Keep learning.
It's the Washington Way.

EMBSYS 105
Programming with Embedded & Real-Time Operating Systems

Instructor: Nick Strathy, nstrathy@uw.edu

TA: Gideon Lee, gideonhlee@yahoo.com

Lecture 4 1/27/2020

PROFESSIONAL &
CONTINUING EDUCATION
UNIVERSITY of WASHINGTON

Looking ahead

Date	Lecture number	Assignment
1/6	L1	A1 due* before L2
1/13	L2	A2 due before L3
1/20	L3	A3 due before L4
1/27	L4	A4 due before L5
2/3	L5	A5 due before L7, Project due before L10
2/10	Holiday (?)	
2/17	L6	
2/24	L7	
3/2	L8	
3/9	L9	
3/16	L10 – Student presentations	

* Assignments are due Sunday night at 11:59 PM

Previous Lecture (L3) Overview

- MicroC/OS-II (uCOS) Introduction
  - Task creation, Task Delay, Sample task code, uCOS startup steps
- Porting uCOS to our board (**Labrosse** *Ch 13*)
  - Key data definitions, enabling and disabling interrupts, critical section implementation, initializing the stack, context switching, C pointers and assembly language
- uCOS Services (**Labrosse** *Ch 16*)
  - Task Management
  - Time Management
  - Interrupt Management
  - Semaphores
  - Mutexes
  - Message Mailboxes
  - Message Queues
  - Event Flags
  - Memory Management
  - User-Defined Functions
  - Miscellaneous Services
- uCOS Configuration (**Labrosse** *Ch 17*)
- Assignment 3 – uCOS Port

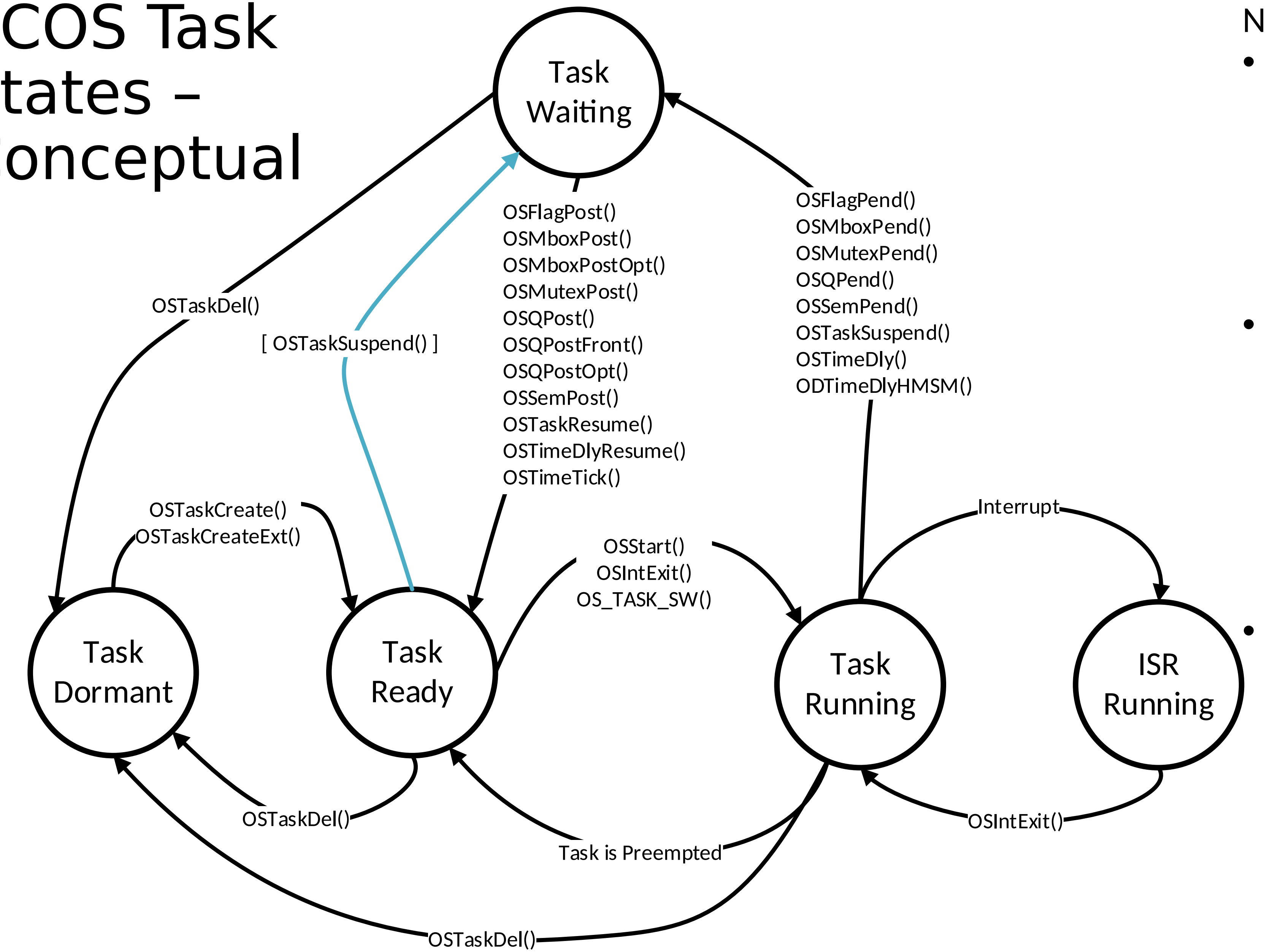
Stopped here last lecture.
Next slide: 64

Current Lecture (L4) Overview

First, finish slides from last week (L3) beginning at slide 64

- uCOS Internals (Labrosse ch 3, 6, 9)
  - Task States
  - Task Control Blocks (TCBs)
  - Ready List
  - Task Scheduling
  - Event Control Blocks
    - Placing a task in the ECB Wait List
    - Removing a Task from the ECB Wait List
    - Find the Highest Priority Task Waiting on the ECB
    - List of Free ECBs
    - Initializing an ECB
    - Making a Task Ready
    - Making a Task Wait for an Event
    - Making a task Ready Because of a Timeout
  - Event Flags
- Task Synchronization Techniques (Tanenbaum, Wikipedia, etc.)
  - Sharing data between ISRs and tasks
  - Producer-Consumer Problem
  - Readers and Writers Problem
  - Deadlock
  - Dining Philosophers Problem
- Event Driven Systems (Wikipedia)
  - Characteristics of Event Driven Systems
  - Data Flow Diagrams
- Assignment 4 – Task Synchronization

uCOS Task States – Conceptual



Notes:

- Labrosse p. 39 does not show Ready Waiting however OSTaskSuspend(prio) can make that transition.
- These states are conceptual – actually there are several Wait states; no actual Running or Dormant states are *tracked* in later versions of uCOS.
- Later versions of uCOS call “ISR Running” “Interrupted”

uCOS Task States – Implementation

```
INT8U            OSTCBStat;        /* Task      status                                             */
```

The above TCB field takes on values chosen from:

```
#define  OS_STAT_RDY               0x00u    /* Ready to run                                            */
#define  OS_STAT_SEM               0x01u    /* Pending on semaphore                                    */
#define  OS_STAT_MBOX              0x02u    /* Pending on mailbox                                      */
#define  OS_STAT_Q                 0x04u    /* Pending on queue                                        */
#define  OS_STAT_SUSPEND           0x08u    /* Task is suspended                                       */
#define  OS_STAT_MUTEX             0x10u    /* Pending on mutual exclusion semaphore                   */
#define  OS_STAT_FLAG              0x20u    /* Pending on event flag group                             */

#define  OS_STAT_PEND_ANY         (OS_STAT_SEM | OS_STAT_MBOX | OS_STAT_Q | OS_STAT_MUTEX | OS_STAT_FLAG)
```

```
INT8U            OSTCBStatPend;    /* Task PEND status                                             */
```

