ECE3849

D-Term 2021

Real Time Embedded Systems

Module 3 Part 1

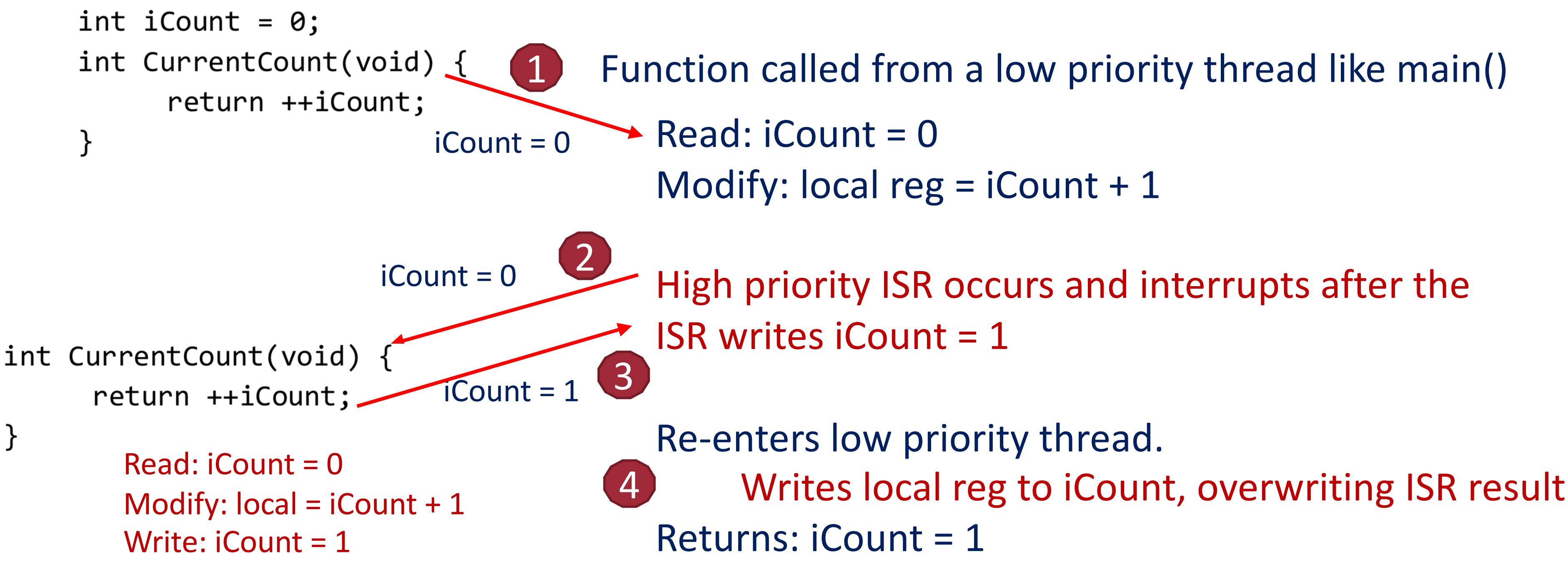
Module 3 Part 1 Overview

- Reentrancy
- Interrupt Controller Functionality
- Introduction to RTOS

Reentrancy

- Functions that can be called from multiple threads are considered a global shared resource.
- A function is reentrant if it can be safely called by more than one thread in a preemptive environment.
  - It can be interrupted and called again (reentered) by a higher priority ISR.
  - Once control returns to the lower priority thread, the interrupted function completes correctly.
- Functions may NOT be reentrant if…
  - It accesses static or global variables non-atomically.
  - It accesses global variables through pointers.
  - It accesses I/O non-atomically.
  - It calls other functions that are not reentrant.

