UNC CHARLOTTE

The μC/OS-II Real-Time Operating System

µC/OS-II

Real-time kernel

- Portable, scalable, preemptive RTOS
- Ported to over 90 processors

Pronounced “microC OS two”

Written by Jean J. Labrosse of Micrium, http://ucos-ii.com

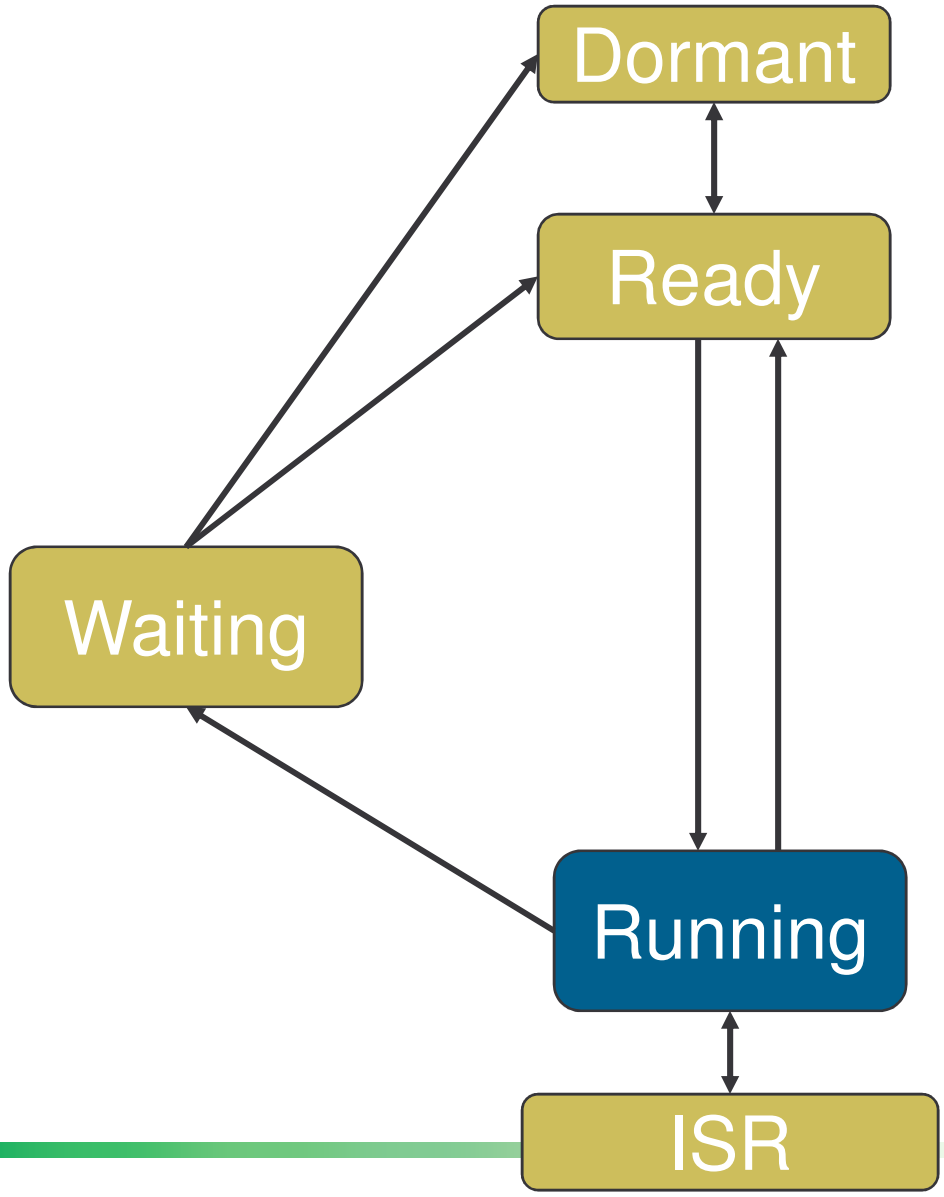
Extensive information in **MicroC/OS-II: The Real-Time Kernel (A complete portable, ROMable scalable preemptive RTOS),** Jean J. LaBrosse, CMP Books

Task States

Five possible states for a task to be in

- Dormant – not yet visible to OS (use OSTaskCreate(), etc.)
- Ready
- Running
- Waiting
- ISR – preempted by an ISR

See manual for details



Task Scheduling

Scheduler runs highest-priority task using OSSched()

- OSRdyTbl has a set bit for each ready task
- Checks to see if context switch is needed
- Macro OS_TASK_SW performs context switch
  - Implemented as software interrupt which points to OSCtxSw
  - Save registers of task being switched out
  - Restore registers of task being switched in

Scheduler locking

- Can lock scheduler to prevent other tasks from running (ISRs still run)
  - OSSchedLock()
  - OSSchedUnlock()
- Nesting of OSSchedLock possible
- Don't lock the scheduler and then perform a system call which could put your task into the WAITING state!

