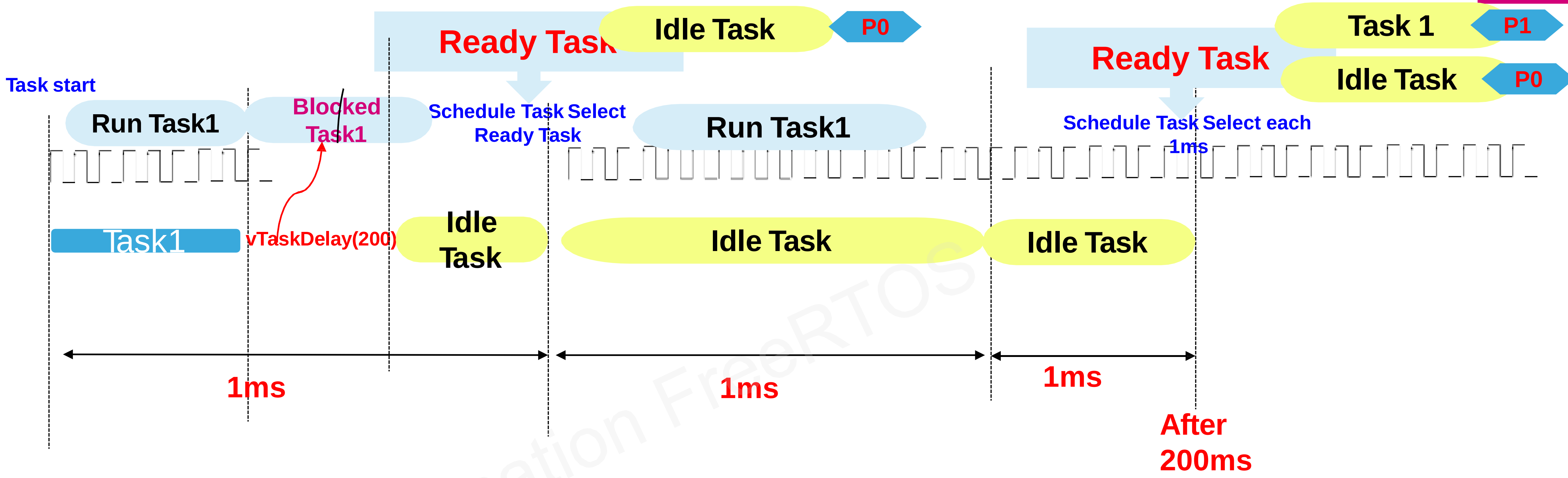
First project



Sysclk=16Mhz **16000 cycle in Run Mode**

Sysclk=160Mhz **160000 cycle in Run Mode**

First project



Sysclk=16Mhz **16000 cycle in Run Mode**

Sysclk=160Mhz **160000 cycle in Run Mode**

Task managment

Task mangement

Task State

API Tasks

Task Hook

Idle Task

Task mangement

Idle Task

- The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run.

- It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

- The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the vTaskDelete() function to ensure the idle task is not starved of processing time. The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions.

- It is possible for application tasks to share the idle task priority (tskIDLE_PRIORITY). See the configIDLE_SHOULD_YIELD configuration parameter for information on how this behaviour can be configured.

Task mangament

Idle Hook (Idle Callback)