The above TCB field takes on values chosen from:

```
#define  OS_STAT_PEND_OK              0u    /* Pending status OK, not pending, or pending complete     */
#define  OS_STAT_PEND_TO              1u    /* Pending timed out                                       */
#define  OS_STAT_PEND_ABORT           2u    /* Pending aborted                                         */
```

Task Control Block (OS_TCB)

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;      /* Pointer to current top of stack                              */

#if OS_TASK_CREATE_EXT_EN > 0
    void            *OSTCBExtPtr;      /* Pointer to user definable data for TCB extension             */
    OS_STK          *OSTCBStkBottom;   /* Pointer to bottom of stack                                   */
    INT32U           OSTCBStkSize;     /* Size of task stack (in number of stack elements)             */
    INT16U           OSTCBOpt;         /* Task options as passed by OSTaskCreateExt()                  */
    INT16U           OSTCBId;          /* Task ID (0..65535)                                           */
#endif

    struct os_tcb   *OSTCBNext;        /* Pointer to next     TCB in the TCB list                      */
    struct os_tcb   *OSTCBPrev;        /* Pointer to previous TCB in the TCB list                      */

#if OS_EVENT_EN|| (OS_FLAG_EN > 0)
    OS_EVENT        *OSTCBEventPtr;    /* Pointer to event control block                               */
#endif

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void            *OSTCBMsg;         /* Message received from OSMboxPost() or OSQPost()              */
#endif
```

Task Control Block (OS_TCB)

```
#if (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE    *OSTCBFlagNode;    /* Pointer to event flag node                                   */
#endif
    OS_FLAGS         OSTCBFlagsRdy;    /* Event flags that made task ready to run                      */
#endif

    INT16U           OSTCBDly;         /* Nbr ticks to delay task or, timeout waiting for event        */
    INT8U            OSTCBStat;        /* Task      status                                             */
    INT8U            OSTCBStatPend;    /* Task PEND status                                             */
    INT8U            OSTCBPrio;        /* Task priority (0 == highest)                                 */

    INT8U            OSTCBX;           /* Bit position in group  corresponding to task priority        */
    INT8U            OSTCBY;           /* Index into ready table corresponding to task priority        */
#if OS_LOWEST_PRIO <= 63
    INT8U            OSTCBBitX;        /* Bit mask to access bit position in ready table               */
    INT8U            OSTCBBitY;        /* Bit mask to access bit position in ready group               */
#else
    INT16U           OSTCBBitX;        /* Bit mask to access bit position in ready table               */
    INT16U           OSTCBBitY;        /* Bit mask to access bit position in ready group               */
#endif
```

Task Control Block (OS_TCB)

```
#if OS_TASK_DEL_EN > 0
    INT8U            OSTCBDelReq;      /* Indicates whether a task needs to delete itself              */
#endif

#if OS_TASK_PROFILE_EN > 0
    INT32U           OSTCBCtxSwCtr;    /* Number of time the task was switched in                      */
    INT32U           OSTCBCyclesTot;   /* Total number of clock cycles the task has been running       */
    INT32U           OSTCBCyclesStart; /* Snapshot of cycle counter at start of task resumption        */
    OS_STK          *OSTCBStkBase;     /* Pointer to the beginning of the task stack                   */
    INT32U           OSTCBStkUsed;     /* Number of bytes used from the stack                          */
#endif

#if OS_TASK_NAME_SIZE > 1
    INT8U            OSTCBTaskName[OS_TASK_NAME_SIZE];
#endif
} OS_TCB;
```

TCB Free List (ucos-ii.h)

```
OS_EXT  OS_TCB           *OSTCBCur;                        /* Pointer to currently running TCB         */
OS_EXT  OS_TCB           *OSTCBFreeList;                   /* Pointer to list of free TCBs             */
OS_EXT  OS_TCB           *OSTCBHighRdy;                    /* Pointer to highest priority TCB R-to-R   */
OS_EXT  OS_TCB           *OSTCBList;                       /* Pointer to doubly linked list of TCBs    */
OS_EXT  OS_TCB           *OSTCBPrioTbl[OS_LOWEST_PRIO + 1];/* Table of pointers to created TCBs        */
OS_EXT  OS_TCB            OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS];   /* Table of TCBs                  */
```

- OSTCBFreeList
  - New tasks get their TCB from this list
  - The new TCB moves from the free list to OSTCBList
- OSTCBPrioTbl
  - New tasks get their TCB added to this table indexed by priority
  - If there is already a non-null entry for a given priority, a task with that priority already exists and error code OS_ERR_PRIO_EXIST is returned by OSTaskCreate()
- All TCBs are preallocated at compile time in array OSTCBTbl

Ready List

```
#define  OS_LOWEST_PRIO 31

#define  OS_RDY_TBL_SIZE ((OS_LOWEST_PRIO) / 8 + 1)

OS_EXT  INT8U OSRdyTbl[OS_RDY_TBL_SIZE];
```

- Rather than maintain a sorted list of Ready tasks with the highest priority task always at the front, uCOS represents the Ready list by a set of lookup tables based on the task priority
- This scheme results in constant time for Ready List insertion/deletion regardless of the number of tasks. It also yields constant time for determining the highest priority Ready task regardless of the number of tasks
- The tables used to maintain the Ready list are **OSRdyTbl , OSRdyGrp, and OSUnMapTbl**
- **OSRdyTbl** contains the set of tasks that are in the Ready state **indexed by task priority**
- Each task is represented by 1 bit in OSRdyTbl where 0 means not Ready, 1 means Ready

Ready List

Add a task to the Ready List (table):

```
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
```

Where these are computed once during task creation:

```
ptcb->OSTCBY          = (INT8U)(prio >> 3);
ptcb->OSTCBX          = (INT8U)(prio & 0x07);

ptcb->OSTCBBitY       = (INT8U)(1 << ptcb->OSTCBY);
ptcb->OSTCBBitX       = (INT8U)(1 << ptcb->OSTCBX);
```

Remove a task from the Ready List (table):

```
y                        =  ptcb>OSTCBY;
OSRdyTbl[y]             &= ~ptcb>OSTCBBitX;
```

Ex: Make Task 20 ready:

prio 20_{10} == 10100_2 binary

ptcb->OSTCBY = 10_2 = 2_{10}

ptcb->OSTCBX = 100_2 = 4_{10}

ptcb->OSTCBBitX = 10000_2

ptcb->OSRdyTbl[2] |= 10000_2

OSRdyTbl:

x y	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	**1**	0	0
2	0	0	0	1	0	0	0	0
3	0	0	1	0	1	0	0	0

Ready List

Finding the highest priority Ready Task

```
INT8U           OSRdyGrp; /* global, tracks high-order 3 bits of Ready task priorities */

OSRdyGrp        |= ptcb->OSTCBBitY; /* updated whenever a task becomes Ready/unReady */
```

Note: OSRdyGrp has bit i set if row i of OSRdyTbl has any bits set.

Where these are computed once during task creation (see previous slide):

```
ptcb->OSTCBY         = (INT8U)(prio >> 3);
ptcb->OSTCBBitY      = (INT8U)(1 << ptcb->OSTCBY);
```

Use OSRdyGrp, OSRdyTbl, and OSUnMapTbl to get highest priority Ready task:

```
INT8U y         = OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
```

Ready List

OSUnMapTable enables lookup of highest priority Ready task given OSRdyGrp and OSRdyTbl

