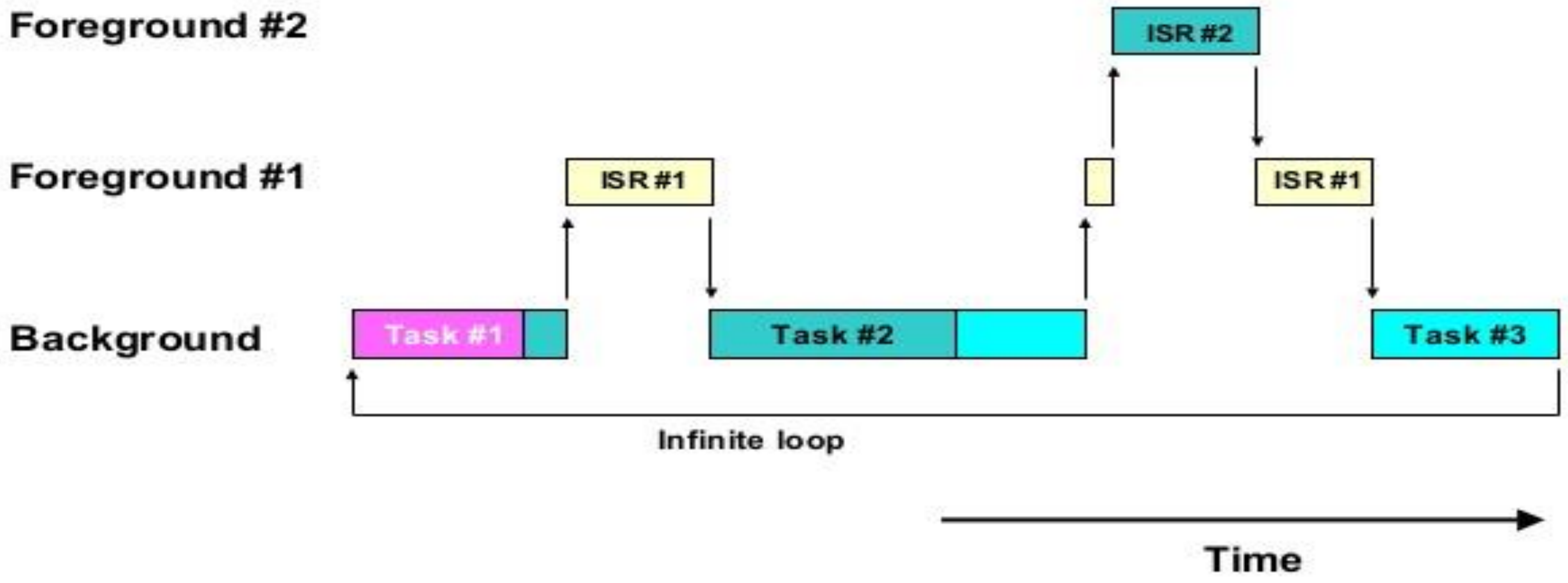
Foreground / Background Systems

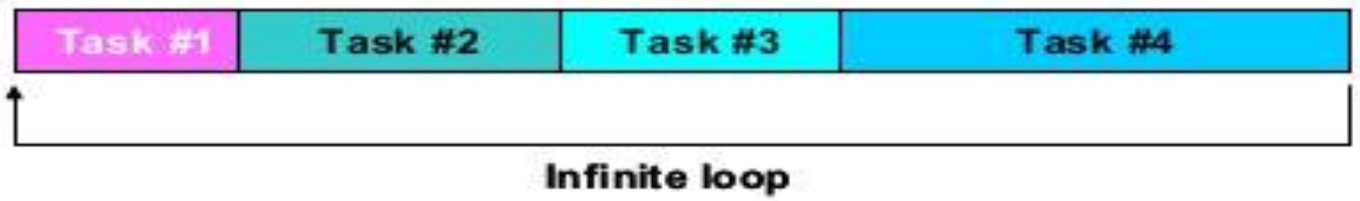
- Small Systems of low complexity.
- Application consists of infinite loop that calls modules to perform desired operations.(Background) – Task Level
- ISR's handle asynchronous events (Foreground) – Interrupt Level
- Critical sections performed by ISR to ensure that they are dealt in timely fashion.

Products without Kernels
Foreground/Background Systems



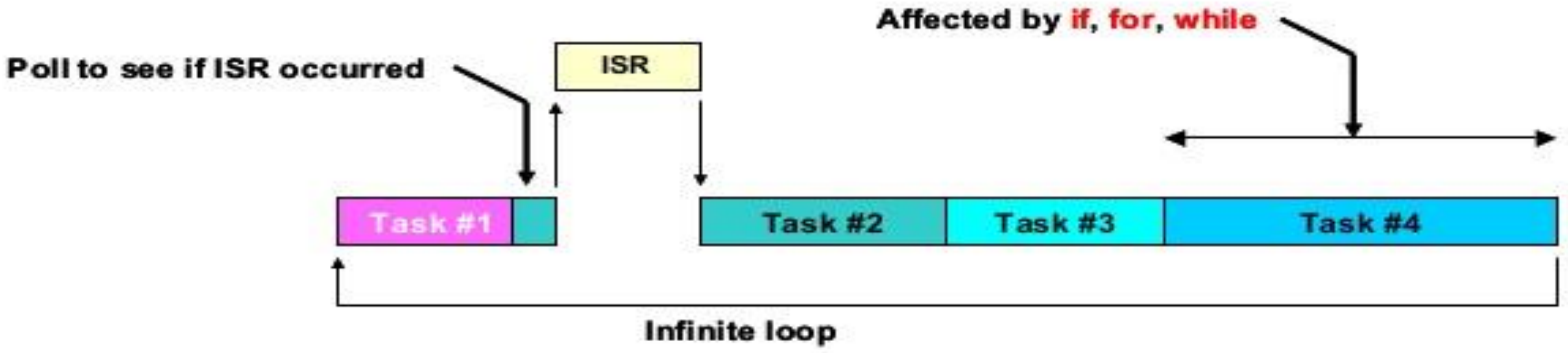
Foreground?/Background Tasks
Disadvantages - Priority

- All ‘tasks’/functions() have the same priority
  - Code executes in sequence
  - If an important event occurs, it’s handled at the same priority as everything else
  - May need to execute the same code often to avoid missing an event



Foreground/Background
Disadvantages – Response Time

- Background response time is the background execution time
  - Non-deterministic
    - Affected by **if**, **for**, **while** ...
  - May not be responsive enough
  - Changes as you change your code



Real Time System

- A system is said to be **Real Time** if it is required to complete it’s work & deliver it’s services on time.

- Example – Flight Control System
  - All tasks in that system must execute on time.

- Non Example – PC system

Hard and Soft Real Time Systems

- Hard Real Time System
  - Failure to meet deadlines is fatal
  - example : Flight Control System

- Soft Real Time System
  - Late completion of jobs is undesirable but not fatal.
  - System performance degrades as more & more jobs miss deadlines
  - Online Databases

Hard and Soft Real Time Systems
(Operational Definition)

- Hard Real Time System
  - Validation by provably correct procedures or extensive simulation that the system always meets the timings constraints

- Soft Real Time System
  - Demonstration of jobs meeting some statistical constraints suffices.

Role of an OS in Real Time Systems

- Standalone Applications
  - Often no OS involved
  - Micro controller based Embedded Systems
- Some Real Time Applications are huge & complex
  - Multiple threads
  - Complicated Synchronization Requirements
  - Filesystem / Network / Windowing support
  - OS primitives reduce the software design time

Real-time and embedded systems operate in constrained environments in which computer memory and processing power are limited.

They often need to provide their services within strict time deadlines to their users and to the surrounding world.

It is these memory, speed and timing constraints that dictate the use of real-time operating systems in embedded software.

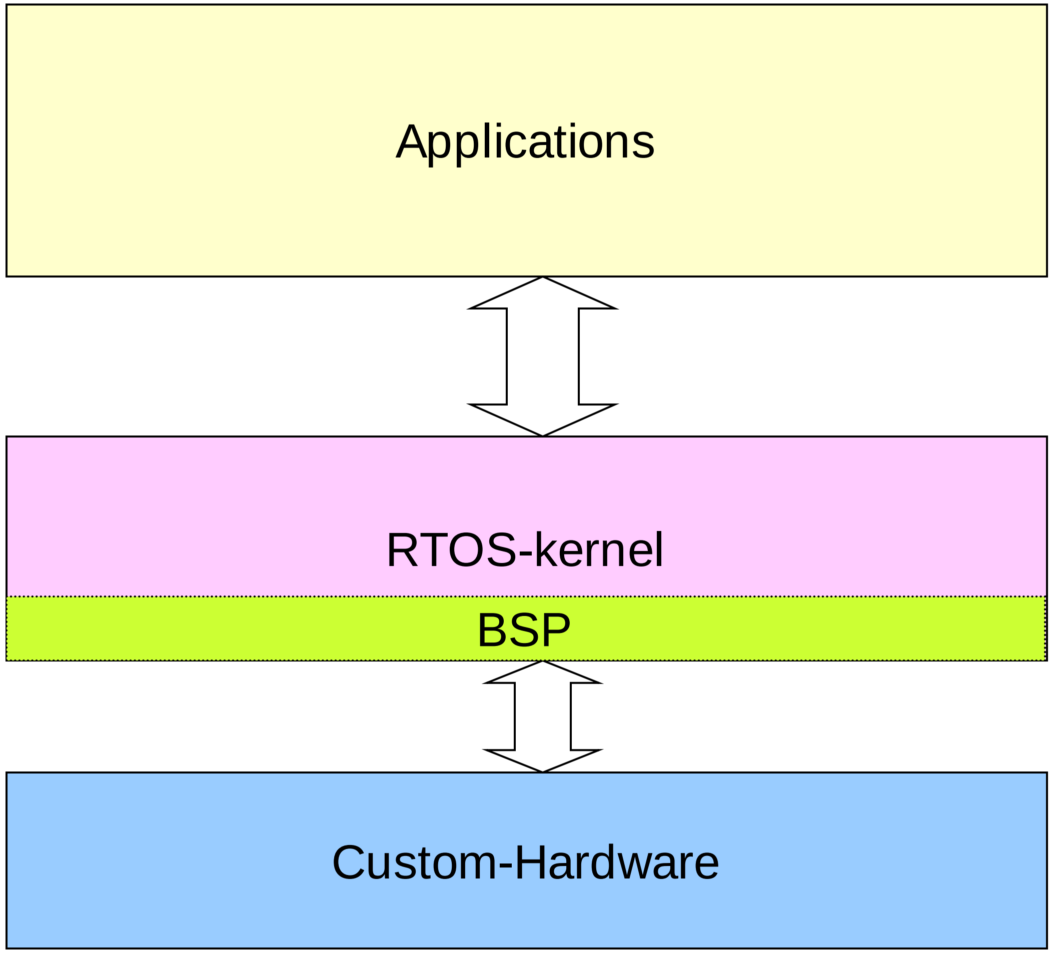
RTOS

A variant of OS that operates in constrained environment in which computer memory and processing power is limited.

Moreover they often need to provide their services in definite amount of time.

Example RTOS: VxWorks, pSOS, Nucleus, RTLinux…

Structure of a RTOS



"kernel" ?

The part of an operating system that provides the most basic services to application software running on a processor.

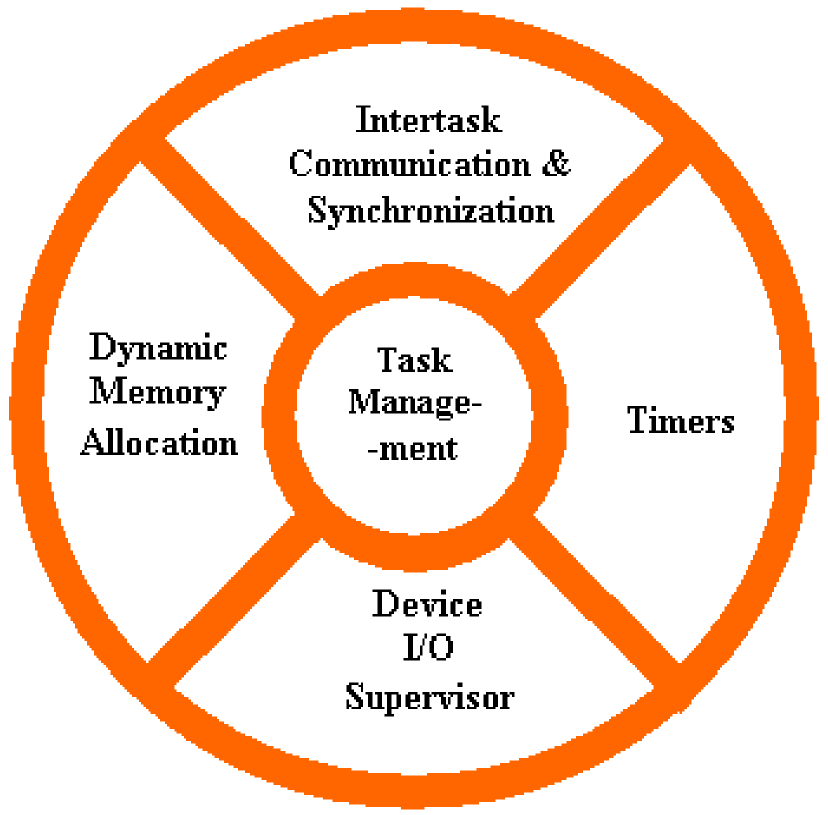
The "kernel" of a real-time operating system ("RTOS") provides an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run.

Components of RTOS

- The most important component of RTOS is its kernel (Monolithic & Microkernel).

- BSP or Board Support Package makes an RTOS target-specific (It’s a processor specific code onto (processor) which we like to have our RTOS running).

RTOS KERNEL


Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Manage- -ment
Timers
Device I/O Supervisor

Task Management:

This set of services allows application software developers to design their software as a number of separate "chunks" of software

-- each handling a distinct topic, a distinct goal, and perhaps its own real-time deadline.

Each separate "chunk" of software is called a "task.“

Services in this category include the ability to launch tasks and assign priorities to them.

The main RTOS service in this category is the scheduling of tasks as the embedded system is in operation.

The Task Scheduler controls the execution of application software tasks, and can make them run in a very timely and responsive fashion.

Intertask Communication and Synchronization:

These services make it possible for tasks to pass information from one to another, without danger of that information ever being damaged.

They also make it possible for tasks to coordinate, so that they can productively cooperate with one another.

Without the help of these RTOS services, tasks might well communicate corrupted information or otherwise interfere with each other.

Timers:

Many embedded systems have stringent timing requirements

Most RTOS kernels also provide some basic Timer services,

such as task delays and time-outs.

Dynamic Memory Allocation:

Many (but not all) RTOS kernels provide Dynamic Memory Allocation services.

This category of services allows tasks to "borrow" chunks of RAM memory for temporary use in application software.

Often these chunks of memory are then passed from task to task, as a means of quickly communicating large amounts of data between tasks.

Some very small RTOS kernels that are intended for tightly memory-limited environments, do not offer Dynamic Memory Allocation services.

Device I/O Supervisor:

Many (but not all) RTOS kernels also provide a "Device I/O Supervisor" category of services.

These services, if available, provide a uniform framework for organizing and accessing the many hardware device drivers that are typical of an embedded system.

In addition to kernel services, many RTOSs offer a number of optional add-on operating system components for such high-level services as

file system organization,

network communication,

network management,

database management,

user-interface graphics, etc.

Each of these add-on components is included in an embedded system only if its services are needed for implementing the embedded application, in order to keep program memory consumption to a minimum.

RTOSs vs. general-purpose operating systems

Real-time operating systems is the need for " deterministic " timing behaviour in the real-time operating systems.

Formally, "deterministic" timing means that operating system services consume only known and expected amounts of time.

General-computing non-real-time operating systems are often quite non-deterministic.

Their services can inject random delays into application software and thus cause slow responsiveness of an application at unexpected times.

Task scheduling

Most RTOSs do their scheduling of tasks using a scheme called "priority-based preemptive scheduling.“

Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness.

Very quick responsiveness is made possible by the "preemptive" nature of the task scheduling.

"Preemptive" means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately.

The basic rule that governs priority-based preemptive scheduling is that at every moment in time, "The Highest Priority Task that is Ready to Run, will be the Task that Must be Running.“

In other words, if both a low-priority task and a higher-priority task are ready to run, the scheduler will allow the higher-priority task to run first.

The low-priority task will only get to run after the higher-priority task has finished with its current work.

Each time the priority-based preemptive scheduler is alerted by an external world trigger (such as a switch closing) or a software trigger (such as a message arrival),

it must go through the following 5 steps:

- Determine whether the currently running task should continue to run.
- If not ? Determine which task should run next.
- Save the environment of the task that was stopped (so it can continue later).
- Set up the running environment of the task that will run next.
- Allow this task to run.

These 5 steps together are called "task switching."

Intertask communication and synchronization

RTOSs, offer a variety of mechanisms for communication and synchronization between tasks.

These mechanisms are necessary in a pre-emptive environment of many tasks.

Without them the tasks might well communicate corrupted information or otherwise interfere with each other.

The most popular kind of communication between tasks in embedded systems is the passing of data from one task to another.

Most RTOSs offer a message passing mechanism

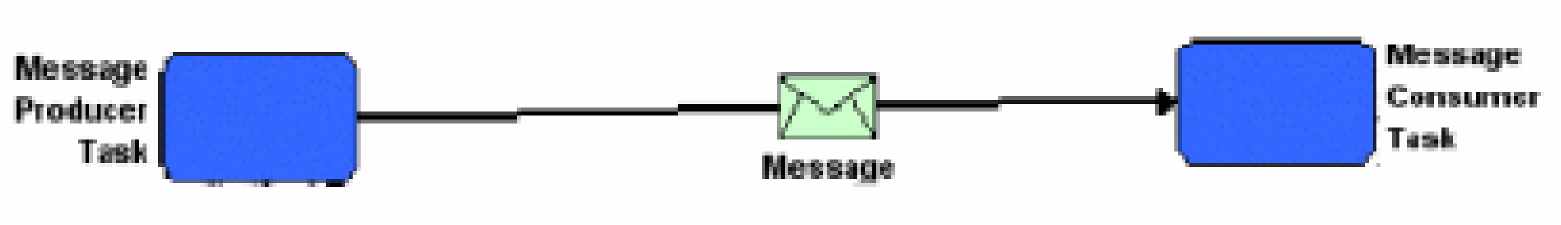


Figure 5: Intertask Message Communication

- Most RTOSs offer a semaphore or mutex mechanism for handling negative synchronization (sometimes called "mutual exclusion").

- These mechanisms allow tasks to lock certain embedded system resources for their use only, and subsequently to unlock the resource when they're done.

- For positive synchronization, different RTOSs offer different mechanisms. Some RTOSs offer event-flags, while others offer signals.

Determinism and high-speed message passing

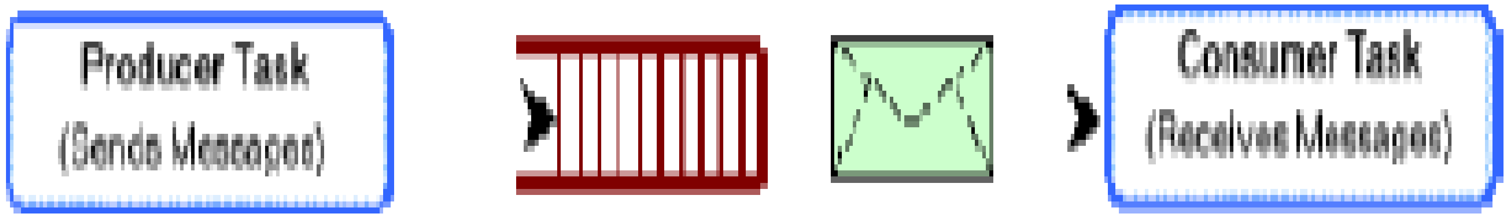


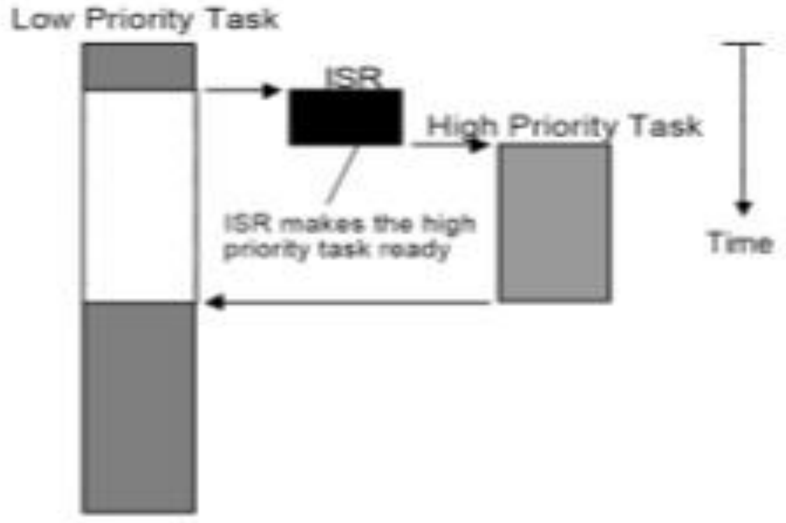
Figure 6: Message Transfer via Message Queue

What is an RTOS?

