The Super Loop

```
int main(void) {
  /* Initialize system */

  while (1) {
    /* Periodic Tasks */
    ADC_Read();
    SPI_Read();
    USB_Packet();
    Audio_Decode();
    File_Write();
  }
}
```

"No interrupts"
"Polling"

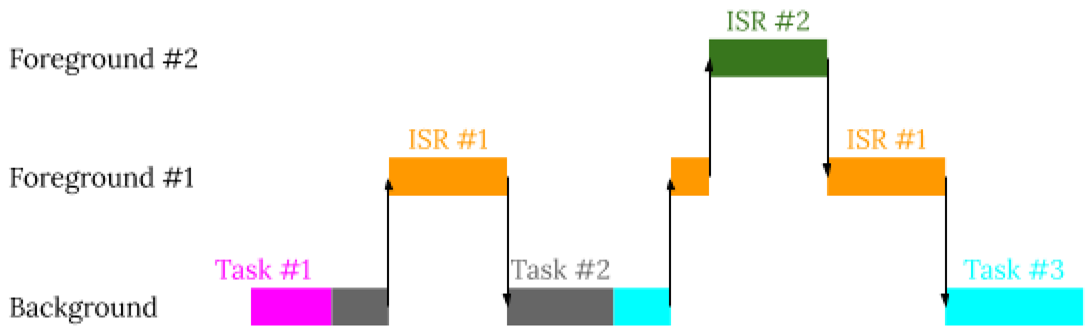
+ Simple
  Minimal HW resources
  Highly portable

- Inaccurate timing
  High power consumption

1 - Determinism
2 - Responsiveness
3 - Polling periodicity

Foreground/Background

```
void USB_ISR(void) {
  Clear interrupt;
  Read packetM
}
```



+ No upfront cost
  Minimal training required
  No need to set aside resources to accommodate RTOS

- Difficult to ensure that each operation will meet its deadline
  High-priority code must be placed in the foreground
  Problems can arise when code is maintained by multiple developers
  Even in projects with a single developer, expanding the application can prove difficult

RTOS

RT = Correct function @ Correct Time
OS = HW + SW manager
RTOS = HW + SW manager that can help us ensure having correct function @ correct time

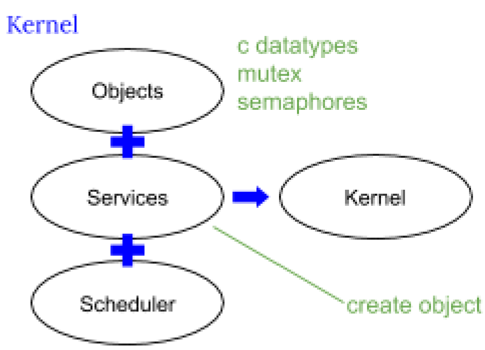
Real-Time System
They can be:
- Hard
- Firm
- Soft

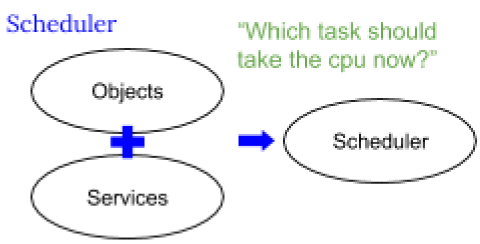
Fast software is not necessarily real-time software

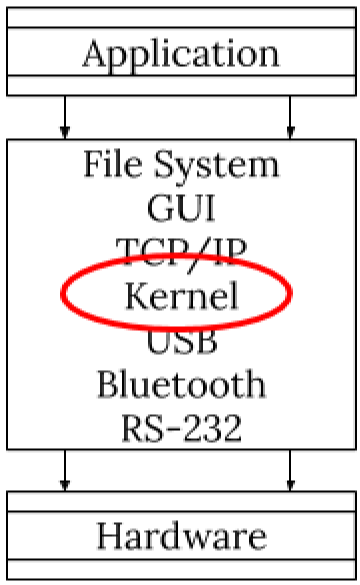
Determinism is a desirable quality in real-time software