```
INT8U y         = OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
```

where:

```
INT8U  const  OSUnMapTbl[256] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x00 to 0x0F          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x10 to 0x1F          */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x20 to 0x2F          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x30 to 0x3F          */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x40 to 0x4F          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x50 to 0x5F          */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x60 to 0x6F          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x70 to 0x7F          */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x80 to 0x8F          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0x90 to 0x9F          */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0xA0 to 0xAF          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0xB0 to 0xBF          */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0xC0 to 0xCF          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0xD0 to 0xDF          */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,       /* 0xE0 to 0xEF          */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0        /* 0xF0 to 0xFF          */
};
```

Task Scheduling

- OS_Sched() performs task-level scheduling
  - Determines the highest priority ready task
  - Transfers control of the CPU to that task
  - Note: uCOS doesn’t explicitly change state from Ready Running i.e. the running task remains in the Ready List.
- OSIntExit() performs ISR-level scheduling
  - Only reschedules to the highest priority ready task if interrupt nesting level is 0 otherwise simply returns from procedure call to nested ISR code which will return from ISR to previous nested ISR

OSSched()

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3                            /* Allocate storage for CPU status register     */
    OS_CPU_SR  cpu_sr = 0;
#endif

    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) {               /* Schedule only if all ISRs done and ...      */
        if (OSLockNesting == 0) {          /* ... scheduler is not locked                 */
            OS_SchedNew(); /* OS_SchedNew() determines highest priority Ready task */
            if (OSPrioHighRdy != OSPrioCur) { /* No Ctx Sw if current task is highest rdy*/
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++; /* Inc. # of context switches to this task*/
#endif
                OSCtxSwCtr++;                  /* Increment context switch counter */
                OS_TASK_SW();   /* Perform a context switch - really a call to OSCtxSw() */
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```

OSSchedNew() – finds highest pri Ready task

```
static  void  OS_SchedNew (void)
{
#if OS_LOWEST_PRIO <= 63                         /* See if we support up to 64 tasks                   */
    INT8U   y;

    y              = OSUnMapTbl[OSRdyGrp];
    OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
#else                                            /* We support up to 256 tasks                         */
    INT8U   y;
    INT16U *ptbl;


    if ((OSRdyGrp & 0xFF) != 0) {
        y = OSUnMapTbl[OSRdyGrp & 0xFF];
    } else {
        y = OSUnMapTbl[(OSRdyGrp >> 8) & 0xFF] + 8;
    }
    ptbl = &OSRdyTbl[y];
    if ((*ptbl & 0xFF) != 0) {
        OSPrioHighRdy = (INT8U)((y << 4) + OSUnMapTbl[(*ptbl & 0xFF)]);
    } else {
        OSPrioHighRdy = (INT8U)((y << 4) + OSUnMapTbl[(*ptbl >> 8) & 0xFF] + 8);
    }
#endif
}
```

OSIntExit()

```
void  OSIntExit (void)
{
#if OS_CRITICAL_METHOD == 3                 /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0;
#endif
    if (OSRunning == OS_TRUE) {
        OS_ENTER_CRITICAL();
        if (OSIntNesting > 0) {             /* Prevent OSIntNesting from wrapping       */
            OSIntNesting--;
        }
        if (OSIntNesting == 0) {            /* Reschedule only if all ISRs complete ... */
            if (OSLockNesting == 0) {       /* ... and not locked.                      */
                OS_SchedNew();
                if (OSPrioHighRdy != OSPrioCur) {  /* No Ctx Sw if current task is highest rdy */
                    OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                    OSTCBHighRdy->OSTCBCtxSwCtr++; /* Inc. # of context switches to this task  */
#endif
                    OSCtxSwCtr++;                  /* Keep track of the number of ctx switches */
                    OSIntCtxSw();                  /* Perform interrupt level ctx switch       */
                }
            }
        }
        OS_EXIT_CRITICAL();
    }
}
```

Event Control Blocks (OS_EVENTs)

- ECBs are used for managing 4 types of uCOS task synchronization “subclasses”
  - Semaphores
  - Mutexes
  - Message Mailboxes
  - Message Queues
- We looked already at the APIs for operating on ECBs
- Now we’ll look at the internal data and operations
- Data consists of
  - ECB type (“subclass”)
  - Data specific to the ECB type
  - Wait list for tasks blocked on the ECB

Event Control Blocks (OS_EVENTs)

```
typedef struct os_event {
    INT8U    OSEventType;                    /* Type of event control block (see OS_EVENT_TYPE_xxxx)    */
    void    *OSEventPtr;                     /* Pointer to message or queue structure                   */
    INT16U   OSEventCnt;                     /* Semaphore Count (not used if other EVENT type)          */
#if OS_LOWEST_PRIO <= 63
    INT8U    OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                */
#else
    INT16U   OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
    INT16U   OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                */
#endif

#if OS_EVENT_NAME_SIZE > 1
    INT8U    OSEventName[OS_EVENT_NAME_SIZE];
#endif
} OS_EVENT;
```

Event Control Blocks (OS_EVENTs)

- OSEventType (“subclass”) is one of
  - OS_EVENT_TYPE_SEM, OS_EVENT_TYPE_MUTEX, OS_EVENT_TYPE_MBOX, OS_EVENT_TYPE_Q
- OSEventPtr points to the message if this is a Mailbox, else a queue structure if this is a Queue
- OSEventCnt is the semaphore counter if this is a semaphore
- OSEventTbl[] and OSEventGrp together implement the wait list using the same scheme as the Ready List.
- OSEventName[] is the name of the ECB.
  - Use OSEventNameSet(), OSEventNameGet()

Event Control Blocks (OS_EVENTs)

- Adding/Removing a task from an ECB wait list
    - This needs to be done when blocking/unblocking a task on the ECB
    - Analogous to adding/removing a task from the Ready List
    - Uses OSEventGrp and OSEventTbl of the OS_EVENT instance instead of OSRdyGrp and OSRdyTbl
    - Note: each ECB has its own OSEventGrp and OSEventTbl
- Finding the highest priority task waiting on an ECB
    - This needs to be done when deciding which task to unblock
    - Analogous to finding the highest priority task in the Ready List
    - Uses OSEventGrp, OSEventTbl and again OSUnMapTbl

Event Control Blocks (OS_EVENTs)

Making a task wait on an ECB

- Below function is called by OSSemPend(), OSMboxPend(), etc. to block the calling task on the ECB

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
    INT8U  y;

    OSTCBCur->OSTCBEventPtr =  pevent;   /* Store pointer to event control block in TCB */
    y                       =  OSTCBCur->OSTCBY;  /* Task no longer ready               */
    OSRdyTbl[y]            &= ~OSTCBCur->OSTCBBitX;
    if (OSRdyTbl[y] == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY; /* Clear bit if this was only task ready in grp*/
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX; /* Put task in wait list*/
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

Event Control Blocks (OS_EVENTs)

Making a task ready because of a Timeout

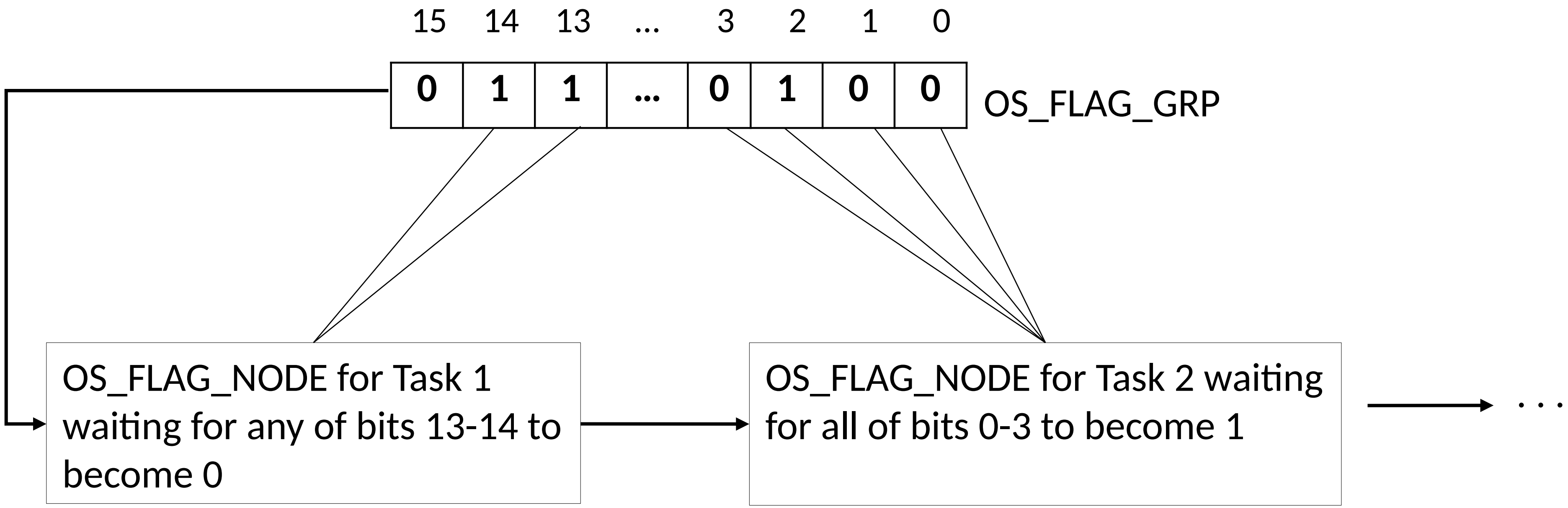
- When we do a blocking wait **with timeout**, ptcb->OSTCBDly is loaded with the timeout ticks
- OSTimeTick() decrements OSTCBDly and if it reaches 0, makes the task Ready
- Below function is called by OSSemPend(), OSMboxPend(), etc. to handle timeout on blocking wait

