μC/OS-II, The Real-Time Kernels and the ARM7 / ARM9
Jean J. Labrosse

www.Micrium.com
My Background

- Master’s Degree in Electrical Engineering
- Wrote two books (μC/OS-II and ESBB)
- Wrote many papers for magazines
  - Embedded Systems Programming
  - Electronic Design
  - C/C++ User’s Journal
  - ASME
  - Xcell Journal
- Have designed Embedded Systems for over 20 years
- President of Micrium
  - Provider of Embedded Software Solutions
Part I

Foreground/Background Systems

\( \mu \text{C/OS-II, The Real-Time Kernels} \)

Task Management
Products without Kernels (Foreground/Background Systems)

Foreground #1

Foreground #2

ISR #1

ISR #2

Task #1

Task #2

Task #3

Infinite loop

Time
/* Background */
void main (void)
{
    Initialization;
    FOREVER {
        Read analog inputs;
        Read discrete inputs;
        Perform monitoring functions;
        Perform control functions;
        Update analog outputs;
        Update discrete outputs;
        Scan keyboard;
        Handle user interface;
        Update display;
        Handle communication requests;
        Other...
    }
}

/* Foreground */
ISR (void)
{
    Handle asynchronous event;
}
Foreground/Background Advantages

- Used in low cost Embedded Applications
- Memory requirements only depends on your application
- Single stack area for:
  - Function nesting
  - Local variables
  - ISR nesting
- Minimal interrupt latency
- Low Cost
  - No royalties to pay to vendors
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

Poll to see if ISR occurred

ISR

Infinite loop

Affected by if, for, while
All ‘tasks’ have the same priority!
- Code executes in sequence
- If an important event occurs it’s handled at the same priority as everything else!
- You may need to execute the same code often to avoid missing an event.

You have to implement all services:
- Time delays and timeouts
- Timers
- Message passing
- Resource management

Code is harder to maintain and can become messy!
Part I

Foreground/Background Systems

μC/OS-II, The Real-Time Kernels
Task Management
What is µC/OS-II?

- 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
  - Intertask communication and synchronization
Why use µC/OS-II?

- To help manage your firmware:
  - GUI (User Interface)
  - File System
  - Protocol Stack
  - Application
  - I/Os
Why use µC/OS-II?

- To be more responsive to real-time events
- To prioritize the work to be done by the CPU
- To simplify system expansion
  - Adding low-priority tasks generally does not change the responsiveness to higher priority tasks!
- To reduce development time
- To easily split the application between programmers
  - Can simplify debugging
- To get useful services from the kernel
  - Services that you would want to provide to your application code
Designing with µC/OS-II
(Splitting an application into Tasks)

- High Priority Task
- Low Priority Task
- Each Task
  - Event
  - Infinite Loop

Importance

Task

© 2008, Micrium, All Rights Reserved
µC/OS-II and the ARM7 / ARM9
μC/OS-II is a Preemptive Kernel

ISR

Interrupt Occurs
Vector to ISR

ISR

ISR make High Priority Task Ready

High Priority Task (HPT)

 ISR Completes (Switch to HP Task)

Low Priority Task (LPT)

ISR Completes (Switch back to LP Task)

HP Task Completes (Switch back to LP Task)
Part I

Foreground/Background Systems
μC/OS-II, The Real-Time Kernels

Task Management
What are 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 are Tasks?