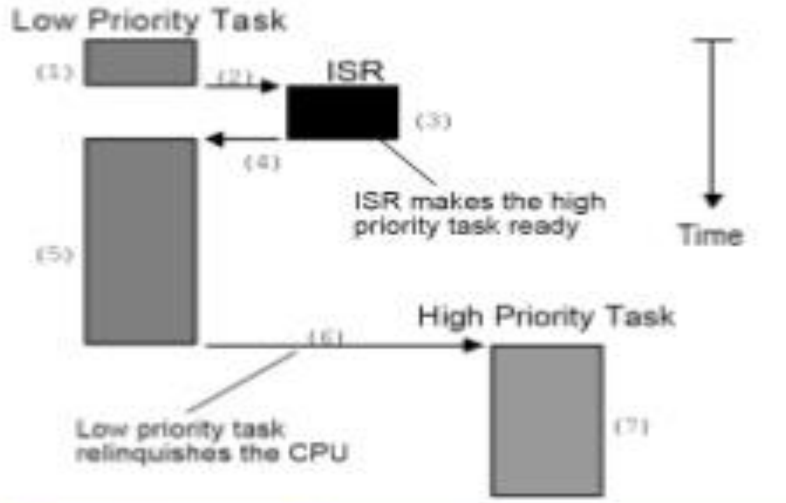
- **R**eal-**T**ime **O**perating **S**ystem
  - Software that manages the time of a microprocessor, microcontroller, or a digital signal processor
  - Prioritizes the work to be done
  - Provides ‘multitasking’
  - Provides ‘services’ to the application
    - Semaphores
    - Message mailboxes and queues
    - Event flags
    - Time delays, timers, and timeouts
    - Task management
    - Memory management
    - Bandwidth assessment ‘idle time’

Types of RTOSs

1. Pre-emptive:
   Always runs the highest available task. Tasks of identical priority share CPU time (fully pre-emptive with round robin time slicing)



2 . Cooperative:
   Context switches only occur if a task blocks, or explicitly relinquishes CPU control


Low Priority Task
(1)
(2)
ISR
(3)
(4)
ISR makes the high priority task ready
Time
(5)
High Priority Task
(6)
(7)
Low priority task relinquishes the CPU

RTOS Advantages

- Software that manages the time of a microprocessor or microcontroller
  - Ensures that the most important code runs first
- Allows Multitasking
  - Do more than one thing at the same time
  - Application is broken down into multiple tasks, each handling one aspect of your application
  - It's like having multiple CPUs
- Provides valuable services to your application
  - Time delays
  - Resource sharing
  - Inter-task communication and synchronization

Available RTOSs for LPC2300/2400

A few of those available

- Keil RTX Real-Time Kernel
  - http://www.keil.com/arm/rl-arm/kernel.asp
- Micrium µC/OS-II
  - http://www.micrium.com/products/rtos/kernel/rtos.html
- CMX-RTX
  - http://www.cmx.com/rtx.htm
- Express Logic ThreadX
  - http://www.rtos.com/
- Segger ebmOS
  - http://www.segger.com/embos_general.html
- uClinux
  - As distributed with the EA LPC2468 board
- FreeRTOS.org
  - http://www.freertos.org/

Why NXP chose Micrium µC/OS-II

For the LPC2468 Industrial Reference Design (IRD) Platform

- Written in ANSI C
  - Source code provided
  - Consistent coding style
- Pre-emptive
- Deterministic
- Royalty-free
  - Licensed on a per-end-product basis
- Extensive Documentation and Support
- Meets requirements of Safety-Critical Systems
- Large user base
  - Used in hundreds of products all over the world

R
T
O
S

INTRODUCTION

❖**MicroC/OS-II** (commonly termed as **µC/OS-II** or **uC/OS-II**), is the acronym for Micro-Controller Operating Systems Version 2.

❖It is a priority-based real-time multitasking operating system kernel for microprocessors, written mainly in the C programming language.

❖It is intended for use in embedded systems.

R
T
O
S

Features of MicroC/OS-II :

- It is a very small real-time kernel.
- Memory footprint is about 20KB for a fully functional kernel.
- Source code is written mostly in ANSI C.
- Highly portable, ROMable, very scalable, preemptive real-time, deterministic, multitasking kernel.
- It can manage up to 64 tasks (56 user tasks available).
- It has connectivity with μC/GUI and μC/FS (GUI and File Systems for μC/OS II).

R
T
O
S

Cont…

- ➢It is ported to more than 100 microprocessors and microcontrollers.
- ➢It is simple to use and simple to implement but very effective compared to the price/performance ratio.
- ➢It supports all type of processors from 8-bit to 64-bit.

Portable:

- µc/os-ii written in highly portable ANSI C.
- And Target specific code in ALP.
- ALP is kept minimum.
- Can be ported to large number of microprocessors.
- Runs on 8, 16,32, 64 bit µp, µc and DSP’s.

ROMable

- µc/os-ii designed for embedded applications.

- With proper tool chain (C compiler, Assembler, and linker), we can embed µc/os-ii as part of product.

Scalable

- Can use only the services needed by the application.
- Allows to reduce the amount of memory.
- Is accomplished with the use of conditional compilation.

Preemptive

- µc/os-ii is fully preemptive real time kernel.

- Always runs the highest priority task that is ready.

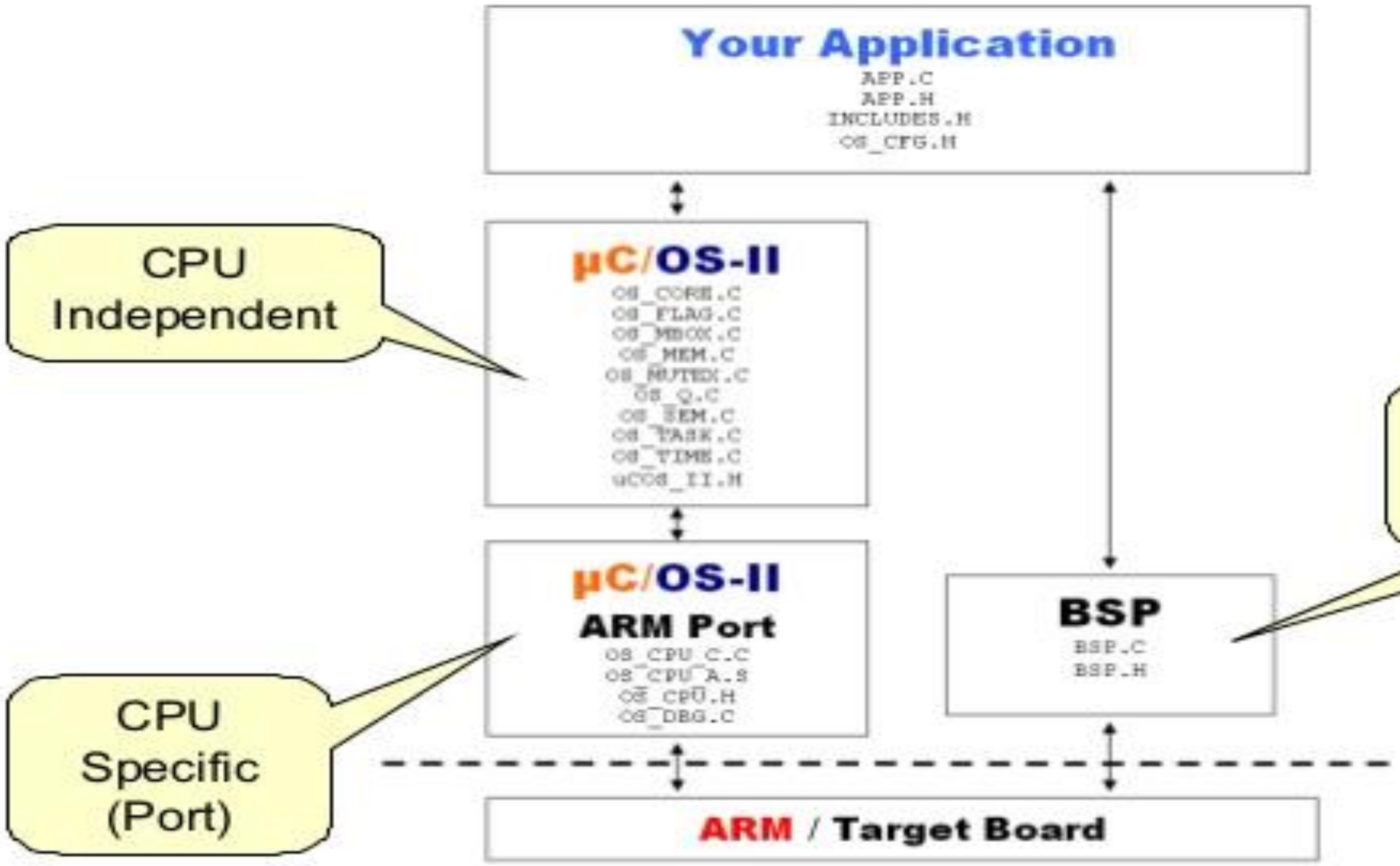
Multitasking, Deterministic, Task Stacks, Services, Interrupt Management, Robust and reliable.

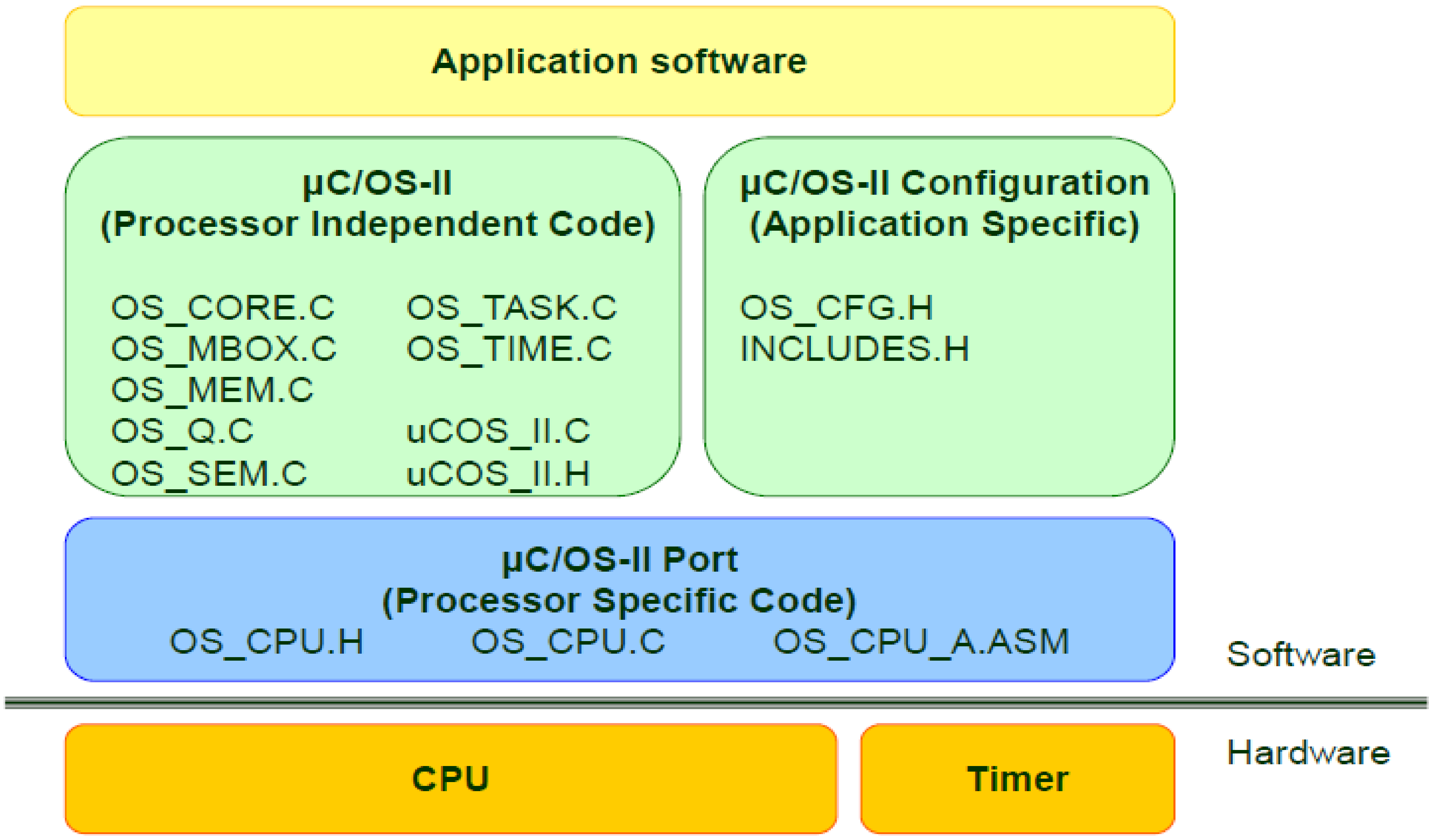
- Manage upto 64 tasks , that is 64 priority levels.
- Can know execution time of function or a service.
- Allows each task to have different stack size.
- Interrupts can be nested up to 255 levels.
- Certified by FAA . OS tested and examined extensively and can be used in safety critical systems.

Additional modules

µC/OS-II *The Real-Time Kernel*	Real-Time Operating System (RTOS)
µC/TCP-IP *Protocol Stack*	TCP-IP protocol stack
µC/USB Host *Universal Serial Bus Host Stack*	Host USB stack
µC/USB Device *Universal Serial Bus Device Stack*	Device USB stack
µC/CAN *CAN Protocol Stack*	CAN protocol stack
µC/Probe *Real-Time Monitoring*	Runtime data monitor
µC/BuildingBlocks *Embedded Software Components*	µC/Shell and µC/LCD driver
µC/FS *Embedded File System*	Microsoft compatible FAT File System

Micrium μC/OS-II and Related Files




Application software
µC/OS-II
(Processor Independent Code)
OS_CORE.C
OS_MBOX.C
OS_MEM.C
OS_Q.C
OS_SEM.C
OS_TASK.C
OS_TIME.C
uCOS_II.C
uCOS_II.H
µC/OS-II Configuration
(Application Specific)
OS_CFG.H
INCLUDES.H
µC/OS-II Port
(Processor Specific Code)
OS_CPU.H
OS_CPU.C
OS_CPU_A.ASM
Software
Hardware
CPU
Timer

RTOS Tasks

- A task is a simple program that thinks it has the CPU all to itself
- Each Task has
  - Its own stack space
  - A priority based on its importance
- A task contains YOUR application code

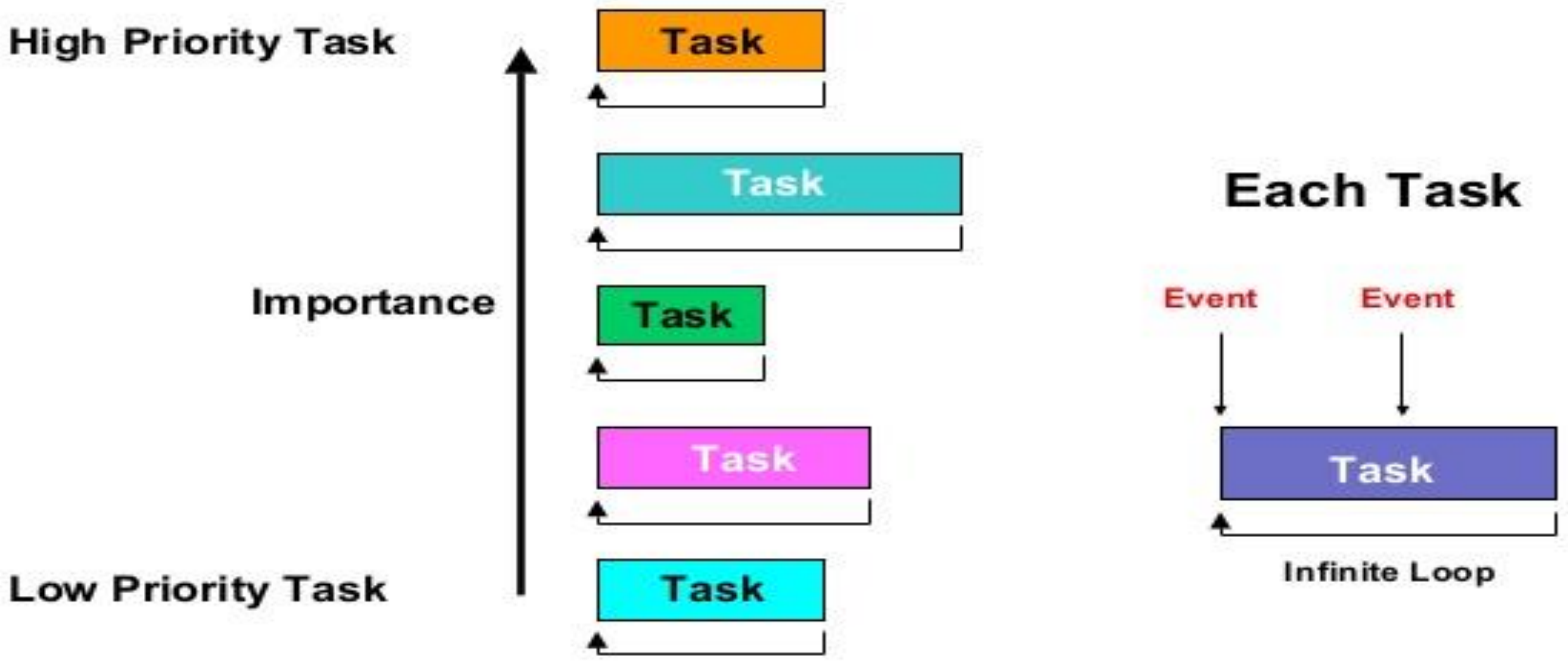
What is a Task?