Non-reentrant: Read-Modify-Write Example



```
int iCount = 0;
int CurrentCount(void) {
      return ++iCount;
}
```

iCount = 0

1 Function called from a low priority thread like main()

Read: iCount = 0

Modify: local reg = iCount + 1

iCount = 0

2 High priority ISR occurs and interrupts after the

```
int CurrentCount(void) {
     return ++iCount;
}
```

Read: iCount = 0

Modify: local = iCount + 1

Write: iCount = 1

ISR writes iCount = 1

3 iCount = 1

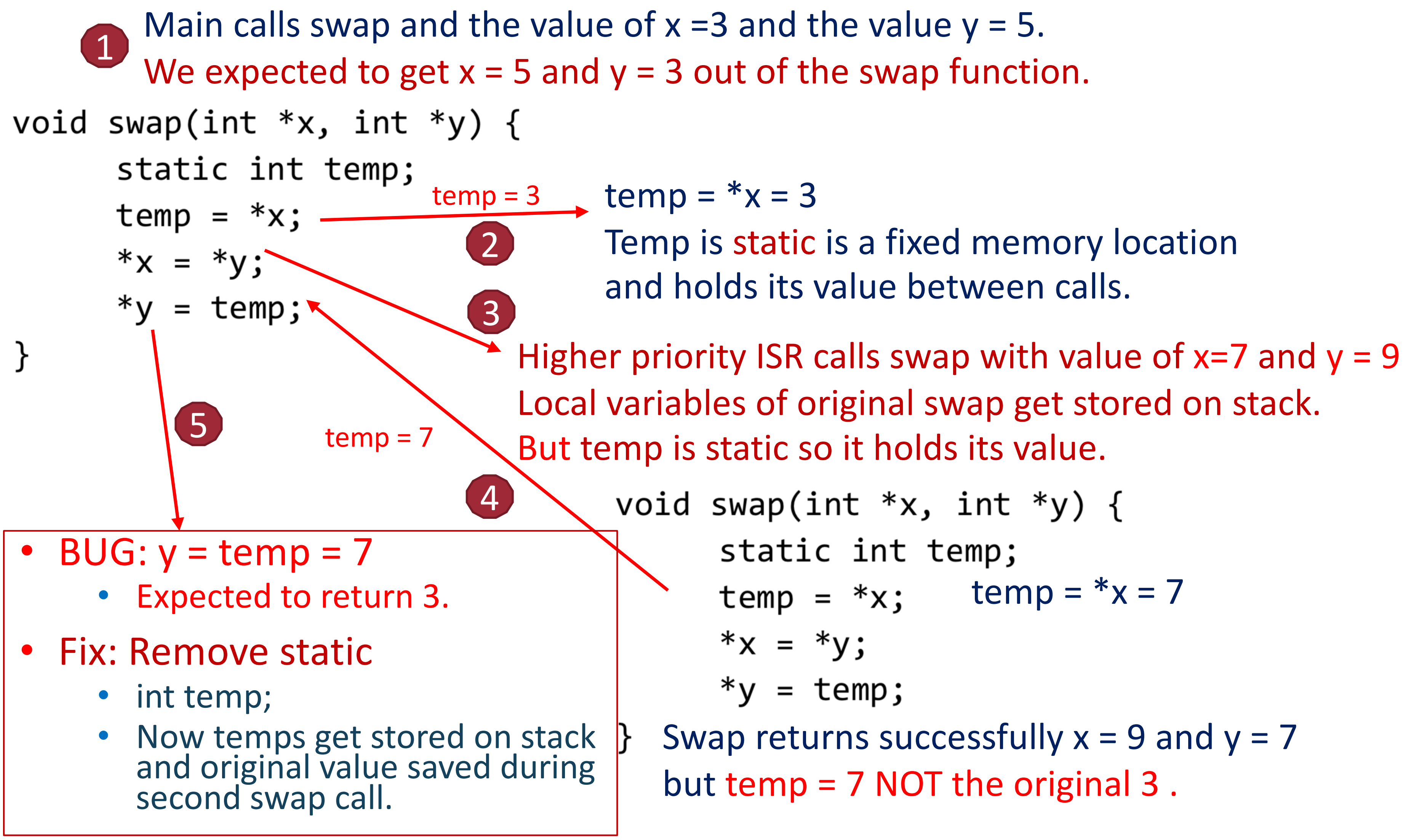
Re-enters low priority thread.