- A task is an infinite loop:

  ```c
  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) */
    }
  }
  }
  ```

- A task can be in one of 5 states...
Task States

- Dormant
- Ready
- Waiting For Event
- Event Occurs Or Timeout
- Running
- Waiting For Event
- Wait For Event
- Wait for time to expire
- Wait for a message
- Wait for a signal
- Delete Task
- Create Task
- Context Switch
- ISR
- Task Interrupted
Tasks needs to be ‘Created’

- To make them 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
‘Creating’ a Task

- You create a task by calling a service provided by the kernel:

  ```c
  INT8U OSTaskCreate(void (*p_task)(void *p_arg),
  void   *p_arg,
  void   *p_stk,
  INT8U   prio);
  ```

- You can create a task:
  - before you start multitasking (at init-time) or,
  - during (at run-time).
Stacks can be checked at run-time to see if you allocated sufficient RAM

Allows you to know the ‘worst case’ stack growth of your task(s)

Stack is cleared when task is created
  – Optional
Deleting a Task

- Tasks can be deleted (return to the ‘dormant’ state) at run-time
  - Task can no longer be scheduled
- Code is NOT actually deleted
- Can be used to ‘abort’ (or ‘kill’) a task
- TCB freed and task stack could be reused.
### µC/OS-II Task APIs

<table>
<thead>
<tr>
<th>µC/OS-II Task APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>µC/OS-II</td>
</tr>
<tr>
<td>µC/OS-II Task APIs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT8U OSTaskChangePrio</td>
<td>(INT8U oldprio, INT8U newprio);</td>
</tr>
<tr>
<td>INT8U OSTaskCreate</td>
<td>(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskCreateExt</td>
<td>(void (*task)(void *p_arg), *p_arg, OS_STK *ptos, INT8U prio, OS_STK *ptos, INT8U id, OS_STK *ptos, INT32U stk_size, void *pext, INT16U opt);</td>
</tr>
<tr>
<td>INT8U OSTaskDel</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskDelReq</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskNameGet</td>
<td>(INT8U prio, INT8U *pname, INT8U *perr);</td>
</tr>
<tr>
<td>void OSTaskNameSet</td>
<td>(INT8U prio, INT8U *pname, INT8U *perr);</td>
</tr>
<tr>
<td>INT8U OSTaskResume</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskSuspend</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskStkChk</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskQuery</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>INT8U OSTaskRegGet</td>
<td>(INT8U prio, INT8U id, INT8U *perr);</td>
</tr>
<tr>
<td>void OSTaskRegSet</td>
<td>(INT8U prio, INT8U id, INT8U *perr);</td>
</tr>
<tr>
<td>void OSTaskQuery</td>
<td>(INT8U id, INT8U value, INT8U *perr);</td>
</tr>
</tbody>
</table>
Part II

Task Scheduling

Context Switching
Servicing Interrupts
Time delays and Timeouts
What is Scheduling?

- Deciding whether there is a more important task to run.

Occurrences:
- When a task decides to wait for time to expire
- When a task sends a message or a signal to another task
- When an ISR sends a message or a signal to a task
  - Occurs at the end of all nested ISRs

Outcome:
- Context Switch if a more important task has been made ready-to-run or returns to the caller or the interrupted task
The µC/OS-II Ready List

OSRdyGrp

OSRdyTbl[ ]

[0] 7 6 5 4 3 2 1 0
[1] 15 14 13 12 11 10 9 8
[2] 23 22 21 20 19 18 17 16
[4] 39 38 37 36 35 34 33 32
[5] 47 46 45 44 43 42 41 40
[6] 55 54 53 52 51 50 49 48
[7] 63 62 61 60 59 58 57 56

Lowest Priority Task (Idle Task)

Task Priority #
Finding the Highest Priority Task Ready

OSRdyGrp

0xF6

OSRdyTbl[ ]

0x78

Y = 1

X = 3

Task Priority

Y = 1

X = 3

Bit Position #11

11

© 2008, Micrium, All Rights Reserved
µC/OS-II and the ARM7 / ARM9
Priority Resolution Table

/****************************
* PRIORITY RESOLUTION TABLE
*
* Note(s): 1) Index into table is bit pattern to resolve
* highest priority.
* 2) Indexed value corresponds to highest priority
* bit position (i.e. 0..7)
************************************************************/