- A task is an infinite loop

```
void Task (void *p_arg)
{
   Do something with ‘argument’ p_arg;
   Task initialization;
   for (;;) {
      /* Processing (Your Code)                              */
      Wait for event;              /* Time to expire ...     */
                                   /* Signal from ISR ...    */
                                   /* Signal from task ...   */
      /* Processing (Your Code)                              */
   }
}
```

Designing with µC/OS-II

Splitting an application into Tasks



R T O S

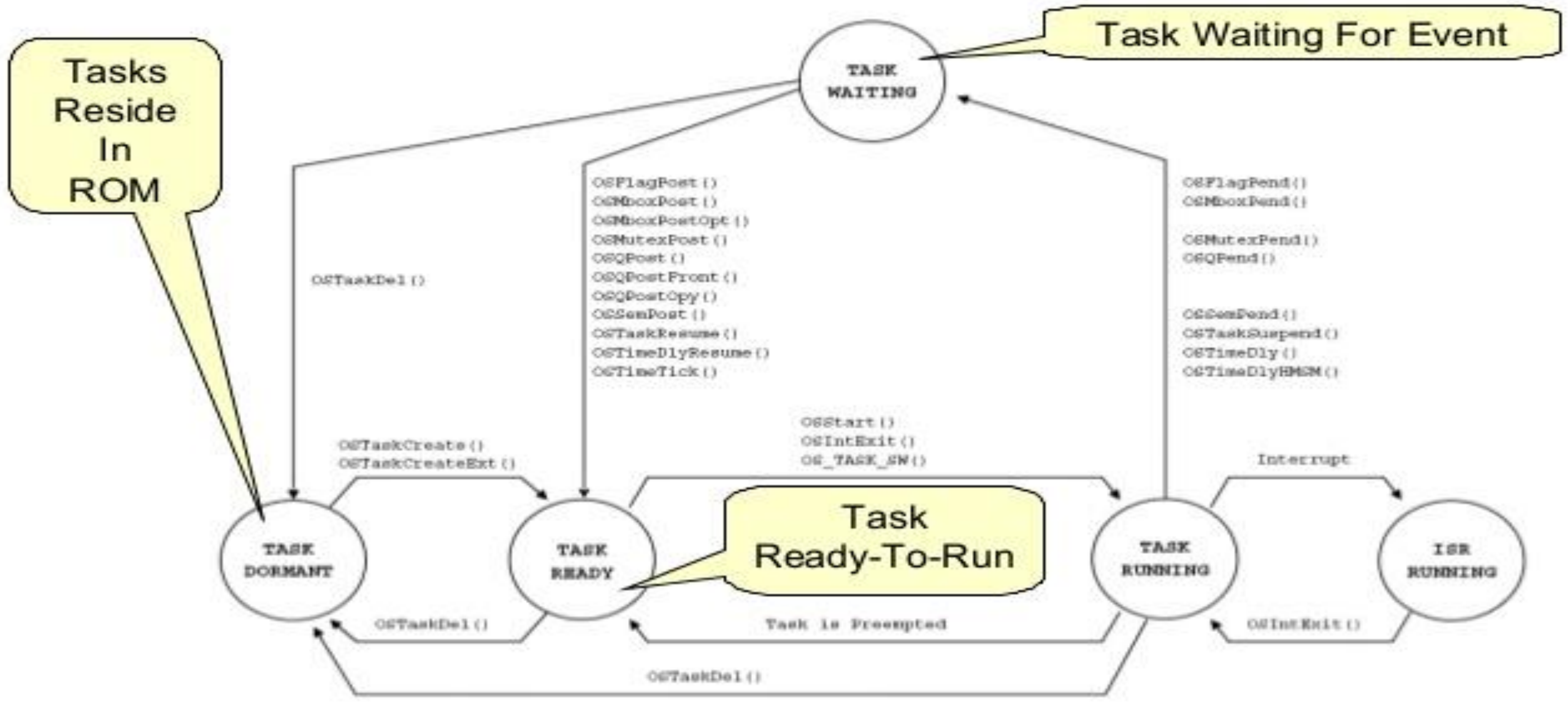
Task states:

μC/OS-II is a multitasking operating system. Each task is an infinite loop and can be in any one of the following five states:

- ❖ Dormant
- ❖ Ready
- ❖ Running
- ❖ Waiting
- ❖ ISR (Interrupt Service Routine)

- Dormant State: Task that resides in the memory but has not been made available to the multitasking kernel.
- Ready: Task is ready when it can execute but its priority is less than the currently running task.
- Running: Task is running when it has control of CPU
- Waiting: A task is waiting when it requires occurrence of an event.(Wait for I/O operation to complete, a shared resource available, a timing pulse to occur, or time to expire)
- ISR state: Task is in ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt.

µC/OS-II Task States



Architecture of the Kernel

- Embedded software consists of operating system and application software.
- The services provided by the operating system are accessed through Application programming Interface (API) to develop application software.
- API are function calls used to access kernel objects and services of kernel.

Kernel Objects

- Tasks
- Task Scheduler
- Interrupt service routine
- Semaphores
- Mutexes
- Mailboxes
- Message queues
- Pipes
- Event Registers
- Signals
- Timers

- Kernel provides services through operations on the kernel objects.

Services are Time management, Memory management, Interrupt handling.

Tasks and Task Scheduler

- Embedded software consists of a number of tasks.
- These tasks include OS tasks and Application specific tasks.
- Tasks in embedded system implemented as infinite loop.
- The task object consists of its
  - name,
  - a unique ID,
  - a priority,
  - a stack and
  - A Task Control Block – that contains information related to task.

Tasks and Task Scheduler-cont..

Kernel has system tasks with priorities. These tasks are

- Start-up task – executed when OS starts
- Exception Handling task – Handle the exceptions
- Logging Task – to log the various system messages
- Idle task – Has lowest priority , Runs when no other tasks runs, ensures that CPU is not idle.

Tasks and Task Scheduler-cont..

Single CPU handles multiple tasks

Tasks have to share CPU time in a disciplined manner.

Each task is assigned a priority.

Task Scheduling – Mechanism that decides the task to be run next.

Object that perform task scheduling – Task Scheduler.

- Tasks Share CPU Time and Resources such as registers, external memory, input / output devices.
- A Task should not corrupt data of another task.
- Reentrant Function – Used by more than one task without data corruption.(Local Variable)
- Non Reentrant Function – Data is corrupted when more than one tasks calls the function. (Global Variable)

Task Synchronization

- Critical Section of code – code should not be interrupted while execution.

- Solution: Disable the interrupts before execution of Critical section.

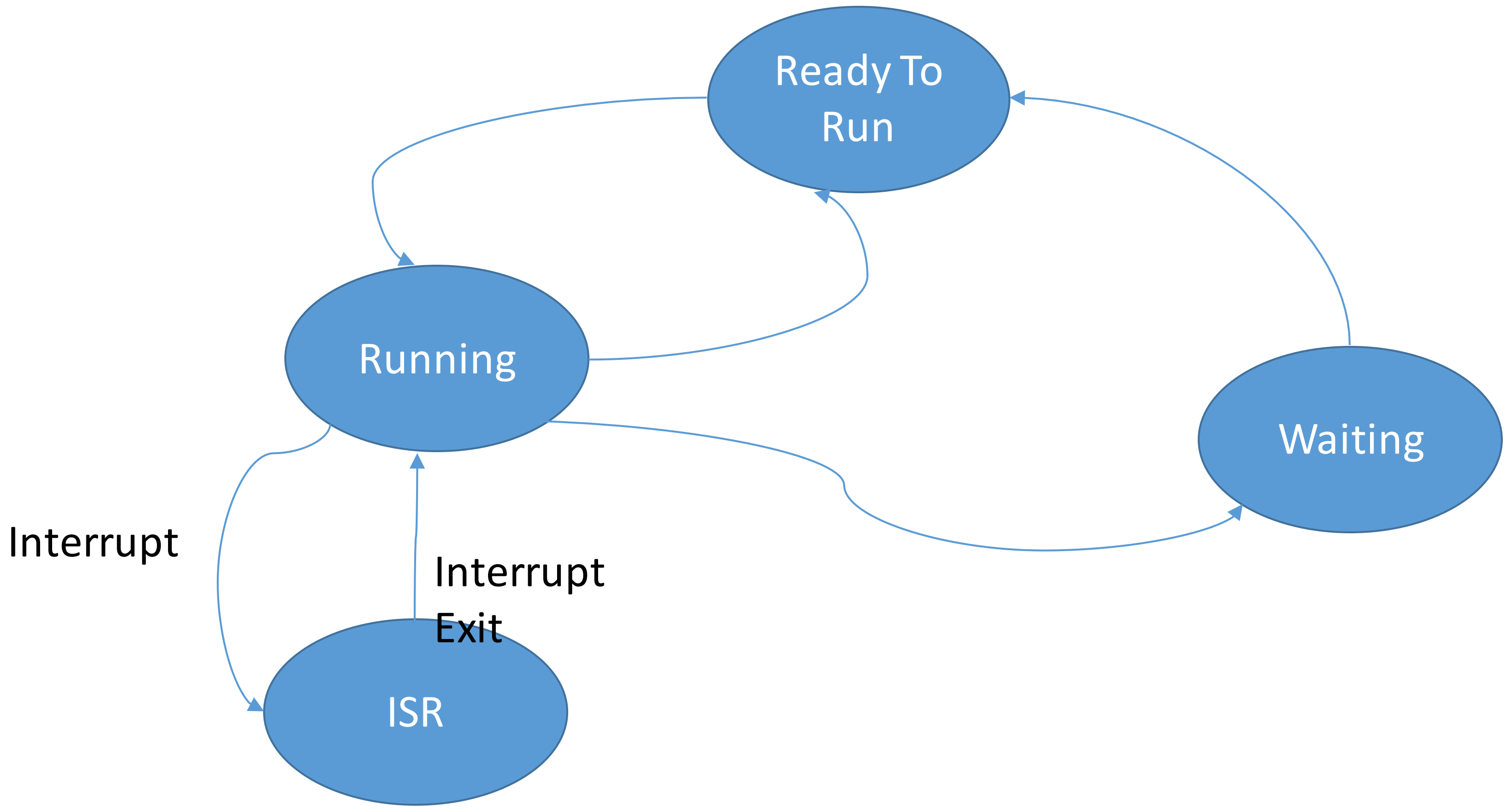
- Kernel Object Semaphores are used to protect the data.

- Shared Resources – Resources shared by many tasks.
- Example – Serial Port, Keyboard, Display or Memory Locations.
- Tasks should maintain discipline to access shared resources.
- Semaphores and Mutexes are used to share resources with discipline.

Inter Task Communication

- Tasks may need to communicate data amongst themselves.
- Example: A Task writes data into an array and Another task has to read data from the array.
- Achieved through Mailboxes, Message Queues, pipes, Event Registers and Signals.

Task States



- Example: Embedded System that obtains data from serial port and converts the data into Ethernet packets.

- Task1 – Reads data from serial port
- Task2 – Converts serial data into packets and puts in buffer.
- Task3 – Sends to Ethernet Interface

- Task2 has to wait for Task1 to complete and then informs to Task3

- At any instant one task is executed by CPU
- Other tasks waits for external event or waits for CPU Time.

- Task can be in one of the state
- Running – Task is being executed by the CPU.

- Waiting – Task waiting for another event to occur.

- Ready to Run – Task waiting in a Queue for CPU time

Task Stack

- Each Task will have stack.
- Stores Local Variables, Function Parameters, Return Addresses and CPU Registers during an interrupt.
- At time of task creation, size of stack has to be specified or a default stack size has to be used.

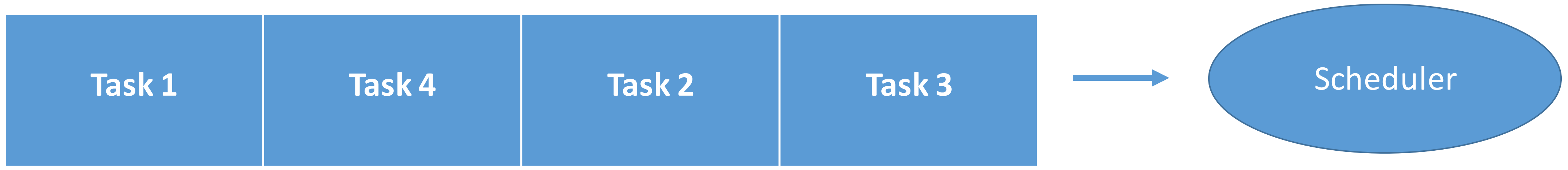
Context Switching

- Context - State of CPU registers when task to be pre – empted.
- Context Switching – Saving the contents of CPU registers and loading new task.

Scheduling Algorithms

- How does a kernel decide which task to run?
- Solution – Scheduling Algorithms.
- First In First Out
- Round Robin Algorithm
- Round Robin with priority
- Shortest Job First
- Non Pre- emptive Multitasking
- Pre- emptive Multitasking

First In First Out

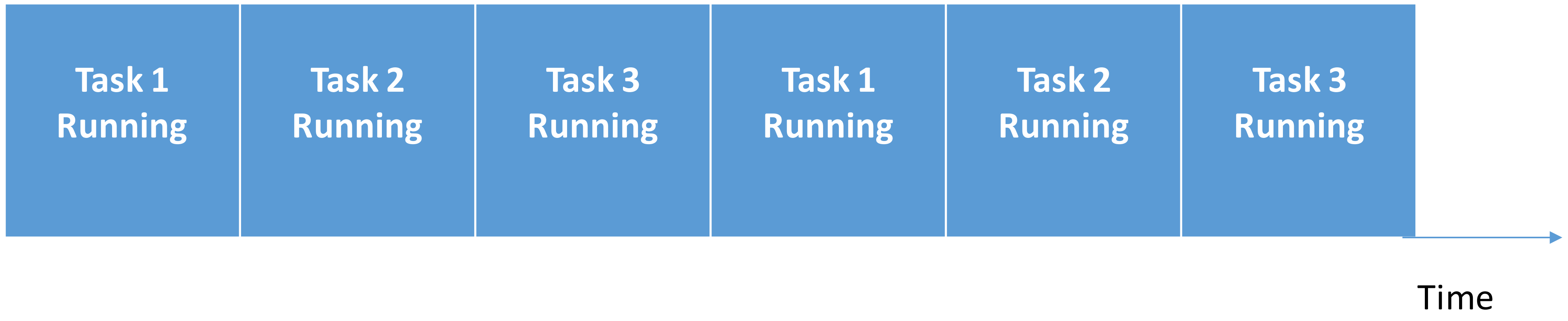


<u>Executed Tasks</u>

Task 3
Task 2
Task 4
Task 1

- Tasks which are Ready to Run are kept in Queue.
- CPU serves the tasks on first-come-first served basis.
- Simple to implement.
- Used for Embedded Applications with small tasks and small execution times.
- Disadvantage – Difficult to estimate the amount of time a task has to wait for being executed.

Round Robin Algorithm



Kernel allocates certain amount of time for each task waiting in the queue.

Time slice allocated is called Quantum.

The Kernel gives control to next task if

a) The current task has completed its work within the time slice.
b) The current task has no work to do.
c) The current task has completed its allocated time slice.

Simple to Implement.

Disadvantage: Cannot be Used for time critical applications.

Round Robin with priority

- Round Robin Method with priority assigned to some or all the tasks.
- A high priority task can interrupt the CPU and can be executed.
- High priority task meets the desired response time.

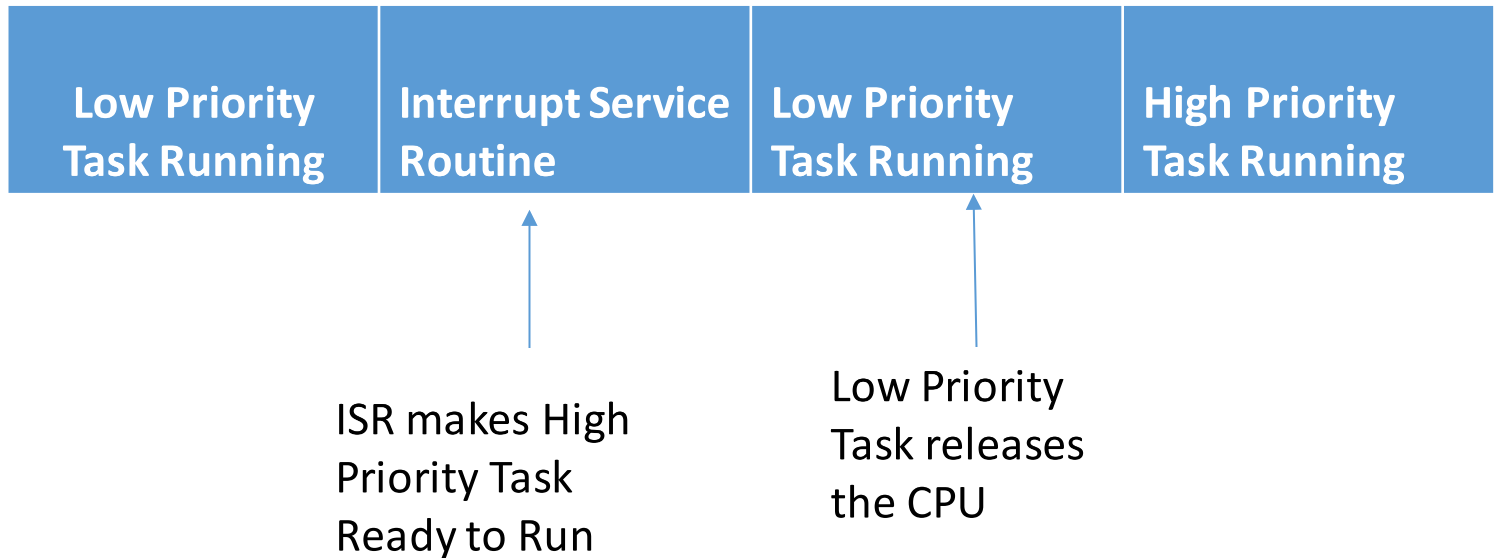
Shortest Job First

- Time for each task can be estimated beforehand.
- CPU executes the task which takes least amount of time.
- Priority is based on execution time.
- Higher the execution time , the lesser the priority.
- Disadvantage: Task with highest amount of time has to wait.

For Hard real time systems none of the above scheduling algorithms can be Used.

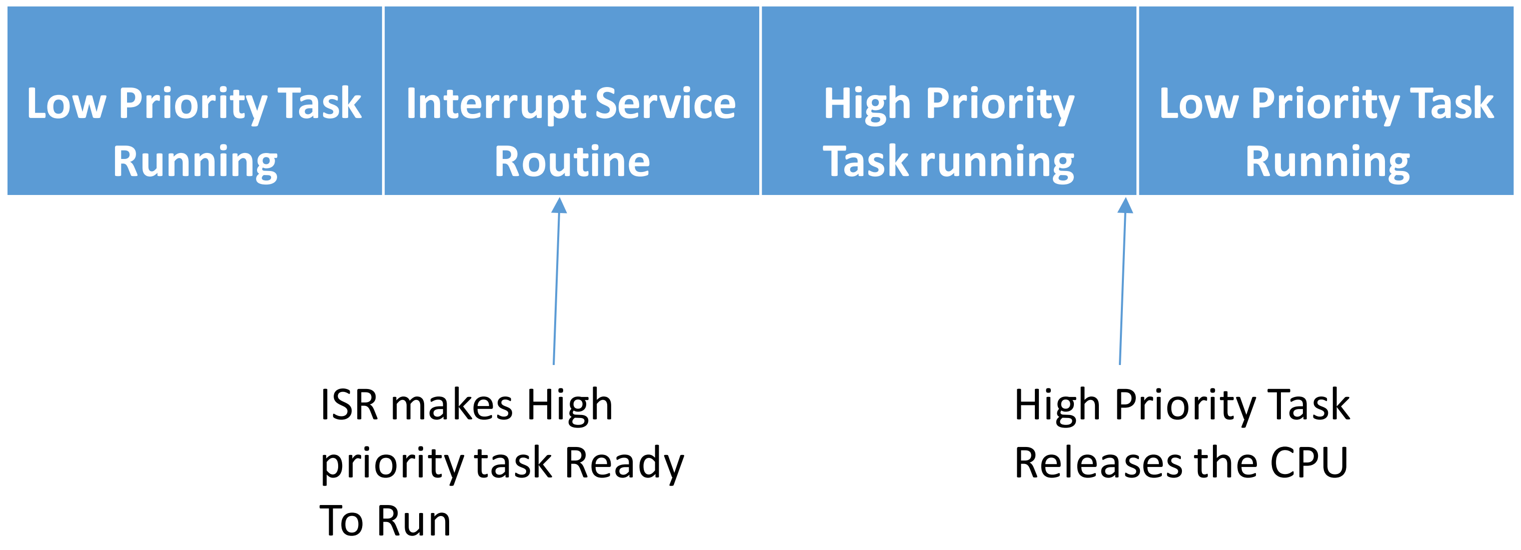
Non Pre- emptive Multitasking

- Tasks Cooperate with each other to get their share of the CPU.
- Each Task has to release the CPU and give control to another task on its own.
- Each Task is given priority.
- Disadvantage: High Priority task wait for long time


Low Priority Task Running
Interrupt Service Routine
Low Priority Task Running
High Priority Task Running
ISR makes High Priority Task Ready to Run
Low Priority Task releases the CPU

Preemptive Multitasking

- Highest priority task is always given the CPU time.
- If Lower priority is running and a higher priority task is in Ready-to-Run state, the running task is pre-empted and the higher priority task is executed.


Low Priority Task Running
Interrupt Service Routine
High Priority Task running
Low Priority Task Running
ISR makes High priority task Ready To Run
High Priority Task Releases the CPU

R
T
O
S

Rate Monotonic Scheduling :

➢ In Rate Monotonic Scheduling tasks with the highest rate of execution are given the highest priority

➢ Assumptions:

- ✓ All tasks are periodic
- ✓ Tasks do not synchronize with one another, share resources, etc.
- ✓ Preemptive scheduling is used (always runs the highest priority task that is ready)

➢ Under these assumptions, let n be the number of tasks, Ei be the execution time of task i, and Ti be the period of task i. Then, all deadlines will be met if the following inequality is satisfied:

$\Sigma Ei / Ti \leq n(2\^{}1/n - 1)$

Rate Monotonic Scheduling :

➢ Suppose we have 3 tasks. Task 1 runs at 100 Hz and takes 2 ms. Task 2 runs at 50 Hz and takes 1 ms. Task 3 runs at 66.7 Hz and takes 7 ms. Apply RMS theory…

$$(2/10) + (1/20) + (7/15) = 0.717 \le 3(21/3 - 1) = 0.780$$

✓Thus, all the deadlines will be met

➢ General Solution?