4 Writes local reg to iCount, overwriting ISR result

Returns: iCount = 1

- BUG: CurrentCount was called 2x, iCount should equal 2
- Fix: Disable Interrupts while doing the ++iCount operation.

Non-reentrant: Static Variables



1 Main calls swap and the value of x =3 and the value y = 5.
We expected to get x = 5 and y = 3 out of the swap function.

```
void swap(int *x, int *y) {
     static int temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
```

2 temp = *x = 3
Temp is static is a fixed memory location and holds its value between calls.

3 Higher priority ISR calls swap with value of x=7 and y = 9
Local variables of original swap get stored on stack.
But temp is static so it holds its value.

```
void swap(int *x, int *y) {
     static int temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
```

temp = *x = 7

Swap returns successfully x = 9 and y = 7
but temp = 7 NOT the original 3 .

- BUG: y = temp = 7
  - Expected to return 3.
- Fix: Remove static
  - int temp;
  - Now temps get stored on stack and original value saved during second swap call.

Non-reentrant: Using Pointers

```
void swap(int *x, int *y) {
      static int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}
```

- x and y are pointers
  - What are they pointing too?
  - Does it matter?

Non-reentrant: Calling non-reentrant functions

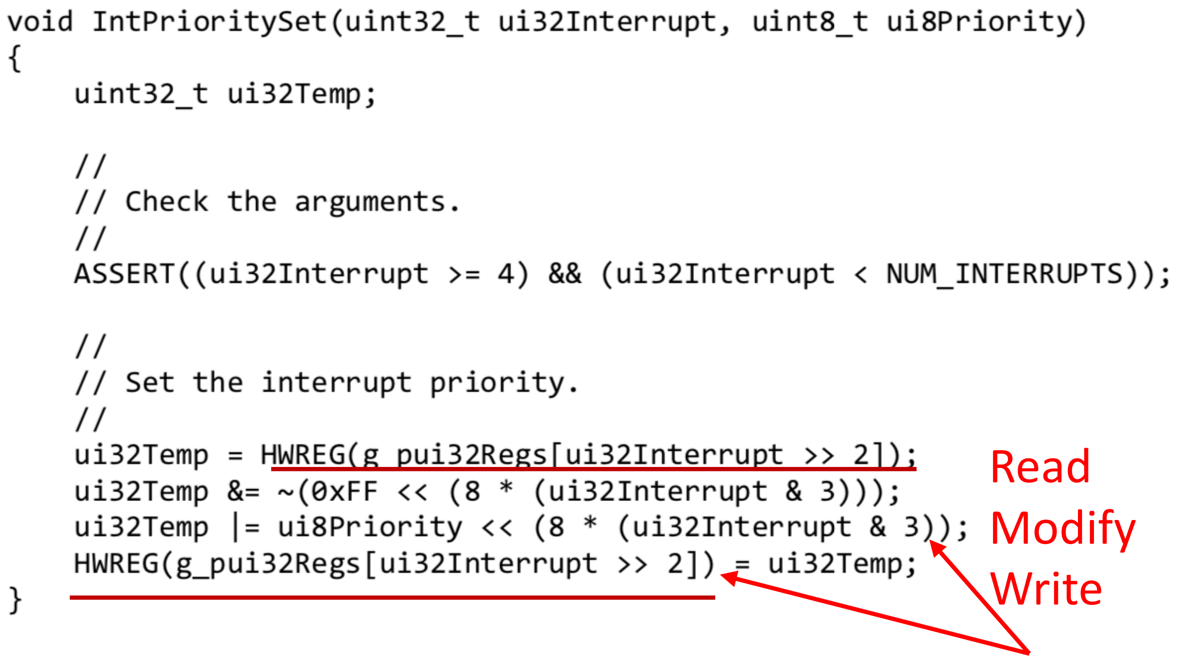
- TimerIntEnable: TivaWare function -> optimized for low latency.
  - HWREG accesses Hardware register directly.
    - Utilizes a read-modify-write -> non-reentrant.
    - Hardware registers are also a shared resource -> non-reentrant.

```
void TimerIntEnable(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_TimerBaseValid(ui32Base));

    //
    // Enable the specified interrupts.
    //
    HWREG(ui32Base + TIMER_O_IMR) |= ui32IntFlags;
}
```

