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 3 1/20/2020

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 (L2) Overview

- Revisit LDREX, STREX
- Embedded Operating System Concepts Part 2 (Labrosse Chapter 2)
  - Multitasking OS concepts
    - The kernel, Preemptive and non-preemptive kernels, The scheduler, Scheduling algorithms, Task states, Parameters used in scheduling – quantum, priority, Priority inversion
  - Real time OS concepts
    - Definition, Hard/soft real time, Jitter, OS timer tick, Interrupt latency
- Context switching on ARM Cortex (Yiu 10.4, 10.5)
- Assignment 2: Context Switch (due in 1 week)

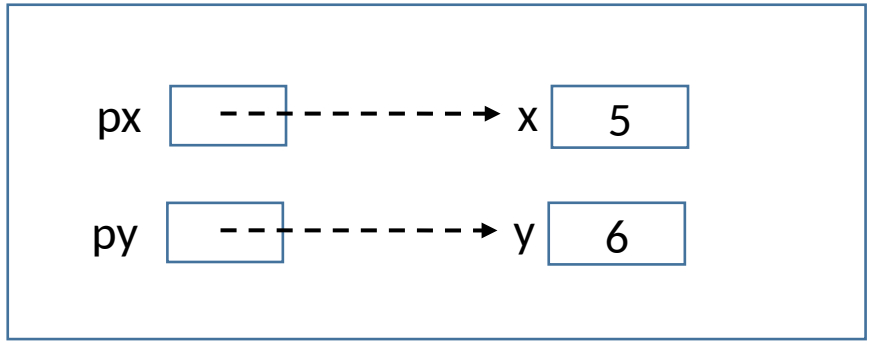
Current Lecture Overview

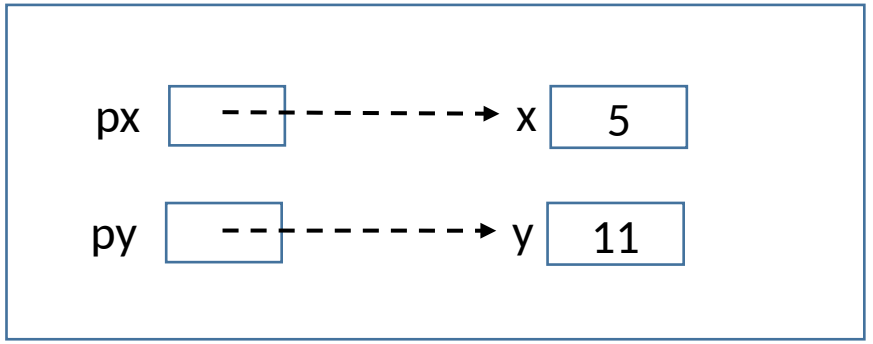
- C pointers
- MicroC/OS-II (uCOS) Introduction
  - Task creation, Task Delay, Sample task code, uCOS startup steps
- Porting uCOS to our board (**Labrosse** *Ch 13*)
  - Key data definitions, enabling and disabling interrupts, critical section implementation, initializing the stack, context switching, 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

C pointers

```
int x = 5;

int* px = &x;    // The value of px is the address of x

int y = 6;

int* py = &y;   // The value of py is the address of y

    *(pointer_to_data) is “the value stored at address data”

printf(“%d”, *px);   //   5       “dereference” px

*py = *px + *py;  // value of y = value of x + value of y

Same result as y = x + y
```

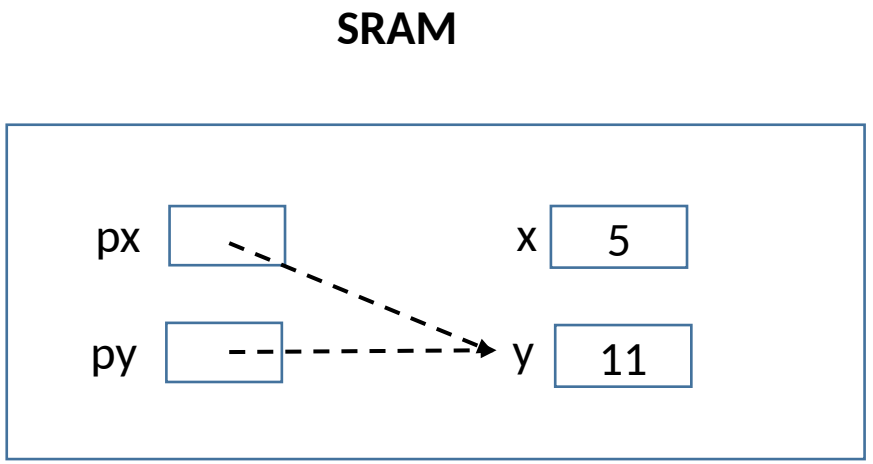
SRAM



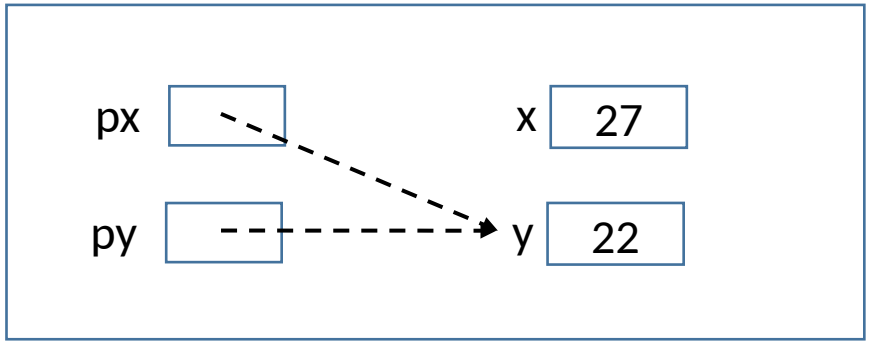


C pointers

px = py; // px points to what py points to

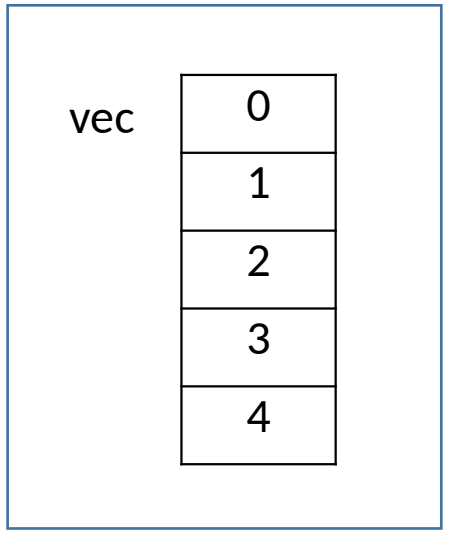


*py = *px + *py;   // 11 + 11 = 22
x += *py;   // 5 + 22 = 27



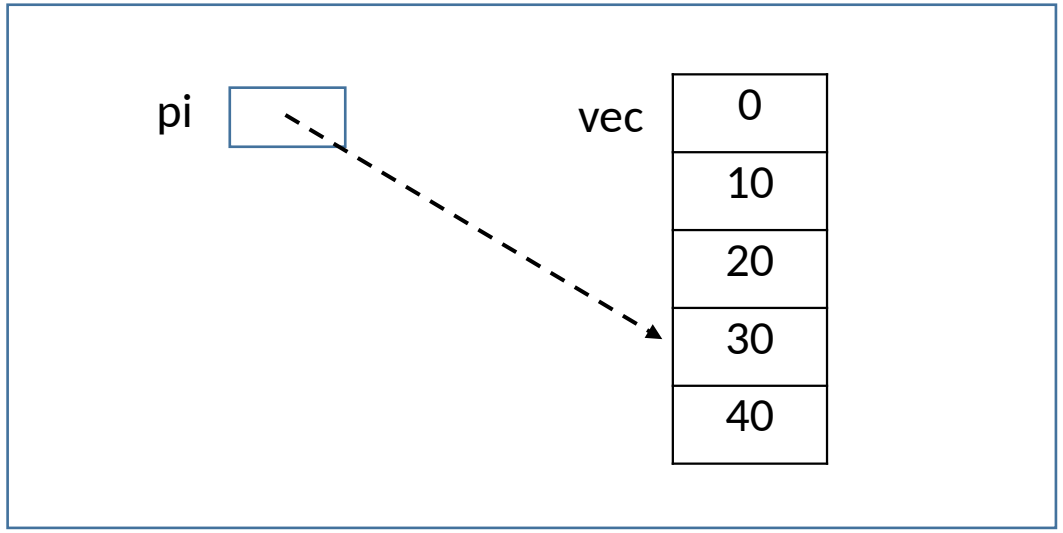
C pointers

```
int vec[5];

for (int i = 0 ; i < 5; ++i) {
        vec[i] = i;
}
```

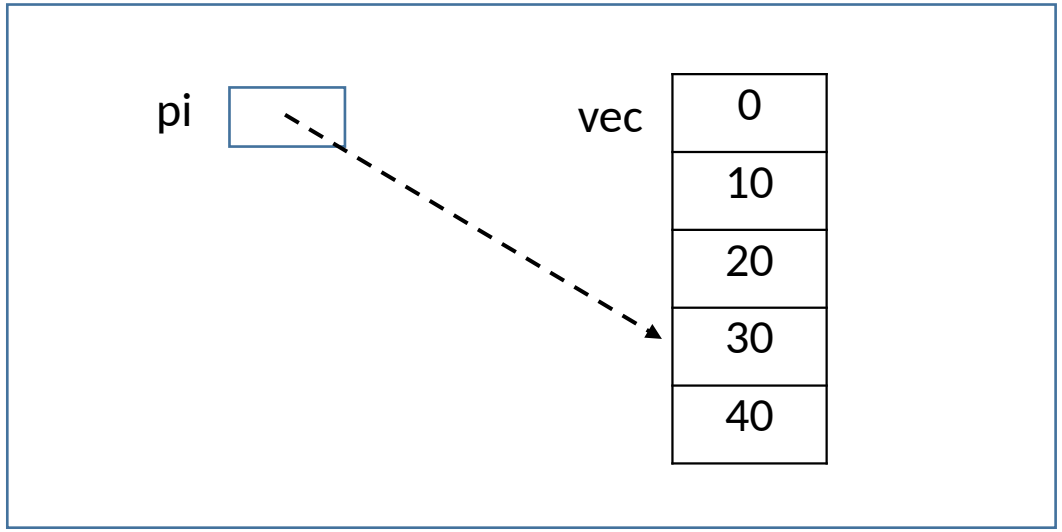


SRAM

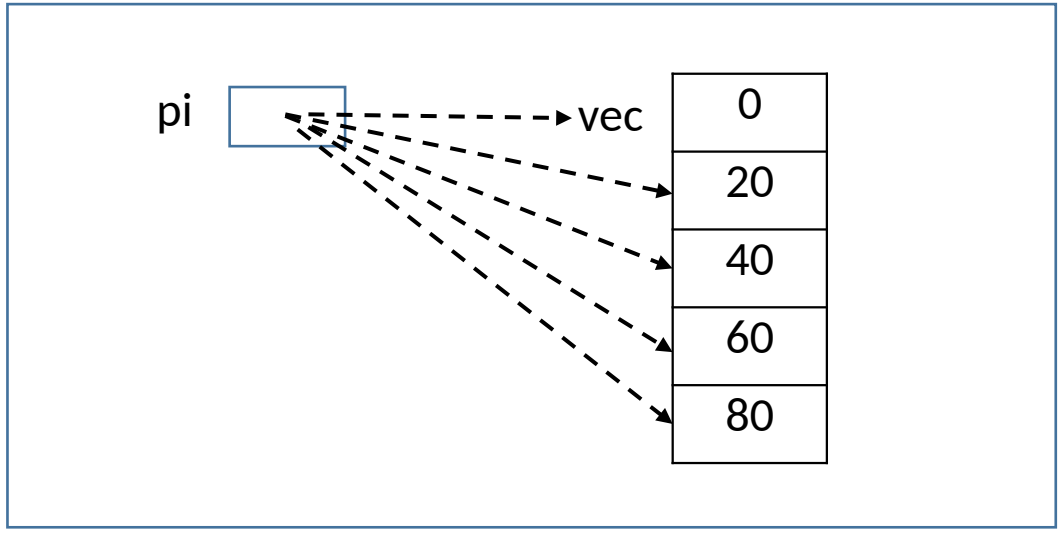
```
int* pi;
pi = &vec[3]; // “address of”
*pi = 30;
pi[1] = 40;
*(pi – 1) = 20;
pi[-2] = 10;
```