```
void  OS_EventTOAbort (OS_EVENT *pevent)
{
    INT8U  y;
    y                       =  OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y]  &= ~OSTCBCur->OSTCBBitX;  /* Remove task from wait list   */
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStatPend =  OS_STAT_PEND_OK;      /* Clear pend status            */
    OSTCBCur->OSTCBStat     =  OS_STAT_RDY;          /* Set status to ready          */
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;         /* No longer waiting for event  */
}
```

Event Flags

High level view of Event Flag implementation (example)

- Implemented as an OS_FLAG_GRP struct which points to a list of OS_FLAG_NODE structs
- Each OS_FLAG_NODE represents a task waiting for a combination of bits in the OS_FLAG_GRP



15	14	13	…	3	2	1	0
0	**1**	**1**	**…**	**0**	**1**	**0**	**0**

OS_FLAG_GRP

OS_FLAG_NODE for Task 1 waiting for any of bits 13-14 to become 0

OS_FLAG_NODE for Task 2 waiting for all of bits 0-3 to become 1

Event Flags

OS_FLAG_GRP typedef

```
typedef struct os_flag_grp {       /* Event Flag Group                  */
    INT8U         OSFlagType;      /* Should be set to OS_EVENT_TYPE_FLAG               */
    void         *OSFlagWaitList;  /* Pointer to first NODE of task waiting on event flag*/
    OS_FLAGS      OSFlagFlags;     /* 8, 16 or 32 bit flags                              */
#if OS_FLAG_NAME_SIZE > 1
    INT8U         OSFlagName[OS_FLAG_NAME_SIZE];
#endif
} OS_FLAG_GRP;
```

Notes:

- OSFlagType must be OS_EVENT_TYPE_FLAG (i.e. not OS_EVENT_TYPE_SEM, OS_EVENT_TYPE_MBOX, etc.)
- OSFlagWaitList points to a doubly linked list of OS_FLAG_NODES (didn’t find a good reason why it’s declared void*)
- OS_FlagFlags is the 16-bit set of flags in this flag group

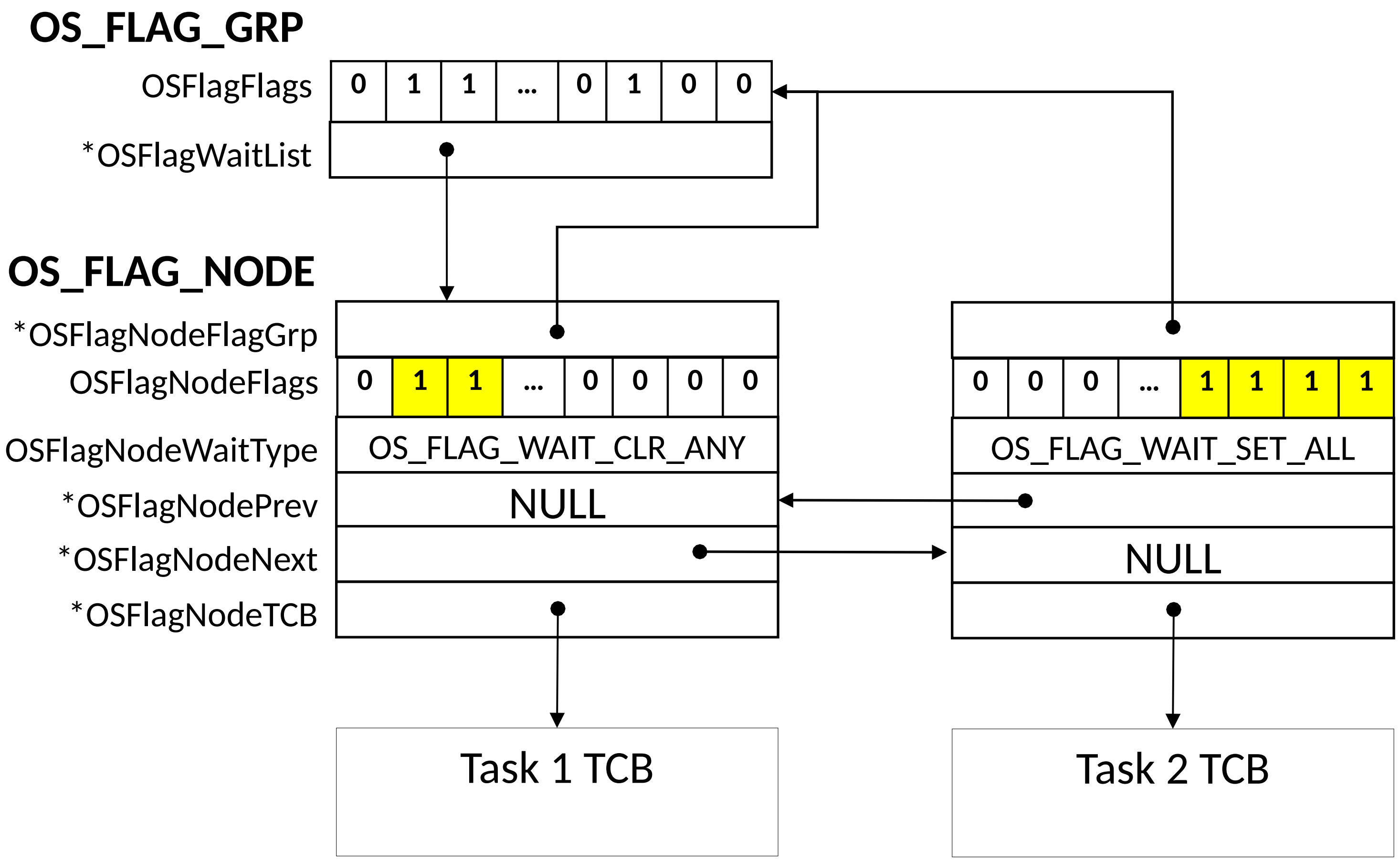
Flag Events

OS_FLAG_NODE typedef