Software that is deterministic has a bounded response time to events

RTOS = Kernel + …







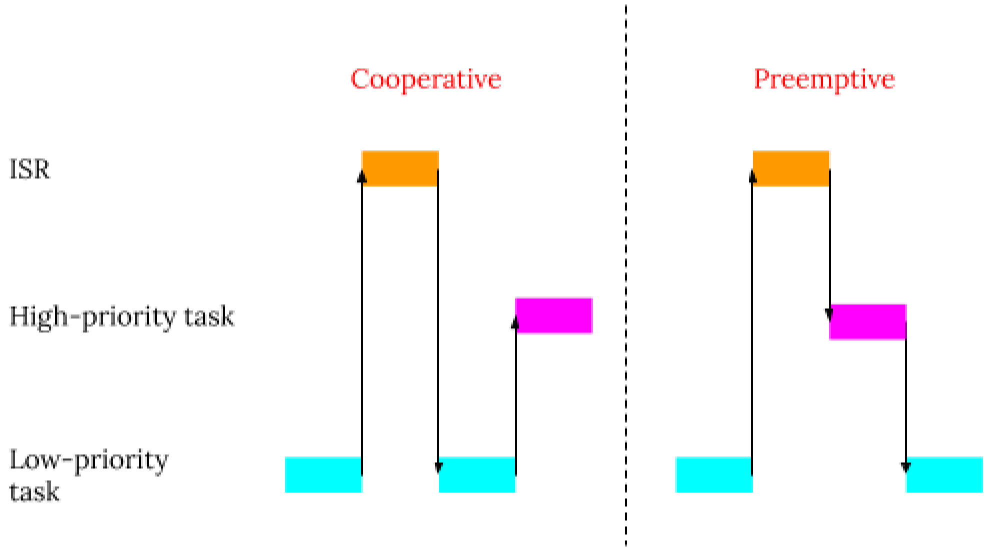
Benefits
Developers who use RTOS are freed from implementing a scheduler and related services.

Typical applications that incorporate RTOS are much easier to expand.

The best RTOS have undergone thorough testing

Types of Kernels

- Cooperative Kernel
- Preemptive Kernel



To RTOS or not to RTOS

Technical and Managerial decision based on many factors includin:

- Expected size of the code
- Complexity of the hardware
- Timing requirements
- Size of the development team

Foreground/Background systems that include any of the following should consider migrating to an RTOS

- Excessive Polling
- Counters for controlling execution rates
- Repetitive function calls in the main loop

```
void ADC_Read(void) {
  while((ADC_ConvComplete()) == 0) {
    ;
  }
  Process analog value;
}
```

Excessive Polling

```
while (1) {
  ADC_Read();
  if((i % 8192) == 0) {
    SPI_Read();
  }
  i++;
}
```

Counters

```
while(1) {
  ADC_Read();
  LCD_Update();
  SPI_Read();
  USB_Packet();
```

Repetitive Function calls

```
    LCD_Update();
    Audio_Decode();
    File_Write();
    LCD_Update();
   }
```

main()

- First function executed in μC/OS-II-based application
- The routines that main() calls lay the ground for multitasking

```
int main(void) {
  OS_Init();
  /* Create @ least 1 task */
  OS_Start();
}
```

OSInit()

- OSInit() must be invoked before any of μC/OS-II's services are used
- This function initialize the operating system's data structures
- μC/OS-II's internal tasks are created by OSInit()
- The extent of initializations depends on configuration constants

μC/OS-II's internal Tasks

Idle Task

- Runs when other tasks are unable to do so
- Invokes a hookfunction OSTaskIdleHook()
- Is automatically assigned the lowest possible priority

Statistics Task

- Runs periodically
- Records CPU utiliation
- Is capable of checking the size of stacks
- Has a slightly higher priority than the Idle task

Creating the First Task

- After OSInit() has been invoked, tasks can be created
- Most applications create just one task in main()
    - If multiple tasks are created, the statistics task will not function properly
- Additional tasks can be created at essentially anytime