✓ As n $\rightarrow \infty$, the right-hand side of the inequality goes to ln(2)=0.6931. Thus, you should design your system to use less than 60-70% of the CPU

RTOS

Task Management (Services):

- Task Feature
- Task Creation
- Task Stack & Stack Checking
- Task Deletion
- Change a Task's Priority
- Suspend and Resume a Task
- Get Information about a Task

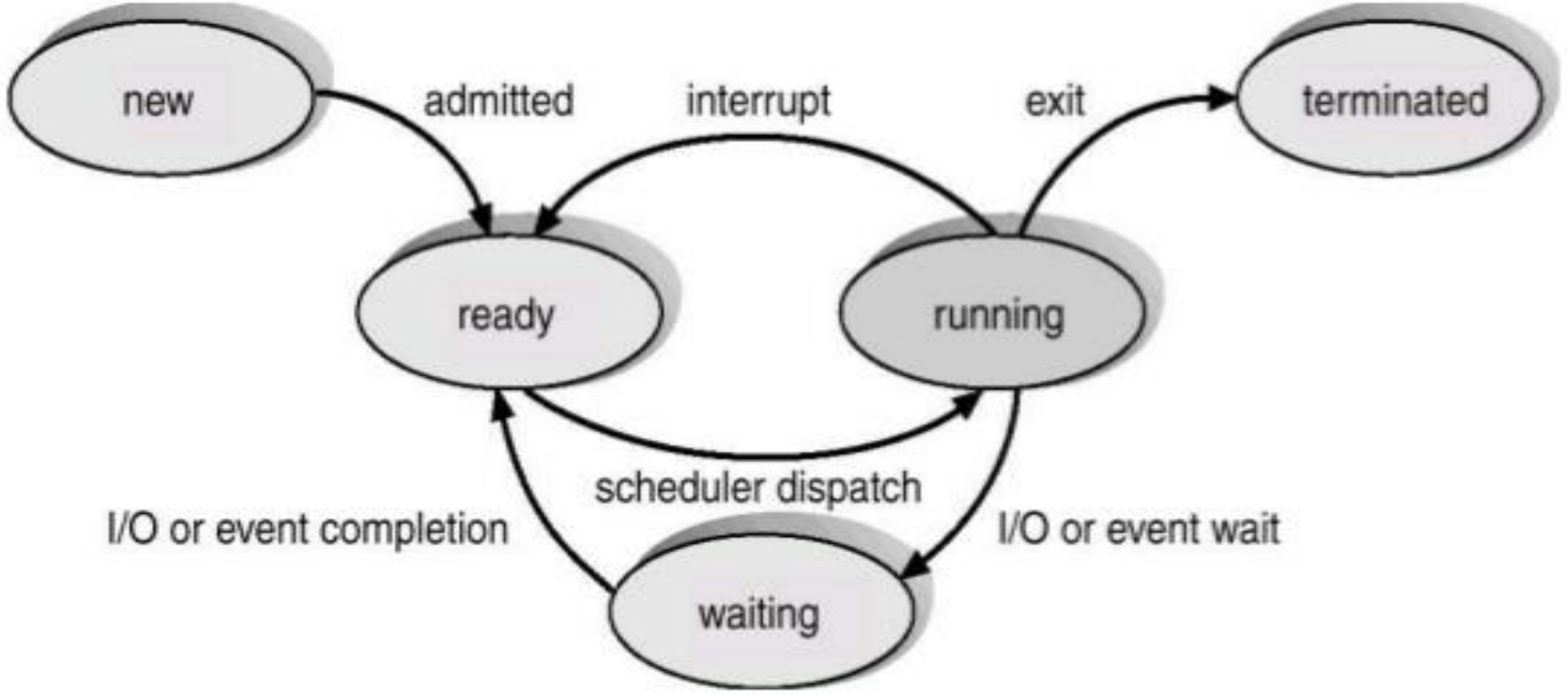
R
T
O
S

Task Feature :

- μC/OS-II can manage up to 64 tasks.
- The four highest priority tasks and the four lowest priority tasks are reserved for its own use. This leaves 56 tasks for applications.
- The lower the value of the priority, the higher the priority of the task. (Something on the lines of Rate Monotonic Scheduling).
- The task priority number also serves as the task identifier.

R
T
O
S

PROCESS CYCLE :



R
T
O
S

Task Creation :

There are two functions for creating a task:

✓OSTaskCreate()

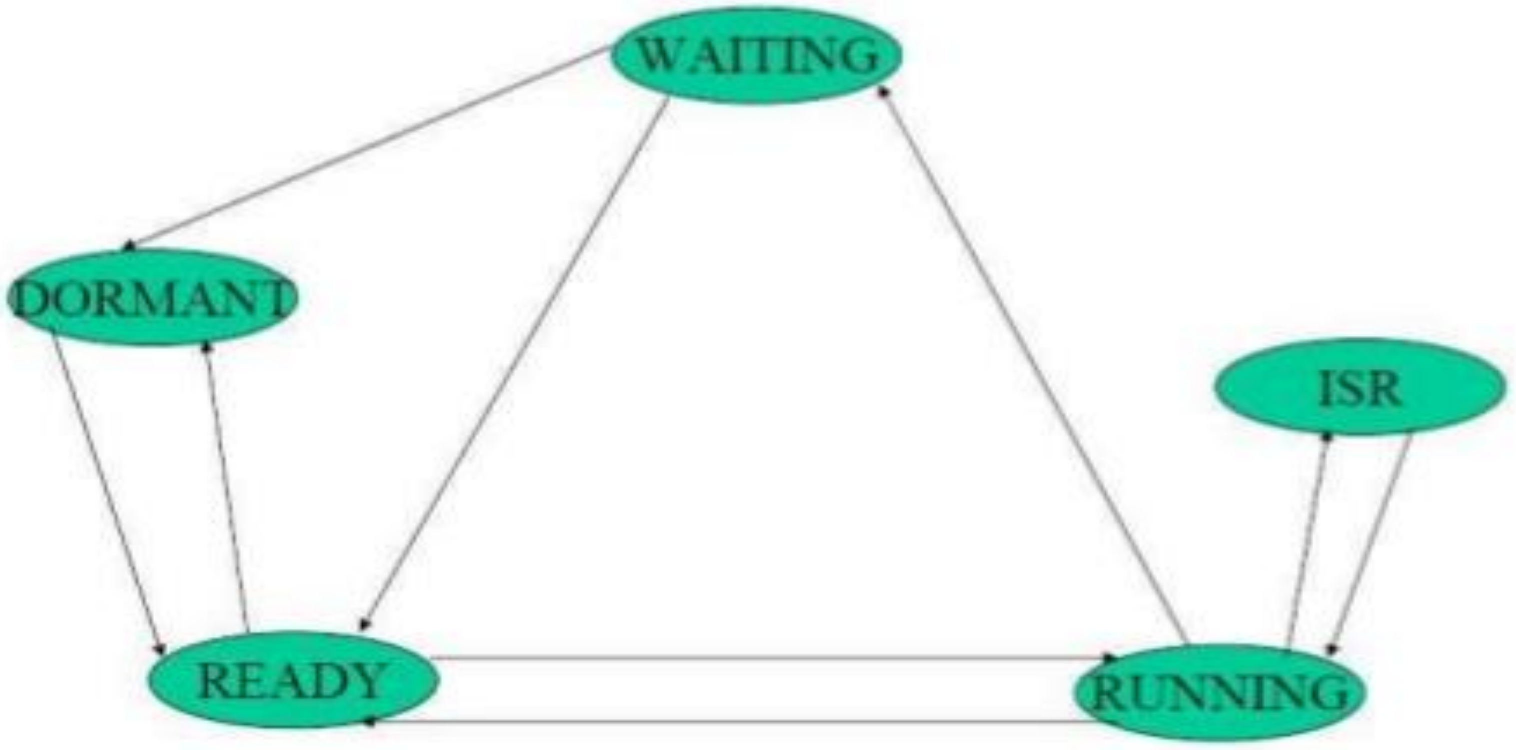
✓ OSTaskCreateExt().

Why 'Create' a Task?

- To make it ready for multitasking
- The kernel needs to have information about your task
  - Its starting address
  - Its top-of-stack (TOS)
  - Its priority
  - Arguments passed to the task
  - Other information about your task

RTOS

Task Management :



R
T
O
S

Task Management :

➢After the task is created, the task has to get a stack in which it will store its data.

➢ A stack must consist of contiguous memory locations.

➢ It is necessary to determine how much stack space a task actually uses.

➢ Deleting a task means the task will be returned to its dormant state and does not mean that the code for the task will be deleted. The calling task can delete itself.

RTOS

Task Management (cont…):

- If another task tries to delete the current task, the resources are not freed and thus are lost. So the task has to delete itself after it uses its resources
- Priority of the calling task or another task can be changed at run time.
- A task can suspend itself or another task, a suspended task can resume itself.
- A task can obtain information about itself or other tasks. This information can be used to know what the task is doing at a particular time.

Task Management

- Creating a task
- Stack checking
- Changing the task priority
- Getting Information about a task
- Suspending a task
- Resuming a task
- Request to delete a task
- Deleting a task

Task creation

There are two functions to create a task:

1. OSTaskCreate()
2. OSTaskCreateExt()

OSTaskCreate( ) requires 4 arguments:

1. ‘task’ is pointer to the task function
2. ‘pdata’ is a pointer to an argument that is passed to the task function when it starts executing
3. ‘ptos’ is a pointer to the top of the stack
4. ‘prio’ is the task priority

OSTaskCreateExt()

OSTaskCreateExt() offers more functionality but at the expense of additional overhead. This function requires 9 arguments. The first four arguments are same as with OSTaskCreate()

Extra arguments:

5. ‘id’ a unique identifier for the task being created. This argument has been added for future expansion and is not used by µC/OS-II, usually set to task priority.
6. ‘pbos’ is a pointer to the task's bottom of stack, used to perform stack checking
7. ‘stk_size’ is the size of the stack in number of elements. This argument is also used for stack checking

OSTaskCreateExt() (continued)

8. ‘pext’ is a pointer to a user supplied data area that can be used to extend the size of the task control block
9. ‘opt’ specifies options to OSTaskCreateExt(). This argument specifies whether stack checking is allowed, whether the stack will be cleared, and whether floating point operations are performed by the task.

The file os_cfg.h contains a list of available options

1. OS_TASK_OPT_STK_CHK
2. OS_TASK_OPT_STK_CLR
3. OS_TASK_OPT_SAVE_FP

Task initialization

OSTaskCreate() calls OSTaskStkInit () to initialize the stack for the task. OSTaskInit() is defined in the file os_cpu.c

µC/OS-II supports processors that have stacks that grow from either high memory to low memory or from low memory to high memory (defined by OS_STK_GROWTH in the file os_cpu.h)

OSTCBInit() initializes the TCB for this task. Defined in os_core.c

OSTaskCreateHook() is a user specified function that extends the functionality of OSTaskCreate() but can increase interrupt latency

Finally, OSTaskCreate() calls OSSched() to determine whether the new task has a higher priority than its creator.

OSStart() -multitasking has started

Task function

A task is typically an infinite loop function

```
void YourTask(void *pdata)
{
    for (;;) {
        /* USER CODE */
        Call one of uC/OS-II's services:
        OSMboxPend();
        OSQPend();
        OSSemPend();
        OSTaskDel(OP_PRIO_SELF);
        OSTaskSuspend(OS_PRIO_SELF);
        OSTimeDly();
        OSTimeDlyHMSM();
        /* USER CODE */
    }
}
```

Task deletion

A task can delete itself upon completion.

```
void YourTask(void *pdata)
{
    /* USER CODE */
    OSTaskDel(OS_PRIO_SELF);
}
```

Task Stack

Each task must have its own stack. A stack must be declared as being of type OS_STK and must consist of contiguous memory locations.

We pass the top of stack pointer in the task create function

Memory can be allocated statically or dynamically

Static allocation

```
OS_STK MyTaskStack[STACKSIZE];
```

Dynamic allocation

```
OS_STK *pstk;

pstk = (OS_STK *) malloc(STACKSIZE);
if (pstk != (OS_STK *) 0) {
    /* Create the task */
}
```

Task Stack (continued)

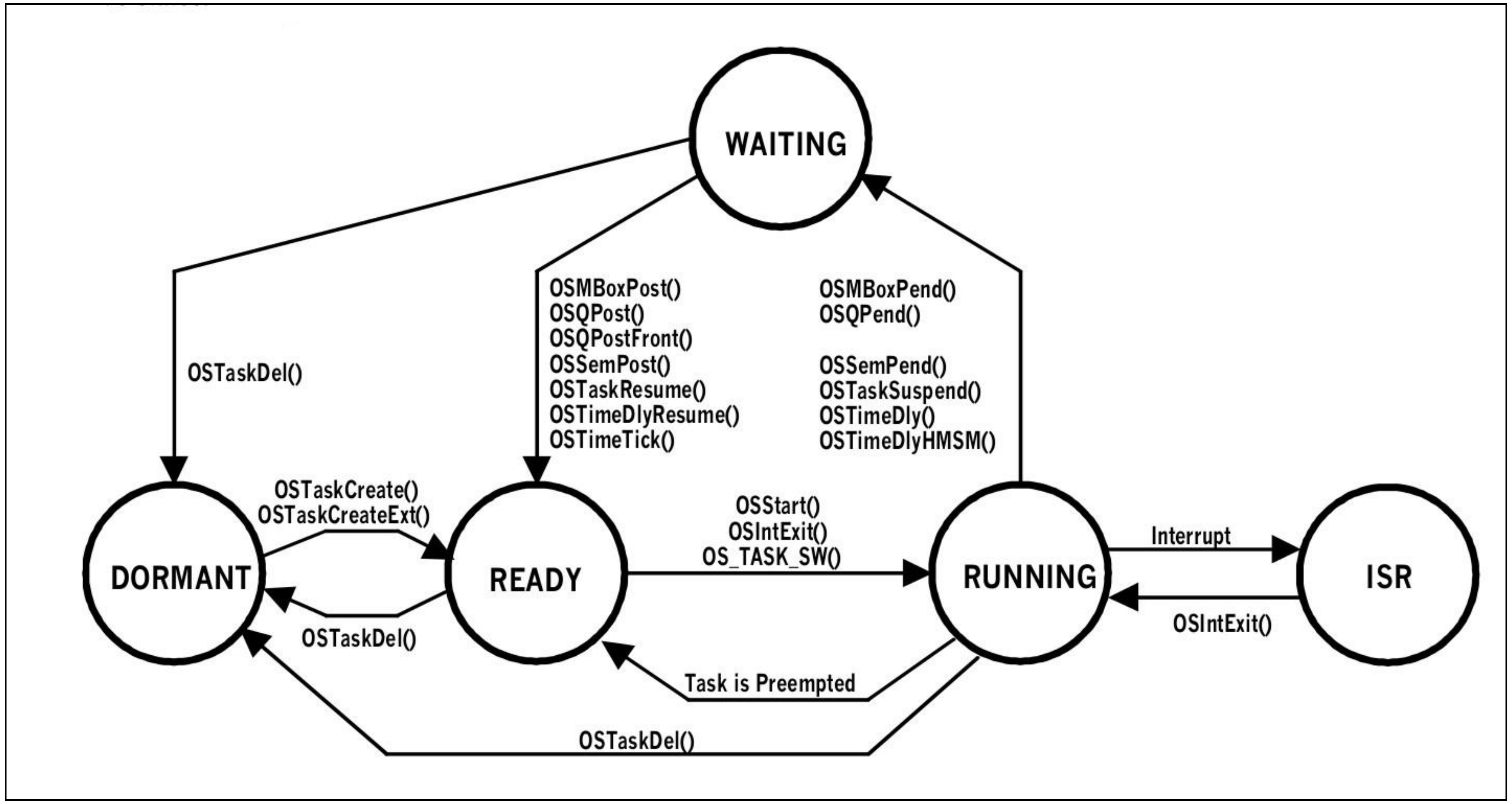
If in file os_cpu.h the constant OS_STK_GROWH is set to 1 you must pass the **highest** memory location of the stack to the task-create function because the stack grows from high to low memory

```
OS_STK TaskStack[STACKSIZE];

OSTaskCreate(task, pdata, &TaskStack[STACKSIZE - 1], prio);
```

If the constant OS_STK_GROWTH is set to 0, you must pass the **lowest** memory location of the stack to the task-create function because the stack grows from low to high memory

```
OS_STK TaskStack[STACKSIZE];

OSTaskCreate(task, pdata, &TaskStack[0], prio);
```

Task states


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Task Control Block

A task control block is a data structure called OS_TCB, that is used by µC/OS-II to store the state of a task when it is preempted.

When the task regains control of the CPU, data in the task control block allows the task to resume execution exactly where it left off.

All OS_TCB's reside in RAM.

The OS_TCB is initialized when a task is created.

OS_TCB structure

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;

#if OS_TASK_CREATE_EX_EN
    void            *OSTCBExtPtr;
    OS_STK          *OSTCBStkBottom;
    INT32U           OSTCBStkSize;
    INT16U           OSTCBOpt;
    INT16U           OSTCBId;
#endif

    struct os_tcb   *OSTCBNext;
    struct os_tcb   *OSTCBPrev;

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT        *OSTCBEventPtr;
#endif
```

Task Scheduling

µC/OS-II always executes the highest priority task that is ready to run.

Which task has highest priority is determined by the scheduler.

Task level scheduling is performed by **`OSSched()`**

ISR level scheduling is performed by **`OSIntExit()`**

µC/OS-II task scheduling time is constant irrespective of the number of tasks created by the application.

Locking & Unlocking the Scheduler

The function **`OSSchedLock()`** is used to prevent task rescheduling until its counterpart, **`OSSchedUnlock()`**, is called.

The task that calls **`OSSchedLock()`** keeps control of the CPU even though other higher priority tasks are ready to run.

Interrupts, however, are still recognized and serviced (assuming interrupts are enabled).

The variable **`OSLockNesting`** keeps track of the number of times **`OSSchedLock()`** has been called to allow for nesting.