```
typedef struct os_flag_node {               /* Event Flag Wait List Node                */
    void         *OSFlagNodeNext;           /* Pointer to next     NODE in wait list    */
    void         *OSFlagNodePrev;           /* Pointer to previous NODE in wait list    */
    void         *OSFlagNodeTCB;            /* Pointer to TCB of waiting task           */
    void         *OSFlagNodeFlagGrp;        /* Pointer to Event Flag Group              */
    OS_FLAGS      OSFlagNodeFlags;          /* Event flag to wait on                    */
    INT8U         OSFlagNodeWaitType;       /* Type of wait:                            */
                                            /*      OS_FLAG_WAIT_AND                    */
                                            /*      OS_FLAG_WAIT_ALL                    */
                                            /*      OS_FLAG_WAIT_OR                     */
                                            /*      OS_FLAG_WAIT_ANY                    */
} OS_FLAG_NODE;
```

Note: Didn’t find a good reason why pointers are all void*

Flag Events



OS_FLAG_GRP

OSFlagFlags: 0 1 1 … 0 1 0 0

*OSFlagWaitList

OS_FLAG_NODE

*OSFlagNodeFlagGrp		
OSFlagNodeFlags	0 1 1 … 0 0 0 0	0 0 0 … 1 1 1 1
OSFlagNodeWaitType	OS_FLAG_WAIT_CLR_ANY	OS_FLAG_WAIT_SET_ALL
*OSFlagNodePrev	NULL	
*OSFlagNodeNext		NULL
*OSFlagNodeTCB		

Task 1 TCB

Task 2 TCB

Waiting on a Flag Group

- Allocate an OS_FLAG_NODE from free list and insert it at front of OSFlagWaitList
- Initialize the fields and point OSFlagNodeTCB to the current TCB
- Remove the current TCB from Ready List and change its status to OS_STAT_FLAG
- Reschedule

Posting to a Flag Group

- Update the OSFlagFlags in OS_FLAG_GRP with the posted value
- Chain through OSFlagWaitList and for any OS_FLAG_NODEs whose bit patterns are satisfied:
  - Make Ready the linked TCB
  - Delete the OS_FLAG_NODE and return it to free list
- Reschedule

Summary of uCOS Internals

- Task States
  - uCOS keeps the Running task in the Ready List
  - Tracks several types of Wait (blocked) states
  - Does not track Dormant, Running, or Interrupted states
- Task Control Blocks (TCBs)
  - The TCB contains all the information needed to manage the Task
  - Are all preallocated at compile time
  - Are initialized individually at Task Creation time
- Ready List
  - Implemented as a set of tables which allow insertion, deletion, and lookup of the highest priority task in constant time regardless of number of tasks.

- Task Scheduling
  - Handled by two routines:
  - OSSched() handles context switch from one task to another
  - OSIntExit() handles context switch from ISR code to the highest priority task but only if interrupt nesting level is 0
- Event Control Blocks
  - Provide a uniform API and management framework for 4 “subclasses” of ECB: semaphores, mutexes, message mailboxes, and message queues.
  - Repurpose the Ready List table lookup scheme to track tasks blocked on any “subclass” instance
- Event Flags
  - Enable tasks to block for arbitrary events signaled by other tasks or ISRs
  - Implemented as a Flag Group node of reference bits referenced by a doubly linked list of blocked TCBs waiting for a TCB-specific bit pattern to occur in the Flag Group.

Task Sync Techniques

- We will look at some common issues for synchronizing tasks
- The goal is to have a bag of tricks for handling synchronization problems
- Synchronization techniques must
  - Avoid lengthy ISRs
  - Avoid shared resource corruption when multiple tasks operate on a shared resource simultaneously
  - Avoid deadlock – when two or more tasks are blocked waiting for a resource held by one of the other tasks.
  - Avoid starvation – where one or more tasks are starved for access to a resource

Task Sync Techniques

Issues around sharing resources between ISRs and tasks

- Typical scenario:
  - interrupt occurs when new data is ready to be processed
  - ISR moves the data to a shared location (buffer) for a task to process
  - How to synchronize the ISR and task to guard the buffer?
- ISRs cannot do blocking waits for resources (uCOS prevents that by returning an error code if you try).
- ISRs should obviously not spin while waiting for a resource
- ISRs simply cannot wait. Period.
- So what should they do?

Task Sync Techniques

A wrong way: ISR moves new data to the buffer and posts to a semaphore to have a task process the new data

- Task code blocks and waits for a post to allow it to process new data
- ISR code moves new data to a shared buffer, then posts to the semaphore to allow it to process the input
- Note: ISRs can’t use uCOS Mutexes: only task code can acquire a mutex and only the owner can release it

ISR Code

```
ISR() {
    // Move new data to the
    // shared buffer
    Post(mySem);
}
```

Task Code

```
while (1) {
    Pend(mySem);
    // Remove data from the
    // shared buffer;
    // process the data
}
```

Task Sync Techniques

Problem:

- We’re using a semaphore to protect our shared resource, so we should be good, right? Wrong! Semaphores always require careful thought
- What happens if another interrupt from the same source occurs while the task is accessing the buffer?
- In general we are not going to keep interrupts disabled while the task is processing the previous data
- How can we guard the shared buffer?

ISR Code

```
ISR() {
    // Move new data to the
    // shared buffer
    Post(mySem);
}
```

Task Code

```
while (1) {
    Pend(mySem);
    // Remove data from the
    // shared buffer;
    // process the data
}
```

Task Sync Techniques

A Solution: **Use a message passing scheme**

- Use a message passing scheme such as a mailbox to pass the buffer pointer back and forth between task and ISR.
- Now if an interrupt from the same source happens before the task has removed the previous message, we'll lose the new data but at least we won't corrupt the old data
- If we can't afford to lose the new data, use a sufficiently long **message queue** instead of a mailbox.

ISR Code

```
ISR() {
    if (Accept(mBox, msg)) {
      // move new data to the
      // msg buffer
      Post(mBox, msg);
    }
}
```

Task Code

```
while (1) {
    msg = Pend(mBox);
    // process the msg
    Post(mBox, msg);
}
```

Task Sync Techniques

Producer-Consumer Problem

- In the absence of message passing services, we can use semaphores
- The classical Producer-Consumer Problem involves synchronizing a Producer Task which deposits data in a buffer and a Consumer Task which takes data out of the same buffer
- With no synchronization, the result is chaos with the producer overwriting the previous buffer contents while the consumer is removing data

Producer Task Code

```
while (1) {
    // get new data
    // deposit data in buffer
}
```

Consumer Task Code

```
While (1) {
    // remove data from buffer
    // process data
}
```

Task Sync Techniques

Producer-Consumer Problem

- Can we protect the buffer with just one semaphore?
- Uh-oh, that's no good – while the Producer is getting new data the Consumer can try to remove a message multiple times
- While the Consumer is processing data the Producer can overwrite the buffer with new data multiple times

Producer Task Code

```
while (1) {
    // get new data
    Pend(guardSem);
        // deposit data in
        // buffer
    Post(guardSem);
}
```

Consumer Task Code