C pointers

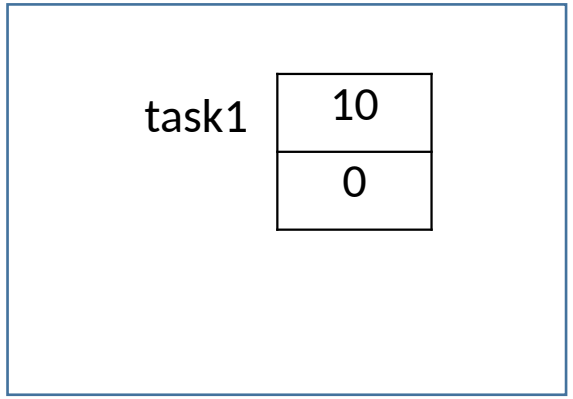



```
for (pi = vec; pi <= &vec[4]; ++pi) {
        *pi = 2 * (*pi);
}
```



C structs

```
enum Status {
    ready,
    running,
    blocked
}

struct TaskControl{
    int id;
    Status status;
};

TaskControl task1;
task1.id = 10;
task1.status = ready;
```



‘Dot’ Operator ‘.’

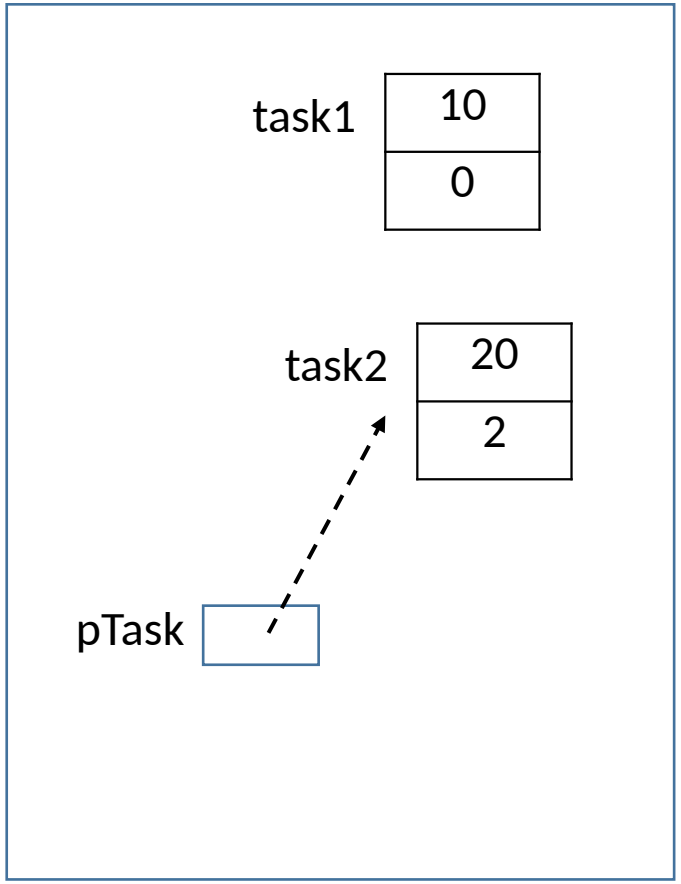
- Given an instance of a struct, use the dot operator to offset to fields in the struct

C structs

‘Arrow’ operator ‘->’

- Given a *pointer* to an instance of a struct, use the arrow operator to offset to fields in the struct

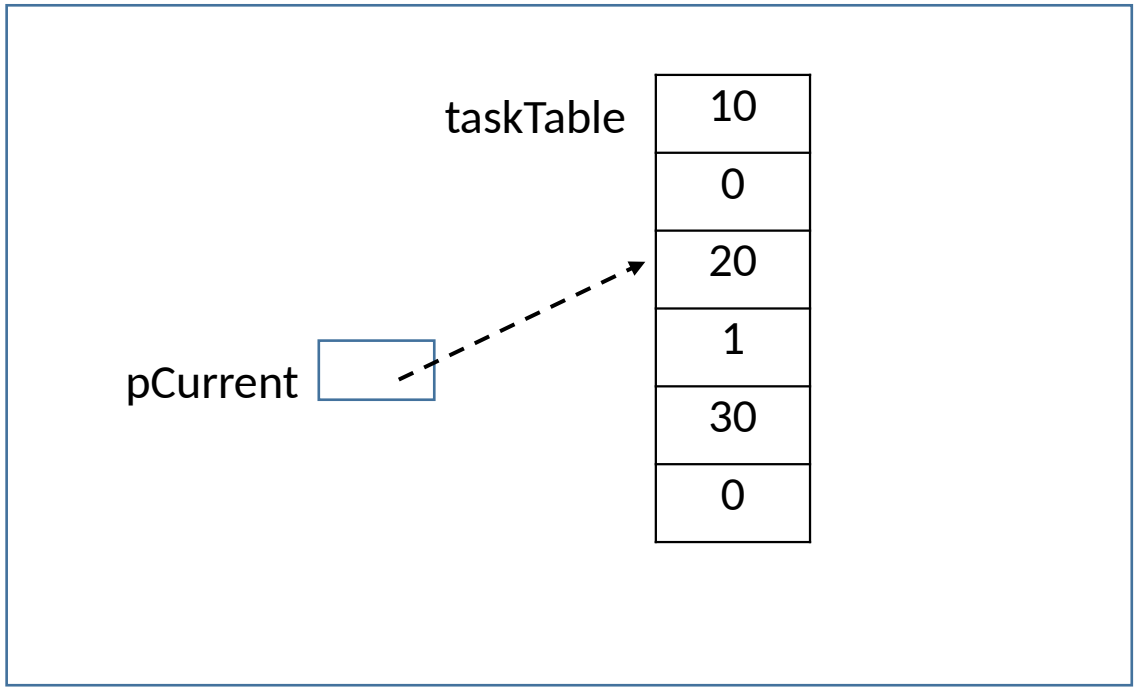
```
TaskControl task2;

TaskControl* pTask = &task2;

pTask->id = 20;
pTask->status = blocked;
```



C structs

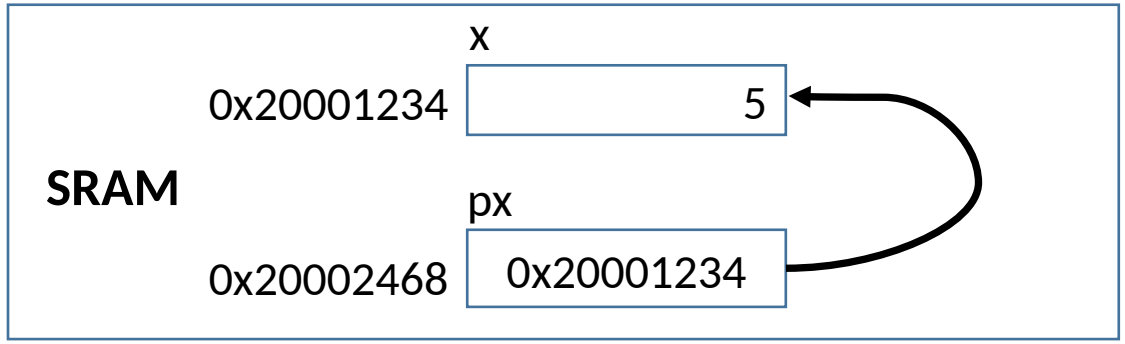
Array of structs

```
TaskControl taskTable[3];

taskTable[0].id = 10;
taskTable[0].status = ready;

taskTable[1].id = 20;
taskTable[1].status = ready;

taskTable[2].id = 30;
taskTable[2].status = ready;

TaskControl* pCurrent;
pCurrent = &taskTable[1];
pCurrent->status = running;
```



SRAM

C pointers and assembly language



Var name	Memory address	value
x	0x20001234	5
px	0x20002468	0x20001234

C code

```
int x;
int* px;

x = 5;
px = &x;    // pointer to x. The value of px is the address of x

Print_uint32(x);      // 5
PrintHex(&px);        // 0x20002468
PrintHex(px);         // 0x20001234
Print_uint32(*px);  // 5
```

Assembly language:

```
LDR   R0,=x       ; R0 = &x
LDR   R1,#5       ; R1 = 5
STR    R1, [R0]    ; x = 5

LDR   R1,=px      ; R1 = address of px = 0x20002468
STR    R0, [R1]    ; px = &x = 0x20001234

LDR   R0,=px      ; R0 = &px = 0x20002468
LDR   R0,[R0]     ; R0 = value of px = 0x20001234 = &x
LDR   R0,[R0]     ; R0 = *px = 5
```

MicroC/OS-II introduction

- I’ll refer to it usually as uCOS
- Is an embedded real time OS
- Certified for safety critical applications by the FAA
- This means every line of code has been tested and verified to be safe for applications where human life is on the line.
- Used in hundreds of products: Avionics, Medical equipment/devices, Data communications equipment, White goods (appliances), Mobile phones, Industrial controls, Consumer electronics, Automotive
- Source code is available and free for noncommercial purposes so can port to new hardware
- Implements preemptive priority scheduling with priority inheritance to avoid priority inversion
- Each task **must** have a **unique priority**

MicroC/OS-II introduction

Task creation

```
INT8U OSTaskCreate(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio)
```

`void (*task)(void *p_arg)`

- **`task`** is the address of the function that implements the task
- Takes one argument so you can pass a pointer to memory with initialization data for the task

`void *p_arg`

- The argument to pass to the task at task startup

`OS_STK *ptos`

- pointer to the task's top-of-stack

`INT8U prio`

- **UNIQUE** priority for the task

Returns `OS_ERR_NONE` on success otherwise returns an error code.