```
void TaskX (void *pdata)
{
      pdata = pdata;
      for (;;) {
         .
         OSSchedLock();         /* Prevent other tasks to run        */
         .
         .                      /* Code protected from context switch */
         .
         OSSchedUnlock();       /* Enable other tasks to run         */
         .
      }
}
```

What is the Idle Task?

If there is no task, then the CPU will execute the idle task.

The idle task cannot be deleted because it should always be present in case there is no other task to run.

The idle task function **`OSTaskIdle()`** has the lowest priority, **`OS_LOWEST_PRIO`**.

`OSTaskIdle()` does nothing but increments a 32-bit counter called **`OSIdleCtr`**.

`OSIdleCtr` is used by the statistics task to determine how much CPU time (in percentage) is actually being used by the application task software

Idle Task

```
void OS_TaskIdle(void *pdata)
{
    pdata = pdata;
    for (;;) {
       OS_ENTER_CRITICAL();
       OSIdleCtr++;
       OS_EXIT_CRITICAL();
    }
}
```

Stack Checking

The OSTaskStkChk() function tells how much free stack space is available for your application.

To enable the µC/OS-II stack checking feature you must set OS_TASK_CREATE_EXT to ‘1’ in file os_cfg.h

Create the task using OSTaskCreateExt() and give the task more space than you think it really needs.

Set ‘opt’ argument in OSTaskCreateExt() to OS_TASK_OPT_STK_CHK

Call OSTaskStkChk() from a task by specifying the priority of the task you want to check. You can inquire about any task stack, not just the running task.

Stack Checking (continued)

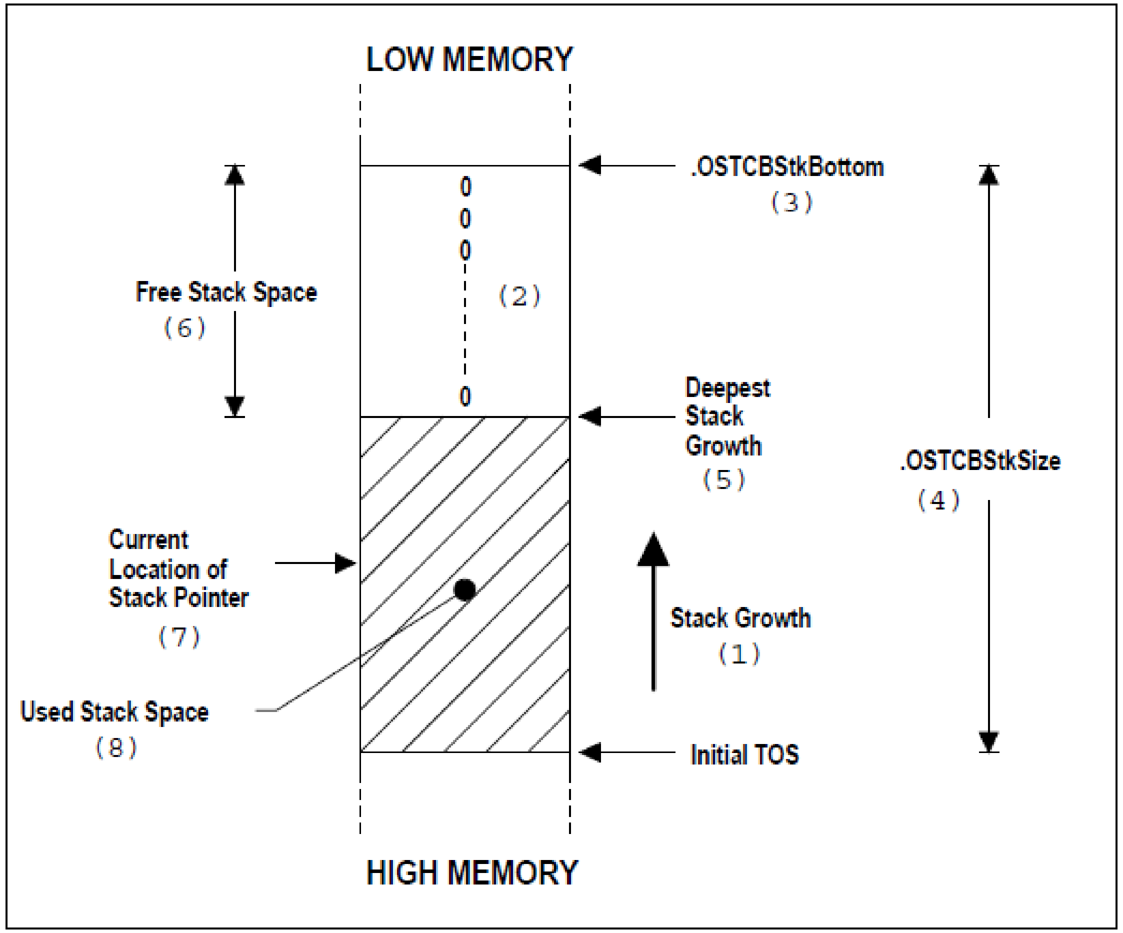
Assume `OS_STK_GROWTH` is set to 1 (Stack grows to lower memory addresses).

To perform stack checking, µC/OS-II requires that the stack is filled with zeros when the task is created

µC/OS-II needs to know the address of the Bottom-Of-Stack (BOS) and the size of the stack that is assigned to the task. These two values are stored in the task's OS_TCB when the task is created.

µC/OS-II determines the free stack space by counting the zero-elements from the Bottom-Of-Stack to the first no-zero element.

Stack Checking (continued)



Suspending a Task

Sometimes we need to explicitly suspend the execution of a task.

Suspension is done by calling OSTaskSuspend()

The suspended task can only be resumed by calling OSTaskResume()

Task suspension is cumulative.

A task can suspend itself or another task.

Resuming a Task

Sometimes we need to resume a task that we have suspended some time before.

A suspended task can only be resumed by calling OSTaskResume()

A task is resumed only if the number of calls to OSTaskResume() matches the calls to OSTaskSuspend()

A task cannot resume itself, only another task can do this.

Changing the task priority

Call the OSTaskChangePrio() function if you want to change priority of task at runtime.

Getting Information about a task

A task can retrieve information about itself or another application tasks by calling OSTaskQuery()

The function OSTaskQuery() gets a copy of specified task's OS_TCB.

To call OSTaskQuery(), your application must first allocate storage for OS_TCB.

After calling OSTaskQuery() this OS_TCB contains a snapshot of the OS_TCB for the desired task.

Requesting to delete a task

When a task deletes another task, there is a risk that there is no systematically de-allocation of the resources created by the other task.

This would lead to memory leaks which are not acceptable.

In this case it is required that a task tells the other task to delete itself when it is done with it's resources.

We can do this by using OSTaskDelReq() function.

Both the requesting task and task to be deleted must call OSTaskDelReq()

Deleting a Task

The OSTaskDel() function deletes a task, but that does not mean that we are deleting the code of the task from memory.

The task is returned to the dormant state. The task is simply no longer scheduled by µC/OS-II.

The OSTaskDel() function ensures that a task cannot not be deleted within an ISR.

Make sure you should never delete the idle task.

A task can delete itself by specifying OS_PRIO_SELF as the argument.

R
T
O
S

Memory Management :

The services of Memory management includes:

- ✓Initializing the Memory Manager.
- ✓Creating a Memory Partition.
- ✓Obtaining Status of a Memory Partition.
- ✓Obtaining a Memory Block.
- ✓Returning a Memory Block.
- ✓Waiting for Memory Blocks from a Memory Partition.

R
T
O
S

Memory Management :

➢Each memory partition consists of several fixed-sized memory blocks.

➢A task obtains memory blocks from the memory partition. A task must create a memory partition before it can be used.

➢Allocation and de-allocation of these fixed-sized memory blocks is done in constant time and is deterministic.

➢Multiple memory partitions can exist, so a task can obtain memory blocks of different sizes.

➢A specific memory block should be returned to its memory partition from which it came.

R T O S

Time Management :

Clock Tick : A clock tick is a periodic time source to keep track of time delays and time outs.

➢Here, tick intervals varies from 10 ~ 100 ms.

➢The faster the tick rate, the higher the overhead imposed on the system.

➢Whenever a clock tick occurs μC/OS-II increments a 32- bit counter, the counter starts at zero, and rolls over to 4,294,967,295 (2^32-1) ticks.

➢A task can be delayed and a delayed task can also be resumed.

R
T
O
S

Time Management :

It involves five services that includes:

✓OSTimeDLY()

✓OSTimeDLYHMSM()

✓OSTimeDlyResume()

✓OSTimeGet()

✓OSTimeSet()

μC/OS-II Time Management Service	**Enabled when set to 1 in OS_CFG.H**
OSTimeDly()	
OSTimeDlyHMSM()	OS_TIME_DLY_HMSM_EN
OSTimeDlyResume()	OS_TIME_DLY_RESUME_EN
OSTimeGet()	OS_TIME_GET_SET_EN
OSTimeSet()	OS_TIME_GET_SET_EN

Time Management configuration constants in OS_CFG.H

Delaying a Task, OSTimeDly()

- µC/OS-II provides a service that allows the calling task to delay itself for a user-specified number of clock ticks.
- Calling this function causes a context switch and forces µC/OS-II to execute the next highest priority task that is ready to run.
- The task calling OSTimeDly() is made ready to run as soon as the time specified expires
- or if another task cancels the delay by calling OSTimeDlyResume().
- Note that this task will run only when it’s the highest priority task.

OSTimeDlyHMSM()

- you can specify time in hours (H), minutes (M), seconds (S), and milliseconds (m), which is more natural.
- calling this function causes a context switch and forces µC/OS-II to execute the next highest priority task that is ready to run.
- The task calling OSTimeDlyHMSM() is made ready to run as soon as the time specified expires or if another task cancels the delay by calling OSTimeDlyResume()
- this task runs only when it again becomes the highest priority task.

OSTimeDlyResume()

- Instead of waiting for time to expire, a delayed task can be made ready to run by another task that cancels the delay.
- This is done by calling OSTimeDlyResume() and specifying the priority of the task to resume.
- OSTimeDlyResume() checks to see if the task is waiting for time to expire.
- Whenever the OS_TCB field .OSTCBDly contains a nonzero value, the task is waiting for time to expire because the task called either OSTimeDly(), OSTimeDlyHMSM(), or any of the PEND functions
- The delay is then canceled by forcing .OSTCBDly to 0.

System Time, OSTimeGet() and OSTimeSet()

- Whenever a clock tick occurs, µC/OS-II increments a 32-bit counter.
- This counter starts at zero when you initiate multitasking by calling OSStart() and rolls over after 4,294,967,295 ticks.
- At a tick rate of 100Hz, this 32-bit counter rolls over every 497 days.
- You can obtain the current value of this counter by calling OSTimeGet().
- You can also change the value of the counter by calling OSTimeSet().

Message Mailbox Management

- A message mailbox (or simply a mailbox) is a µC/OS-II object that allows a task or an ISR to send a pointer-sized variable to another task.
- µC/OS-II provides seven services to access mailboxes:
- OSMboxCreate(), OSMboxDel(), OSMboxPend(), OSMboxPost(), OSMboxPostOpt(), OSMboxAccept(), and OSMboxQuery().

µC/OS-II Mailbox Service	Enabled when set to 1 in OS_CFG.H
OSMboxAccept()	OS_MBOX_ACCEPT_EN
OSMboxCreate()	
OSMboxDel()	OS_MBOX_DEL_EN
OSMboxPend()	
OSMboxPost()	OS_MBOX_POST_EN
OSMboxPostOpt()	OS_MBOX_POST_OPT_EN
OSMboxQuery()	OS_MBOX_QUERY_EN

Table - Table 10.1 Mailbox configuration constants in OS_CFG.H.

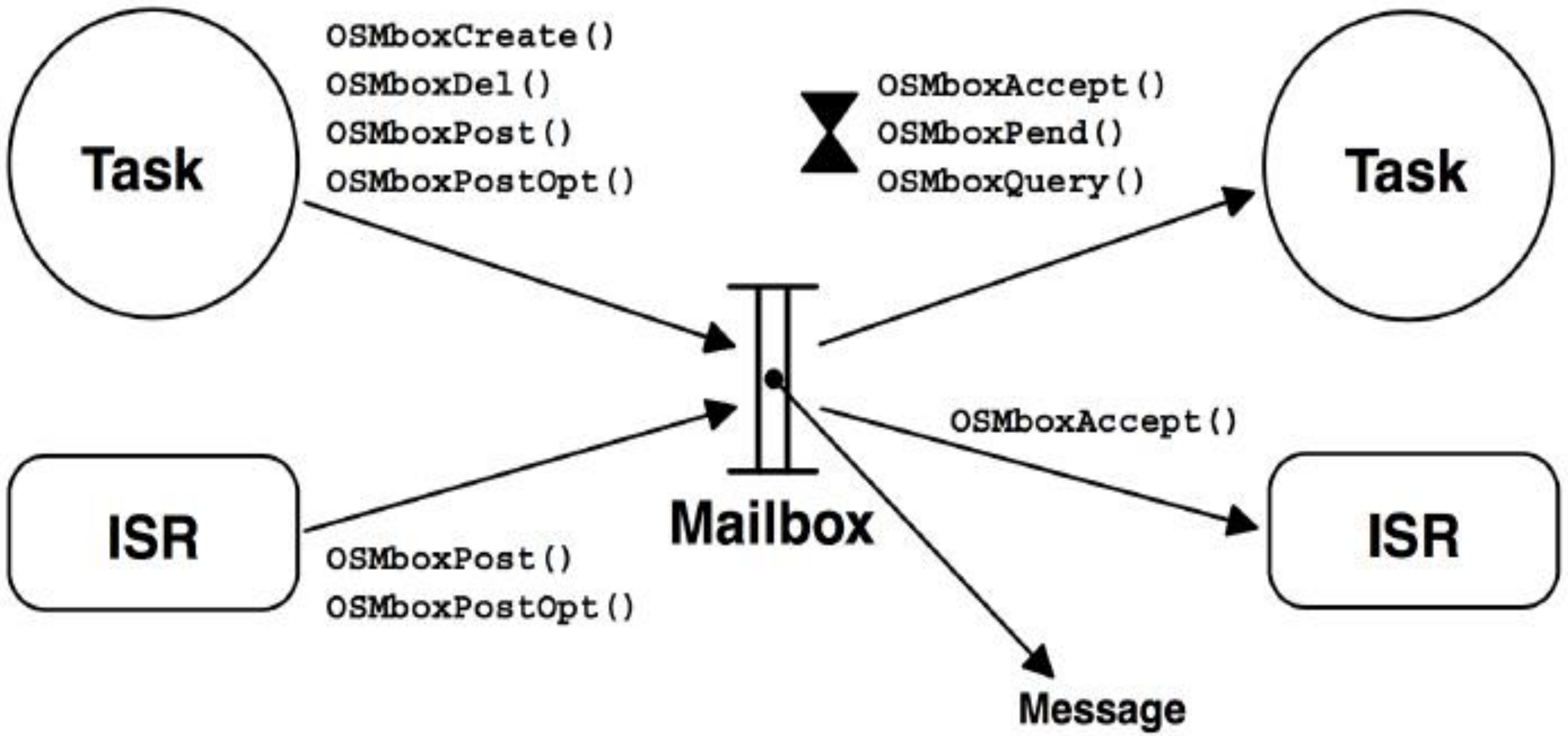


Figure - Figure 10.1 Relationships between tasks, ISRs, and a message mailbox.

OSMboxAccept()

- OSMboxAccept() allows you to see if a message is available from the desired mailbox.
- Unlike OSMboxPend(), OSMboxAccept() does not suspend the calling task if a message is not available.
- In other words, OSMboxAccept() is non-blocking. If a message is available, the message is returned to your application, and the content of the mailbox is cleared.
- This call is typically used by ISRs because an ISR is not allowed to wait for a message at a mailbox.

OSMboxCreate()

OS_EVENT *OSMboxCreate(void *msg);

- OSMboxCreate() creates and initializes a mailbox.
- A mailbox allows tasks or ISRs to send a pointer-sized variable (message) to one or more tasks.
- **Arguments**
- msg is used to initialize the contents of the mailbox. The mailbox is empty when msg is a NULL pointer. The mailbox initially contains a message when msg is non-NULL.
- **Returned Value**
- A pointer to the event control block allocated to the mailbox. If no event control block is available, OSMboxCreate() returns a NULL pointer.

```
OS_EVENT *CommMbox;



void main (void)
{
        .
        .
        OSInit();                               /* Initialize µC/OS-II  */
        .
        .
        CommMbox = OSMboxCreate((void *)0);    /* Create COMM mailbox  */
        OSStart();                              /* Start Multitasking   */
}
```

OSMboxDel()

OS_EVENT *OSMboxDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

- OSMboxDel() is used to delete a message mailbox.

- This function is dangerous to use because multiple tasks could attempt to access a deleted mailbox. You should always use this function with great care.

- Generally speaking, before you delete a mailbox, you must first delete all the tasks that can access the mailbox.

```
OS_EVENT *DispMbox;

void Task (void *pdata)
{
    INT8U  err;

    pdata = pdata;
    while (1) {
        .
        .
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        if (DispMbox == (OS_EVENT *)0) {
            /* Mailbox has been deleted */
        }
        .
        .
    }
}
```

OSMboxPend()

void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

- OSMboxPend() is used when a task expects to receive a message. The message is sent to the task either by an ISR or by another task.
- The message received is a pointer-sized variable, and its use is application specific.
- If a message is present in the mailbox when OSMboxPend() is called, the message is retrieved, the mailbox is emptied, and the retrieved message is returned to the caller.
- If no message is present in the mailbox, OSMboxPend() suspends the current task until either a message is received or a user-specified timeout expires.

```
OS_EVENT *CommMbox;


void CommTask(void *pdata)
{
        INT8U  err;
        void   *msg;


        pdata = pdata;
        for (;;) {
            .
            .
            msg = OSMboxPend(CommMbox, 10, &err);
            if (err == OS_NO_ERR) {
                .
                .   /* Code for received message                    */
                .
            } else {
                .
                .   /* Code for message not received within timeout */
                .
            }
            .
            .
        }
}
```

OSMboxPost()

INT8U OSMboxPost(OS_EVENT *pevent, void *msg);

- OSMboxPost() sends a message to a task through a mailbox. A message is a pointer-sized variable and, its use is application specific.
- If a message is already in the mailbox, an error code is returned indicating that the mailbox is full. OSMboxPost() then immediately returns to its caller, and the message is not placed in the mailbox.
- If any task is waiting for a message at the mailbox, the highest priority task waiting receives the message.
- If the task waiting for the message has a higher priority than the task sending the message, the higher priority task is resumed, and the task sending the message is suspended. In other words, a context switch occurs.

```
OS_EVENT *CommMbox;
INT8U     CommRxBuf[100];

void CommTaskRx (void *pdata)
{
      INT8U  err;

      pdata = pdata;
      for (;;) {
          .
          .
           err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
          .
          .
      }
}
```

Message Queue Management

- A message queue (or simply a queue) is a µC/OS-II object that allows a task or an ISR to send pointer-sized variables to another task.

- Each pointer typically is initialized to point to some application-specific data structure containing a message.
- µC/OS-II provides nine services to access message queues:
- OSQCreate(), OSQDel(), OSQPend(), OSQPost(), OSQPostFront(), OSQPostOpt(), OSQAccept(), OSQFlush(), and OSQQuery().

µC/OS-II Queue Service	Enabled when set to 1 in OS_CFG.H
OSQAccept()	OS_Q_ACCEPT_EN
OSQCreate()	
OSQDel()	OS_Q_DEL_EN
OSQFlush()	OS_Q_FLUSH_EN
OSQPend()	
OSQPost()	OS_Q_POST_EN
OSQPostFront()	OS_Q_POST_FRONT_EN
OSQPostOpt()	OS_Q_POST_OPT_EN
OSQQuery()	OS_Q_QUERY_EN

Table - Table 11.1 Message queue configuration constants in OS_CFG.H.

OSQAccept()

void *OSQAccept(OS_EVENT *pevent, INT8U *err);

- OSQAccept() checks to see if a message is available in the desired message queue. Unlike OSQPend(), OSQAccept() does not suspend the calling task if a message is not available.
- In other words, OSQAccept() is non-blocking.
- If a message is available, it is extracted from the queue and returned to your application.
- This call is typically used by ISRs because an ISR is not allowed to wait for messages at a queue.
- **Returned Value**
- A pointer to the message if one is available; NULL if the message queue does not contain a message or the message received is a NULL pointer.

```
OS_EVENT *CommQ;

void Task (void *pdata)
{
        void *msg;

        pdata = pdata;
        for (;;) {
            msg = OSQAccept(CommQ);         /* Check queue for a message   */
            if (msg != (void *)0) {
                .                           /* Message received, process   */
                .
            } else {
                .                           /* Message not received, do .. */
                .                           /* .. something else           */
            }
            .
            .
        }
}
```

OSQCreate()

OS_EVENT *OSQCreate(void **start, INT8U size);

- OSQCreate() creates a message queue. A message queue allows tasks or ISRs to send pointer-sized variables (messages) to one or more tasks. The meaning of the messages sent are application specific.
- **Arguments**
- start is the base address of the message storage area. A message storage area is declared as an array of pointers.
- size is the size (in number of entries) of the message storage area.
- **Returned Value**
- OSQCreate() returns a pointer to the event control block allocated to the queue. If no event control block is available, OSQCreate() returns a NULL pointer.

```
OS_EVENT *CommQ;
void     *CommMsg[10];




void main (void)
{
        OSInit();                                         /* Initialize µC/OS-II
*/
        .
        .
        CommQ = OSQCreate(&CommMsg[0], 10);               /* Create COMM Q
*/
        .
        .
        OSStart();                                        /* Start Multitasking
*/
}
```

```
OS_EVENT *DispQ;



void Task (void *pdata)
{
    INT8U  err;



    pdata = pdata;
    while (1) {
        .
        .
        DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
        if (DispQ == (OS_EVENT *)0) {
            /* Queue has been deleted */
        }
        .
        .
    }
}
```

OSQDel()

- OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *err);
- OSQDel() is used to delete a message queue.
- This function is dangerous to use because multiple tasks could attempt to access a deleted queue. You should always use this function with great care.
- Generally speaking, before you delete a queue, you must first delete all the tasks that can access the queue.

OSQPend()

- void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
- OSQPend() is used when a task wants to receive messages from a queue.
- The messages are sent to the task either by an ISR or by another task.
- The messages received are pointer-sized variables, and their use is application specific.
- If at least one message is present at the queue when OSQPend() is called, the message is retrieved and returned to the caller. If no message is present at the queue, OSQPend() suspends the current task until either a message is received or a user-specified timeout expires.