x |= y => x = x | y ➔ Read-Modify-Write

Non-reentrant: non-atomic IO resource

- IntPrioritySet(): TivaWare function
  - Utilizes a read-modify-write -> non-reentrant.
  - Hardware registers are a shared resource -> non-reentrant.



```
void IntPrioritySet(uint32_t ui32Interrupt, uint8_t ui8Priority)
{
    uint32_t ui32Temp;

    //
    // Check the arguments.
    //
    ASSERT((ui32Interrupt >= 4) && (ui32Interrupt < NUM_INTERRUPTS));

    //
    // Set the interrupt priority.
    //
    ui32Temp = HWREG(g_pui32Regs[ui32Interrupt >> 2]);
    ui32Temp &= ~(0xFF << (8 * (ui32Interrupt & 3)));
    ui32Temp |= ui8Priority << (8 * (ui32Interrupt & 3));
    HWREG(g_pui32Regs[ui32Interrupt >> 2]) = ui32Temp;
}
```

Read

Modify

Write

If IntPrioritySet is called from ISR here.
The register value will be over written on return.

Reentrancy: Libraries

- Never assume any library function is reentrant.
  - Check datasheet to verify if reentrant or thread safe.
  - TivaWare libraries are written to minimize latency and are NOT reentrant.
- Disable interrupts if…
  - Using non-reentrant function.
  - Using reentrant libraries that use global data or shared peripherals.
- Avoid calling same function from multiple threads.
  - Reentrancy is not required and latency is not impacted.