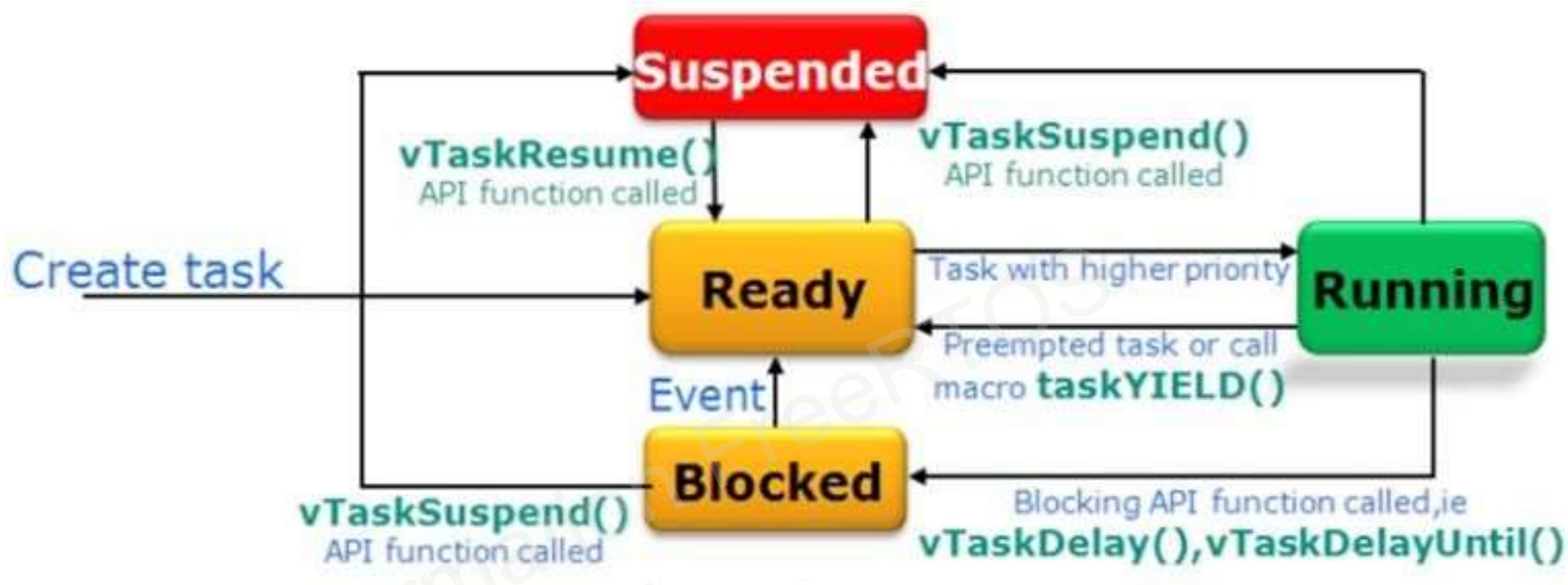
- The idle task runs at the very lowest priority, so such an idle hook function will only get executed when there are no tasks of higher priority that are able to run. This makes the idle hook function an ideal place to put the processor into a low power state

- Set configUSE_TICK_HOOK 1
  ```
  void vApplicationIdleHook( void );
  ```

- The idle hook is called repeatedly as long as the idle task is running. It is paramount that the idle hook function does not call any API functions that could cause it to block. Also, if the application makes use of the vTaskDelete() API function then the idle task hook must be allowed to periodically return (this is because the idle task is responsible for cleaning up the resources that were allocated by the RTOS kernel to the task that has been deleted).

Task managment

Task State



- **vTaskDelay()**
- **vTaskDelayUntil()**

Queues

Semaphores

Mutex

All these kernel objects support API which can block Task during operation

Task managment

Task State Blocking State

- A Task which is Temporarily or permanently chosen not to RUN on CPU

Generate delay for 10ms

```
For (int i=0;i<10000;i++);
```

CPU
Engaged

- This code runs on CPU continuously for 10ms, starving other low priority tasks
- Never use such delay implementations.

```
vTaskDelay(10);
```

CPU
Not-Engaged

- This is blocking delay API which block the Task for 10ms
- That means for the next 10 ms other lower priority tasks can RUN
- After 10ms the Task will RUN

Importance of Delay APIs (Converting Time to Ticks)

The resolution between 2 tick interrupt is 1ms
1ms takes 1 tick interrupt.
10ms will take 10 tick interrupt
Let’s assume **configTICK_RATE_HZ=250Hz** (250 tick interrupts in 1sec)
TICK_RATE_MS= 1/250 = 4ms == > that means the tick interrupt happens for every 4ms.so the resultion between two tick is 4ms

Tick= Time(ms) /TICK_RATE_MS

```
void vTaskFunction( void * pvParameters )
{
/* Block for 500ms. */
const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

     for( ;; )
     {
          /* Simply toggle the LED every 500ms, blocking between each toggle. */
          vToggleLED();
          vTaskDelay( xDelay );
     }
}
```

Task mangement

Task API

- They fall under 3 categories:

Creation
```
xTaskCreate
vTaskDelete
```

Control
```
vTaskDelay
vTaskDelayUntil
uxTaskPriorityGet
vTaskPrioritySet
vTaskSuspend
vTaskResume
xTaskResumeFromIS
R
```