```
OS_EVENT *CommQ;


void CommTask(void *data)
{
        INT8U  err;
        void   *msg;


        pdata = pdata;
        for (;;) {
            .
            .
            msg = OSQPend(CommQ, 100, &err);
            if (err == OS_NO_ERR) {
                .
                .          /* Message received within 100 ticks!         */
                .
            } else {
                .
                .          /* Message not received, must have timed out  */
                .
            }
            .
            .
        }
}
```

OSQPost()

- INT8U OSQPost(OS_EVENT *pevent, void *msg);
- OSQPost() sends a message to a task through a queue. A message is a pointer-sized variable, and its use is application specific.
- If the message queue is full, an error code is returned to the caller. In this case, OSQPost() immediately returns to its caller, and the message is not placed in the queue.
- If any task is waiting for a message at the queue, the highest priority task receives the message. If the task waiting for the message has a higher priority than the task sending the message, the higher priority task resumes, and the task sending the message is suspended; that is, a context switch occurs.
- Message queues are first-in first-out (FIFO), which means that the first message sent is the first message received.

```
OS_EVENT *CommQ;
INT8U      CommRxBuf[100];


void CommTaskRx (void *pdata)
{
        INT8U  err;


        pdata = pdata;
        for (;;) {
            .
            .
            err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
            switch (err) {
                 case OS_NO_ERR:
                      /* Message was deposited into queue      */
                      break;

                 case OS_Q_FULL:
                      /* Queue is full                         */
                      Break;
                .
            }
            .
            .
        }
}
```

Message Mailbox Management

- A message mailbox (or simply a mailbox) is a µC/OS-II object that allows a task or an ISR to send a pointer-sized variable to another task.
- The pointer is typically initialized to point to some application specific data structure containing a "message."
- µC/OS-II provides seven services to access mailboxes: OSMboxCreate(), OSMboxDel(), OSMboxPend(), OSMboxPost(), OSMboxPostOpt(), OSMboxAccept(), and OSMboxQuery().

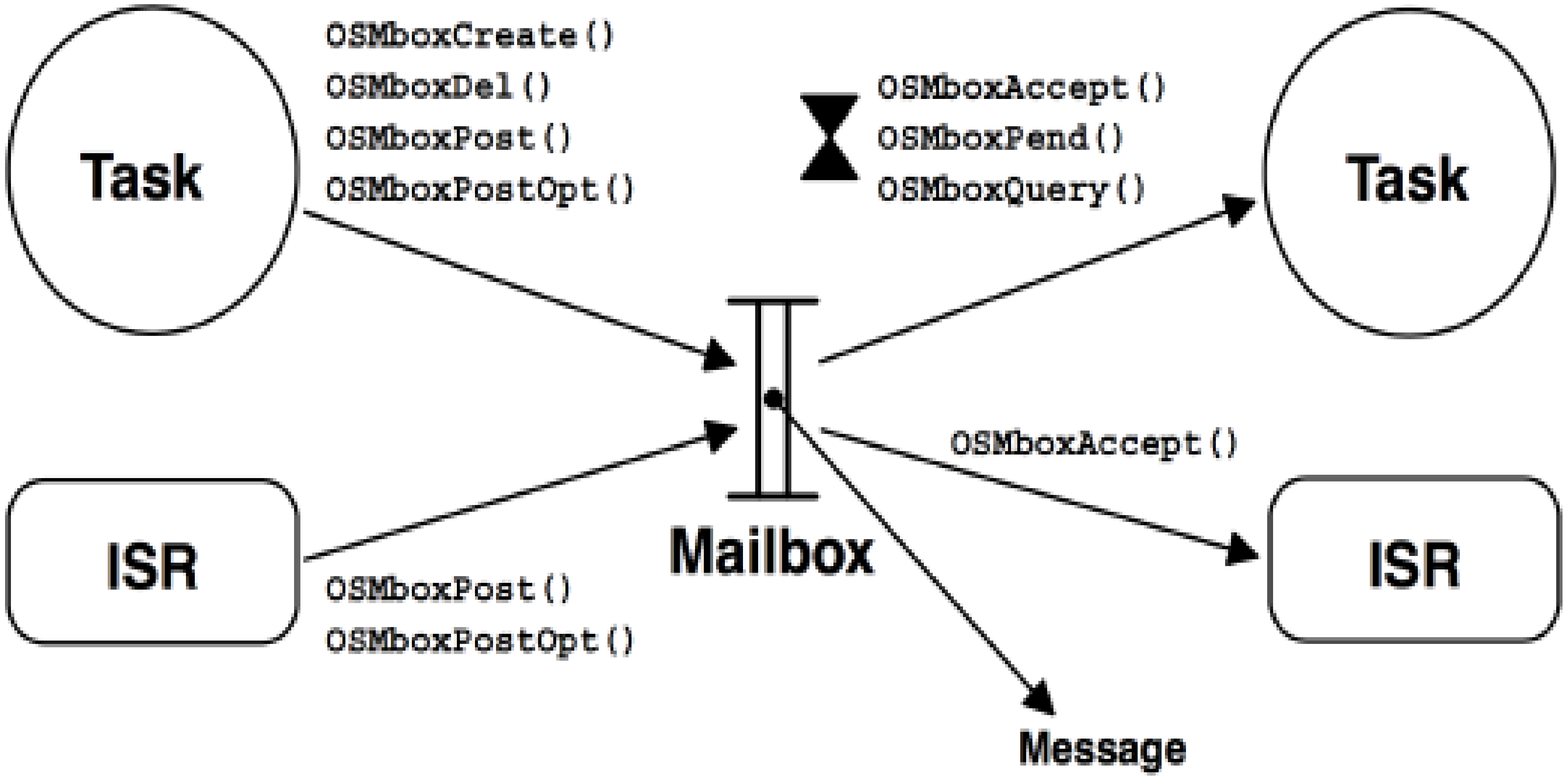


Figure - Figure 10.1 Relationships between tasks, ISRs, and a message mailbox.

OSMboxCreate()

- OS_EVENT *OSMboxCreate(void *msg);
- OSMboxCreate() creates and initializes a mailbox. A mailbox allows tasks or ISRs to send a pointer-sized variable (message) to one or more tasks.
- **Arguments**
- msg is used to initialize the contents of the mailbox. The mailbox is empty when msg is a NULL pointer. The mailbox initially contains a message when msg is non-NULL.
- **Returned Value**
- A pointer to the event control block allocated to the mailbox. If no event control block is available, OSMboxCreate() returns a NULL pointer.

```c
OS_EVENT *CommMbox;



void main (void)
{
        .
        .
        OSInit();                              /* Initialize µC/OS-II  */
        .
        .
        CommMbox = OSMboxCreate((void *)0);   /* Create COMM mailbox  */
        OSStart();                             /* Start Multitasking   */
}
```

OSMboxPost()

- INT8U OSMboxPost(OS_EVENT *pevent, void *msg);
- OSMboxPost() sends a message to a task through a mailbox. A message is a pointer-sized variable and, its use is application specific.
- If a message is already in the mailbox, an error code is returned indicating that the mailbox is full.
- OSMboxPost() then immediately returns to its caller, and the message is not placed in the mailbox.
- If any task is waiting for a message at the mailbox, the highest priority task waiting receives the message.

- **Arguments**
- pevent is a pointer to the mailbox into which the message is deposited. This pointer is returned to your application when the mailbox is created [see OSMboxCreate()].
- msg is the actual message sent to the task. msg is a pointer-sized variable and is application specific.
- You must never post a NULL pointer because this pointer indicates that the mailbox is empty.
- **Returned Value**
- OSMboxPost() returns one of these error codes:
- OS_NO_ERR if the message is deposited in the mailbox.
- OS_MBOX_FULL if the mailbox already contains a message.
- OS_ERR_EVENT_TYPE if pevent is not pointing to a mailbox.
- OS_ERR_PEVENT_NULL if pevent is a pointer to NULL.
- OS_ERR_POST_NULL_PTR if you are attempting to post a NULL pointer. By convention a NULL pointer is not supposed to point to anything.

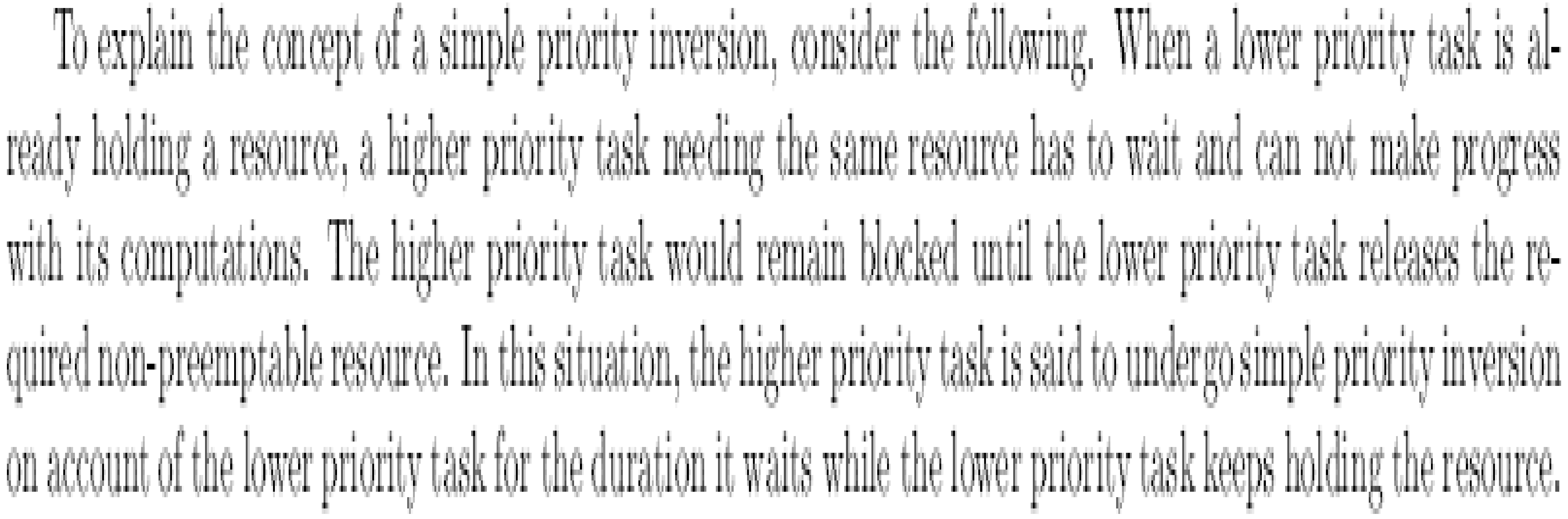
OSMboxPend()

- void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
- OSMboxPend() is used when a task expects to receive a message. The message is sent to the task either by an ISR or by another task. The message received is a pointer-sized variable, and its use is application specific.
- If a message is present in the mailbox when OSMboxPend() is called, the message is retrieved, the mailbox is emptied, and the retrieved message is returned to the caller.
- If no message is present in the mailbox, OSMboxPend() suspends the current task until either a message is received or a user-specified timeout expires.

```
OS_EVENT *CommMbox;
INT8U     CommRxBuf[100];


void CommTaskRx (void *pdata)
{
      INT8U  err;


      pdata = pdata;
      for (;;) {
          .
          .
          err = OSMboxPost(CommMbox, (void *)&CommRxBuf[0]);
          .
          .
      }
}
```

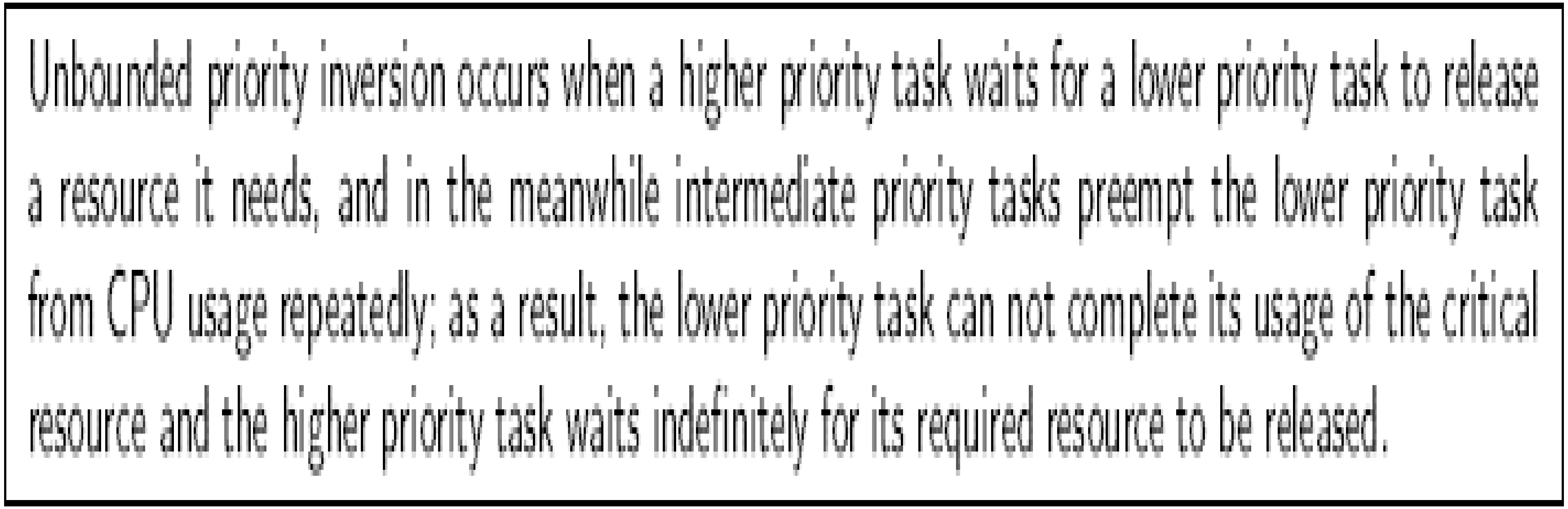
```
OS_EVENT *CommMbox;


void CommTask(void *pdata)
{
        INT8U  err;
        void   *msg;