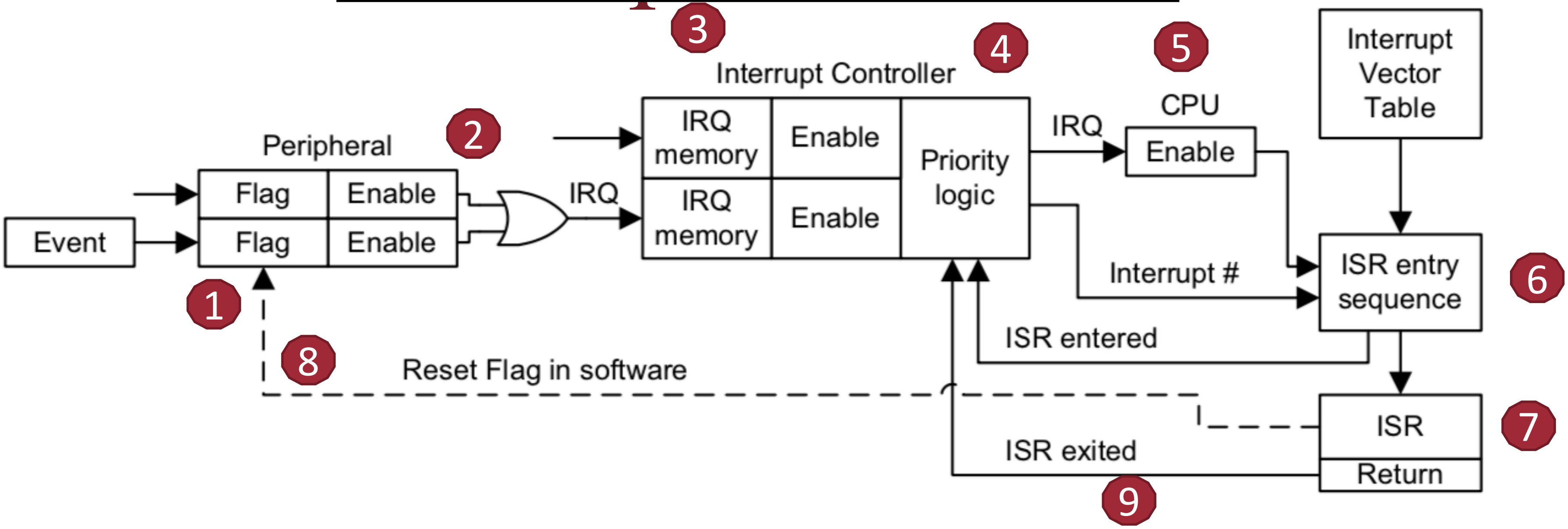
Reentrancy: Best Practices

- Use local storage as much as possible.
  - Includes register, stack, dynamically allocated memory.
  - Local variables and function arguments are OK.
    - Arguments passed as pointers should be avoided as they may point to shared data.
  - Global or static constants are OK, because they don't change.
- Properly handle access to global or static variables, they are considered shared data.
  - Access them atomically.
  - Disable interrupts when accessing them non-atomically.
- Do not call other non-reentrant functions without disabling interrupts.

Thread safety

- A thread safe function is safe to call from multiple threads concurrently.
    - In a multiprocessor environment, where two functions may run entirely in parallel thread safety must be guaranteed.
- If running under an OS, thread safety may be implemented using blocking semaphore / mutex objects.
    - However, ISRs are not permitted to block so can not call blocking functions.
    - In ISRs, mutual exclusion can only be achieved by disabling interrupts.
- A function can be reentrant, thread safe, both or neither.
    - Thread safe does not imply reentrant, or visa versa.

Interrupt Controller



1 Peripheral event sets an interrupt flag.

Flip-flop latches high when event occurs.

2 The peripheral interrupt type must be enabled.

TimerIntEnable()

The enabled peripheral interrupts are OR'ed into a single IRQ.

3 IRQ input signal from all peripherals connect to the controller and are enabled. IntEnable()

4 Controller prioritizes and outputs the interrupt number for the highest priority and asserts its output IRQ to the CPU.

5 The CPU global interrupt must be enabled.

IntMasterEnable()

6 The CPU preforms the ISR entry sequence, using the interrupt number to fetch the address of the ISR stored in the vector table.

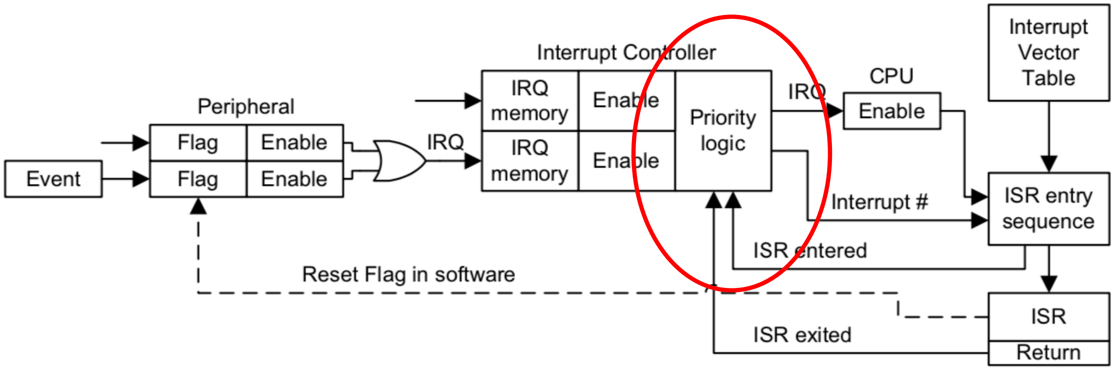
7 ISR Routine starts interrupt and entry status is provided back to the interrupt controller.