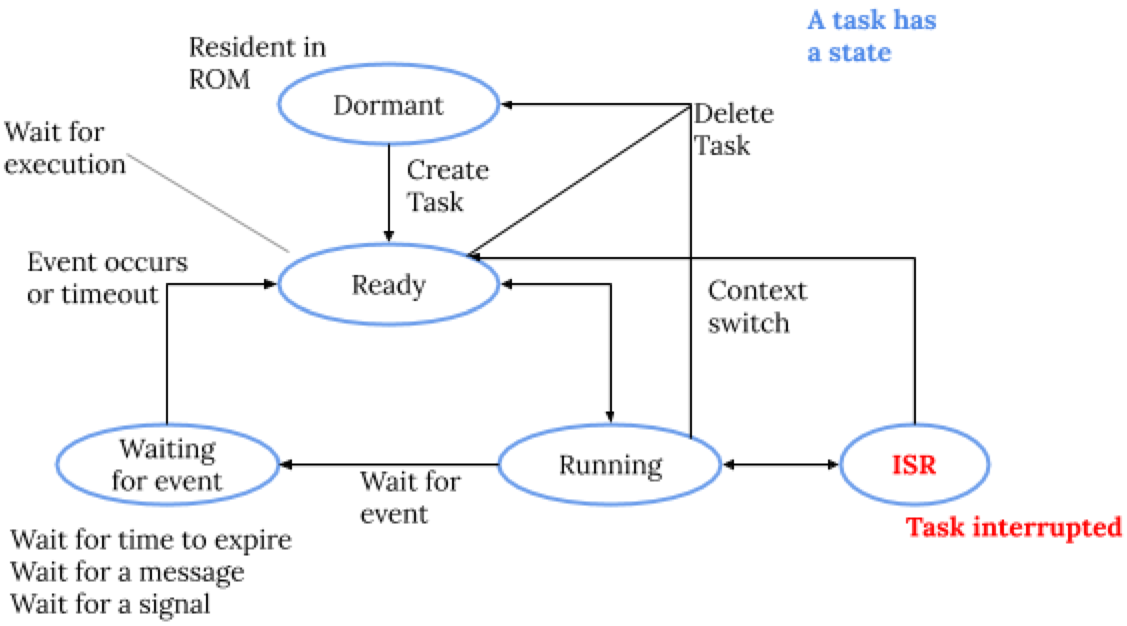
OSStart()

- main() must call OSStart() in order to initiate multitasking
- OSStartHighRdy() which is an assembly-language routine invoked by OSStart() runs the first task
- Unless errors occurs, OSStart() should be the last function called from main()

Task

- There are few required components
  - Each task is given its own stack space
  - A task must be assigned a priority
- Normally, a task involves an infinite loop
  - Task cannot return
- Initializations might precede the loop

```
void App_TaskExample(void *p_data) {
  Task initialiation;
  for(;;) {
    Work toward task's goal;
    Wait for events;
  }
}
```



Task stack

- Each task should have its own stack
  - It must be declared as OS_STK and made of consistent and contiguous memory
  - Can be allocated statically or dynamically
    - OS_STK MyStack[???]

Stack sizing

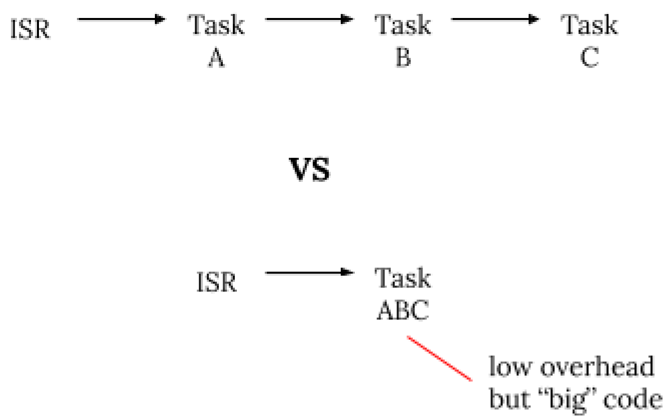
- Not all tasks requires the same amount of stack space
- Application developers are responsible for sizing their stacks

Task Control Block

- TCB's for task management
- TCB's reside in RAM
- Every task is assigned a TCB when created
  - The task's priority
  - The state of the task
  - A pointer to the task's stack

Application Partitioning

- Non-trivial
- A poorly partitioned application may fail to meet performance requirements
- Look for activities that can execute in parallel
- Beware of excessive communication and large number of tasks
- Many methods exists



Divide and Conquer (appl. partitioning)

Divide

- Identify code based on the following:
  - IO bound
  - CPU bound
  - Periodicity
- Functions can be either IO or CPU bounded
- Functions can be periodic or aperiodic

Conquer (Groups)

- Group functions into tasks based on the following:
  - Function cohesion
  - Time cohesion
  - Periodic cohesion
- Function cohesion → single task or sequential tasks
- Time cohesion → Separate parallel tasks
- Periodic cohesion → Same task with counters or different tasks