INT8U const OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x00-0x0F
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x10-0x1F
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x20-0x2F
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x30-0x3F
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x40-0x4F
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x50-0x5F
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x60-0x6F
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x70-0x7F
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x80-0x8F
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0x90-0x9F
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0xA0-0xAF
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0xB0-0xBF
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0xC0-0xCF
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0xD0-0xDF
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, // 0xE0-0xEF
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 // 0xF0-0xFF
};

(Step #1) Y = @ [0xF6]  
(i.e. 0xF6 = OSRdyGrp)

(Step #2) X = @ [0x78]  
(i.e. 0x78 = OSRdyTbl[1])
void TaskAtPrio0 (void *p_arg) {
    while (TRUE) {
        .
        OSTimeDlyHMSM(0, 0, 1, 0);
        .
    }
}

Task at Priority 0 runs

Task needs to suspend for 1 second

μC/OS-II clears the Ready bit
Scheduling

OSRdyGrp

OSRdyTbl[ ]

OSTCBPrioTbl[ ]

Old TCB

New TCB

HPT Ready (Bit 11)

(1) Find Highest Priority Task Ready

(2) Index to Find TCB

11
Task Scheduling

Context Switching

Servicing Interrupts

Time delays and Timeouts
Context Switch
(or Task Switch)

- Once the µC/OS-II finds a NEW ‘High-Priority-Task’, µC/OS-II performs a Context Switch.
- The context is the ‘volatile’ state of a CPU
  - Generally the CPU registers
Context Switch (or Task Switch)

SVC Mode

Before

After

© 2008, Micrium, All Rights Reserved

µC/OS-II and the ARM7 / ARM9
Part II

Task Scheduling
Context Switching
Servicing Interrupts
Time delays and Timeouts
Interrupts

- Interrupts are always more important than tasks!
- Interrupts are always recognized
  - Except when they are disabled by µC/OS-II or your application
  - Your application can disable interrupts for as much time as µC/OS-II does without affecting latency
- You should keep ISRs (Interrupt Service Routines) as short as possible.
  - Acknowledge the device
  - Signal a task to process the device
Servicing Interrupts

Interrupts → ISR

- Save Task's Context
- Interrupt Response
- Task Response

ISR → Interrupted Task
- End of ISR, Task Resumed
- End of ISR, Context Switch
- New HPT Task
- Kernel Disables Interrupts
µC/OS-II requires a periodic interrupt source

- Through a hardware timer
  - Between 10 and 1000 ticks/sec. (Hz)
- Could be the power line frequency
  - 50 or 60 Hz
- Called a ‘Clock Tick’ or ‘System Tick’
- Higher the rate, the more the overhead!

The tick ISR calls a service provided by µC/OS-II to signal a ‘tick’
Why keep track of Clock Ticks?

- To allow tasks to suspend execution for a certain amount of time
  - In integral number of ‘ticks’
    - OSTimeDly(ticks)
  - In Hours, Minutes, Seconds and Milliseconds
    - OSTimeDlyHMSM(hr, min, sec, ms)

- To provide timeouts for other services (more on this later)
  - Avoids waiting forever for events to occur
  - Eliminates deadlocks
Part II

Task Scheduling
Context Switching
Servicing Interrupts

Time delays and Timeouts
µC/OS-II Time Delays

- µC/OS-II allows for a task to be delayed:
  - OSTimeDly(ticks)
  - OSTimeDlyHMSM(hr, min, sec, ms)
  - *Always forces a context switch*
  - Suspended task uses little or *no* CPU time

- If the tick rate is 100 Hz (10 mS), a keyboard scan every 100 mS requires 10 ticks:

```c
void Keyboard_Scan_Task (void *p_arg)
{
    for (;;) {
        OSTimeDly(10); /* Every 100 mS */
        Scan keyboard;
    }
}
```
Pending on events allow for timeouts
  – To prevent waiting forever for events