```
While (1) {
    Pend(guardSem);
        // remove data from
        // buffer
    Post(guardSem);
    // process data
}
```

Task Sync Techniques

Producer-Consumer Problem

- How about adding another semaphore to block the Producer till the Consumer is ready for new data?
- Initialization
  - guardSem <- 1
  - emptySem <- 1
- Uh-oh the Consumer can still spin removing the same message multiple times while the Producer is getting new data

Producer Task Code

```
while (1) {
    // get new data
    Pend(emptySem);
    Pend(guardSem);
        // deposit data in
        // buffer
    Post(guardSem);
}
```

Consumer Task Code

```
While (1) {
    Pend(guardSem);
        // remove data from
        // buffer
    Post(guardSem);
    // process data
    Post(emptySem);
}
```

Task Sync Techniques

Producer-Consumer Problem

- How about one more semaphore to block the consumer till the Producer has new data?
- Initialization
  - guardSem <- 1
  - emptySem <- 1
  - fullSem <- 0
- Now we're OK.

Producer Task Code

```
while (1) {
    // get new data
    Pend(emptySem);
    Pend(guardSem);
        // deposit data in
        // buffer
    Post(guardSem);
    Post(fullSem);
}
```

Consumer Task Code

```
While (1) {
    Pend(fullSem);
    Pend(guardSem);
        // remove data from
        // buffer
    Post(guardSem);
    Post(emptySem);
    // process data
}
```

Task Sync Techniques

Producer-Consumer Problem

- Can generalize this to where the message buffer is a queue with **N** slots
- Initialization
  - guardSem <- 1
  - fullSem <- 0
  - **emptySem <- N**
- Allows up to N messages to be deposited in the queue before blocking the Producer until the Consumer removes a message
- Meanwhile the Consumer can remove a message any time fullSem is > 0

Producer Task Code

```
while (1) {
     // get new data
     Pend(emptySem);
     Pend(guardSem);
          // deposit data in
          // queue
     Post(guardSem);
     Post(fullSem);
}
```

Consumer Task Code

```
While (1) {
     Pend(fullSem);
     Pend(guardSem);
          // remove data from
          // queue
     Post(guardSem);
     Post(emptySem);
     // process data
}
```

Task Sync Techniques

Producer-Consumer Problem

- How about if the Producer is an ISR?
- ISRs can’t do blocking waits. Can we do it with non-blocking waits?
- **No: can’t use this scheme as-is if Producer is an ISR**
- There may be available slots in the queue but if the Consumer is actively removing data from the queue the guardSem semaphore will not be available and the ISR will have to throw away the new data
- **Solution**: rather than use guardSem, disable interrupts while adding/removing data from the queue …

ISR Producer Code

```
ISR() {
    // get new data
    if (Accept(emptySem)) {
        if (Accept(guardSem)) {
            // deposit data in
            // queue
            Post(guardSem);
            Post(fullSem);
        }
        else Post(emptySem);
    }
}
```

Task Consumer Code

```
While (1) {
    Pend(fullSem);
    Pend(guardSem);
        // remove data from
        // queue
    Post(guardSem);
    Post(emptySem);
    // process data
}
```

Task Sync Techniques

Producer-Consumer Problem

- Solution if Producer is an ISR
- Deposit and removal from the queue must be done with interrupts off
- Otherwise if we use a semaphore the consumer can be active in the critical section thus preventing the ISR from operating on the queue

ISR Producer Code

```
ISR() {
    // get new data
    if (Accept(emptySem)) {
        DisableInterrupts();
            // deposit data in
            // queue
        EnableInterrupts();
        Post(fullSem);
     }
}
```

Task Consumer Code

```
While (1) {
    Pend(fullSem);
    DisableInterrupts();
        // remove data from
        // queue
    EnableInterrupts();
    Post(emptySem);
    // process data
}
```

Task Sync Techniques

Readers and Writers Problem

- This problem has multiple readers reading from shared data who need to synchronize with writers who update the shared data
- If no writer is active, any number of readers may read the data simultaneously
- Only one writer may update the data at a time and no readers may read while the writer is writing
- Initialize
    - rc <- 0   (int reader count)
- This solution (from Tanenbaum) may starve writers. True for uCOS?
- Other solutions can be found in the literature

Writers Code

```
while (1) {
    // get new data
    Pend(wMutex);
        // write new data
        // to shared data
    Post(wMutex);
}
```

Readers Code

```
While (1) {
    Pend(rcMutex);
    rc += 1;
    if (rc == 1) Pend(wMutex);
    Post(rcMutex);
    // read shared data
    Pend(rcMutex);
    rc -= 1;
    if (rc == 0) Post(wMutex);
    Post(rcMutex);
    // process the data
}
```

Task Sync Techniques

Deadlock

- Deadlock or Deadly Embrace can happen when multiple tasks try to acquire a set of resources in different orders:
  - Task 1 gets mutexA
  - Task1 gets preempted by Task 2
  - Task2 gets mutexB
  - deadlock
- One way to reduce the chance of deadlock is to always acquire in name-sorted order
- Easier said than done if one semaphore is acquired at one place in the code and a second one later at some distant place in the code.

Task 1

```
while (1) {
    Pend(mutexA);
    Pend(mutexB);
        // use resources A and B
    Post(mutexB);
    Post(mutexA);
}
```

Task 2

```
while (1) {
    Pend(mutexB);
    Pend(mutexA);
        // use resources A and B
    Post(mutexA);
    Post(mutexB);
}
```

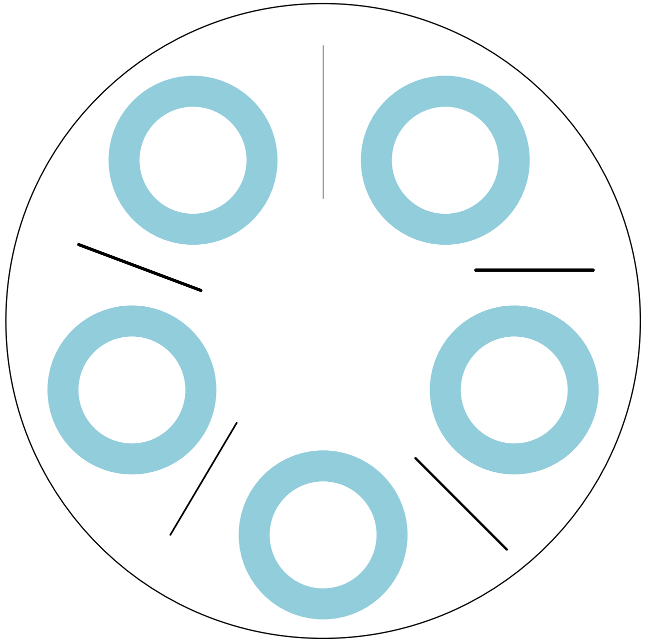
Task Sync Techniques

Non-solution:

```
void philosopher(int i) {
  while (1) {
    think();
    takeChopstick(i);
    takeChopstick((i+1) % N);
    eat();
    putChopStick(i);
    putChopStick((i+1) % N);
  }
}
```

Deadlock/Starvation Avoidance

- **Dining Philosophers Problem**
- N philosophers each spend their time in an endless loop: think; eat; repeat;
- The catch is that chopsticks are shared resources



- Solutions to such problems can require much thought to avoid both deadlock in contention for resources and starvation where one task rarely or never gets access to shared resources.
- There are numerous solutions to the Dining Philosphers. See Tanenbaum, Wikipedia, etc.

Task Sync Techniques

Summary of Task Sync Techniques

- Wrong ways and right ways to share data between ISRs and tasks
- Producer-Consumer Problem
    - Task-based exploration
    - ISR-as-Producer exploration
- Readers and Writers Problem
- Deadlock and Starvation
    - Example deadlock due to incorrect acquisition order of mutexes
    - Name-sorted-order acquisition of mutexes as deadlock avoidance technique
    - Dining Philosophers Problem

Event Driven Systems

An Event Driven System is composed of

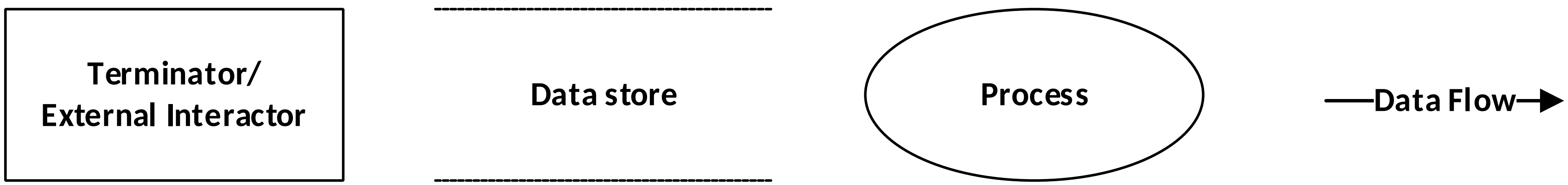
- Event Emitters (or Agents)
    - responsible for detecting, gathering and transferring events
- Event Consumers (or Sinks)
    - responsible for providing a reaction to a received event. The reaction may consist of filtering and forwarding the filtered event
- Event Channels
    - conduct the events from Emitter to Consumer

The Graphical User Interface is the canonical example of an Event Driven System