MicroC/OS-II introduction

Task delay

```
void  OSTimeDly (INT16U ticks)
```

- Delays the task for the given number of timer ticks
- A function like this is often used in priority scheduling kernels as a way to allow lower priority tasks to run

MicroC/OS-II introduction

Sample task code

```
void TaskOne(void* pdata)
{
    char buf[BUFSIZE];
    PrintWithBuf(buf, BUFSIZE, "TaskOne: starting\n");

    while(1)
    {
       OSTimeDly(100);
       PrintWithBuf(buf, BUFSIZE, "TaskOne: Turn LED On\n");
        SetLED(OS_TRUE);
    }
}
```

MicroC/OS-II introduction

Example: Creating a task

```
static OS_STK   Task1Stk[APP_CFG_TASK_START_STK_SIZE];

OSTaskCreate(
   TaskOne,
   (void*)0,
   &Task1Stk[APP_CFG_TASK_START_STK_SIZE-1],
APP_TASK_TEST1_PRIO);
```

MicroC/OS-II introduction

uCOS Startup steps

In main():

1. Initialize the hardware and disable interrupts
2. Initialize uCOS internals by calling OSInit()
3. Create one uCOS task – the “startup” task
4. Call OSStart() to have the “startup” task resume and kick off uCOS

MicroC/OS-II introduction

The “startup” task

1. Starts the tick timer
2. Creates your app tasks
3. Calls OSTaskDel(OS_PRIO_SELF) to have the “startup” task remove itself. Its job is done and it should be removed from the possibility of further execution.

MicroC/OS-II introduction

Example “startup” task

```
void StartupTask(void* pdata)
{
   // Start the system tick (do it here rather than hw_Init())
   SysTick_Config(CLOCK_HSI / OS_TICKS_PER_SEC);

   OSTaskCreate(TaskOne, (void*)0, &Task1Stk[APP_CFG_TASK_START_STK_SIZE-1], APP_TASK_TEST1_PRIO);
   OSTaskCreate(TaskTwo, (void*)0, &Task2Stk[APP_CFG_TASK_START_STK_SIZE-1], APP_TASK_TEST2_PRIO);
   OSTaskCreate(TaskThree, (void*)0, &Task3Stk[APP_CFG_TASK_START_STK_SIZE-1], APP_TASK_TEST3_PRIO);

   // Delete ourselves, letting the work be done in the new tasks.
   OSTaskDel(OS_PRIO_SELF);
}
```

MicroC/OS-II introduction

Notes about startup

- Notice that there was no step required to initialize the task's stack in the startup steps
- However the "startup" task was resumed i.e. had its context switched to.
- How was that? – because task stack initialization is handled internally by uCOS using a hardware dependent function we have to supply as part of porting uCOS to our board.

MicroC/OS-II introduction

Summary of uCOS introduction

- Task creation
- Task Delay
- Sample task code
- uCOS startup steps
- Start up task
  - Kicks off tick interrupt
  - Launches all your application tasks
  - Removes itself

Next – Porting uCOS

Porting uCOS

- See Labrosse Ch 13
- We can build the OS using our toolchain, load it onto our board and use it to run applications
- uCOS Porting Requirements (applies to most any multitasking OS)
  - A compiler that can generate reentrant (thread safe) code (see next slide)
  - Processor supports interrupts, particularly a timer tick in the range 10-100 Hz
  - Interrupts can be enabled/disabled from C
  - Processor supports a hardware stack with capacity for potentially many kilobytes
  - Processor has instructions to load and store the CPU registers and stack pointer.

Porting uCOS

What makes code generated by a compiler reentrant?

```
MyFunction(int arg1)
{
    int var1
    …
}
```

- Reentrancy requires that function arguments and local variables be allocated on the task's stack so even if two or more threads are active inside MyFunction at the same time, they each get their own instances of arg1 and var1 and so don't interfere with each other.
- Of course even if a compiler generates thread safe code a function can still be rendered non-reentrant (non-thread-safe) if it operates on global or static data without critical section protection.

Porting uCOS

- Most of uCOS is written in C
- But uCOS source code won't actually work out-of-the-box – it's missing hardware specific code for interrupt handling, context switching and stack initialization
- Coding conventions
  - The book advocates for one big header containing all necessary definitions:

    ```
    #include "includes.h"
    ```

  - uCOS has since backed off that so we don't use includes.h, however we have bsp.h which fulfills an equivalent purpose for us.
  - "Namespace": most all uCOS identifiers are named with the prefix "OS" – OS_ENTER_CRITICAL(), OS_EXIT_CRITICAL(), OSTaskCreate(), etc.

Porting uCOS

- The changes required to port the OS to new hardware consist of implementing a relatively small number of uCOS interfaces.
- There are 5 key areas that require coding
  - Data type definitions like INT32S, INT32U, INT8U, etc.
  - Code to implement critical sections (by disabling/enabling interrupts)
  - Code to initialize a task's stack
  - Code to switch context
  - Code to implement ISRs, especially the OS timer tick

Porting uCOS

- Assumptions/simplifications for porting uCOS to Cortex-M4
  - No floating point
    - no stacking of floating point registers for context switching
    - Caveat: don’t use floating point in your applications
  - Use only Main stack pointer
    - Avoids debugging issues involving two hardware stacks
    - all code executes at privileged elevation

Porting uCOS

Key data definitions

- Stack element type – in our case each stack entry is 32 bits wide

```
typedef unsigned int   OS_STK;
```

- Stack growth direction: 0 means low memory to high, 1 means high memory to low. Ours is:

```
#define  OS_STK_GROWTH          1
```

- CPU status register type - used for saving interrupt status before entering critical section – ours is 32 bits wide (PRIMASK)

```
typedef unsigned int   OS_CPU_SR;
```

Porting uCOS

Enabling and disabling interrupts

- uCOS defines 3 ways of handling the “CPU status” when disabling interrupts – implementer chooses one to implement depending on hardware capabilities

1. Simple – no saving CPU status before disable, no restoring CPU status after re-enable – i.e. no capability of handling nested interrupts
2. Stack based – CPU status is pushed before disabling interrupts, popped when re-enabling – allows nested interrupts
3. Save/restore using a local variable – CPU status is saved to a local variable before disabling, then restored from that variable when re-enabling – allows nested interrupts

- **Our uCOS port uses option 3 – save/restore using a local variable**

Porting uCOS

Critical section implementation in uCOS

```
#define  OS_ENTER_CRITICAL()  {cpu_sr = CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {CPU_SR_Restore(cpu_sr);}
```

CPU_SR_Save() internal helper function

- Coded in assembly language

1. Disables interrupts
2. Returns PRIMASK value before interrupts were disabled
3. Caller's local var *cpu_sr* holds the returned value

CPU_SR_Restore(CPU_SR cpu_sr) internal helper function

- Coded in assembly language
- Restores PRIMASK from cpu_sr
- Returns nothing

Porting uCOS

Critical section usage

```
OS_CPU_SR  cpu_sr = 0;  // declare local var cpu_sr
OS_ENTER_CRITICAL();
    // interrupts are now disabled, PRIMASK is saved in cpu_sr
    // access shared resource
OS_EXIT_CRITICAL();
// PRIMASK is now restored from cpu_sr
```

Porting uCOS

OSTaskStkInit() – C code to initialize the stack for a new task

- uCOS will call this function when it creates new tasks
- Next slide shows the code

```
OS_STK *OSTaskStkInit (
  void (*task)(void *p_arg),
  void *p_arg,
  OS_STK *ptos,
  INT16U opt)
{
    OS_STK  *p_stk;

    // Stack is full-descending.
    // Align stack-top on an 8-byte
boundary:
    p_stk      = ptos + 1u;

    p_stk      = (OS_STK *)((OS_STK)(p_stk)
                 & 0xFFFFFFF8u);


    *(--p_stk) = (OS_STK)0x01000000uL;
//xPSR
    *(--p_stk) = (OS_STK)task;         // PC

    *(--p_stk) = (OS_STK)OS_TaskReturn;// LR

    *(--p_stk) = (OS_STK)0x12121212uL; //
R12
    *(--p_stk) = (OS_STK)0x03030303uL; // R3
```

```
    *(--p_stk) = (OS_STK)0x11111111uL; //R11

    *(--p_stk) = (OS_STK)0x10101010uL; //R10

    *(--p_stk) = (OS_STK)0x09090909uL; //R9

    *(--p_stk) = (OS_STK)0x08080808uL; //R8

    *(--p_stk) = (OS_STK)0x07070707uL; //R7

    *(--p_stk) = (OS_STK)0x06060606uL; //R6

    *(--p_stk) = (OS_STK)0x05050505uL; //R5

    *(--p_stk) = (OS_STK)0x04040404uL; //R4
```

Porting uCOS

Context switching in uCOS

Implemented by 4 assembly language functions in the port to Cortex-M4

- OSCtxSw()
  - Always called from task level uCOS code (not ISR level code)
  - Does nothing but set PendSV interrupt to trigger a context switch
- OSIntCtxSw()
  - Gets called by uCOS after servicing an interrupt but only if no more interrupts are pending
  - Does nothing but set PendSV interrupt to trigger a context switch
- ContextSwitch()
  - The actual PendSV handler
  - Ensures that current task context is saved
  - Restores context of highest priority ready task and resumes execution
- OSStartHighReady()
  - See next slide