8 The ISR must clear original interrupt at the peripheral.

TimerIntClear()

9 ISR Routine exists providing status back to the interrupt controller.

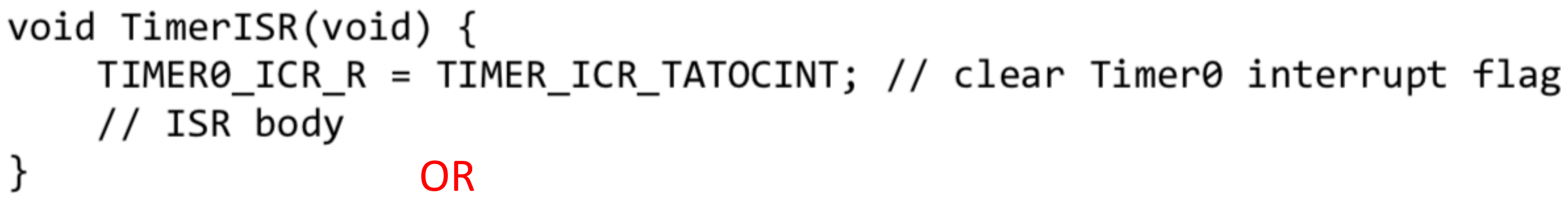
Interrupt Controller: Prioritization



- To trigger an interrupt, the interrupt must be enabled in the controller as well as in the peripheral and the global interrupt signal to the CPU must be enabled.
- The Interrupt controller receives an IRQ input signal from each peripheral.
  - The peripheral status is stored in the IRQ memory.
  - Even if the input IRQ signal de-asserts before the ISR is called, the controller keeps it in pending state.
  - When the ISR is entered, the state is set to active.
  - When the ISR exits,
    - If the interrupt is still asserted it places it back to pending.
    - If the interrupt is not asserted it places it into an inactive state.
- The controller keeps track of ISR nesting.
  - All interrupts are allowed while running foreground code in main().
  - All interrupts are disabled while running the highest priority ISR.
  - While running background ISRs, only higher priority interrupts are allowed while an ISR is in process.
    - If a new interrupt request is higher priority than the current ISR, the CPU IRQ line will assert to request an interrupt.

Clearing the Peripheral ISR Flag

- The ISR routine is responsible for clearing the peripherals interrupt flags.
  - This should be done immediately at the start of the ISR.
- Depending on the time critical nature of the ISR it can be done using driver calls or by direct access to the hardware register.



```
void TimerISR(void) {
    TIMER0_ICR_R = TIMER_ICR_TATOCINT; // clear Timer0 interrupt flag
    // ISR body
}
```

OR

TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

- What would happen if the ISR did not clear the flag?
  - ?
- Why is it important that it happen as the first step?
  - ?

Older Devices

- Older devices may have a less integrated interrupt controller.
- The ISR routines may have to explicitly call commands that are automated in our controller.
    - May explicitly need to tell the controller it is entering and exiting the ISR routine.
    - May not support full prioritization of tasks.
        - Requiring ISR to explicitly enable preemption during the ISR to allow higher priority interrupts to preempt.
- May require an RTOS to fill in the functions provided by a more sophisticated interrupt controller.
    - Cost of RTOS is additional complexity and slower response times.

Real-Time Operating System (RTOS)

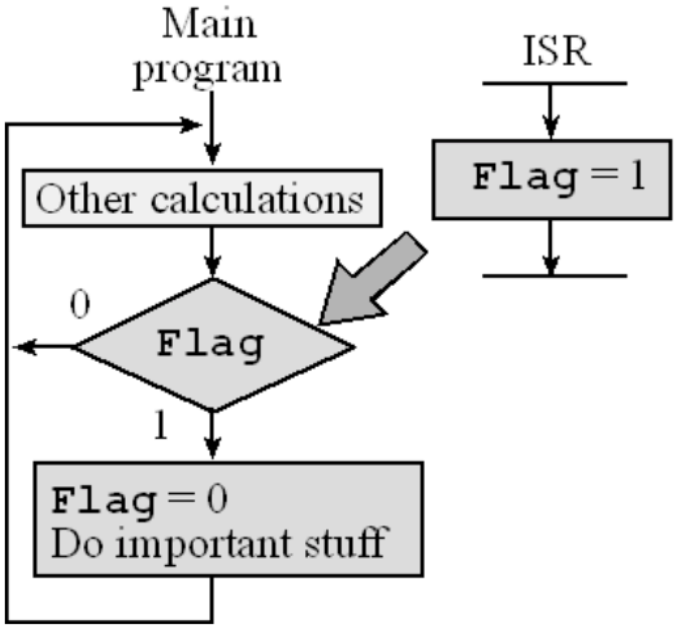
- RTOSes vary widely in capabilities.
  - We are using TI-RTOS.
- Some core features are universal.
  - Multitasking.
  - Preemptive priority scheduler for tasks.
  - Task synchronization via semaphores and related objects that can block and unblock certain tasks.
- Non-core features that are sometimes supported.
  - Networking support.
  - GUI features.
  - Database functionality.