Idle task

- Runs when nothing else is ready
- Automatically has prioirty OS_LOWEST_PRIO
- Only increments a counter for use in estimating processor idle time

Where Is The Code Which Makes It Work?

- Selecting a thread to run
  - OSSched() in os_core2.c
- Context switching
  - OS_TASK_SW in os_cpu.h
  - OSCtxSw in os_cpu_a.a30
- What runs if no tasks are ready?
  - OSTaskIdle() in os_core2.c

TCB for µC/OS-II

```
typedef struct os_tcb {                  /* Extended TCB code is italicized */
   OS_STK *OSTCBStkPtr;                  /* Pointer to current top of stack */
   void   *OSTCBExtPtr;                  /* Pointer to user definable data for
   TCB
                                            extension */
   OS_STK *OSTCBStkBottom;               /* Pointer to bottom of stack - last
                                            valid address */
   INT32U OSTCBStkSize;                  /* Size of task stack (in bytes) */
   INT16U OSTCBOpt;                      /* Task options as passed by
                                             OSTaskCreateExt() */
   INT16U OSTCBId;                       /* Task ID (0..65535) */
   struct os_tcb *OSTCBNext;             /* Pointer to next TCB in the TCB list
   */
   struct os_tcb *OSTCBPrev;             /* Pointer to previous TCB in list */
   OS_EVENT *OSTCBEventPtr;              /* Pointer to event control block */
   void *OSTCBMsg;                       /* Message received from OSMboxPost() or
                                            OSQPost() */
   INT16U OSTCBDly;                      /* Nbr ticks to delay task or, timeout
                                            waiting for event */
   INT8U OSTCBStat;                      /* Task status */
   INT8U OSTCBPrio;                      /* Task priority (0 == highest,
                                            63 == lowest) */
   … ready table position information …
   BOOLEAN OSTCBDelReq;                  /* Indicates whether a task needs to
                                            delete itself */
} OS_TCB;
```