To avoid deadlocks

Example:
  – Read ‘slow’ ADC
  – Timeout indicates that conversion didn’t occur within the expected time.

```c
void ADCTask (void *p_arg)
{
    void *p_msg;
    OS_ERR err;

    for (;;) {
        Start ADC;
        p_msg = OSMboxPend(., ., 10, &err);
        if (err == OS_NO_ERR) {
            Read ADC and Scale;
        } else {
            /* Problem with ADC converter! */
        }
    }
}
```

Timeout of 10 ticks.
### Time Delays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>OSTimeDly</td>
</tr>
<tr>
<td>INT8U</td>
<td>(INT16U ticks);</td>
</tr>
<tr>
<td>void</td>
<td>OSTimeDlyHMSM</td>
</tr>
<tr>
<td>INT8U</td>
<td>(INT16U hours, INT16U minutes, INT16U seconds, INT16U milli);</td>
</tr>
<tr>
<td>INT8U</td>
<td>OSTimeDlyResume</td>
</tr>
<tr>
<td>INT32U</td>
<td>(INT8U prio);</td>
</tr>
<tr>
<td>void</td>
<td>OSTimeGet</td>
</tr>
<tr>
<td>void</td>
<td>OSTimeSet</td>
</tr>
<tr>
<td>INT32U</td>
<td>(INT32U ticks);</td>
</tr>
</tbody>
</table>

### Timers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>OSTmrCreate</td>
</tr>
<tr>
<td>INT32U</td>
<td>(INT32U dly, INT32U period, INT8U opt, INT8U OS_TMR_CALLBACK callback, INT8U callback_arg, INT8U *pname, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrDel</td>
</tr>
<tr>
<td>INT8U</td>
<td>(OS_TMR INT8U, OS_TMR INT8U INT8U, *ptmr, INT8U *perr, INT8U *ptmr, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrNameGet</td>
</tr>
<tr>
<td>INT8U</td>
<td>(OS_TMR INT8U, OS_TMR INT8U INT8U, *ptmr, INT8U *perr, INT8U *ptmr, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrRemainingGet</td>
</tr>
<tr>
<td>INT8U</td>
<td>(OS_TMR INT8U, OS_TMR INT8U INT8U, *ptmr, INT8U *perr, INT8U *ptmr, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrStateGet</td>
</tr>
<tr>
<td>INT8U</td>
<td>(OS_TMR INT8U, OS_TMR INT8U INT8U, *ptmr, INT8U *perr, INT8U *ptmr, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrStart</td>
</tr>
<tr>
<td>INT8U</td>
<td>(OS_TMR INT8U, OS_TMR INT8U INT8U, *ptmr, INT8U *perr, INT8U *ptmr, INT8U *perr);</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>OSTmrStop</td>
</tr>
<tr>
<td>void</td>
<td>(OS_TMR INT8U, INT8U INT8U INT8U void *callback_arg, INT8U *perr);</td>
</tr>
</tbody>
</table>
Part III

Resource Sharing and Mutual Exclusion

Task Synchronization
Task Communication
YOU MUST ensure that access to common resources is protected!

- µC/OS-II only gives you mechanisms

You protect access to common resources by:

- Disabling/Enabling interrupts
- Lock/Unlock
- MUTEX (Mutual Exclusion Semaphores)
Resource Sharing (Disable and Enable Interrupts)

- When access to resource is done quickly
  - Must be less than µC/OS-II’s interrupt disable time!
  - Be careful with Floating-point!
- Disable/Enable interrupts is the fastest way!

```
rpm = 60.0 / time;
CPU_CRITICAL_ENTER();
Global RPM = rpm;
CPU_CRITICAL_EXIT();
```
Resource Sharing
(Lock/Unlock the Scheduler)

- ‘Lock’ prevents the scheduler from changing tasks
  - Interrupts are still enabled
  - Can be used to access non-reentrant functions
  - Can be used to reduce priority inversion
  - Same effect as making the current task the Highest Priority Task
  - Don’t Lock for too long
    - Defeats the purpose of having µC/OS-II.