Application Partitioning Sanity Check

For each task regardless of the execution schedule

$$T \le D \le P$$

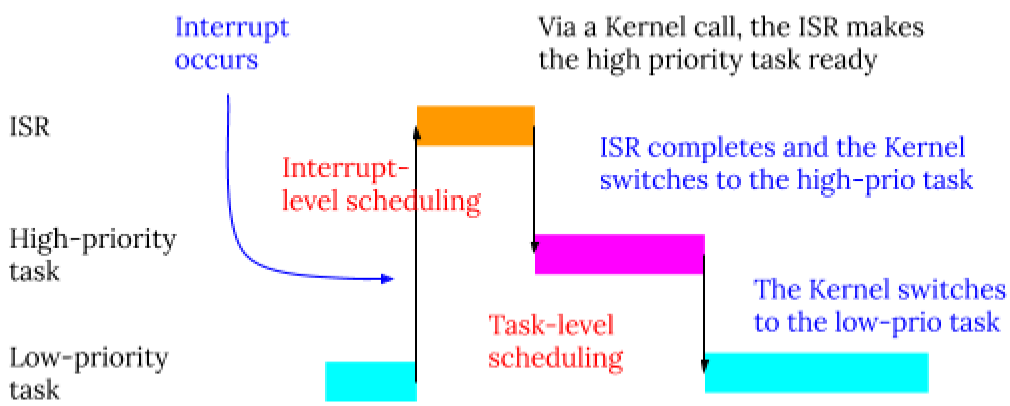
T = WCET (worst case execution time)
D = Deadline
P = Period

Assigning Priorities
Gomaa Criterion

finish quickly ↓ finish correctly ↓

Urgent	Critical	Priority
No	No	Lowest
No	Yes	Lower
Yes	No	Higher
Yes	Yes	Highest

RTOS scheduling



Delay Functions

```
void  OSTimeDly       (INT16U    ticks);
INT8U OSTimeDlyHMSM   (INT8U     hours,
                       INT8U     minutes,
                       INT8U     seconds,
                       INT16U    milli);
```

Shared Resources

- A global variable or data structure that is used by multiple contexts
- Peripheral devices are shared resources
- Race condition, major problem in multi-context execution

Protecting Shared Resources

- Shared resources are not atomically accessed and will never be atomically accessed
- Shared resources should be exclusively accessed
- Locking is the used mechanism
- Every shared resource is associated with a lock
    - A single lock can be used with multiple shared resources

- Access is granted to a context if it can open the lock

Disabling Interrupts

- 2 macros provided OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
- Used by µC/OS-II code
- Can be used by your application
  - Not a good coding style
  - Application may crash
- Can be implemented in 1 of 3 ways and selected through configuration

~~Disabling Interrupts Method 1~~

...

~~Disabling Interrupts Method 2~~

...

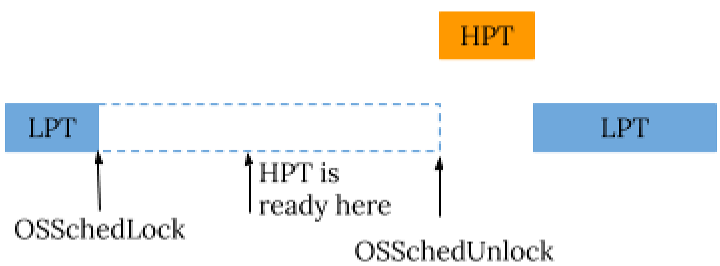
Disabling Interrupts Method 3

- Compiler must support
  - Storing PSW in a C variable
  - Loading PSW from a C variable
  - Disabling interrupts in from C

```
void App_TaskExample(void* p_arg) {
#if OS_CRITICAL_METHOD==3
  OS_CPU_SR cpu_sr = 0;
#endif
  while(1) {
    OS_ENTER_CRITICAL();
    Access shared resource;
    OS_EXIT_CRITICAL();
  }
}
```

Schedule Locking

- A task can lock the scheduler to keep control of the cpu, even if there are higher priority tasks ready
- During scheduler locking, your application should not call any service that suspends execution
  - Application will crash!