Utilities
```
tskIDLE_PRIORITY
xTaskGetTickCount
uxTaskGetNumberOfTasks
vTaskList
vTaskGetRunTimeStats
vTaskStartTrace
ulTaskEndTrace
uxTaskGetStackHighWaterMark
vTaskSetApplicationTaskTag
xTaskGetApplicationTaskTag
xTaskCallApplicationTaskHook
```

Task managment

Task Implementation Function(Task Function)

Pass data to task function-Pass pointer of the data

```
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 managment

API to create and schedule task

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

Create TCB (Task control block)
Create associated stack space for it

usStackDepth : The number of words (not bytes!) to allocate for use as the task's stack. example, if the stack is 32-bits wide and usStackDepth is 400 then 1600 bytes will be allocated for use as the task's stack.
pvParameters : A value that will passed into the created task as the task's parameter. If pvParameters is set to the address of a variable then the variable must still exist when the created task executes – so it is not valid to pass the address of a stack variable.
pxCreatedTask :Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL
Returns: If the task was created successfully then pdPASS is returned. Otherwise errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY is returned

Task mangement

Two tasks share the same task function

```c
/* Define the strings that will be passed in as the task parameters.  These are
defined const and off the stack to ensure they remain valid when the tasks are
executing. */
const char *pcTextForTask1 = "Task 1 is running\n";
const char *pcTextForTask2 = "Task 2 is running\n";
```

```c
/* Create one of the two tasks. */
xTaskCreate(    vTaskFunction,          /* Pointer to the function that implements the task. */
                "Task 1",               /* Text name for the task.  This is to facilitate debugging only. */
                240,                    /* Stack depth in words. */
                (void*)pcTextForTask1,  /* Pass the text to be printed in as the task parameter. */
                1,                      /* This task will run at priority 1. */
                NULL );                 /* We are not using the task handle. */

/* Create the other task in exactly the same way.  Note this time that we
are creating the SAME task, but passing in a different parameter.  We are
creating two instances of a single task implementation. */
xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 1, NULL );