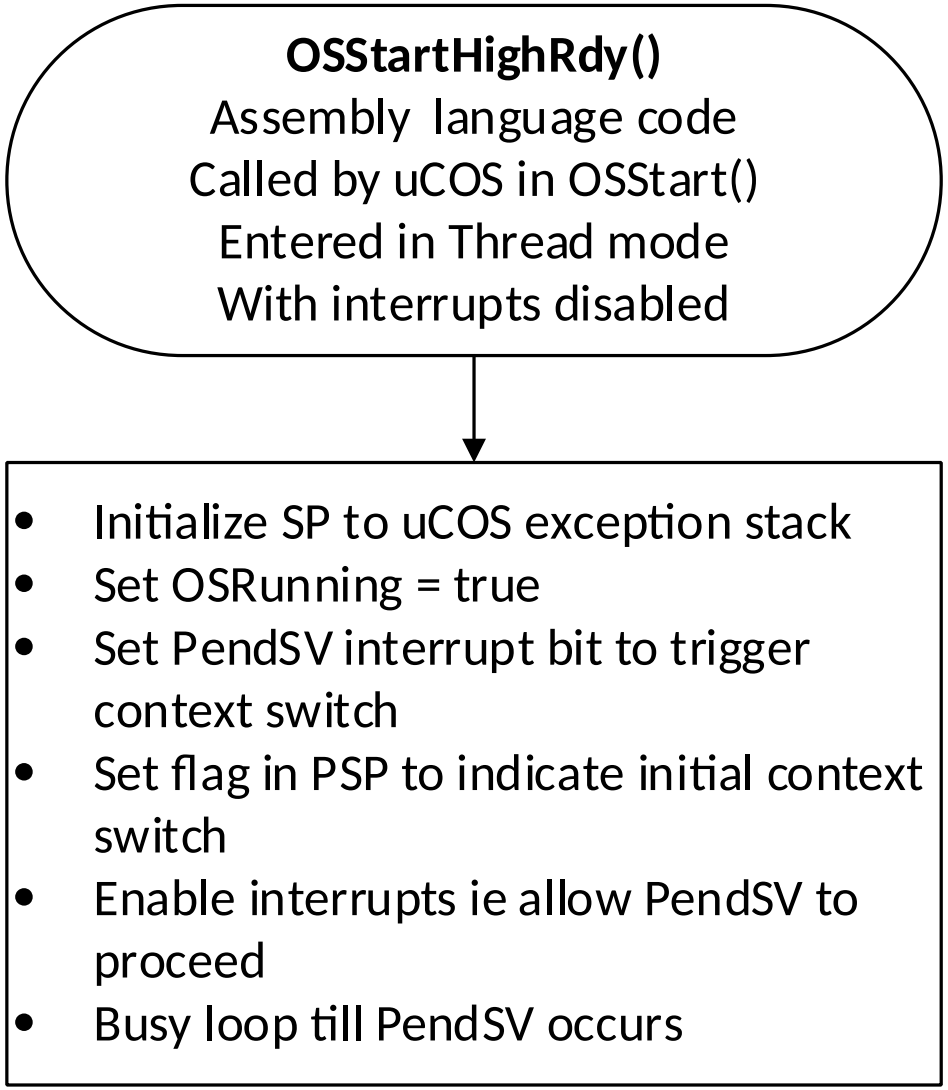
Porting uCOS

Context switching in uCOS

- OSStartHighReady()
  - Has the job of kicking off multitasking
  - Cortex-M4 architecture and context switching considerations
    - Context switches must be done exclusively by the PendSV handler at lowest priority
    - The only way to trigger PendSV is to set the PendSV interrupt bit via software
    - Reminder: context switch consists of saving context of **current task** and restoring context of next task
    - When kicking off multitasking **there is no current task**
    - For the second homework we launched a first task manually however uCOS has no provision for manually launching the first task.
    - The approach used is for the context switch code to use a flag to determine if this is the first context switch. If yes, then don’t save the current context before restoring the next context otherwise save the current context before restoring the next context.

Porting uCOS

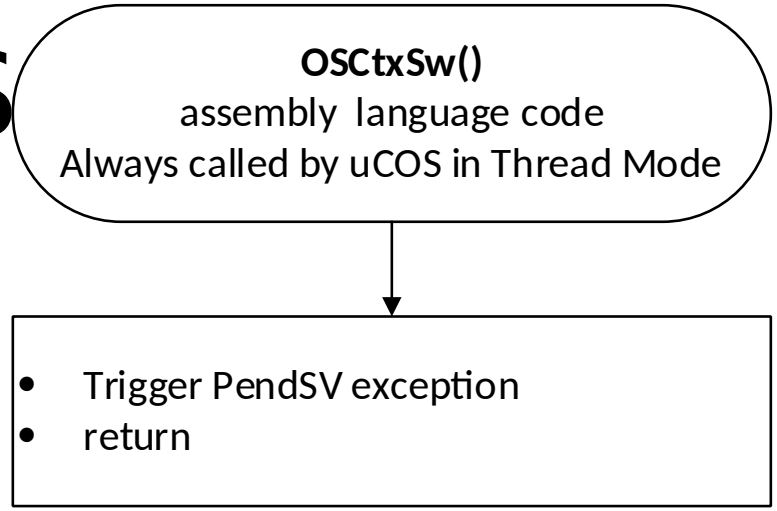
- **OSStartHighRdy**() is called to start the first task running in uCOS
- It turns on the uCOS flag **OSRunning** which indicates internally within uCOS that multitasking has begun.
- Since we only use SP_Main throughout uCOS, we can repurpose SP_Process as a flag to indicate whether a subsequent context switch is the initial one or not

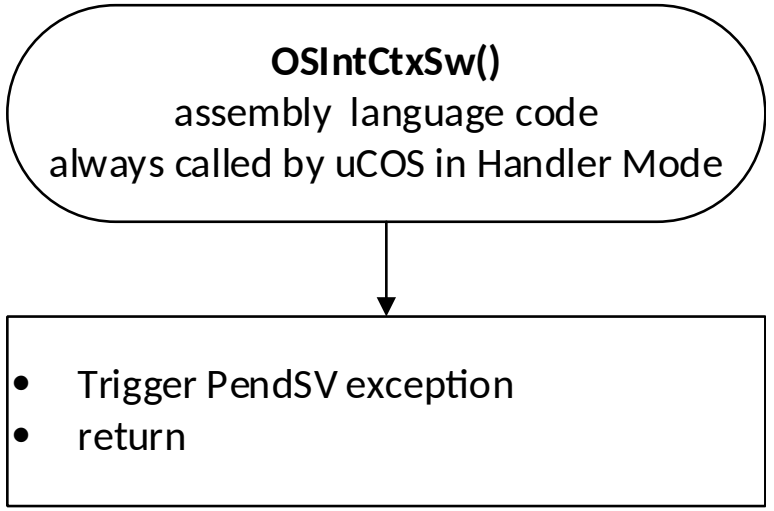


Porting uCOS

OSCtxSw() and **OSINTCtxSw()**

- These are called by uCOS when it determines that a context switch is necessary.
- They are identical
- The only difference is that uCOS calls OSCtxSw() in **Thread** mode to switch from one task to another; it calls OSIntCtxSw() in **Handler** mode to switch from the interrupted task to the highest priority ready task (which may be the interrupted task)

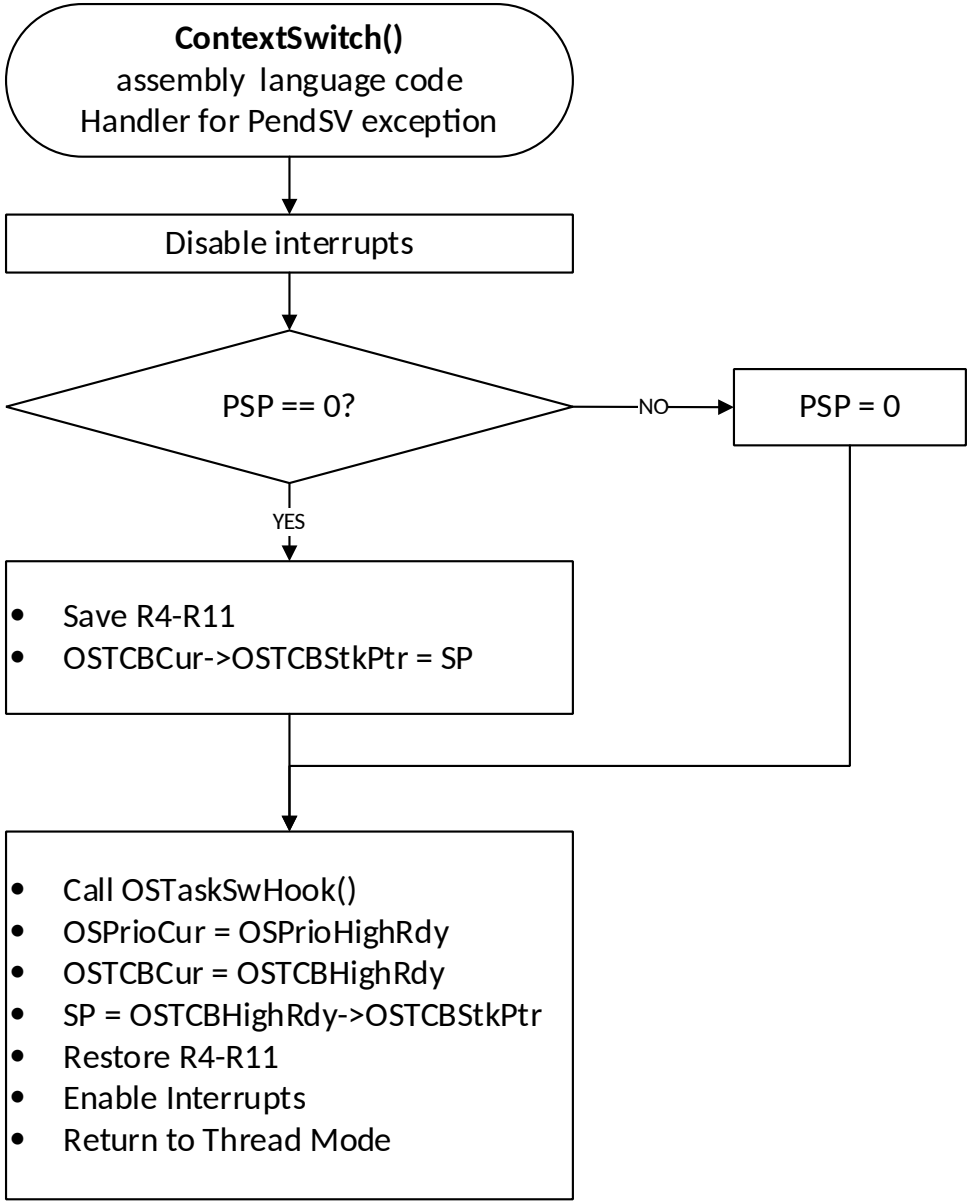




Porting uCOS

ContextSwitch()

- Written in assembly
- This is the handler for PendSV.
- PendSV is triggered by one of OSCtxSw() or OSINTCtxSw()
- Uses PSP as flag to indicate the first context switch
- The first context switch must not try to save context back to uCOS