Semaphores

- Semaphores are based on counters
- A semaphore can be classified as either binary or counting
- Have 2 operations:
  - Pend = while the semaphore's counter has a value of zero, allow other tasks to run
  - Post = Increment the semaphore's counter
- Semaphores are implemented with event control blocks (ECBs)
  - These structures are somewhat similar to TCBs
- ECBs are used to implement semaphores, mutexes, mailboxes, and queues

Semaphore API

```
OS_EVENT    *OSSemCreate      (INT16U     cnt);
void        OSSemPend         (OS_EVENT   *pevent,
                                INT16U     timeout,
                                INT8U      *perr);
INT8U       OSSemPost         (OS_EVENT   *pevent);
```

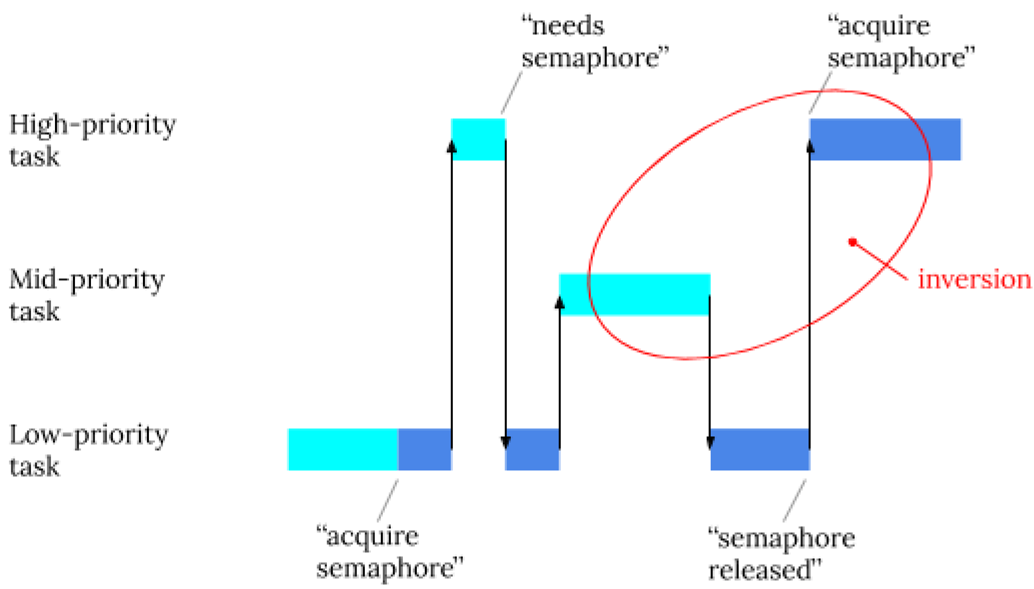
Semaphore Problems

1 - Starvation
2 - Dead look ⇒ Ordered locking
3 - Priority inversion

Deadlock problem

```
Task1() {                          Task2() {
 lock   1                           lock   2  ->  1
 lock   2                           lock   1  ->  2
 unlock 2                           unlock 1  ->  2
 unlock 1                           unlock 2  ->  1
}                                  }
```

Priority Inversion

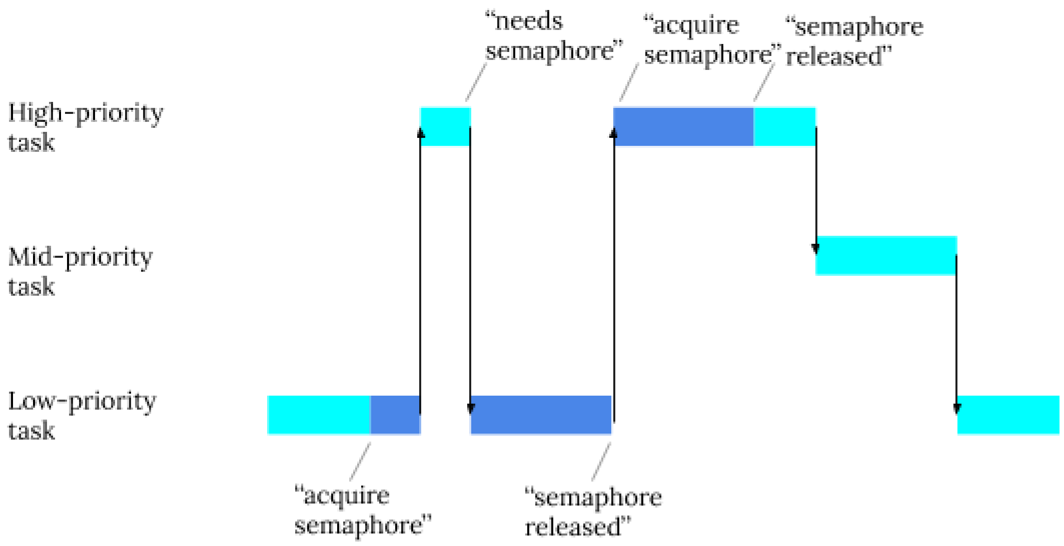


Mutex = Binary Semaphore used for Shared Resources Protection

+

Priority Inversion Solution
(Inheritance or ceiling)