        pdata = pdata;
        for (;;) {
            .
            .
            msg = OSMboxPend(CommMbox, 10, &err);
            if (err == OS_NO_ERR) {
                .
                .  /* Code for received message                        */
                .
            } else {
                .
                .  /* Code for message not received within timeout */
                .
            }
            .
            .
        }
}
```

Priority Inversion



To explain the concept of a simple priority inversion, consider the following. When a lower priority task is already holding a resource, a higher priority task needing the same resource has to wait and can not make progress with its computations. The higher priority task would remain blocked until the lower priority task releases the required non-preemptable resource. In this situation, the higher priority task is said to undergo simple priority inversion on account of the lower priority task for the duration it waits while the lower priority task keeps holding the resource.

Unbounded Priority Inversion



Unbounded priority inversion occurs when a higher priority task waits for a lower priority task to release a resource it needs, and in the meanwhile intermediate priority tasks preempt the lower priority task from CPU usage repeatedly; as a result, the lower priority task can not complete its usage of the critical resource and the higher priority task waits indefinitely for its required resource to be released.

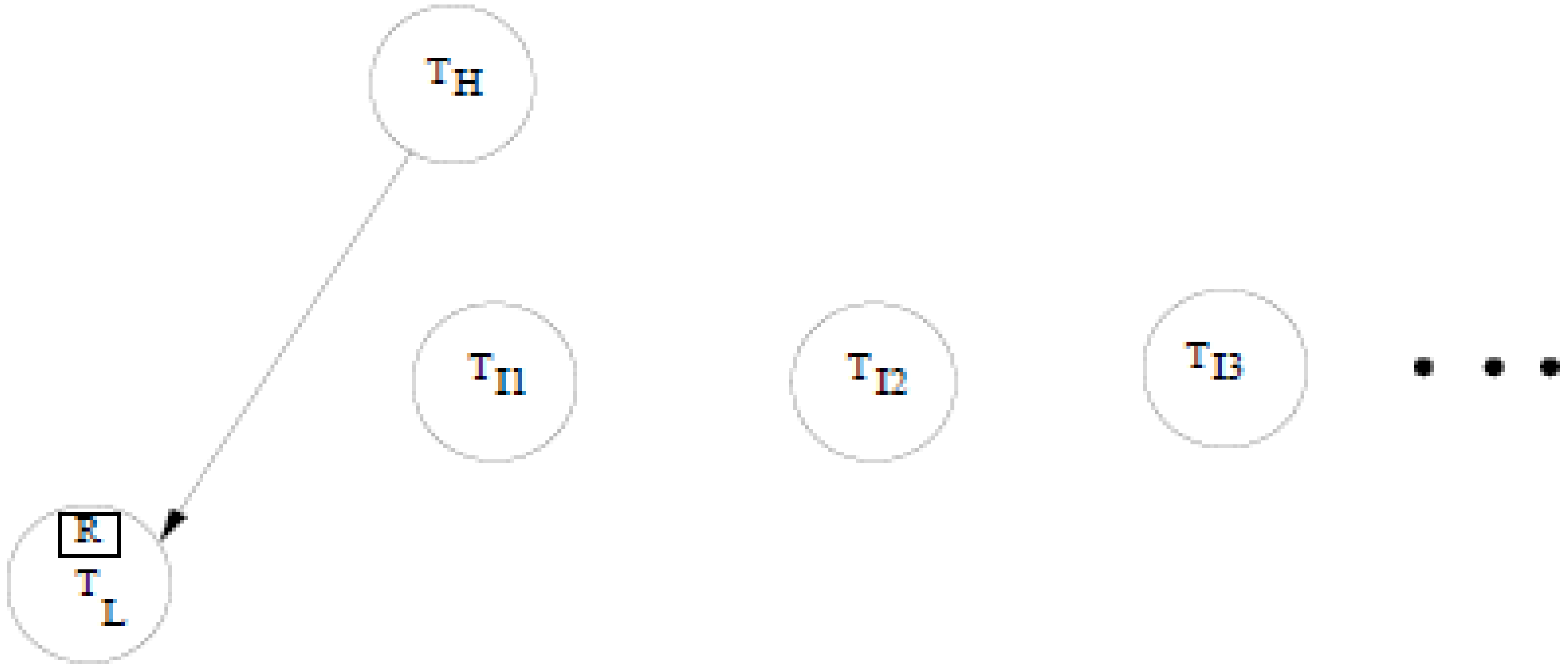


Figure 1: Unbounded Priority Inversion

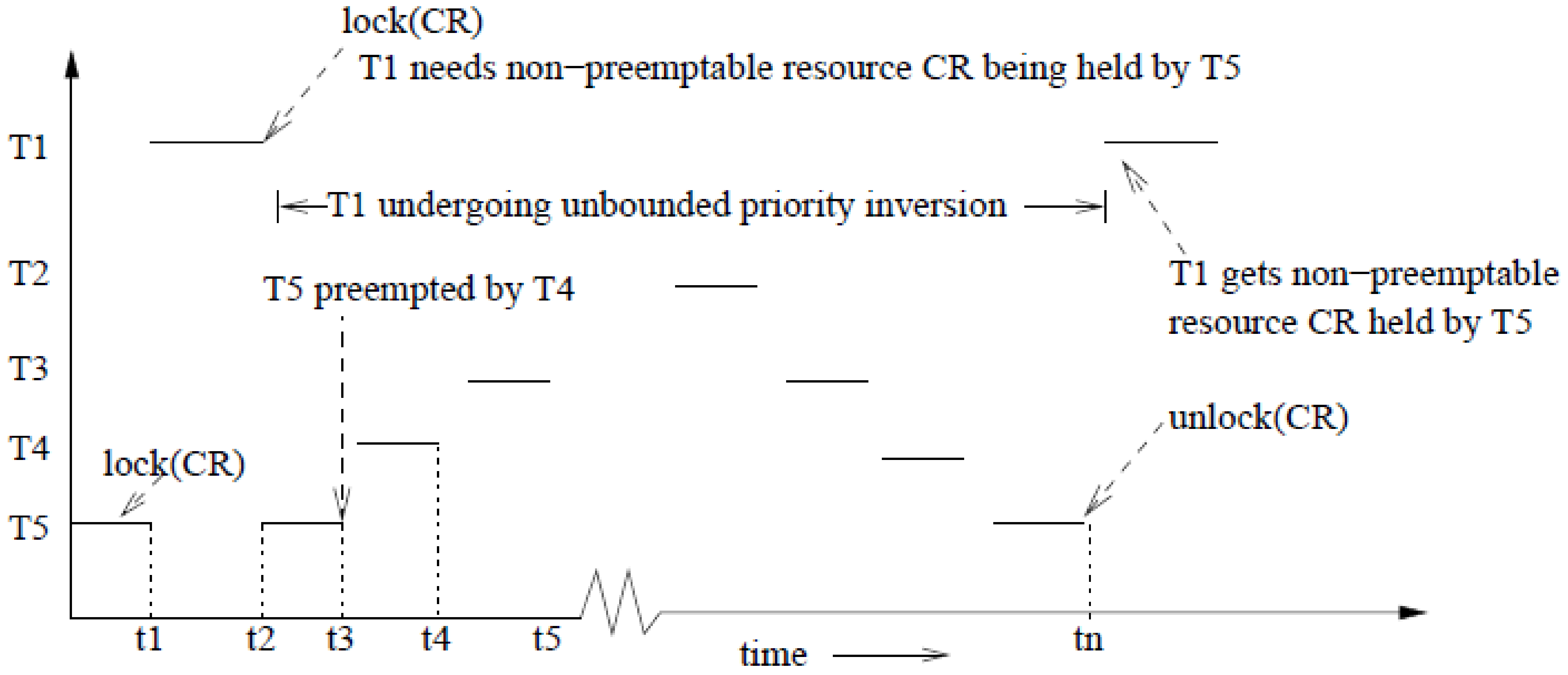
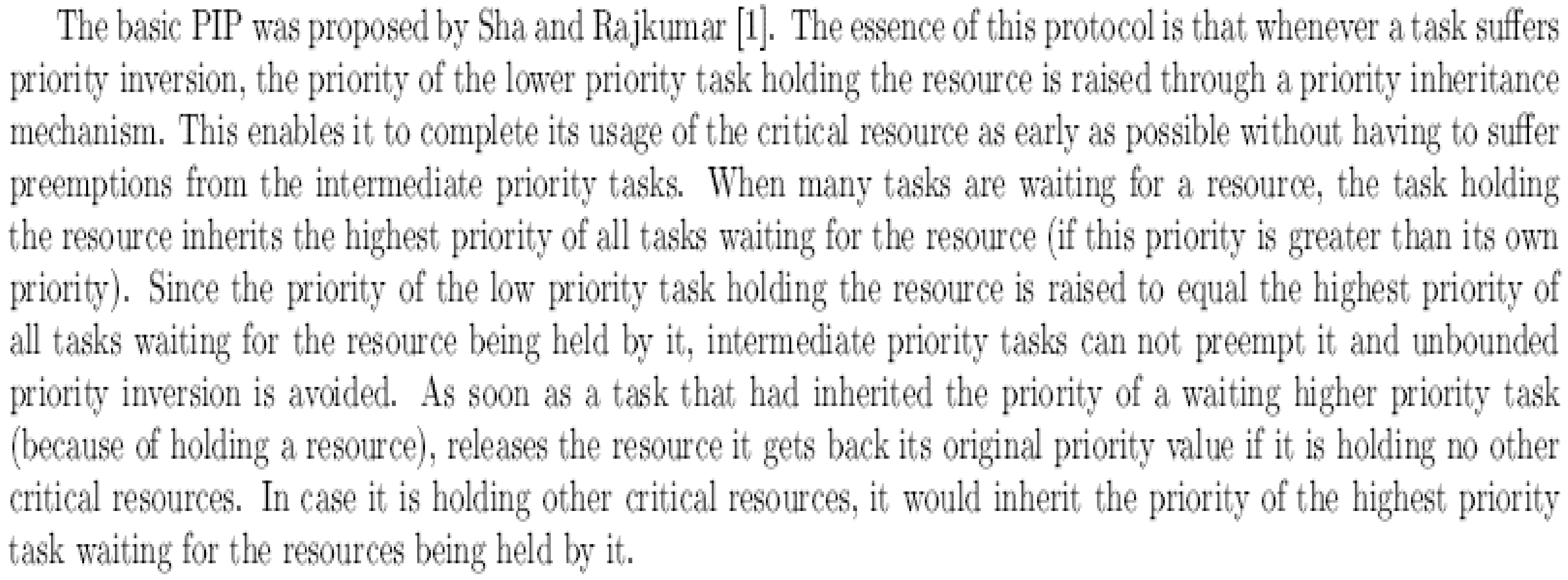

lock(CR)
T1 needs non−preemptable resource CR being held by T5
T1
T1 undergoing unbounded priority inversion
T2
T5 preempted by T4
T1 gets non−preemptable resource CR held by T5
T3
unlock(CR)
T4
lock(CR)
T5
t1
t2
t3
t4
t5
time
tn

Figure 2: Unbounded Priority Inversion

Let us now examine more closely what is meant by unbounded priority inversion and how it arises through a simple example. Consider a real-time application having a high priority task T_H and a low priority task T_L. Assume that T_H and T_L need to share a critical resource R. Besides T_H and T_L, assume that there are several tasks T_{I1}, T_{I2}, T_{I3}, ... that have priorities intermediate between T_H and T_L, and do not need the resource R in their computations. These tasks have schematically been shown in Fig. 1. Assume that the low priority task (T_L) starts executing at some instant and locks the non-preemptable resource as shown in Fig. 1. Suppose soon afterward, the high priority task (T_H) becomes ready, preempts T_L and starts executing. Also assume that it needs the same non-preemptable resource. It is clear that T_H would block for the resource as it is already being held by T_L and T_L would continue its execution. But, the low priority task T_L may be preempted (from CPU usage) by other intermediate priority tasks (T_{I1}, T_{I2}, ...) which become ready and do not require R. In this case, the high priority task (T_H) would have to wait not only for the low priority task (T_L) to complete its use of the resource, but also all intermediate priority tasks (T_{I1}, T_{I2}, ...) preempting the low priority task to complete their computations. This might result in the high priority task having to wait for the required resource for a considerable period of time. In the worst case, the high priority task might have to wait indefinitely for a low priority task to complete its use of the resource in the face of repeated preemptions of the low priority tasks by the intermediate priority tasks not needing the resource. In such a scenario, the high priority task is said to undergo *unbounded priority inversion*.

Priority Inheritance Protocol



The basic PIP was proposed by Sha and Rajkumar [1]. The essence of this protocol is that whenever a task suffers priority inversion, the priority of the lower priority task holding the resource is raised through a priority inheritance mechanism. This enables it to complete its usage of the critical resource as early as possible without having to suffer preemptions from the intermediate priority tasks. When many tasks are waiting for a resource, the task holding the resource inherits the highest priority of all tasks waiting for the resource (if this priority is greater than its own priority). Since the priority of the low priority task holding the resource is raised to equal the highest priority of all tasks waiting for the resource being held by it, intermediate priority tasks can not preempt it and unbounded priority inversion is avoided. As soon as a task that had inherited the priority of a waiting higher priority task (because of holding a resource), releases the resource it gets back its original priority value if it is holding no other critical resources. In case it is holding other critical resources, it would inherit the priority of the highest priority task waiting for the resources being held by it.

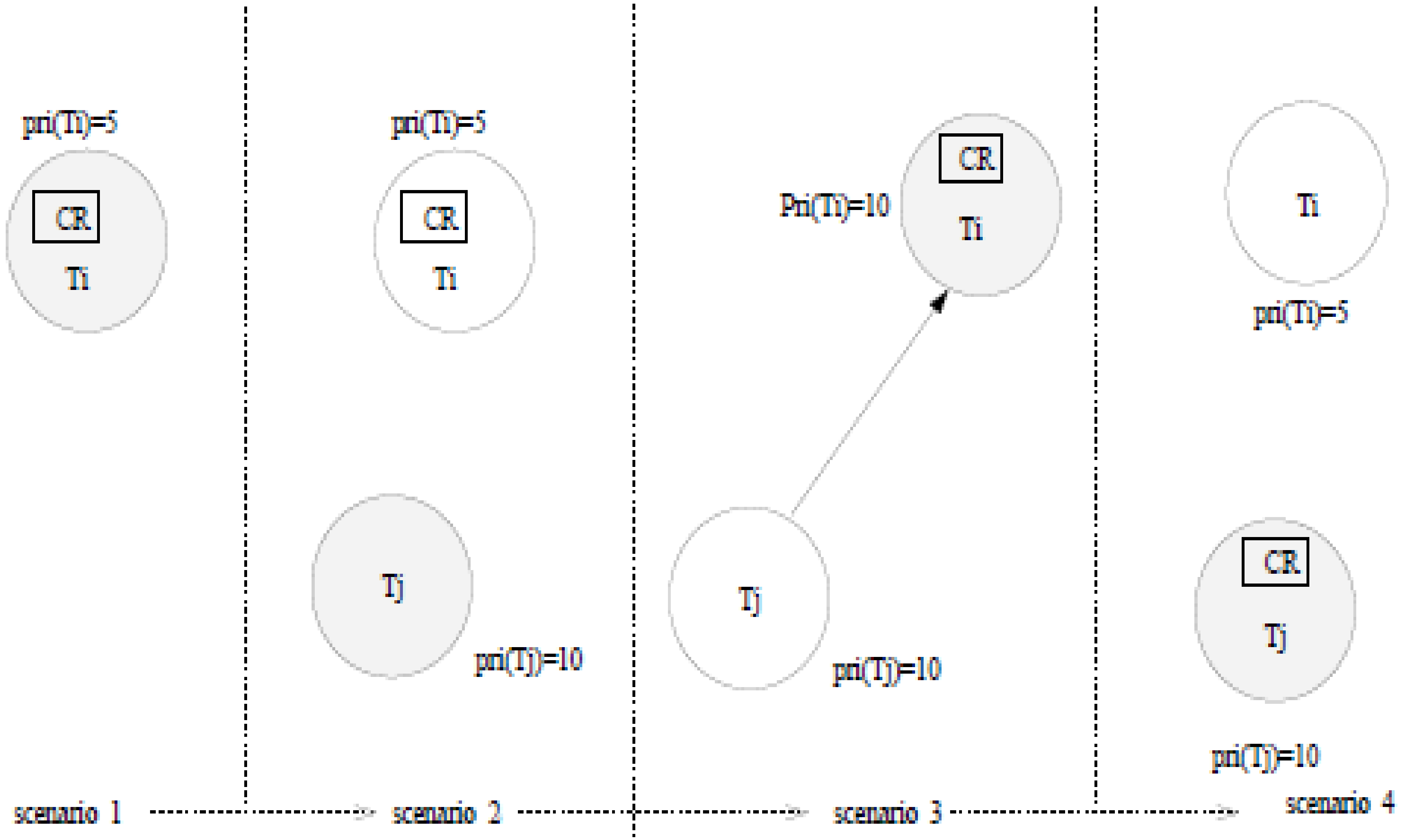


Figure 3: Snapshots Showing Working of Priority Inheritance Protocol

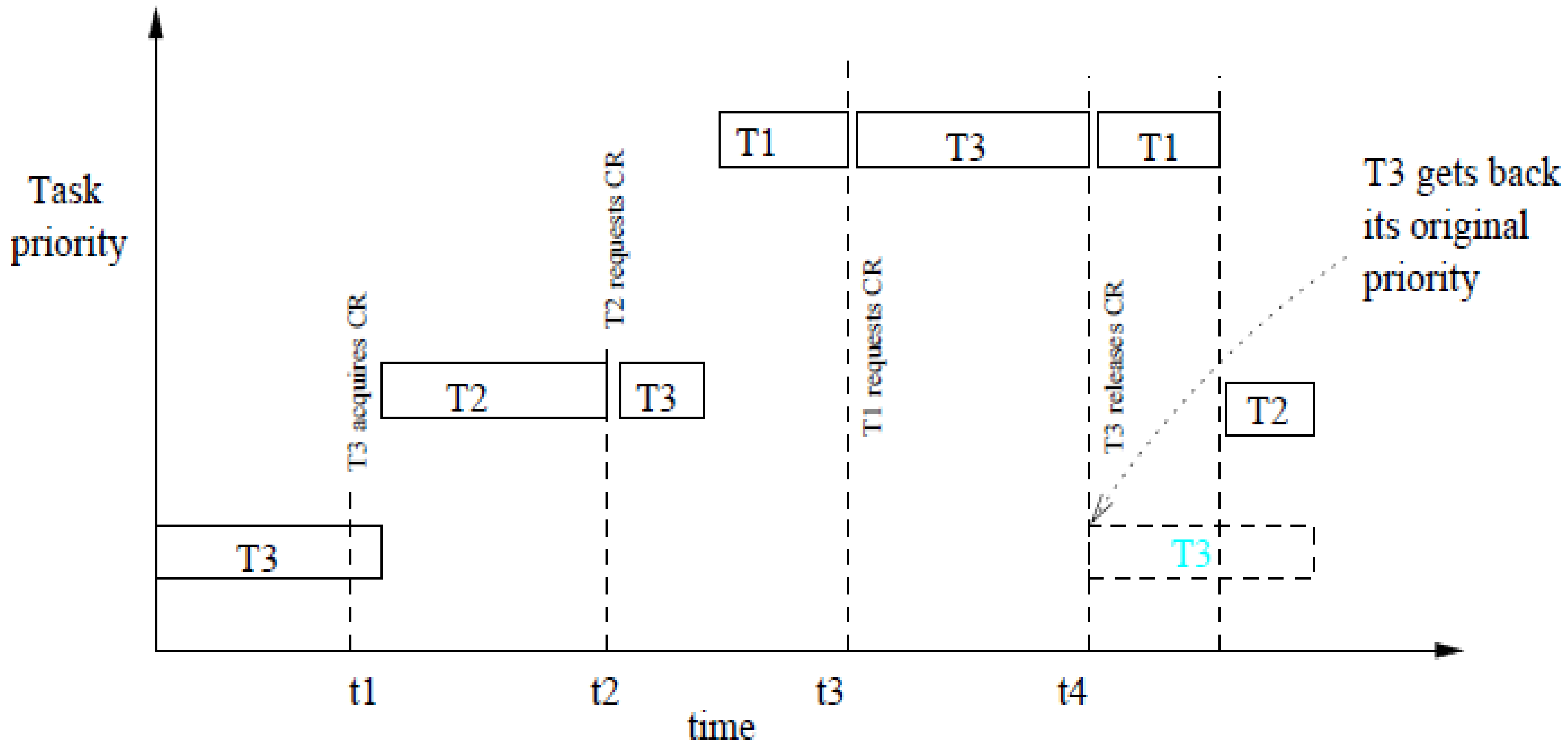


Figure 4: Priority Changes Under Priority Inheritance Protocol

PIP is susceptible to chain blocking and it does nothing to prevent deadlocks.

Deadlock: The basic priority inheritance protocol (PIP) leaves open the possibility of deadlocks. In the following we illustrate how deadlocks can occur in PIP using an example. Consider the following sequence of actions by two tasks T_1 and T_2 which need access to two shared critical resources CR1 and CR2.