Porting uCOS

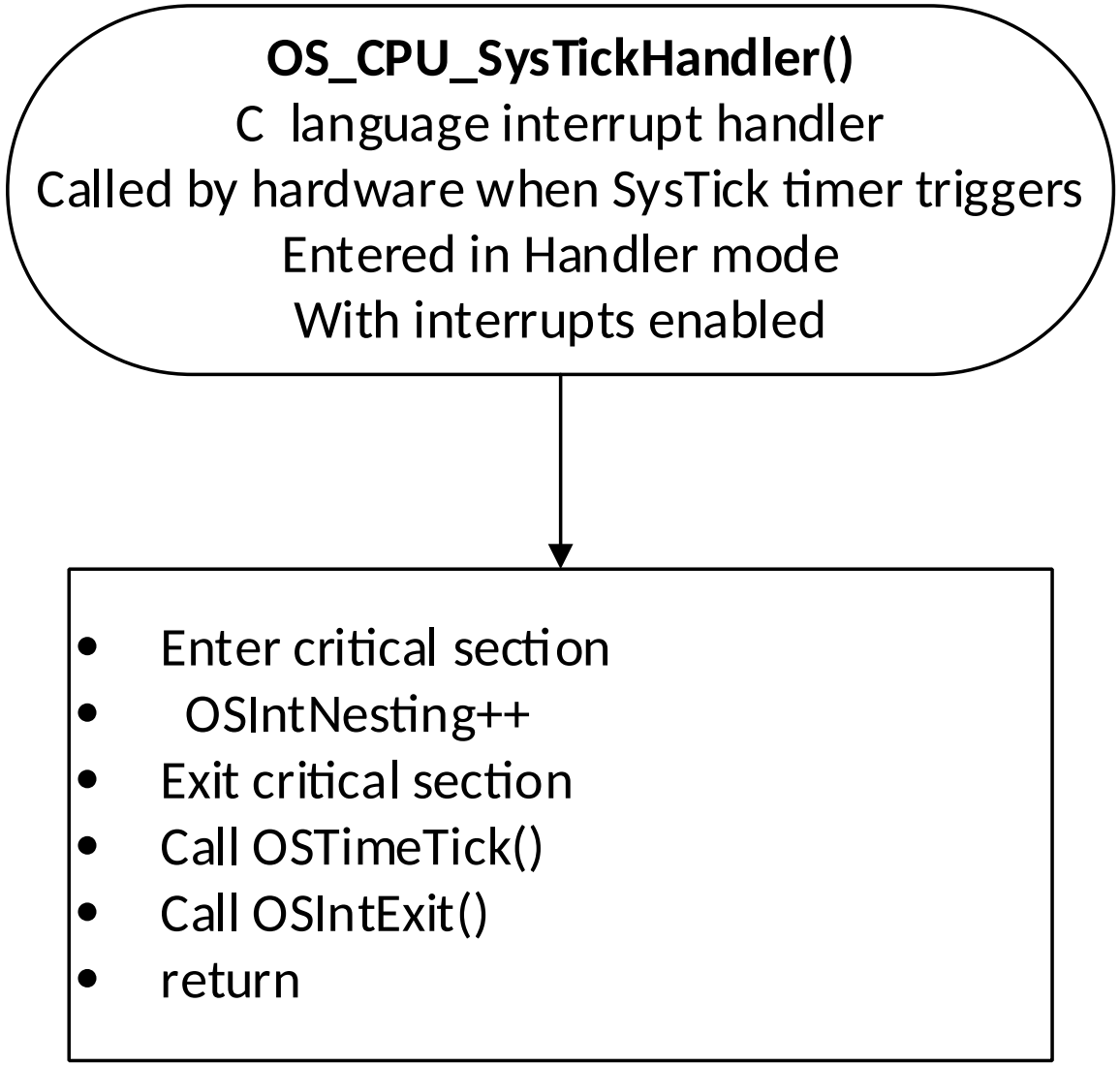
uCOS global variables used in ContextSwitch()

- **OSTCBCur**: points to the task control block (**TCB**) of the current task
  - The uCOS TCB is a struct containing many fields
  - The field we are interested in is **OSTCBStkPtr** which is where the task's stack pointer is saved during the context switch
  - Note that since **OSTCBStkPtr** is the first field in the TCB it is very easy to access in assembly language since it is located at the same address as the TCB itself (no offset needed).
- **OSPrioCur**: the priority of the current task
- **OSTCBHighRdy**: points to the TCB of the highest priority task whose status is ready

Porting uCOS

OS_CPU_SysTickHandler()

- Written in C code
- This is the handler for SysTick.
- On entry, all interrupt handlers must increment OSIntNesting to keep uCOS up to date on current nesting level of interrupts
- Before returning, all handlers must call OSIntExit() which decrements OSIntNesting and if necessary sets PendSV to switch context.



Porting uCOS

Summary of porting uCOS

- Data type definitions like INT32S, INT32U, INT8U, etc.
- Code to implement critical sections via disabling/enabling interrupts
- Code to initialize a task's stack
- Code to switch context
- Code to implement ISRs, especially the OS timer tick

Assignment 3 - Porting uCOS

- Your job is to read the explanatory comments and add missing code to file **os_cpu_a.asm** to achieve the following:
  - **Implement OS_CPU_SR_Save()**. This function gets called throughout uCOS when entering a critical section. It should capture the current interrupt enable/disable status given by register PRIMASK, then disable interrupts, then return the captured status as the function's return value.
  - **Implement OS_CPU_SR_Restore().** This function gets called throughout uCOS when exiting a critical section. It takes an argument consisting of the interrupt enable/disable status and copies it to the PRIMASK register, then returns.
  - **Implement ContextSwitch().** This code handles the PendSV exception. PendSV is only triggered by software in uCOS and only executes when no other interrupt is active. UCOS triggers PendSV by calling either OSCtxSw() or OSIntCtxSw() if it determines that a context switch is necessary.

Assignment 3 - Porting uCOS

Accessing special registers

- MRS - Move to Register from Special Register moves the value from the selected special-purpose register into a general-purpose register.
  - MRS<c> <Rd>,<spec_reg>
  - MRS R0,PSP ; R0 = PSP
- MSR - Move to Special Register from general purpose Register
  - MSR<c> <spec_reg>,<Rn>
  - MSR PSP, R0 ; PSP = R0

Assignment 3 - Porting uCOS

Conditional execution of an instruction

- Cortex-M4 allows for a condition to be appended to an instruction to optionally skip execution of the instruction (treats it like a NOP)
- ARM 32-bit instructions permit most any instruction to have an appended condition
- Most ARM Thumb instructions only allow conditional execution in an “IT” (IF-THEN) block
- (This appears to be a very ad hoc feature in Thumb)

Assignment 3 - Porting uCOS

IT block

IT{x{y{z}}} <firstcond>

- Where firstcond is a condition mnemonic eg EQ, NE, GT …
- x, y, z are chosen from {T,E} standing for then and else
- Allows up to 4 conditional instructions to follow which are either the same or opposite to firstcond.

Assignment 3 - Porting uCOS

IT block example

```
    SUBS   R0,R0,#5     ; Subtract 5 from R0 and set APSR flags
    ITTE   LE           ; Set up conditional execution
                        ; “TTE” means 3 cond. instructions to follow
                        ; T: 1st instruction must use LE condition
                        ; T: 2nd instruction must use LE condition
                        ; E: 3rd instruction must use GT condition
    ADDLE R1,R1,#1      ; if (R0 <= 0) R1++; else NOP;
    SUBLE R2,R2,#1      ; if (R0 <= 0) R2--; else NOP;
    ADDGT R3,R3,R2      ; if (R0 > 0) R3 = R3 + R2; else NOP;
```

Assignment 3 - Porting uCOS

IT block

Some Thumb instructions, notably branch instructions (B, BX) don’t have to be in an IT block to have an appended condition.

```
CMP R0,#5      ; Compare R0 to 5 and set APSR flags
BGE Label1     ; if (R0 >= 5) branch to Label1; else NOP;
```

uCOS Service Categories

- Task Management – OSTaskCreate(), etc.
- Time Management – OSTimeDly(), etc.
- Interrupt Management – OSIntEnter(), OSIntExit()
- Task Synchronization/Communication/Event Handling
  - Semaphores – OSSemPend(), etc.
  - Mutexes – OSMutexPend(), etc.
  - Message Mailboxes – OSMboxPend(), etc.
  - Message Queues – OSQPend(), etc.
  - Event Flags – OSFlagPend(), etc.

uCOS Service Categories

- Memory Management – OSMemGet(), etc.
- User-Defined Functions (hooks or call-backs) – OSTimeTickHook(), etc.
- Miscellaneous – OSStart(), etc.

uCOS Services – Task Management

- **INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)**
  - Tasks can be created before OSStart() is called – eg, the “Startup task”
  - Can be created by other tasks
  - Cannot be created by ISR code i.e. if interrupt nesting depth > 0
  - Priority - should not use 0…3, (OS_LOWEST_PRIORITY-3)…OS_LOWEST_PRIORITY. They are reserved for use by uCOS.
  - Returns OS_ERR_NONE or an error code
- **INT8U OSTaskChangePrio(INT8U oldprio, INT8U newprio)**
  - Changes the priority of a task (tasks are identified by their priority)
  - Remember priorities are unique in uCOS
  - See priority notes for OSTaskCreate()
  - Returns OS_ERR_NONE or an error code

uCOS Services – Task Management

- **INT8U OSTaskCreateExt(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio,**
  - The above 4 parameters are same as for OSTaskCreate() plus these additional parameters:

  INT16U id, // task ID number – future use – use same value as prio

  OS_STK *pbos, // pointer to bottom of stack

  INT32U stk_size, // number of stack entries in stack

  void *pext, // pointer to your custom data

  INT16U opt) // option bits – enable stack checking for this task, etc.

uCOS Services – Task Management

- **INT8U OSTaskDel(INT8U prio)**
  - Deletes the task identified by its priority
  - Specify OS_PRIO_SELF to have task delete itself
  - Task enters dormant state
  - Task can be recreated by calling OSTaskCreate() or OSTaskCreateExt()
  - If not deleting self, probably should not use unless you're sure the task is not holding any resources – see OSTaskDelReq()
  - Returns OS_ERR_NONE or an error code
- **INT8U OSTaskDelReq(INT8U prio)**
  - Requests that the specified task delete itself
  - Safer than OSTaskDel() since the task to be deleted can ensure that it has released all resources first.
  - Requires that task to be deleted checks periodically for incoming deletion request

uCOS Services – Task Management

- **INT8U OSTaskQuery(INT8U prio, OS_TCB *pdata)**
  - Gets a snapshot of the TCB of the task identified by its priority
  - pdata must point to an instance of OS_TCB which contains dozens of fields used to control the task
  - Can be useful for debugging
  - Returns OS_ERR_NONE or an error code
- **INT8U OSTaskResume(INT8U prio)**
  - Resumes the task identified by its priority
  - The only way to resume a task that was suspended by a call to OSTaskSuspend()
  - Returns OS_ERR_NONE or an error code

uCOS Services – Task Management

- **INT8U OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata)**
  - Determines the stack statistics for the task identified by its priority
  - Task must have been created with OSTaskCreateExt() (not OSTaskCreate())
  - OS_TASK_OPT_STK_CHK must be specified when the task is created
  - Must ensure that stack contents are initialized to 0
  - pdata must point to an instance of OS_STK_DATA which contains
    - OSFree – number of bytes free on the stack
    - OSUsed – number of bytes used on the stack
  - Returns OS_ERR_NONE or an error code
- **INT8U OSTaskSuspend(INT8U prio)**
  - Unconditionally suspends (blocks) the task identified by its priority
  - A suspended task can only be resumed by OSTaskResume()
  - Can suspend self but can’t resume self
  - Returns OS_ERR_NONE or an error code

uCOS Services – Time Management

- **void OSTimeDly(INT16U ticks)**
  - Delays the calling task by the specified number of timer ticks
  - Due to jitter if you need to ensure that the task delays *at least* n ticks, specify ticks=n+1
  - uCOS time resolution is governed by uCOS config parameter OS_TICKS_PER_SEC
- **void OSTimeDlyHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)**
  - Delays the calling task by the specified time span
  - uCOS time resolution is governed by uCOS config parameter OS_TICKS_PER_SEC

uCOS Services – Time Management

- **INT8U OSTimeDlyResume(INT8U prio)**
  - Resumes the task identified by its priority and which has called either OSTimeDly() or OSTimeDlyHMSM()
  - Returns OS_ERR_NONE if no error, OS_TIME_NOT_DLY if the specified task was not waiting for time to expire, or another error code
  - Caution: calling this on a task that is waiting for an event makes the task look like a timeout occurred.
- **INT32U OSTimeGet()**
  - Gets current value of uCOS system clock.
  - 32-bit counter that counts number of ticks since power on or clock was last set.