Priority Inheritance



Mutex

- A mutex is yet another mechanism for protecting resources
- In µC/OS-II, mutexes provide built-in protection from priority inversion
    - A modified priority inheritance protocol is used
- Unlike a semaphore, a µC/OS-II mutex does not incorporate a counter
    - The mutex is either available or in use

```
OS_EVENT    *OSMutexCreate    (INT8U      prio,
                               INT8U      *perr);
void        OSMutexPend       (OS_EVENT   *pevent,
                               INT16U     timeout,
                               INT8U      *perr);
INT8U       OSMutexPost       (OS_EVENT   *pevent);
```

Task Interaction

- The tasks comprising a µC/OS-II-based application are not necessarily self-contained
- In order for the application's objectives to be met, tasks may need to interact with each other (and possibly with ISRs)
- A typical RTOS provides services that facilitate such interaction

Synchronizing a Task to an ISR

- Most applications must manage a collection of peripheral devices
- The interrupt service routines (ISRs), associated with system's peripheral devices should be kept brief
- In applications that incorporate a real-time kernel, ISRs can use synchroniation primitives to signal tasks
- Using a semaphore, a task can synchronize to another task or to an ISR!

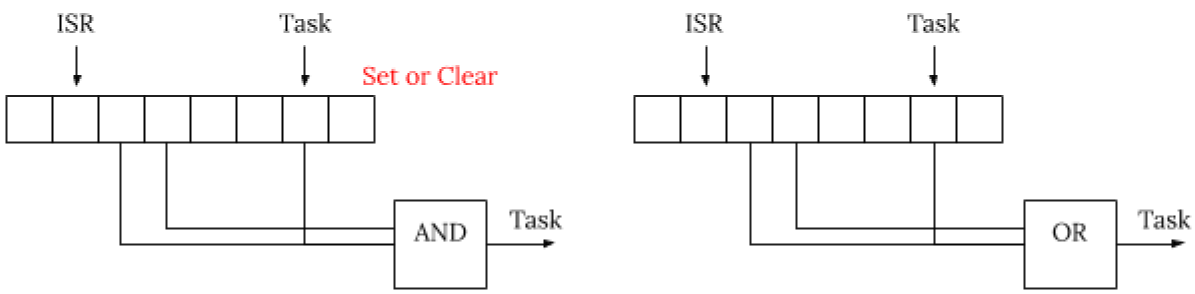
```
OS_EVENT *App_SemADC;
// Init code ...
App_SemADC = OSSemCreate(0);
```

```
void App_ISRADC(void) {
  Clear interrupt;
  OSSemPost(App_SemADC);
}
```

```
void App_TaskADC(void *arg) {
  perform init;
  while(1) {
    Start conversion;
    OSSemPend(App_SemADC, 0,
              &err);
    Process converted value;
  }
}
```

Event flags

- Using Event flags, a task can easily wait for multiple events to take place




```
OS_FLAG_GRP  *OSFlagCreate     (OS_FLAGS         flags,
                                 INT8U            *perr);
OS_FLAGS     OSFlagPend        (OS_FLAG_GRP      *pgrp,
                                 OS_FLAGS         flags,
                                 INT8U            wait_type,
                                 INT16U           timeout,
                                 INT8U            *perr);
OS_FLAGS     OSFlagPost        (OS_FLAG_GRP      *pgrp,
                                 OS_FLAGS         flags,
                                 INT8U            opt,
                                 INT8U            *perr);
```

Inter-Task Communication Services

- Tasks in a µC/OS-II-based application can send and receive messages using services that the kernel provides
- Application developers determine the content of these messages