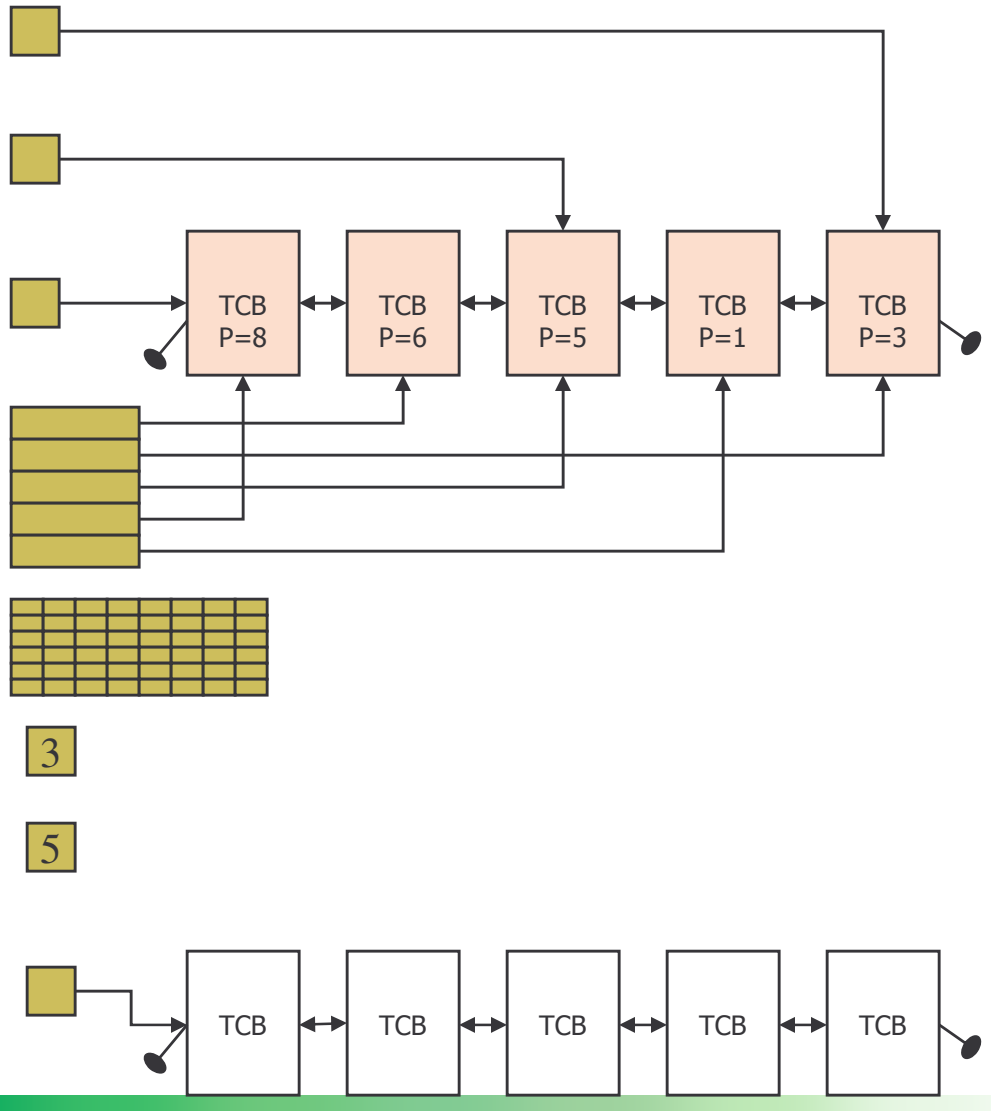
Task States

Task status OSTCBStat

```
/* TASK STATUS (Bit definition for OSTCBStat)                */
#define  OS_STAT_RDY          0x00   /* Ready to run         */
#define  OS_STAT_SEM          0x01   /* Pending on semaphore */
#define  OS_STAT_MBOX         0x02   /* Pending on mailbox   */
#define  OS_STAT_Q                   0x04   /* Pending on queue   */
#define  OS_STAT_SUSPEND   0x08   /* Task is suspended */
```

Data Structures for μC/OS-II

- OSTCBCur - Pointer to TCB of currently running task
- OSTCBHighRdy - Pointer to highest priority TCB ready to run
- OSTCBList - Pointer to doubly linked list of TCBs
- OSTCBPrioTbl[OS_LOWEST_PRIO + 1] - Table of pointers to created TCBs, ordered by priority
- OSReadyTbl - Encoded table of tasks ready to run
- OSPrioCur – Current task priority
- OSPrioHighRdy – Priority of highest ready task
- OSTCBFreeList - List of free OS_TCBs, use for creating new tasks



Enabling Interrupts

Macros OS_ENTER_CRITICAL, OS_EXIT_CRITICAL

Note: three methods are provided in os_cpu.h

- #1 doesn't restore interrupt state, just renables interrupts
- #2 saves and restores state, but stack pointer must be same at enter/exit points – ***use this one!***
- #3 uses a variable to hold state
  - Is not reentrant
  - Should be a global variable, not declared in function StartSystemTick()

System Clock Tick

- OS needs periodic timer for time delays and timeouts
- Recommended frequency 10-200 Hz (trade off overhead vs. response time (and accuracy of delays)
- Must enable these interrupts after calling OSStart()
- Student exercise
  - Which timer is used for this purpose on the QSK62P?
  - What is the frequency?
- OSTick() ISR
  - Calls OSTimeTick()
    - Calls hook to a function of your choosing
    - Decrements non-zero delay fields (OSTCBDly) for all task control blocks
    - If a delay field reaches zero, make task ready to run (unless it was suspended)
  - Increments counter variable OSTime (32-bit counter)
  - Then returns from interrupt
- Interface
  - OSTimeGet(): Ticks (OSTime value) since OSStart was called
  - OSTimeSet(): Set value of this counter

Overview of Writing an Application

Scale the OS resources to match the application
- See os_cfg.h

Define a stack for each task

Write tasks

Write ISRs

Write main() to Initialize and start up the OS (main.c)
- Initialize MCU, display, OS
- Start timer to generate system tick
- Create semaphores, etc.
- Create tasks
- Call OSStart()

Configuration and Scaling

For efficiency and code size, default version of OS supports limited functionality and resources

When developing an application, must verify these are sufficient (or may have to track down strange bugs)

- Can't just blindly develop program without considering what's available

Edit ***os_cfg.h*** to configure the OS to meet your application's needs

- # events, # tasks, whether mailboxes are supported, etc.