uCOS Services – Time Management

- **void OSTimeSet(INT32U ticks)**
  - Sets the uCOS system clock to the given value
- **void OSTimeTick()**
  - Calling this function allows uCOS to process a clock tick
  - Normally called by the timer ISR (true in our case)
  - Can be called at the task level by a high priority task if timer interrupt not available
  - The execution time of OSTimeTick() is directly proportional to the number of tasks created in your application

uCOS Services – Interrupt Management

- **void OSIntEnter()**
  - Notifies uCOS that an interrupt is being entered
  - Purpose is to allow uCOS to increment interrupt nesting level
  - Overhead of calling this can be avoided if instead each ISR increments the uCOS interrupt nesting variable OSIntNesting itself. (Our uCOS port does that.)
- **void OSIntExit()**
  - Called by ISRs to notify uCOS that an interrupt handler has finished processing
  - uCOS decrements the interrupt nesting variable and if it is 0, determines the highest priority ready task and transfers control of the CPU to it, otherwise resumes processing the next shallower level of nested interrupt.

uCOS Services – Task Sync - Semaphores

- **OS_EVENT *OSSemCreate(INT16U value)**
  - Allocates a new semaphore and initializes its counter to the given value
  - Returns a pointer to the allocated semaphore or NULL if none available
  - Semaphores must be created before they can be used
- **INT16U OSSemAccept(OS_EVENT *pevent)**
  - Does a non-blocking wait on the specified semaphore
  - Decrements the semaphore's counter (if > 0) and returns the value *before* it was decremented
  - If the value returned is > 0, the caller can access the resource the semaphore is guarding, otherwise the resource is not available.

uCOS Services – Task Sync - Semaphores

- **void OSSemPend(OS_EVENT *pevent, INT16U timeout, Int8U *err)**
  - Does a blocking wait on the specified semaphore. If the semaphore's count is greater than 0, decrements it and returns, otherwise places the task in the waiting list of the semaphore
  - timeout: if 0, waits forever, otherwise waits the specified number of ticks before timing out
  - err: OS_ERR_NONE if no error, otherwise an error code
- **INT8U OSSemPost(OS_EVENT *pevent)**
  - Signals the given semaphore. If tasks are waiting on the semaphore, removes the highest priority task from the semaphore's wait list and makes it ready, otherwise increments the semaphore's counter.
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync - Semaphores

- **INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata)**
  - Gets information on the current state of the specified semaphore
  - pdata must point to an instance of OS_SEM_DATA which will be populated with the value of the semaphore's counter and information you can use to determine how many tasks are waiting on the semaphore
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync - Semaphores

- **OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, Int8U *err)**
  - Deletes the specified semaphore
  - Caution: should only be used if all tasks that use the semaphore are also being deleted otherwise those tasks may continue to try to reference the deleted semaphore.
  - opt allows you to specify OS_DEL_NO_PEND which causes deletion to occur only if no tasks are pending (waiting) on the semaphore, or OS_DEL_ALWAYS to force deletion.
  - err is OS_ERR_NONE if no error, otherwise an error code
  - Returns NULL if successful otherwise pevent

uCOS Services – Task Sync – Semaphores Example

```
OS_EVENT *mySem;
int sharedRes;


void StartupTask(void *pdata)
{
    mySem = OSSemCreate(1);
    ...
}


Void TaskOne(void *pdata)
{
    ...
    INT8U err;
    OSSemPend(mySem, 0, &err);
      sharedRes += 1;
    OSSemPost(mySem);
    ...
}
```

```
Void TaskTwo(void *pdata)
{
    ...
    INT8U err;
    OSSemPend(mySem, 0, &err);
      sharedRes += 1;
    OSSemPost(mySem);
    ...
}
```

uCOS Services – Task Sync - Mutexes

- **OS_EVENT *OSMutexCreate(INT8U prio, INT8U *err)**
  - Allocates and initializes a mutex with counter value 1.
  - Like a semaphore but the maximum counter value is 1.
  - A mutex is used to gain exclusive access to a resource.
  - It provides **priority inheritance** to avoid **priority inversion**.
  - Concept of ownership: Only the task that currently owns a mutex can release it.
  - prio is the priority inheritance priority (PIP). If a higher-priority-task attempts to acquire a mutex held by a lower-priority-task, the priority of the lower-priority-task is temporarily raised to the PIP value until the resource is released. In uCOS, the PIP must be a unique value not used by any other task or mutex in the system. The mutex's PIP is initialized when the mutex is created and does not change thereafter.
  - err is OS_ERR_NONE if there was no error, otherwise an error code
  - Returns the pointer to the allocated mutex or NULL of none available
  - Mutexes must be created before they can be used

uCOS Services – Task Sync - Mutexes

- **INT8U OSMutexAccept(OS_EVENT *pevent, INT8U *err)**
  - Does a non-blocking wait on the specified mutex
  - Decrements the mutex's counter (if > 0) and returns the value *before* it was decremented. For a mutex the value is only either 0 or 1.
  - If the value returned is 1, the caller can access the resource the mutex is guarding, otherwise the resource is not available.
  - err is OS_ERR_NONE if there was no error otherwise an error code
  - Returns 1 if the mutex is available otherwise 0.

uCOS Services – Task Sync - Mutexes

- **void *OSMutexPend(OS_EVENT *pevent, INT16U timeout, Int8U *err)**
  - Does a blocking wait on the specified mutex.
  - If the mutex's count is 1, decrements it and returns, otherwise places the task in the waiting list of the mutex.
  - If the mutex is already owned by a lower priority task than the calling task, the **priority of the owner is raised to the PIP value specified when the mutex was created**.
  - timeout: if 0, waits forever, otherwise waits the specified number of ticks before timing out
  - err: OS_ERR_NONE if no error, otherwise an error code

uCOS Services – Task Sync - Mutexes

- **INT8U OSMutexPost(OS_EVENT *pevent)**
  - Signals the given mutex.
  - If tasks are waiting on the mutex, removes the highest priority task from the mutex's wait list and makes it ready, otherwise increments the mutex's counter back to 1.
  - If the priority of the owner task was raised due to a preceding Pend by a higher priority task, the priority is restored to its original value.
  - Only the owner of the mutex can successfully post.
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync - Mutexes

- **INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata)**
  - Gets information on the current state of the specified semaphore
  - pdata must point to an instance of OS_SEM_DATA which will be populated with the mutex's counter (0 or 1), owner priority, PIP, and information you can use to determine how many and which tasks are waiting on the semaphore
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync - Mutexes

- **OS_EVENT *OSMutexDel(OS_EVENT *pevent, INT8U opt, Int8U *err)**
  - Deletes the specified mutex
  - Caution: should only be used if all tasks that use the mutex are also being deleted otherwise those tasks may continue to try to reference the deleted mutex.
  - opt allows you to specify OS_DEL_NO_PEND which causes deletion to occur only if no tasks are pending (waiting) on the semaphore, or OS_DEL_ALWAYS to force deletion.
  - err is OS_ERR_NONE if no error, otherwise an error code
  - Returns NULL if successful otherwise pevent

uCOS Services – Task Sync – Mutexes Example

```
#define PIP 5
OS_EVENT *myMutex;
int sharedRes;


void StartupTask(void *pdata)
{
    INT8U err;
    myMutex = OSMutexCreate(PIP, &err);
    ...
}
```

```
Void TaskOne(void *pdata)
{
    ...
    INT8U err;
    OSMutexPend(myMutex, 0, &err);
        sharedRes += 1;
    OSMutexPost(myMutex);
    ...
}

Void TaskTwo(void *pdata)
{
    ...
    INT8U err;
    OSMutexPend(myMutex, 0, &err);
        sharedRes += 1;
    OSMutexPost(myMutex);
    ...
}
```

uCOS Services – Task Sync – Mailboxes

- **OS_EVENT *OSMboxCreate(void *msg)**
  - Allocates a new mailbox and initializes its contents to the given value (may be NULL)
  - A mailbox can hold up to one message (queues can hold more)
  - Returns a pointer to the allocated mailbox or NULL if none available
  - Mailboxes must be created before they can be used
- **void *OSMboxAccept(OS_EVENT *pevent)**
  - Does a non-blocking wait for a message on the specified mailbox
  - Returns a pointer to a message if there is one available otherwise NULL.

uCOS Services – Task Sync – Mailboxes

- **void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, Int8U *err)**
  - Does a blocking wait on the specified mailbox.
  - If the mailbox contains a message, removes the message from the mailbox and returns it, otherwise places the task in the wait list of the mailbox
  - timeout: if 0, waits forever, otherwise waits the specified number of ticks before timing out
  - err: OS_ERR_NONE if no error, otherwise an error code
  - Returns a pointer to the message if successful otherwise NULL

uCOS Services – Task Sync – Mailboxes

- **INT8U OSMboxPost(OS_EVENT *pevent, void *msg)**
  - Attempts to place the given message in the given mailbox.
  - If the mailbox is empty, deposits the message and returns
  - If the mailbox already contains a message, returns OS_MBOX_FULL
  - If tasks are waiting on the mailbox, the highest priority task gets the message.
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Mailboxes

- **INT8U OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt)**
  - Same behavior as for OSMboxPost() except for optional behavior specified by opt which allows for message broadcasting.
  - opt:
    - OS_POST_OPT_NONE: same behavior as OSMboxPost()
    - OS_POST_OPT_BROADCAST: posts the message to all tasks waiting on the mailbox

uCOS Services – Task Sync – Mailboxes

- **INT8U OSMBoxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata)**
  - Gets information on the current state of the specified mailbox
  - pdata must point to an instance of OS_MBOX_DATA which will be populated with the value of the mailbox message and information you can use to determine which tasks are waiting on the mailbox
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Mailboxes