- ‘Unlock’ invokes the scheduler to see if a High-Priority Task has been made ready while locked

```c
OSSchedLock();
Code with scheduler disabled;
OSSchedUnlock();
```
Mutual Exclusion

Mutexes

- Used when time to access a resource is longer than µC/OS-II’s interrupt disable time!
- Mutexes are binary semaphores and are used to access a shared resource
- Mutexes reduce unbounded ‘priority inversions’
Using a Mutex
(Time-of-Day Clock)

ClockTask(void)
{
    while (TRUE) {
        OSTimeDlyHMSM(0, 0, 1, 0);
        OSMutexPend(&ClkMutex, 0);
        Update clock;
        OSMutexPost(&ClkMutex);
    }
}

AppTask(void)
{
    while (TRUE) {
        OSMutexPend(&ClkMutex, 0);
        Get Time Of Day;
        OSMutexPost(&ClkMutex);
        :
    }
}
μC/OS-II’s Mutexes
Priority Ceiling

High Priority Task (HPT)

Medium Priority Task (MPT)

Low Priority Task (LPT)

Task Gets Mutex

HPT Needs Mutex

μC/OS-II raises LPT’s Priority

HPT is done

HPT is done

MPT is done

LPT is done with Mutex

High Priority Task Preempts Low One
## µC/OS-II
### Resource Sharing APIs

## Mutual Exclusion Semaphores

**Mutual Exclusion Semaphores**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSMutexAccept</td>
<td>BOOLEAN, OS_EVENT*</td>
</tr>
<tr>
<td>OSMutexCreate</td>
<td>OS_EVENT*, INT8U, *perr;</td>
</tr>
<tr>
<td>OSMutexDel</td>
<td>OS_EVENT*, INT8U, opt, *perr;</td>
</tr>
<tr>
<td>OSMutexPend</td>
<td>OS_EVENT*, INT16U, timeout, *perr;</td>
</tr>
<tr>
<td>OSMutexPost</td>
<td>OS_EVENT*, INT8U, *perr;</td>
</tr>
<tr>
<td>OSMutexQuery</td>
<td>OS_EVENT*, OS_MUTEX_DATA, *p_mutex_data;</td>
</tr>
</tbody>
</table>

## Scheduler Lock/Unlock

**Scheduler Lock/Unlock**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSSchedLock</td>
<td>void</td>
</tr>
<tr>
<td>OSSchedUnlock</td>
<td>void</td>
</tr>
</tbody>
</table>
Part III

Resource Sharing and Mutual Exclusion

Task Synchronization

Task Communication
Semaphores to signal tasks
(Analog-Digital Conversion)

Read_Analog_Input_Channel_Cnts(channel#, *adc_counts)
{
    Select the desired analog input channel
    Wait for MUX output to stabilize
    Start the ADC Conversion
    Wait for signal from ADC ISR (with timeout)
    if (timed out)
        Return error code to caller
    else
        Read ADC counts
        Return ADC counts to caller
}

ADC_ISR(void)
{
    Signal Event
    Clear EOC interrupt
}
Synchronization of tasks with the occurrence of multiple events

Events are grouped
- 8, 16 or 32 bits per group (compile-time configurable)

Types of synchronization:
- Disjunctive (OR): Any event occurred
- Conjunctive (AND): All events occurred

Task(s) or ISR(s) can either Set or Clear event flags

Only tasks can Wait for events
Event Flags

ISRsg Tasks

Events
(8, 16 or 32 bits)