/* Start the scheduler so our tasks start executing. */
vTaskStartScheduler();
```

Task mangement

Two tasks share the same task function

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile unsigned long ul;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        printf( "%s\n" ,pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Task managment

- Task API

portSWITCH_TO_USER_MODE()
vTaskAllocateMPURegions()
xTaskAbortDelay()
xTaskCallApplicationTaskHook()
xTaskCheckForTimeOut()
xTaskCreate()
xTaskCreateStatic()
xTaskCreateRestricted()
vTaskDelay()
vTaskDelayUntil()
vTaskDelete()

taskDISABLE_INTERRUPTS()
taskENABLE_INTERRUPTS()
taskENTER_CRITICAL()
taskENTER_CRITICAL_FROM_ISR()
taskEXIT_CRITICAL()
taskEXIT_CRITICAL_FROM_ISR()
xTaskGetApplicationTaskTag()
xTaskGetCurrentTaskHandle()
xTaskGetIdleTaskHandle()
xTaskGetHandle()
uxTaskGetNumberOfTasks()

vTaskGetRunTimeStats()
xTaskGetSchedulerState()
uxTaskGetStackHighWaterMark()
eTaskGetState()
uxTaskGetSystemState()
vTaskGetTaskInfo()
pvTaskGetThreadLocalStoragePointer()

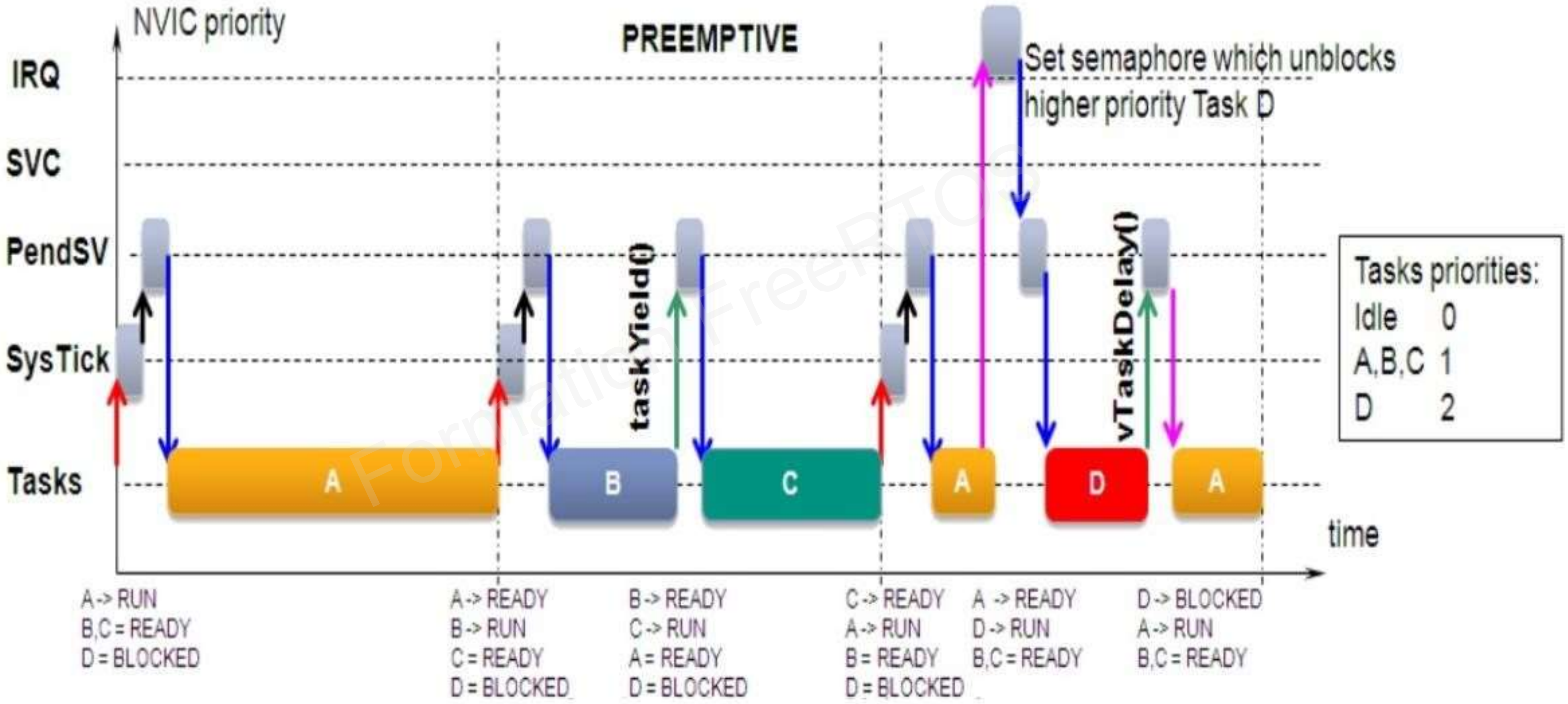
pcTaskGetName()
xTaskGetTickCount()
xTaskGetTickCountFromISR()
vTaskList()
xTaskNotify()
xTaskNotifyAndQuery()
xTaskNotifyAndQueryFromISR()
xTaskNotifyFromISR()
xTaskNotifyGive()
vTaskNotifyGiveFromISR()

xTaskNotifyStateClear()
ulTaskNotifyTake()
xTaskNotifyWait()
uxTaskPriorityGet()
vTaskPrioritySet()
vTaskResume()
xTaskResumeAll()

xTaskResumeFromISR()
vTaskSetApplicationTaskTag()
vTaskSetThreadLocalStoragePointer()
vTaskSetTimeOutState()
vTaskStartScheduler()
vTaskStepTick()
vTaskSuspend()
vTaskSuspendAll()
taskYIELD()

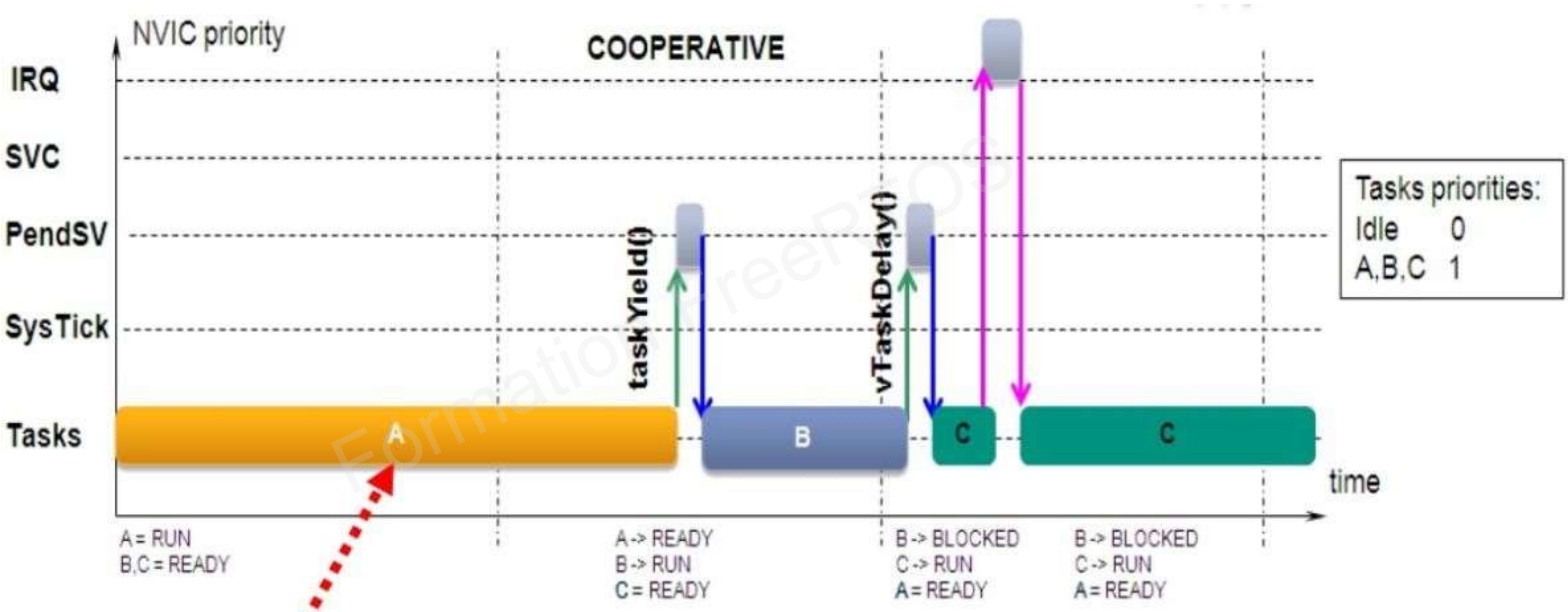
Task managment

FreeRTOS Preemptif scheduler



Task managment

FreeRTOS Cooperatif Scheduler



TaskA Will Leave the CPU only if it enters the Blocking state or by calling taskYIELD()

FreeRTOSconfig.h