- **OS_EVENT *OSMboxDel(OS_EVENT *pevent, INT8U opt, Int8U *err)**
  - Deletes the specified mailbox
  - Caution: should only be used if all tasks that use the mailbox are also being deleted otherwise those tasks may continue to try to reference the deleted mailbox.
  - opt allows you to specify OS_DEL_NO_PEND which causes deletion to occur only if no tasks are pending (waiting) on the mailbox, or OS_DEL_ALWAYS to force deletion.
  - err is OS_ERR_NONE if no error, otherwise an error code
  - Returns NULL if successful otherwise pevent

uCOS Services – Task Sync – Mailboxes Example

```
OS_EVENT *mbox;


void StartupTask(void *pdata)
{
    ...
    mbox= OSMboxCreate(NULL);
    ...
}
```

```
Void TaskOne(void *pdata)
{
    ...
    char *msgTx = “hello”;
    OSMboxPost(mbox, (void*)msgTx);
    ...
}

Void TaskTwo(void *pdata)
{
    ...
    INT8U err;
    char *msgRx;
    msgRx = (char*)OSMboxPend(mbox, 0, &err);
    ...
}
```

uCOS Services – Task Sync – Queues

- **OS_EVENT *OSQCreate(void **start, INT8U size)**
  - Creates a circular message queue holding a maximum of size messages
  - Each message consists of a pointer-wide data item (32 bits)
  - start is an array of void pointers you must allocate and is the storage for the queue
  - size is the number of entries in the start array and represents the maximum number of messages the queue can hold
  - Returns a pointer to the queue event control block if successful otherwise NULL
- **void *OSQAccept(OS_EVENT *pevent)**
  - Does a non-blocking wait for a message on the specified queue
  - Returns a pointer to a message if there is one available otherwise NULL.
  - uCOS queues are default FIFO – first in first out

uCOS Services – Task Sync – Queues

- **void *OSQPend(OS_EVENT *pevent, INT16U timeout, Int8U *err)**
  - Does a blocking wait on the specified queue.
  - If the queue contains a message, does a FIFO removal of a message from the queue and returns it, otherwise places the task in the wait list of the queue
  - timeout: if 0, waits forever, otherwise waits the specified number of ticks before timing out
  - err: OS_ERR_NONE if no error, otherwise an error code
  - Returns a pointer to the message if successful otherwise NULL

uCOS Services – Task Sync – Queues

- **INT8U OSQPost(OS_EVENT *pevent, void *msg)**
  - Attempts to insert the given message in the given queue.
  - If the queue is not full, deposits the message at the tail and returns
  - If the queue is full, returns OS_Q_FULL
  - If tasks are waiting on the queue, the highest priority task gets the message.
  - Returns OS_ERR_NONE if there was no error otherwise an error code
- **INT8U OSQPostFront(OS_EVENT *pevent, void *msg)**
  - Same behavior as for OSMboxPost() except that the message is inserted at the head of the queue instead of the tail.

uCOS Services – Task Sync – Queues

- **INT8U OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt)**
  - Same behavior as for OSQPost() except for optional behavior specified by opt which allows for message broadcasting and insertion at front.
  - opt:
    - OS_POST_OPT_NONE: same behavior as OSQPost()
    - OS_POST_OPT_BROADCAST: posts the message to all tasks waiting on the queue
    - OS_POST_OPT_FRONT: same behavior as OSQPostFront()
    - OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST: combined options result in broadcast to the front of the queue for all waiting tasks.

uCOS Services – Task Sync – Queues

- **INT8U *OSQFlush(OS_EVENT *pevent)**
  - Empties the contents of the specified queue
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Queues

- **INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata)**
  - Gets information on the current state of the specified message queue
  - pdata must point to an instance of OS_Q_DATA which will be populated with
    - the value of the next queue message,
    - number of messages in the queue,
    - size of the queue,
    - and information you can use to determine which tasks are waiting on the queue
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Queues

- **OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, Int8U *err)**
  - Deletes the specified message queue
  - Caution: should only be used if all tasks that use the queue are also being deleted otherwise those tasks may continue to try to reference the deleted queue.
  - opt allows you to specify OS_DEL_NO_PEND which causes deletion to occur only if no tasks are pending (waiting) on the queue, or OS_DEL_ALWAYS to force deletion.
  - err is OS_ERR_NONE if no error, otherwise an error code
  - Returns NULL if successful otherwise pevent

uCOS Services – Task Sync – Queues Example

```
#define QMAXENTRIES 2
void * qMsgVPtrs[QMAXENTRIES];
OS_EVENT *qMsg;



void StartupTask(void *pdata)
{
    INT8U err;
    qMsg = OSQCreate(qMsgVPtrs, QMAXENTRIES);
    ...
}
```

```
Void TaskOne(void *pdata)
{
    ...
    INT8U err;
    char *msgTx1 = “hello”;
    char *msgTx2 = “world”;
    err = OSQPost(qMsg, (void*)msgTx1);
    err = OSQPost(qMsg, (void*)msgTx2);
    ...
}

Void TaskTwo(void *pdata)
{
    ...
    INT8U err;
    char *msgRx1, *msgRx2;
    msgRx1 = (char*)OSQPend(qMsg, 0, &err);
    msgRx2 = (char*)OSQPend(qMsg, 0, &err);
    ...
}
```

uCOS Services – Task Sync – Event Flags

- **OS_FLAG_GRP *OSFlagCreate(OS_FLAGS flags, INT8U *err)**
  - Creates and initializes an event flag group
  - flags contains the initial value of the flags in the group
  - err is OS_ERR_NONE if there was no error otherwise an error code
  - Returns a pointer to the flag group if successful otherwise NULL

uCOS Services – Task Sync – Event Flags

- **OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err)**
  - Does a non-blocking wait on specified bits in a group of flag bits
  - pgrp is the group of bits on which to do the non-blocking wait.
  - flags specifies which bits in pgrp you want to check.
  - wait_type specifies whether the bits you want to check should be 0 or 1
    - OS_FLAG_WAIT_CLR_ALL – waits for all the bits you want to check to become 0
    - OS_FLAG_WAIT_CLR_ANY – waits for any of the bits you want to check to become 0
    - OS_FLAG_WAIT_SET_ALL – waits for all the bits you want to check to become 1
    - OS_FLAG_WAIT_SET_ANY – waits for any of the bits you want to check to become 1
    - OS_FLAG_CONSUME OR this to reset bits after successful wait
  - err is OS_ERR_NONE if the desired wait bits were satisfied, otherwise an error code
  - Returns the current state of the flag group without blocking.

uCOS Services – Task Sync – Event Flags

- **OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err)**
  - Does a blocking wait on specified bits in a group of flag bits
  - pgrp is the group of bits on which to do the non-blocking wait.
  - flags specifies which bits in pgrp you want to check.
  - wait_type specifies whether the bits you want to check should be 0 or 1
    - OS_FLAG_WAIT_CLR_ALL – waits for all the bits you want to check to become 0
    - OS_FLAG_WAIT_CLR_ANY – waits for any of the bits you want to check to become 0
    - OS_FLAG_WAIT_SET_ALL – waits for all the bits you want to check to become 1
    - OS_FLAG_WAIT_SET_ANY – waits for any of the bits you want to check to become 1
    - OS_FLAG_CONSUME OR this to reset bits after successful wait
  - timeout is the number of timer ticks to wait before timing out.
  - Returns the state of the flag group or 0 if timed out.

uCOS Services – Task Sync – Event Flags

- **OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)**
  - Sets or clears specified bits in the given flag group
  - pgrp is the group of bits to operate on
  - flags specifies the bits to set or clear. You specify bits by setting them to 1
  - opt
    - OS_FLAG_SET – bits specified in flags are set to 1 in pgrp
    - OS_FLAG_CLR – bits specified in flags are set to 0 in pgrp
  - err is OS_ERR_NONE if no error occurred otherwise an error code
  - Returns the new value of the flag group

uCOS Services – Task Sync – Event Flags

- **OS_FLAGS OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err)**
  - Gets the current value of the event flags in the group.
  - Does not return the wait list at this time
  - err is OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Event Flags

- **OS_FLAG_GRP *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, Int8U *err)**
  - Deletes the flag group
  - Caution: should only be used if all tasks that use the flag group are also being deleted otherwise those tasks may continue to try to reference the deleted flag group.
  - opt allows you to specify OS_DEL_NO_PEND which causes deletion to occur only if no tasks are pending (waiting) on the flag group, or OS_DEL_ALWAYS to force deletion.
  - err is OS_ERR_NONE if no error, otherwise an error code
  - Returns NULL if successful otherwise pevent

uCOS Services – Task Sync – Event Flags Example

```
OS_FLAG_GRP *myFlags;

void StartupTask(void *pdata)
{
    INT8U err;
    myFlags = OSFlagCreate(0x3, &err);
    ...
}

Void TaskOne(void *pdata)
{
    INT8U err;
    // clear bit 1:
    OSFlagPost(myFlags, 0x2,
        OS_FLAG_CLR, &err);
    // wait for bit 0 to clear:
    OSFlagPend(myFlags, 0x1,
        OS_FLAG_WAIT_CLR_ALL, 0, &err);
    ...
}
```

```
Void TaskTwo(void *pdata)
{
    INT8U err;
    // clear bit 0:
    OSFlagPost(myFlags, 0x1,
        OS_FLAG_CLR, &err);
    // wait for bit 1 to clear:
    OSFlagPend(myFlags, 0x2,
        OS_FLAG_WAIT_CLR_ALL, 0, &err);
    ...
}
```

uCOS Services – Memory Management

- **OS_MEM *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err)**
  - Creates and initializes a uCOS memory partition.
  - Uses the services of a uCOS memory partition control block.
  - addr is the address of an array of nblks entries where each entry is of size blksize. You must allocate this array before passing it to uCOS.
  - uCOS will then manage the partition so you can later dynamically allocate individual entries from it
  - err is OS_ERR_NONE if there was no error otherwise an error code
  - Returns the allocated memory partition or NULL if no memory partition control block is available.

uCOS Services – Memory Management

- **void *OSMemGet(OS_MEM *pmem, INT8U *err)**
  - Allocates a memory block from the specified memory partition
  - pmem is a pointer to a memory partition previously created by OSMemCreate()
  - err is OS_ERR_NONE if there was no error otherwise an error code
  - Returns a pointer to the allocated memory block

uCOS Services – Memory Management

- **INT8U OSMemPut(OS_MEM *pmem, void *pblk)**
  - Returns the specified memory block to the specified memory partition
  - pmem is a pointer to the memory partition from which the memory block was previously allocated by OSMemGet()
  - pblk is a pointer to the memory block to return
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Memory Management