Set or Clear

Tasks

Wait

Tasks

Wait

OR

AND
## µC/OS-II

### Task Synchronization APIs

#### Event Flags

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_FlagAccept</td>
<td>OS_FLAG_GRP, OS_FLAGS, INT8U, INT8U, INT8U, *pgrp, flags, wait_type, *perr</td>
</tr>
<tr>
<td>OS_FlagCreate</td>
<td>OS_FLAGS, INT8U, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagDel</td>
<td>OS_FLAG_GRP, OS_FLAGS, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagNameGet</td>
<td>INT8U, INT8U, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagNameSet</td>
<td>void, INT8U, INT8U, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagPend</td>
<td>OS_FLAG_GRP, OS_FLAGS, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagPendGetFlagsRdy</td>
<td>(void);</td>
</tr>
<tr>
<td>OS_FlagPost</td>
<td>OS_FLAG_GRP, OS_FLAGS, INT8U, INT8U, INT8U, *pgrp, flags, *perr</td>
</tr>
<tr>
<td>OS_FlagQuery</td>
<td>OS_FLAG_GRP, OS_FLAGS, INT8U, INT8U, INT8U, *pgrp, *perr</td>
</tr>
</tbody>
</table>

#### Counting Semaphores

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSSemAccept</td>
<td>OS_EVENT, INT16U, *pevent, opt, *perr</td>
</tr>
<tr>
<td>OSSemCreate</td>
<td>INT16U, *pevent, opt, *perr</td>
</tr>
<tr>
<td>OSSemDel</td>
<td>INT16U, INT16U, *pevent, opt, *perr</td>
</tr>
<tr>
<td>OSSemPend</td>
<td>INT8U, *pevent, timeout, *perr</td>
</tr>
<tr>
<td>OSSemPendAbort</td>
<td>INT8U, *pevent, timeout, *perr</td>
</tr>
<tr>
<td>OSSemPost</td>
<td>INT8U, *pevent, cnt, *perr</td>
</tr>
<tr>
<td>OSSemQuery</td>
<td>INT8U, *pevent, cnt, *perr</td>
</tr>
<tr>
<td>OSSemSet</td>
<td>INT16U, INT16U, *pevent, cnt, *perr</td>
</tr>
</tbody>
</table>
Part III

Resource Sharing and Mutual Exclusion
Task Synchronization

Task Communication
Message Queues

- **Message passing**
  - Message is a pointer
  - Pointer can point to a variable or a data structure
- **FIFO** (*First-In-First-Out*) type queue
  - Size of each queue can be specified to the kernel
- **LIFO** (*Last-In-First-Out*) also possible
- **Tasks or ISR can ‘send’ messages**
- **Only tasks can ‘receive’ a message**
  - Highest-priority task waiting on queue will get the message
- **Receiving task can timeout if no message is received within a certain amount of time**
RPM_ISR()
{
    Read Timer;
    DeltaCounts = Counts - PreviousCounts;
    PreviousCounts = Counts
    Post DeltaCounts;
}

RPMTask()
{
    while (1)
        Wait for message from ISR (with timeout);
        if (timed out)
            RPM = 0;
        else
            RPM = 60 * Fin / counts;
        Compute average RPM;
        Check for overspeed/underspeed;
        Keep track of peak RPM;
        etc.
}
### µC/OS-II

#### Task Communication APIs

**Message Mailboxes**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void OSMboxAccept(OS_EVENT *pevent)</code></td>
<td>Accept a mailbox event</td>
</tr>
<tr>
<td><code>OS_EVENT *OSMboxCreate(void *pmsg)</code></td>
<td>Create a mailbox</td>
</tr>
<tr>
<td><code>INT8U OSMboxDel(OS_EVENT *pevent, INT8U *perr)</code></td>
<td>Delete a mailbox</td>
</tr>
<tr>
<td><code>void OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *perr)</code></td>
<td>Pending a mailbox event</td>
</tr>
<tr>
<td><code>INT8U OSMboxPendAbort(OS_EVENT *pevent, INT8U opt, INT8U *perr)</code></td>
<td>Abort pending a mailbox event</td>
</tr>
<tr>
<td><code>INT8U OSMboxPost(OS_EVENT *pevent, void *pmsg)</code></td>
<td>Post a message to a mailbox</td>
</tr>
<tr>
<td><code>INT8U OSMboxPostOpt(OS_EVENT *pevent, void *pmsg, INT8U opt)</code></td>
<td>Post a message to a mailbox with options</td>
</tr>
<tr>
<td><code>INT8U OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *p_mbox_data)</code></td>
<td>Query a mailbox</td>
</tr>
</tbody>
</table>