```
T1:  Lock CR1, Lock CR2, Unlock CR2, Unlock CR1
T2:  Lock CR2, Lock CR1, Unlock CR1, Unlock CR2
```

Assume that T_1 has higher priority than T_2. T_2 starts running first and locks critical resource CR2 (T_1 was not ready at that time). After some time, T_1 arrives, preempts T_2, and starts executing. After some time, T_1 locks CR1 and then tries to lock CR2 which is being held by T_2. As a consequence T_1 blocks and T_2 inherits T_1's priority according to the priority inheritance protocol. T_2 resumes its execution and after some time needs to lock the resource CR1 being held by T_1. Now, T_1 and T_2 are both deadlocked.

Chain Blocking: A task is said to undergo chain blocking, if each time it needs a resource, it undergoes priority inversion. Thus if a task needs *n* resources for its computations, it might have to undergo priority inversions n times to acquire all its resources. Let us now explain how chain blocking can occur using the example shown in Fig. 5. Assume that a task T_1 needs several resources. In the first snapshot (shown in Fig. 5), a low priority task T_2 is holding two resources CR1 and CR2, and a high priority task T_1 arrives and requests to lock CR1. It undergoes priority inversion and causes T_2 to inherit its priority. In the second snapshot, as soon as T_2 releases CR1 its priority reduces to its original priority and T_1 is able to lock CR1. In the third snapshot, after executing for some time, T_1 requests to lock CR2. This time it again undergoes priority inversion since T_2 is holding CR2. T_1 waits until T_2 releases CR2. From this example, we can see that chain blocking occurs when a task undergoes multiple priority inversions to lock its required resources.

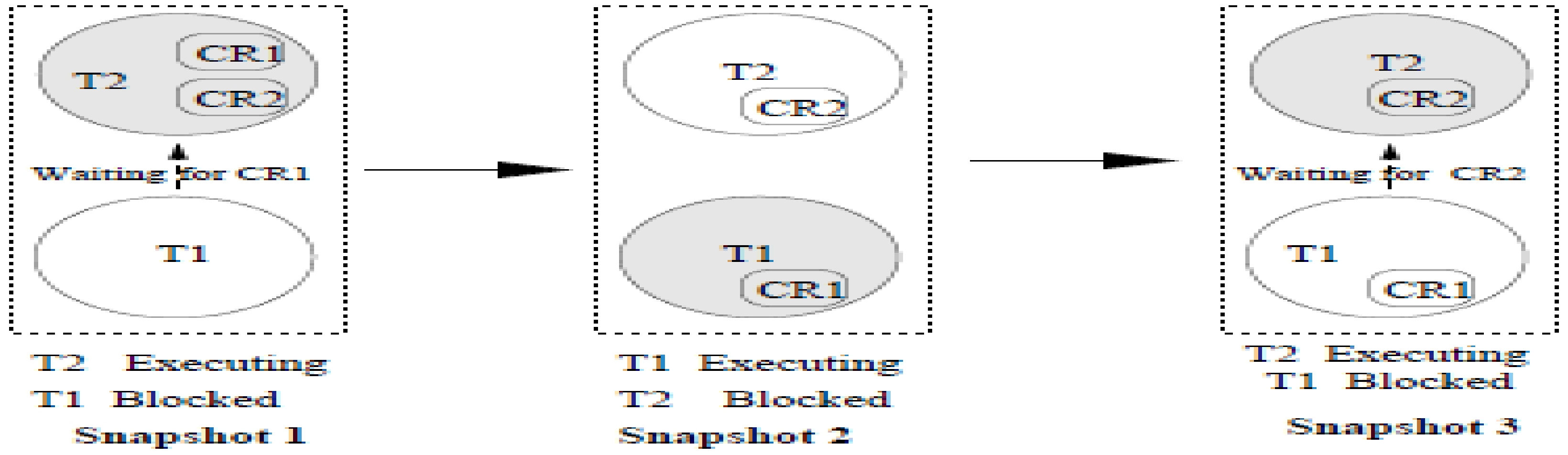


Figure 5: Chain Blocking in Priority Inheritance Protocol

R
T
O
S

Inter-Task Communication :

➢Inter-task or inter process communication in μC/OS takes place using:

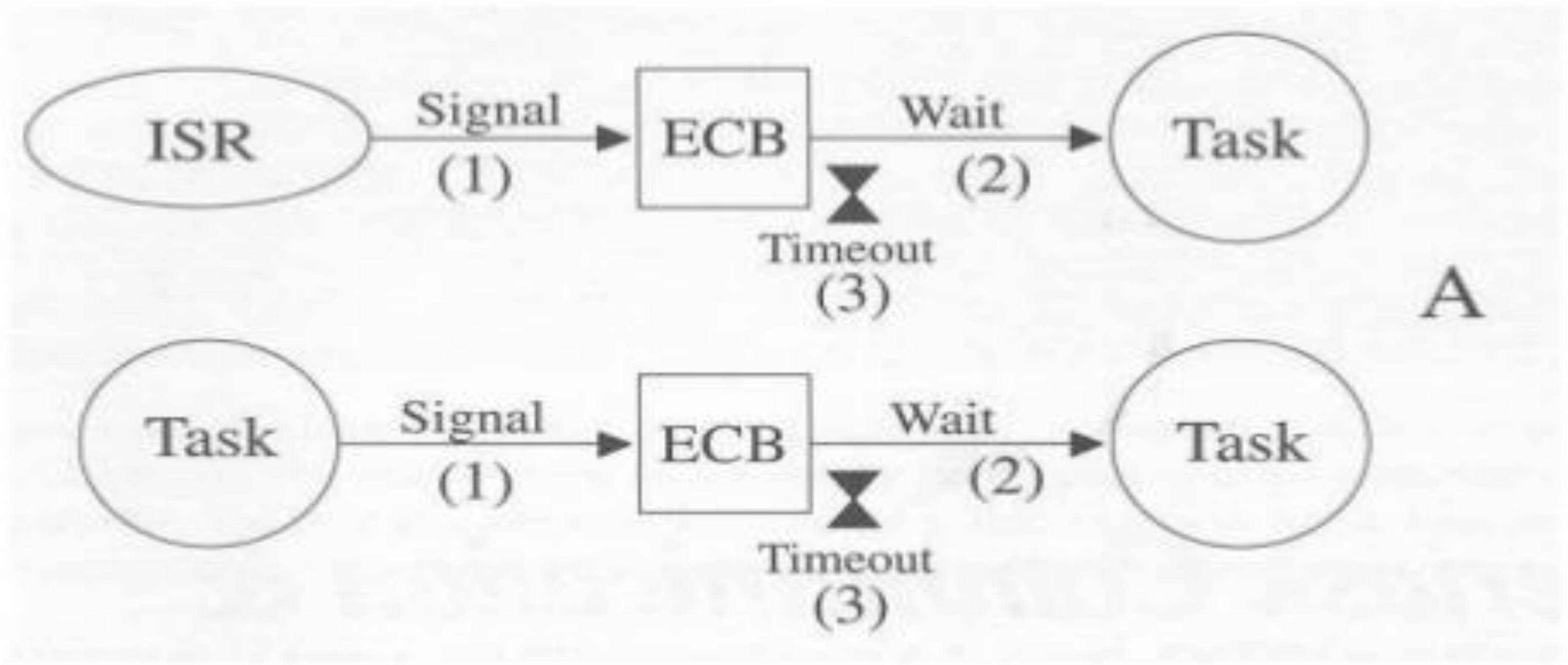
- ✓Semaphores
- ✓Message mailbox
- ✓Message queues

➢Tasks and Interrupt service routines (ISR). They can interact with each other through an ECB (event control block).

RTOS

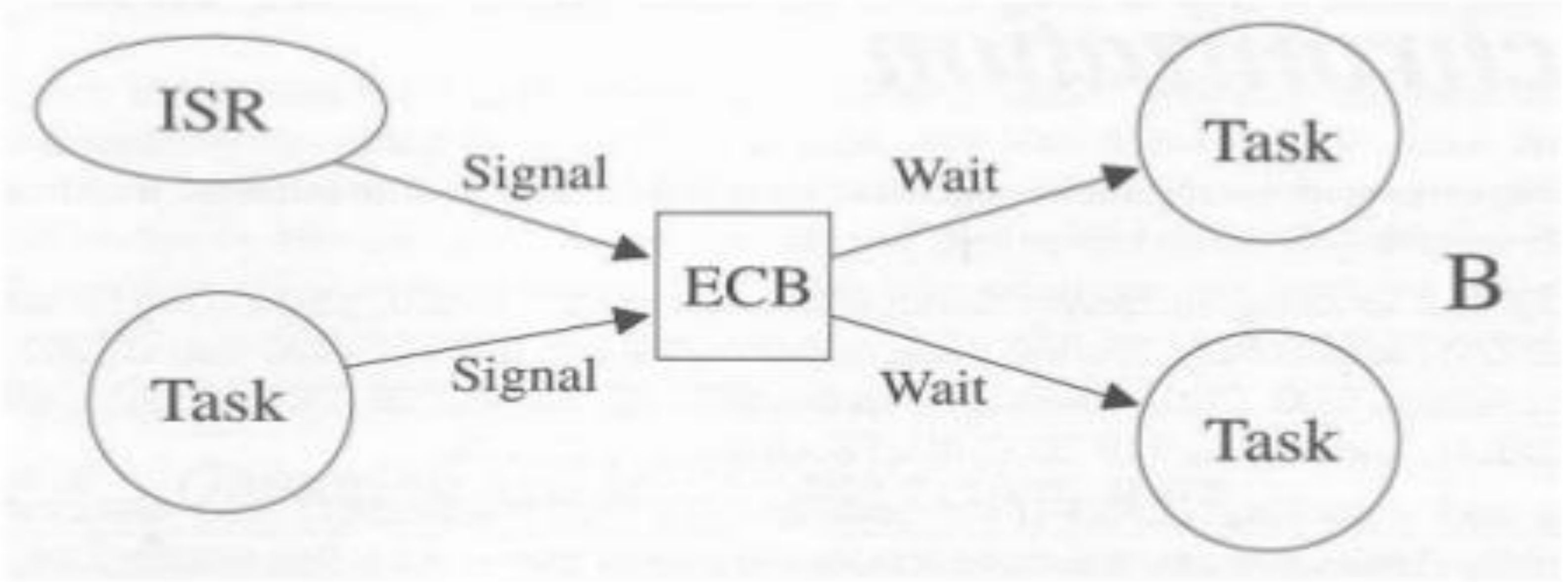
Inter-Task Communication :

➢Single task waiting



Inter-Task Communication :

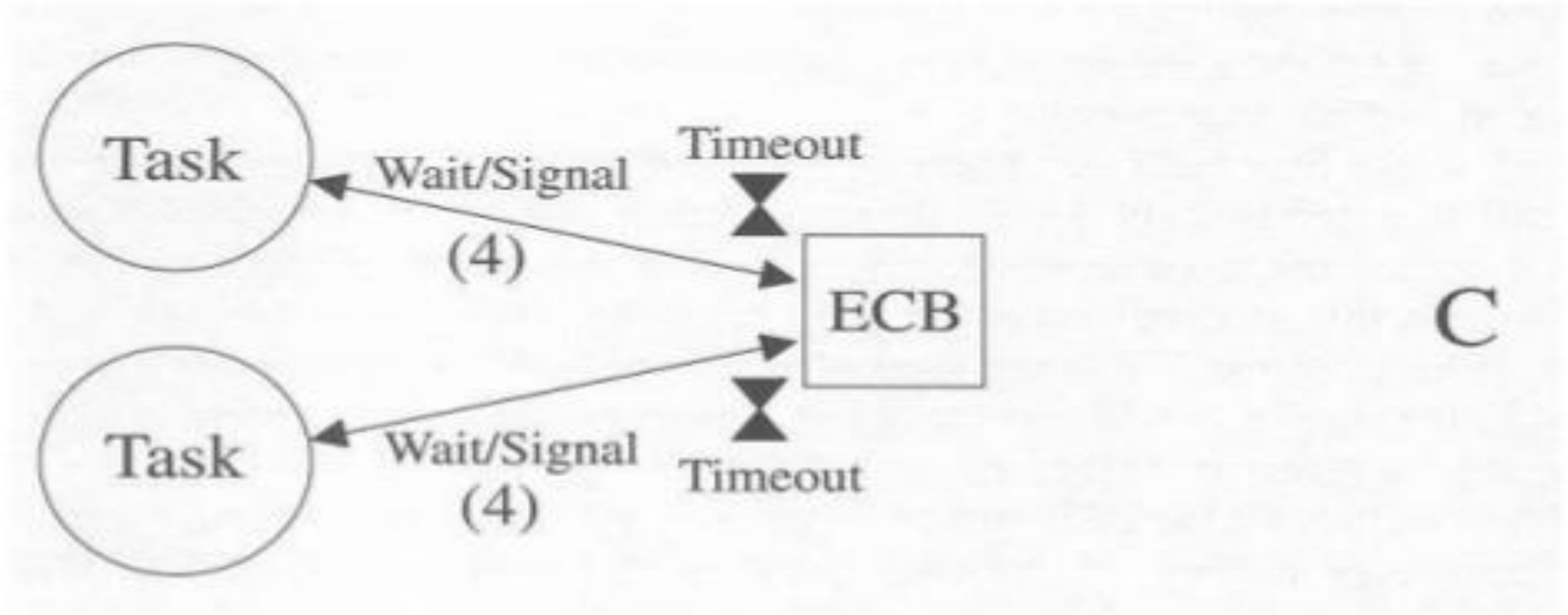
➢Multiple tasks waiting and signaling



R
T
O
S

Inter-Task Communication :

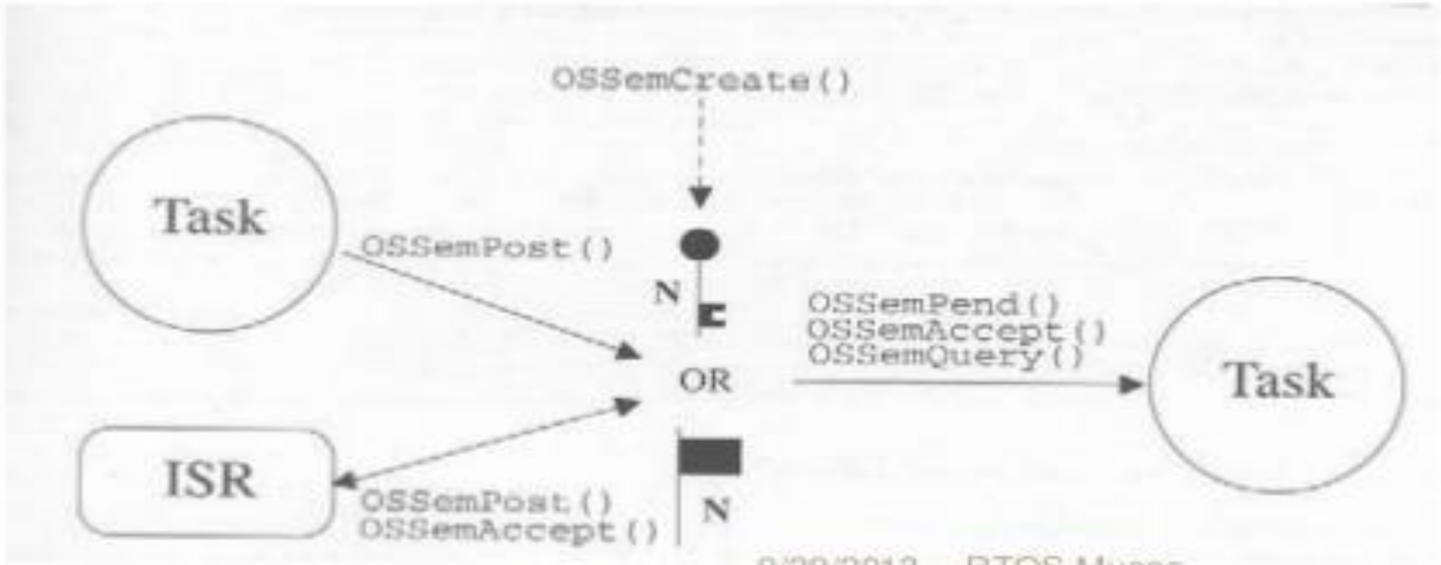
➢Tasks can wait and signal along with an optional time out



R T O S

Inter-Task Communication :

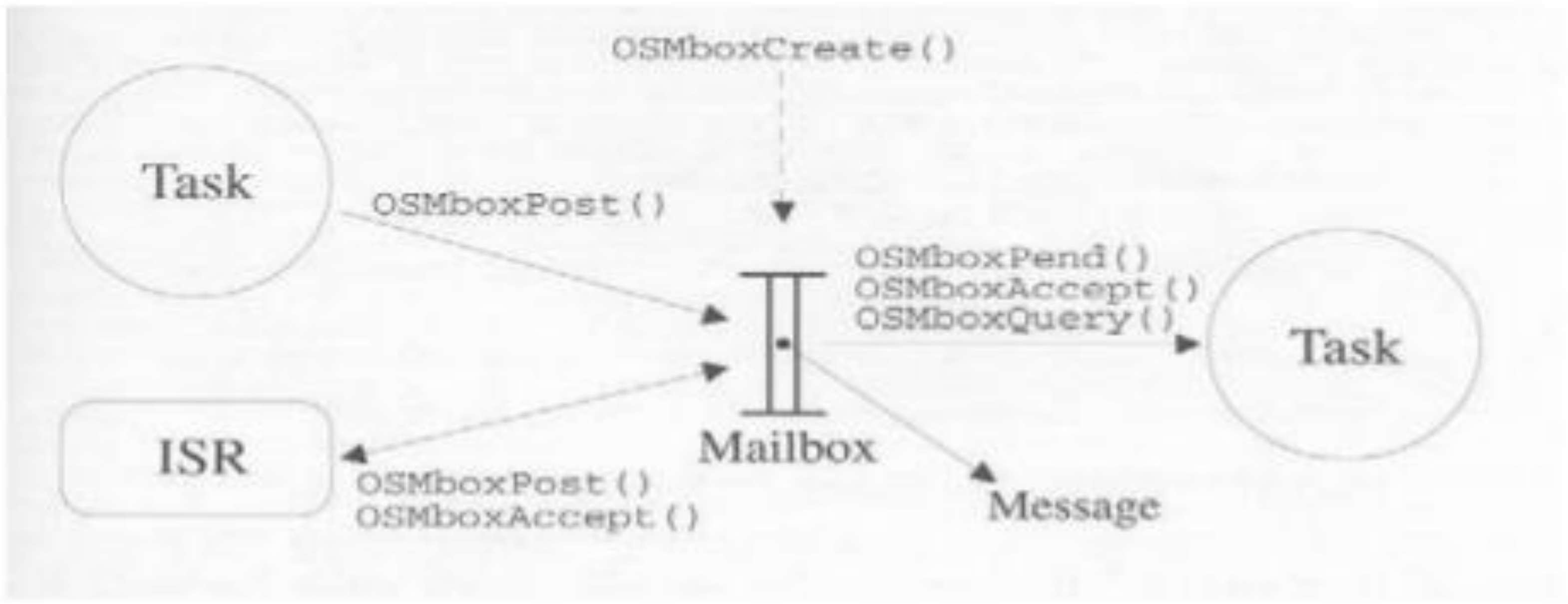
- ➢μC/OS-II semaphores consist of two elements
  - ✓16-bit unsigned integer count
  - ✓list of tasks waiting for semaphore
- ➢μC/OSII provides
  - ✓Create, post, pend accept and query services



R T O S

Inter-Task Communication :

➢μC/OS-II message-mailboxes: an μC/OSII object that allows a task or ISR to send a pointer sized variable (pointing to a message) to another task.



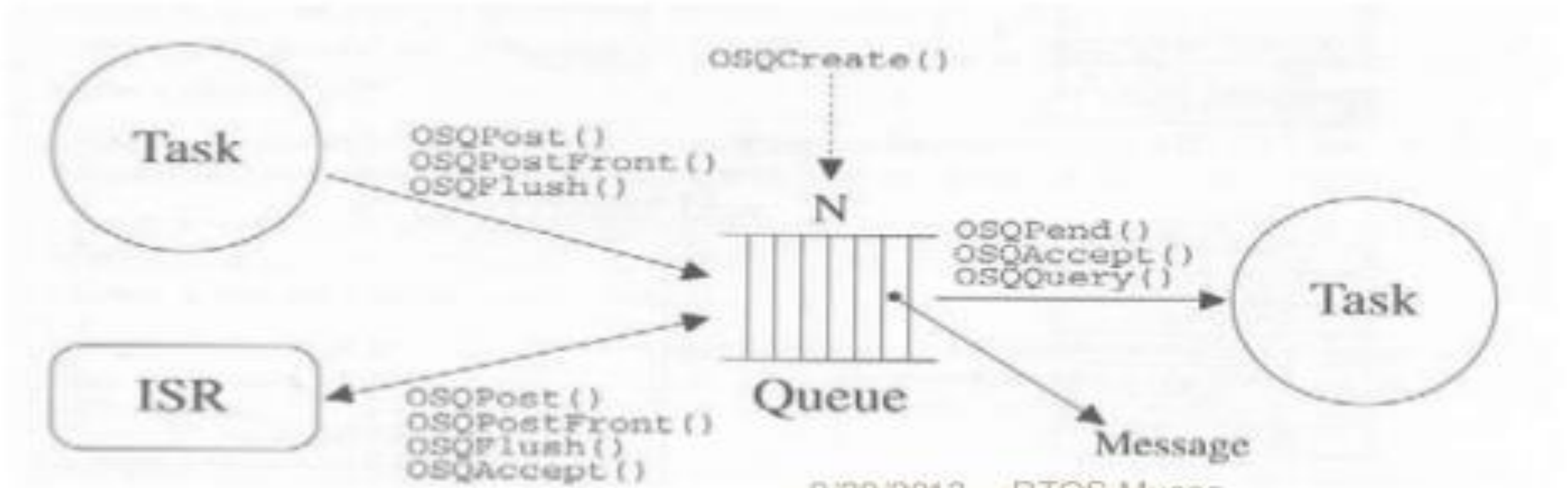
R
T
O
S

Inter-Task Communication :

➢ µC/OS-II message-queues

➢Available services: Create, Post (FIFO), PostFront (LIFO), Pend, Accept, Query, Flush

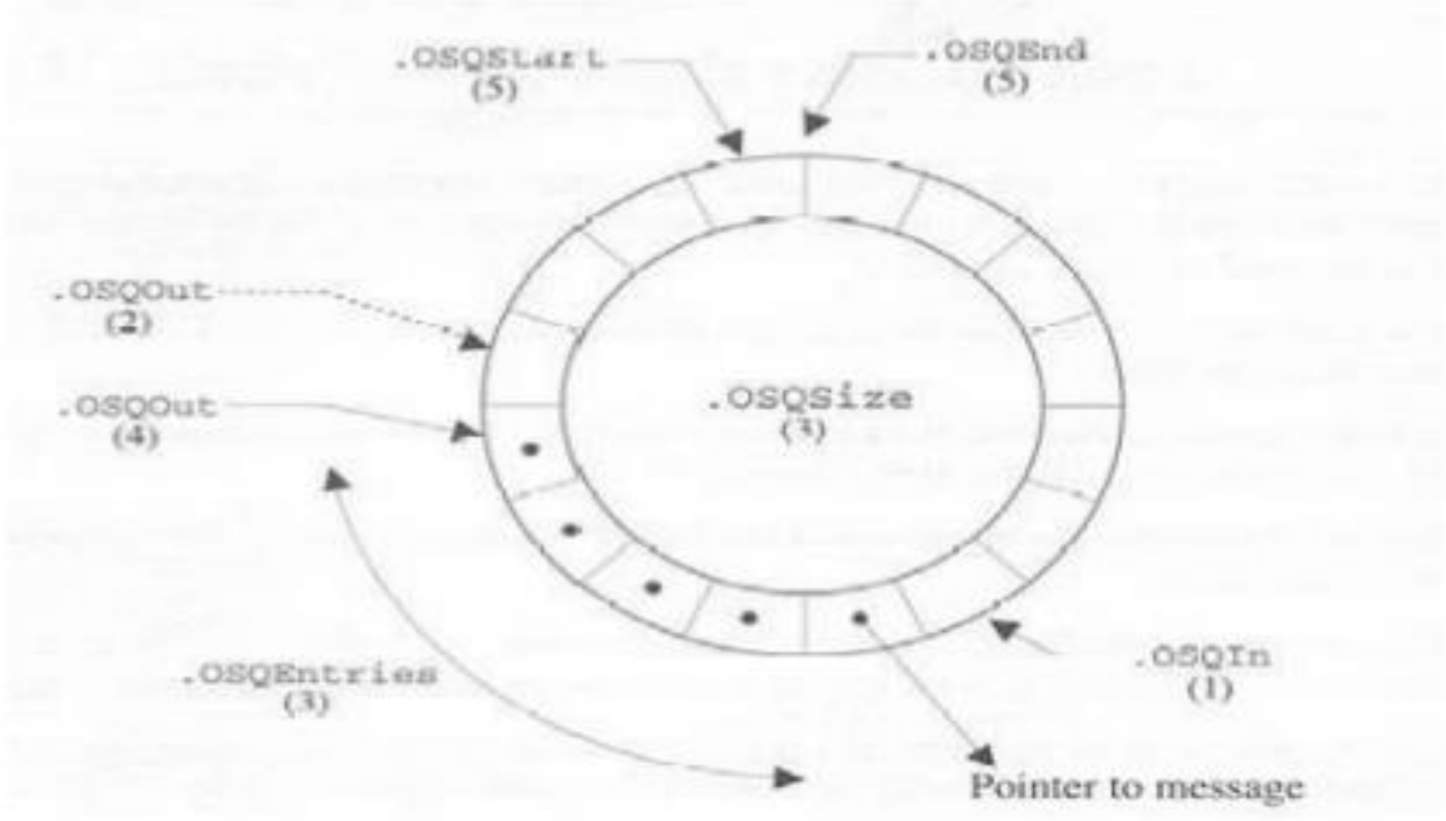
▪ N = #of entries in the queue: Queue full if Post or PostFront called N times before a Pend or Accept



R T O S

Inter-Task Communication :

➢µC/OS-II message-queues organized as circular buffers.



R
T
O
S

Writing Applications Under µC/OS -II:

Tasks running under a multitasking kernel should be written in one of two ways:

1. A non-returning forever loop. For example:

```
void Task (void *)
{
DoInitStuff();
 while (1)
{ do this;
  do that;
  do the other thing;
  call OS service ();   // e.g. OSTimeDelay, OSSemPend, etc.
 }
}
```

R
T
O
S

2. A task that deletes itself after running.

For example:

```
void Task (void *)
 { do this;
  do that;
  do the other thing;
  call OS service ();  // e.g. OSTimeDelay, OSSemPend, etc.
  OSTaskDelete();    // Ask the OS to delete the task
}
```