- **INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata)**
  - Returns information about the specified memory partition
  - pmem is a pointer to the memory partition to query
  - pdata must point to an instance of OS_MEM_DATA which will be populated with details including number of free blocks, number of in-use blocks, list of free blocks.
  - Returns OS_ERR_NONE if there was no error otherwise an error code

uCOS Services – Task Sync – Memory Management Example

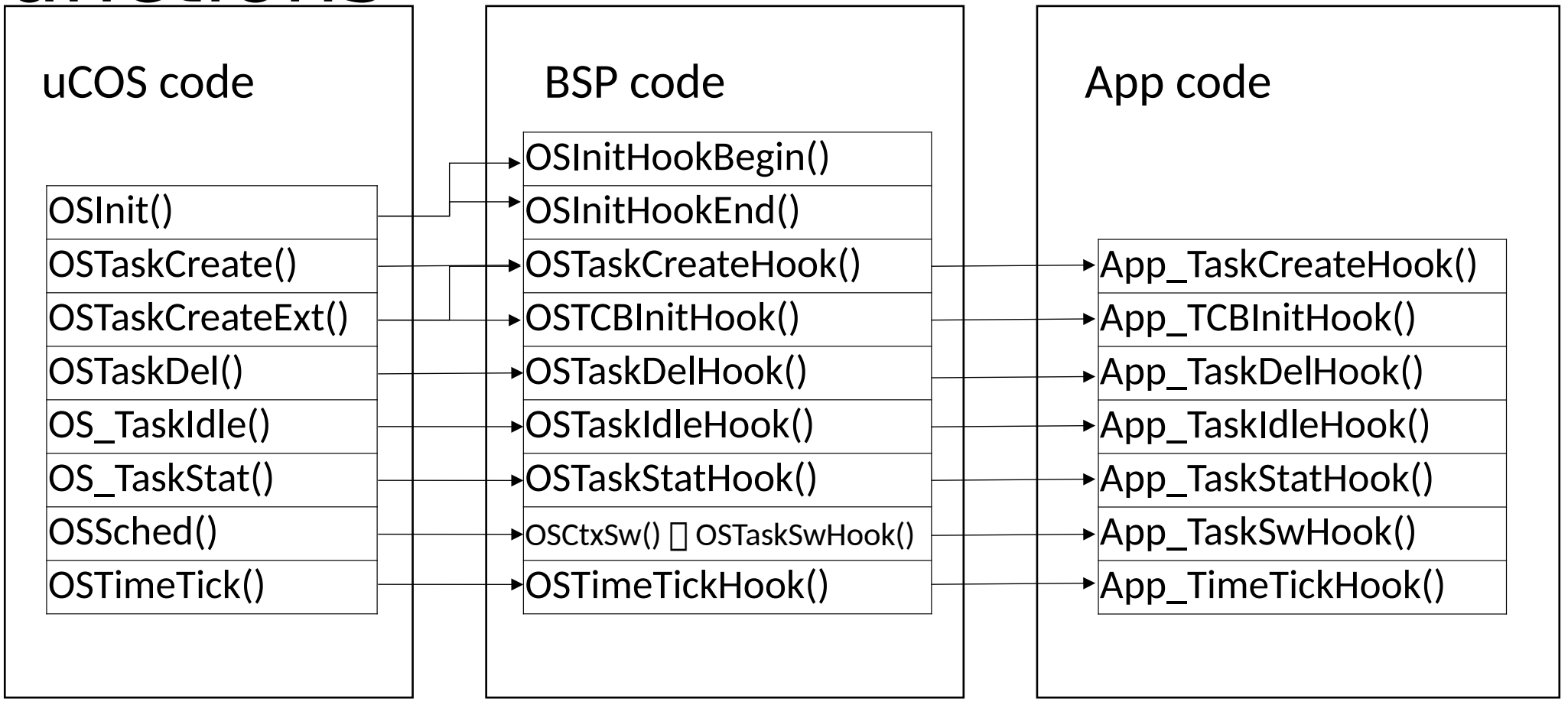
```
#define MAXBLOCKS 5

typedef struct
{
    char *msg;
    INT32U count;
}MyData_t;

MyData_t myDataBlocks[MAXBLOCKS];

OSMem *memPartition;
```

```
void StartupTask(void *pdata)
{
    INT8U err;
    memPartition = OSMemCreate(
        myDataBlocks,
        MAXBLOCKS,
        Sizeof(MyData_t),
        &err);
    ...
}

Void TaskOne(void *pdata)
{
    ...
    INT8U err;
    MyData_t *pdata;
    pdata = (MyData_t*) OSMemGet(
        memPartition, &err);
    ...
    OSMemPut(memPartition, pdata);
    ...
}
```

uCOS Services – User-Defined Functions

- These uCOS call-backs allow you to insert your own code in key locations of uCOS to handle hardware-, application-, or debugging-specific issues.
- There are 2 levels of callbacks
  - BSP level – called by hardware independent kernel code
    - Found in os_cpu_c.c
    - Example: if your hardware supports floating point operations you might need to insert code to allocate memory and save floating point context during context switches
  - Application level – called by BSP kernel code
    - Found in app_hooks.c
    - Examples: you might want to insert some code to be executed by the Idle task, or the Stats task

uCOS Services – User-Defined Functions



uCOS Services – Miscellaneous

- **void OSInit()**
  - Initializes uCOS
  - Must be called prior to calling OSStart() which starts multitasking
- **void OSSchedLock()**
  - Disables the scheduler until OSSchedUnLock() is called
  - The calling task keeps control of the CPU until OSSchedUnlock() is called
  - If there are nested calls to OSSchedLock() there must be an equal number of OSSchedUnlock() calls before scheduling is resumed
  - ISRs will still execute, assuming interrupts are enabled
  - Caution: if the task that called OSSchedLock() gets blocked e.g. on a semaphore, the system will lock up because the scheduler cannot transfer control to any other task

uCOS Services – Miscellaneous

- **void OSSchedUnlock()**
  - Reenables task scheduling whenever it is paired with OSSchedLock()
  - Caution: see OSSchedLock()
- **void OSStart()**
  - Starts multitasking under uCOS.
  - Should only be called once
  - Calling it more than once has no effect after the first call.

uCOS Services – Miscellaneous

- void OSStatInit()
  - Initializes the system for the Stat task which monitors CPU usage of the system
  - **Must** be called in the “startup task” before any other tasks are created
  - It determines the maximum value a 32-bit counter, OSIdleCtr can reach in 0.1 sec. when just one task (the startup task) is running
  - Later, the Stat task will use it to compute CPU usage of the system:

$$\text{CPU Usage (\%)} = 100 * \left(1 - \frac{\text{OSIdleCtr}}{\text{OSIdleCtrMax}}\right)$$

uCOS Services – Miscellaneous

- **UBT16U OSVersion()**
  - Gets the current version of uCOS
  - Returns the version as x.yy multiplied by 100
  - Example: our version v2.91 is returned as 291

uCOS Configuration – os_cfg.h

- uCOS provides the capability to fine tune how many of each component such as TCBs or Semaphores your application will use
- These are #define configuration parameters that get used when the OS is built – i.e. they are not run time parameters
- These parameters allow control over how much RAM the compiled OS occupies
- Example: if your RAM is very small and your application does not use any Mailboxes, don’t waste precious RAM by including code in the OS to handle Mailboxes
- Removal of some features reduces the amount of CPU used by the OS – e.g. the system Statistics task.

uCOS Configuration – os_cfg.h

#define Constant	Example Value
OS_APP_HOOKS_EN 1 enables user application hooks which are called from uCOS. Hook stubs are found in app\hooks.c in our port (App_TaskSwHook(), App_TimeTickHook(), etc.). If 0, conserves CPU time, RAM.	1
OS_ARG_CHK_EN If 1, turns on argument checking. If 0, conserves RAM, CPU time.	0
OS_CPU_HOOKS_EN If 1, enables user hooks (OSTaskSwHook(), OSTimeTickHook(), etc.). If 0, conserves CPU time, RAM.	1
OS_LOWEST_PRIO Range is 3..63. uCOS reserves 2 priorities for itself. uCOS takes lowest for Idle task, and second lowest if Stats task is enabled. Remember: smaller number is higher priority. Must be > number of tasks. Reducing it conserves RAM.	63

uCOS Configuration – os_cfg.h

#define Constant	Example Value
OS_MAX_EVENTS Max number of event control blocks. 1 ECB needed per task sync item like semaphore, mailbox.	10
OS_MAX_FLAGS Max number of event flags.	5
OS_MAX_MEM_PART Max number of memory partitions (simple pools of memory blocks for dynamic allocation).	5
OS_MAX_QS Max number of message queues.	4
OS_MAXTASKS Max number of *application* tasks. Range is 2..62. Not 63 because uCOS has 2 system tasks. Reducing it conserves RAM.	16

uCOS Configuration – os_cfg.h

#define Constant	Example Value
OS_TASK_IDLE_STK_SIZE Number of stack entries, not bytes. Needs to be big enough to hold context for deepest interrupt nesting allowed in application.	128
OS_TASK_STAT_EN If 1, enables the statistics task which computes the CPU % of the application every second.	1
OS_TASK_STAT_STK_SIZE Same comment as for **OS_TASK_IDLE_STK_SIZE**	128
OS_SCHED_LOCK_EN If 1, enables functions OSSchedLock() and OSSchedUnlock().	1
OS_TICKS_PER_SEC Informs uCOS the rate at which OSTimeTick() will be called. Used by uCOS stats task and OSTimeDlyHMSM().	1000

uCOS Configuration – os_cfg.h

Fine tuning category	#define Constants	Our Value
Event Flags	OS_FLAG_EN OS_FLAG_WAIT_CLR_EN OS_FLAG_ACCEPT_EN OS_FLAG_DEL_EN OS_FLAG_QUERY_EN	All 1
Message Mailboxes	OS_MBOX_EN OS_MBOX_ACCEPT_EN OS_MBOX_DEL_EN OS_MBOX_POST_EN OS_MBOX_POST_OPT_EN OS_MBOX_QUERY_EN	All 1
Mutexes	OS_MUTEX_EN OS_MUTEX_ACCEPT_EN OS_MUTEX_DEL_EN OS_MUTEX_QUERY_EN	All 1

uCOS Configuration – os_cfg.h

Fine tuning category	#define Constants	Our Value
Message Queues	OS_Q_EN OS_Q_ACCEPT_EN OS_Q_DEL_EN OS_Q_FLUSH_EN OS_Q_PEND_ABORT_EN OS_Q_POST_EN OS_Q_POST_FRONT_EN OS_Q_POST_OPT_EN OS_Q_QUERY_EN	All 1
Semaphores	OS_SEM_EN OS_SEM_ACCEPT_EN OS_SEM_DEL_EN OS_SEM_PEND_ABORT_EN OS_SEM_QUERY_EN OS_SEM_SET_EN	All 1

uCOS Configuration – os_cfg.h

Fine tuning category	#define Constants	Our Value
Task Management	OS_TASK_CHANGE_PRIO_EN OS_TASK_CREATE_EN OS_TASK_CREATE_EXT_EN OS_TASK_DEL_EN OS_TASK_QUERY_EN OS_TASK_SUSPEND_EN	All 1
Time Management	OS_TIME_DLY_HMSM_EN OS_TIME_DLY_RESUME_EN OS_TIME_GET_SET_EN	All 1