**Message Queues**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void OSQAccept(OS_EVENT *pevent, INT8U *perr)</code></td>
<td>Accept a queue event</td>
</tr>
<tr>
<td><code>OS_EVENT *OSQCreate(void **start, INT16U size)</code></td>
<td>Create a queue</td>
</tr>
<tr>
<td><code>OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *perr)</code></td>
<td>Delete a queue</td>
</tr>
<tr>
<td><code>INT8U OSQFlush(OS_EVENT *pevent)</code></td>
<td>Flush a queue</td>
</tr>
<tr>
<td><code>void OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *perr)</code></td>
<td>Pending a queue event</td>
</tr>
<tr>
<td><code>INT8U OSQPendAbort(OS_EVENT *pevent, INT8U opt, INT8U *perr)</code></td>
<td>Abort pending a queue event</td>
</tr>
<tr>
<td><code>INT8U OSQPost(OS_EVENT *pevent, void *pmsg)</code></td>
<td>Post a message to a queue</td>
</tr>
<tr>
<td><code>INT8U OSQPostFront(OS_EVENT *pevent, void *pmsg)</code></td>
<td>Post a message to the front of a queue</td>
</tr>
<tr>
<td><code>INT8U OSQPostOpt(OS_EVENT *pevent, void *pmsg, INT8U opt)</code></td>
<td>Post a message to a queue with options</td>
</tr>
<tr>
<td><code>INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *p_q_data)</code></td>
<td>Query a queue</td>
</tr>
</tbody>
</table>
Configuration and Initialization
Debugging with Kernels
**µC/OS-II Configuration**

- Allows you to specify which services are available
  - Done through #defines in application specific file: `OS_CFG.H`

- Memory footprint depends on configuration
  - On ARM, 6K to 24K of code space
  - RAM depends on kernel objects used
void main (void)
{
    /* User initialization */
    OSInit();    /* Kernel Initialization */

    /* Install interrupt vectors */

    /* Create at least 1 task (Start Task) */
    /* Additional User code */

    OSStart();   /* Start multitasking */
}
Initialization

- **μC/OS-II** creates 1 to 3 internal tasks:
  - **OS_TaskIdle()**
    - Runs when no other task runs
    - Always the lowest priority task
    - Cannot be deleted
  - **OS_TaskStat()**
    - Computes run-time statistics
      - CPU Usage
      - Check stack usage of other tasks
  - **OS_TmrTask()**
    - If you enabled the ‘timer’ services

- **Initializes other data structures**
Part IV

Configuration and Initialization

Debugging with Kernels
Debugging
IAR’s μC/OS-II Kernel Awareness

- Free with IAR’s EWARM
- Static Tool
- Shows value of kernel structures:
  - Task list
  - Kernel objects
Debugging
μC/Probe, Run-Time Data Monitor

μC/OS-II and the ARM7 / ARM9
© 2008, Micrium, All Rights Reserved
Debugging
Other Techniques

- Use a DAC (Digital to Analog Converter)
  - Output a value based on which task or ISR is running
    - Shows execution profile of each task/ISR running (oscilloscope)
    - Can be used to measure task execution time

- Use output ports for time measurements

- Use TRACE tools
  - Some processors allow you to capture execution traces
  - Some debugger captures and display run-time history
References

Mark H. Klein, Kluwer Academic Publishers

Jean J. Labrosse, CMP Books
ISBN 1-57820-103-9

“Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C”
Jean J. Labrosse, CMP Books
ISBN 0-97930-604-1