- Event Channel is a message passing queue where Emitters post messages to the queue and Consumers consume messages from the queue
- Emitters are keyboard, mouse
    - Events are key-press messages, mouse movement- and click-messages
- Consumers are windows which block for messages placed in the event channel by Emitters

Event Driven Systems

Data Flow Diagrams (DFDs)

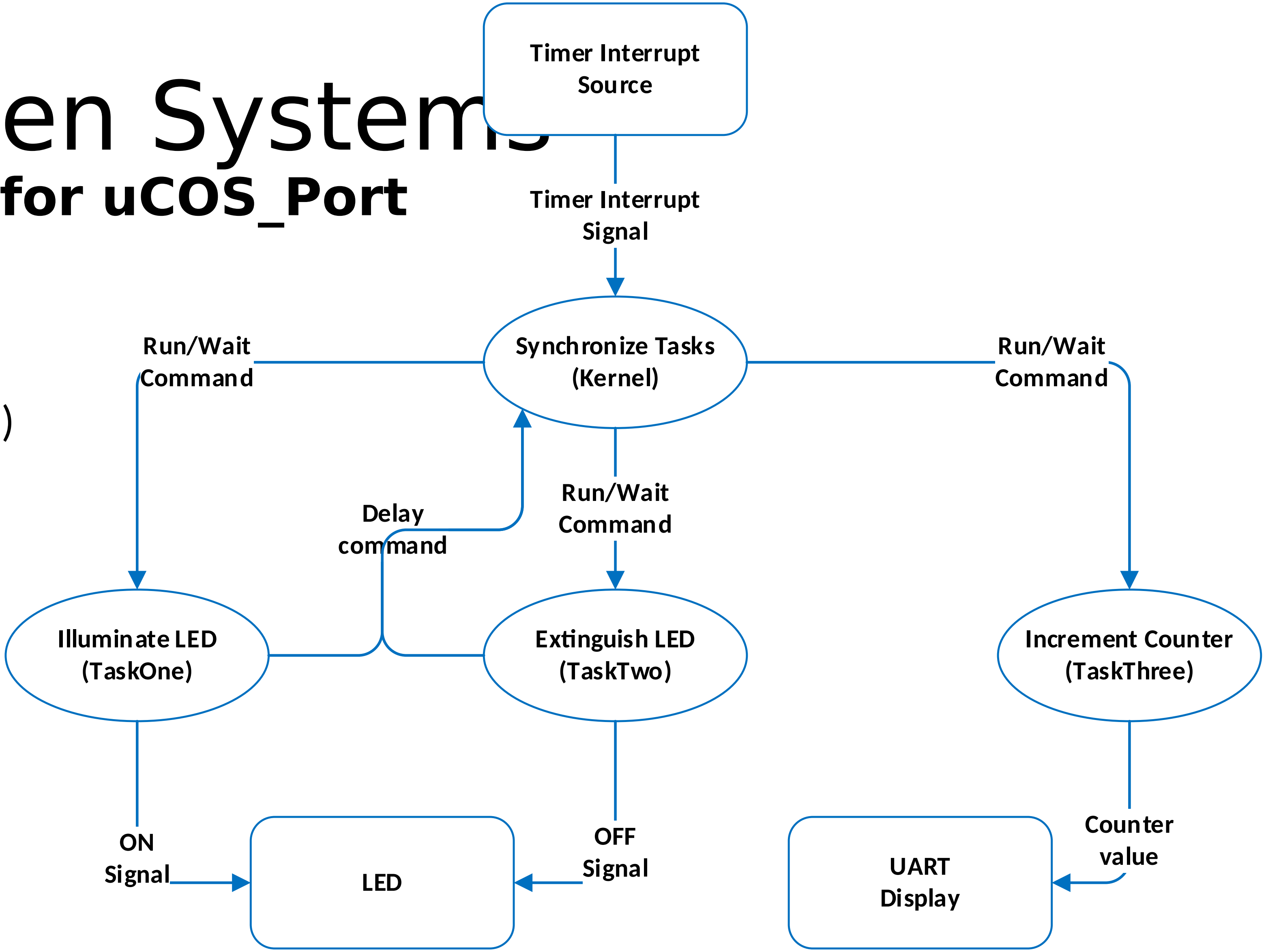


- A useful abstraction tool for many kinds of software systems (not just Event Driven Systems)
- Composed of 4 basic primitives: Terminators, Data stores, Processes, and Data Flows
- Less detailed than Flow Charts
- Useful at the design level for thinking through the design
- Not a specification but a way of explaining the behavior of the system
- In Embedded Event Driven Systems, *Process* nodes can often map to Tasks and *Data Flow* links can map to passed messages/command-codes

Event Driven Systems

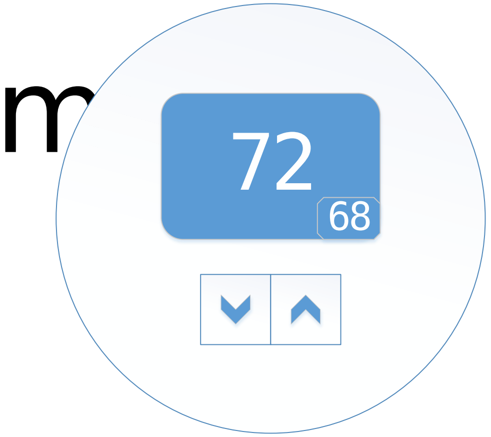
DFD Abstraction for uCOS_Port

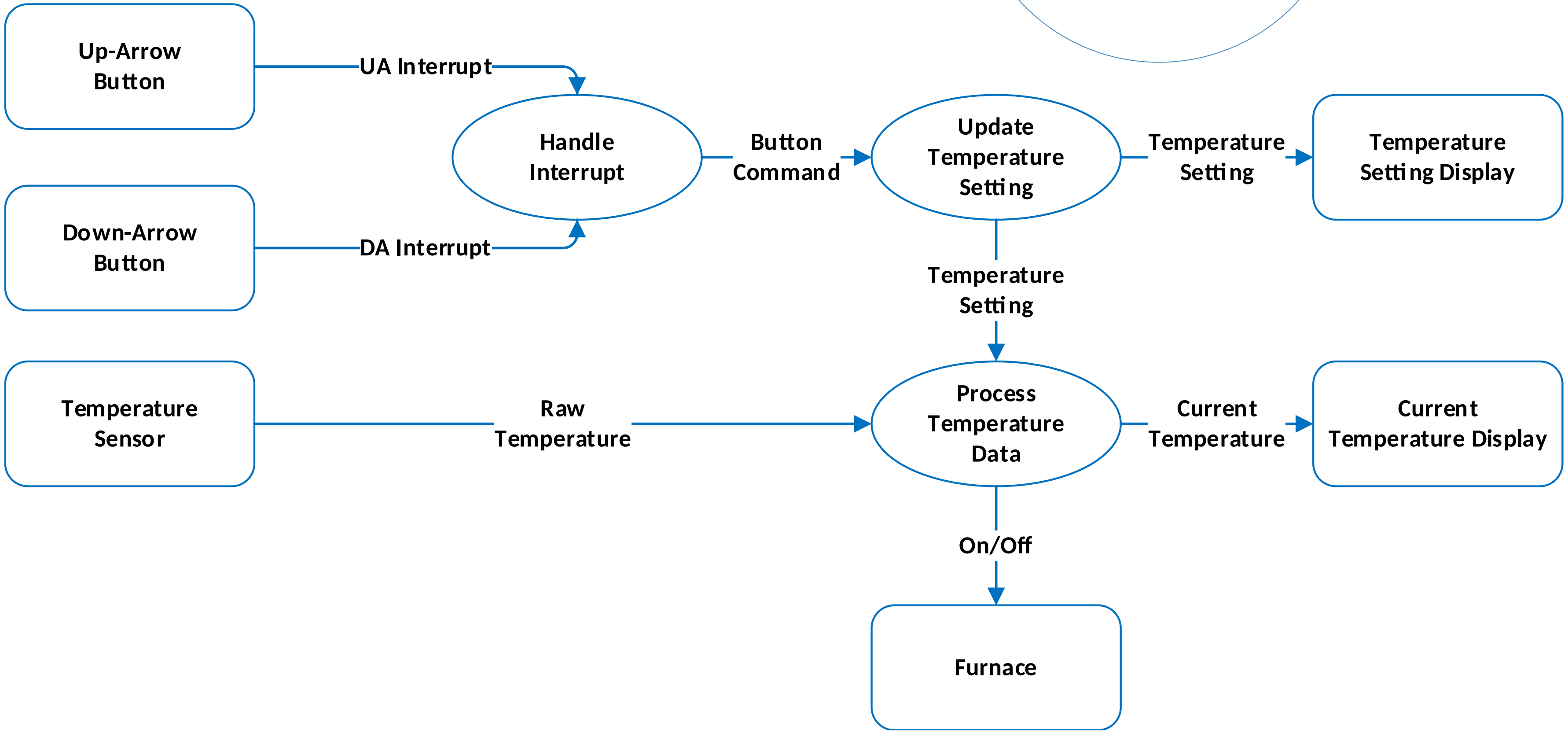
- Processes (ovals) get imperative verb labels
- Terminators (rectangles) get noun labels
- Data flows (arrows) get noun labels



Event Driven System

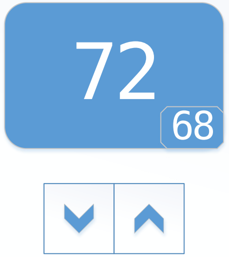
DFD for button-driven system - interrupt driven buttons

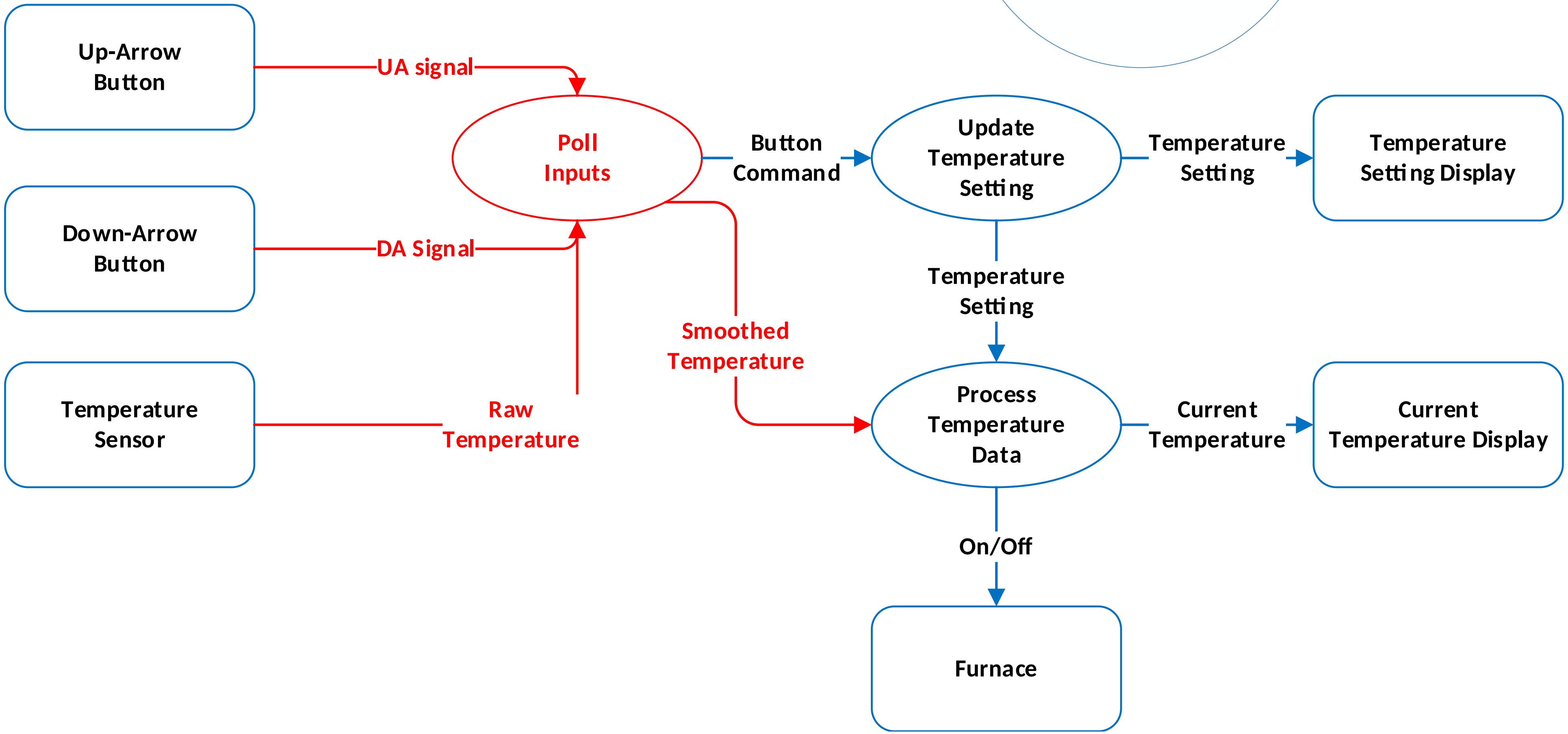




Event Driven Systems

DFD for button-driven system
- No button interrupts (polling of b




Up-Arrow Button
UA signal
Poll Inputs
Down-Arrow Button
DA Signal
Temperature Sensor
Raw Temperature
Button Command
Smoothed Temperature
Update Temperature Setting
Temperature Setting
Temperature Setting Display
Temperature Setting
Process Temperature Data
Current Temperature
Current Temperature Display
On/Off
Furnace