Writing ISRs

- Structure needed
  - Save CPU registers – *NC30 compiler adds this automatically*
  - Call OSIntEnter() or increment OSIntNesting (faster, so preferred)
    - OSIntEnter uses OS_ENTER_CRITICAL and OS_EXIT_CRITICAL, so make sure these use method 2 (save on stack)
  - Execute code to service interrupt – *body of ISR*
  - Call OSIntExit()
    - Has OS find the highest priority task to run after this ISR finishes (like OSSched())
  - Restore CPU registers – *compiler adds this automatically*
  - Execute return from interrupt instruction – *compiler adds this automatically*
- Good practices
  - Make ISR as quick as possible. Only do time-critical work here, and defer remaining work to task code.
  - Have ISR notify task of event, possibly send data
    - OSSemPost – raise flag indicating event happened
    - OSMboxPost – put message with data in mailbox (1)
    - OSQPost – put message with data in queue (n)
      - Example: Unload data from UART receive buffer (overflows with 2 characters), put into a longer queue (e.g. overflows after 128 characters) which is serviced by task

Writing Tasks

Define a stack for each task

- Must be a global (static) array of base type OS_STK

Task structure: two options

- Function with infinite loop (e.g. for periodic task)
  - Each time the loop is executed, it must call an OS function which can yield the processor (e.g. OSSemPend(), OSMboxPend(), OSQPend(), OSTaskSuspend(), OSTimeDly(), OSTimeDlyHMSM())
- Function which runs once and then deletes itself from scheduler
  - Task ends in OSTaskDel()

Task Creation

OSTaskCreate() in os_task.c

- Create a task
- Arguments: pointer to task code (function), pointer to argument, pointer to top of stack (use TOS macro), desired priority (unique)

OSTaskCreateExt() in os_task.c

- Create a task
- Arguments: same as for OSTaskCreate(), plus
  - id: user-specified unique task identifier number
  - pbos: pointer to bottom of stack. Used for stack checking (if enabled).
  - stk_size: number of elements in stack. Used for stack checking (if enabled).
  - pext: pointer to user-supplied task-specific data area (e.g. string with task name)
  - opt: options to control how task is created.

More Task Management

- OSTaskSuspend()
  - Task will not run again until after it is resumed
  - Sets OS_STAT_SUSPEND flag, removes task from ready list if there
  - Argument: Task priority (used to identify task)
- OSTaskResume()
  - Task will run again once any time delay expires and task is in ready queue
  - Clears OS_STAT_SUSPEND flag
  - Argument: Task priority (used to identify task)
- OSTaskDel()
  - Sets task to DORMANT state, so no longer scheduled by OS
  - Removed from OS data structures: ready list, wait lists for semaphores/mailboxes/queues, etc.
- OSTaskChangePrio()
  - Identify task by (current) priority
  - Changes task's priority
- OSTaskQuery()
  - Identify task by priority
  - Copies that task's TCB into a user-supplied structure
  - Useful for debugging

Time Management

Application-requested delays

- Task A calls OSTimeDly or OSTimeDlyHMSM() in os_time.c
- TCB->OSTCBDly set to indicate number of ticks to wait
- Rembember that OSTickISR() in os_cpu_a.a30, OSTimeTick() in os_core2.c decrement this field and determine when it expires
- Task B can resume Task A by calling OSTimeDlyResume()

Example: uC/OSII Demo

Tasks
- Task 1
  - Flashes red LED
  - Displays count of loop iterations on LCD top line
- Task 2
  - Flashes green LED
- Task 3
  - Flashes yellow LED

Debugging with an RTOS

Did you scale the RTOS to your application?
- Number of tasks, semaphores, queues, mailboxes, etc.

Always check result/error codes for system calls
- Light an LED if there's an error

Why doesn't my thread run?
- Look at scheduler's data structures via debugger
  - OSReadyTbl: Table of tasks ready to run
    - Bitfield array
  - TCB: Task control block
    - OSTCBStat: status field
- If the error LED goes on, set a breakpoint there and see what happened

Does your thread have enough stack space?
- sprintf takes a lot. Floating point math does too.

Did you remember to call OSTaskCreate for your thread?

Is your thread structured as an infinite loop with an OS call on which to block?

Are interrupts working properly? Try substituting a polling function to isolate the problem.

Is there enough memory for your program? Check the .map file
- RAM: 0400h to 0137Eh
- Flash ROM: 0F0000h to 0FF8FFh

Summary

Basics of using uC/OS-II

- Task states and scheduling
- Time tick
- How to structure an application
- How to create and manage tasks
- How to delay a task

How to debug when using an RTOS