Message Queues

- In µC/OS-II, message queues are circular buffers
  - The kernel manages each buffer
  - Through API functions, tasks can request the insertion or removal of messages
- A message is a void pointer
- When a task is waiting on a message, the kernel runs other tasks

```
OS_EVENT    *OSQCreate  (void       **start,
                         INT16U     size);
void        *OSQPend    (OS_EVENT   *pevent,
                         INT16U     timeout,
                         INT8U      *perr);
INT8U       OSQPost     (OS_EVENT   *pevent,
                         void       *pmsg);
```

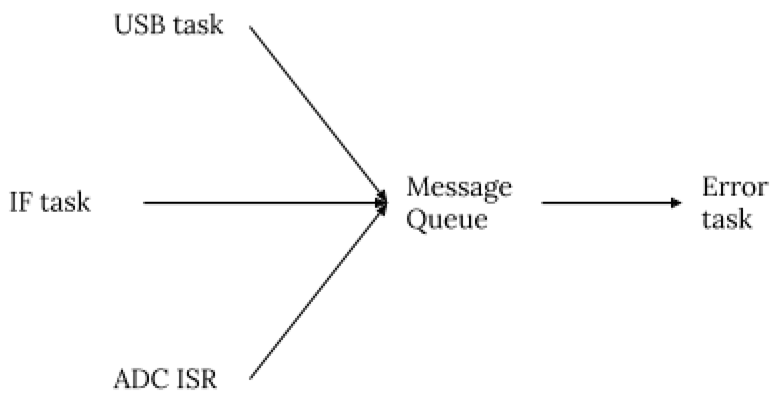
Q = Semaphore + circular buffer (Messages)

```
OS_EVENT *App_QUSB;
void     *App_BufUSB[APP_BUF_SIZE];
// init code
App_QUSB = OSQCreate(&AppBufUSB[0],
                     APP_BUF_SIE);

void App_ISRUSB(void) {
  Clear USB (or DMA) interrupt;
  OSQPost(App_QUSB, p_buf);
}

void App_TaskUSB(void *p_arg) {
  while(1) {
    p_buf = OSQPend(App_QUSB, 0, &err);
    process packet;
  }
}
```

Many-to-1 communication



Message Mailbox

- A message mailbox is another mechanism for inter-task communication
- Essentially, a mailbox is a queue that has a capacity of just one message
- In terms of overhead, mailboxes have a slight advantages over queues

```
OS_EVENT    *OSMboxCreate     (void       *pmsg);
void        *OSMboxPend       (OS_EVENT   *pevent,
                               INT16U     timeout,
                               INT8U      *perr);
```

```
INT8U       OSMboxPost        (OS_EVENT   *pevent,
                               void       *pmsg);
```

1-Way Non-Interlocked Communication

```
OS_EVENT *App_MboxRPM;
// init code
App_MboxRPM = OSMboxCreate((void*) 0);

void App_ISRTimer(void) {
  Clear interrupt;
  Read timer value;
  OSMboxPost(App_MboxRPM, (void*)timer_val);
}

void App_TaskRPM(void *p_arg) {
  Initialize timer;
  while(1) {
    timer_val = (int *)OSMboxPend(App_MboxRPM, 0, &err);
    calculate RPM;
  }
}
```

Memory Manager

- Fixed size memory block management
  - To prevent fragmentation
- Multiple partitions can be created with different sizes
- Blocks allocated from a certain partition must be returned back to the same partition

```
OS_MEM      *OSMemCreate      (void       *addr,
                               INT32U     nblks,
                               INT32U     blksize,
                               INT8U      *perr);
void        *OSMemGet         (OS_MEM     *pmem,
                               INT8U      *perr);
INT8U       OSMemPut          (OS_MEM     *pmem,
                               void       *pblk);
```

Timer Management API

```
OS_TMR      *OSTmrCreate      (INT32U     dly,
                               INT32U     period,
                               INT8U      opt,
                               OS_TMR_CALLBACK callback,
                               void       *callback_arg,
                               INT8U      *pname;
                               INT8U      *perr);
BOOLEAN     OSTmrStart        (OS_TMR     *ptmr,
                               INT8U      *perr);
BOOLEAN     OSTmrStop         (OS_TMR     *ptmr,
                               INT8U      opt,
                               void       *callback_arg,
                               INT8U      *perr)
```

```
INT32U      OSTmrRemainGet    (OS_TMR     *ptmr,
                               INT8U      *perr);
INT8U       OSTmrStateGet     (OS_TMR     *ptmr,
                               INT8U      *perr);
```