```
#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configSTACK_DEPTH_TYPE                  uint16_t
#define configMESSAGE_BUFFER_LENGTH_TYPE        size_t

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0
```

configUSE_PREEMPTION

Set to 1 to use the preemptive RTOS scheduler, or 0 to use the cooperative RTOS scheduler.

configMINIMAL_STACK_SIZE

The size of the stack used by the idle task. Generally this should not be reduced from the value set in the FreeRTOSConfig.h file provided with the demo application for the port you are using.Like the stack size parameter to the xTaskCreate() and xTaskCreateStatic() functions, the stack size is specified in words, not bytes

FreeRTOSconfig.h


```
#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configSTACK_DEPTH_TYPE                  uint16_t
#define configMESSAGE_BUFFER_LENGTH_TYPE        size_t

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0
```

configUSE_TIME_SLICING

By default (if configUSE_TIME_SLICING is not defined, or if configUSE_TIME_SLICING is defined as 1) FreeRTOS uses prioritised preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE_TIME_SLICING is set to 0 then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt has occurred

FreeRTOSconfig.h

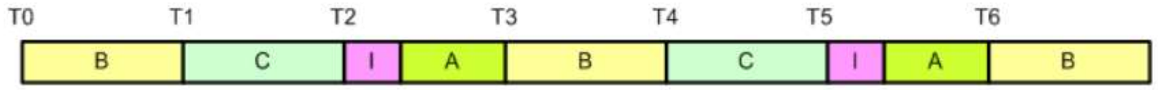
```
#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configSTACK_DEPTH_TYPE                  uint16_t
#define configMESSAGE_BUFFER_LENGTH_TYPE        size_t

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK       0
```

configIDLE_SHOULD_YIELD

This parameter controls the behaviour of tasks at the idle priority. It only has an effect if:

- The preemptive scheduler is being used.
- The application creates tasks that run at the idle priority.

When tasks share the idle priority the behaviour can be slightly different. If configIDLE_SHOULD_YIELD is set to 1 then the idle task will yield immediately if any other task at the idle priority is ready to run. This ensures the minimum amount of time is spent in the idle task when application tasks are available for scheduling. This behaviour can however have undesirable effects (depending on the needs of your application) as depicted below:



four tasks that are all running at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task