RTOS: Multitasking

- RTOS keeps track of multiple threads, called tasks.
  - Each task is similar to main() which contains initialization at the beginning and a infinite while loop.
- Tasks have the following characteristics.
  - They share code and global data.
  - They have separate stacks and CPU register values for context switching.
    - Interrupts share one stack.
  - They are prioritized below ISRs, but higher priority tasks preempt lower priority tasks.
  - When switching context,
    - The task registers are pushed to its own stack.
    - The stack pointer is changed to a different task.
    - When the task starts running again, the task registers are popped from the stack and context is restored.

RTOS: Semaphores

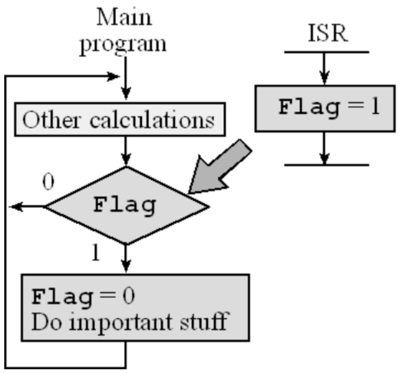
- Review: A semaphore is a mechanism for synchronizing two threads to protect access to shared resources.
  - One thread performs the signal action, by setting a flag.
    - The signal action is implemented with a post() command.
  - One thread performs a wait action, waiting for the flag to be set, clearing it and then performing the shared data operations.
    - The wait action is implemented with a pend() command.



RTOS: High-Level SW Structure

```
void main(void) {
  <init>;
  start_RTOS();  // under TI-RTOS: BIOS_start();
}

void ISR_event1(void) {
  post(semaphore1); // OS call
}

void ISR_event2(void) {
  post(semaphore2); // OS call
}

void OS_task1(void) {
  <init>;
  while (1) {
    pend(semaphore1); // blocking OS call
    <handle event 1>;
  }
}

void OS_task2(void) {
  <init>;
  while (1) {
    pend(semaphore2); // blocking OS call
    <handle event 2>;
  }
}
```

- Main function starts RTOS
  - Starts ISR and task functions running.
- This example has two events, each with an ISR.
- Each ISR is associated with a semaphore.
  - ISR_event1 accesses semaphore1.
  - ISR_event2 accesses semaphore2.
- Each ISR also has a related task with a while loop to act as an event handler.
  - The task also accesses the related semaphore.



Example RTOS Semaphore

```
    void main(void) {
      <init>;
      start_RTOS();  // under TI-RTOS: BIOS_start();
    }

    void ISR_event1(void) {
3     post(semaphore1); // OS call
    }

    void ISR_event2(void) {
      post(semaphore2); // OS call
    }

    void OS_task1(void) {
1     <init>;
      while (1) {
        pend(semaphore1); // blocking OS call
4       <handle event 1>;
      }
    }

    void OS_task2(void) {
      <init>;
2     while (1) {
        pend(semaphore2); // blocking OS call
        <handle event 2>;
      }
    }
```

1 OS_task1() start up.

- Initializes its functions.
- Enters while loop.
- OS_task1 performs wait action using pend() command.
- Stays in pend() waiting for signal action happens on semaphore1.
  - While pending the task is blocked and allows other tasks to execute.

2 OS_task2() starts up and pends on semaphore2.

3 Event1 triggers ISR_event1.

- ISR_event1 performs a signal action using the post() command for semaphore1.

4 OS_task1()

- Sees the post and unblocks handling the event1 functionality.
- Performs the important stuff.
- Returns to the pend() state and is blocked.

- Same functionality can be seen with OS_task2() and ISR_event2().

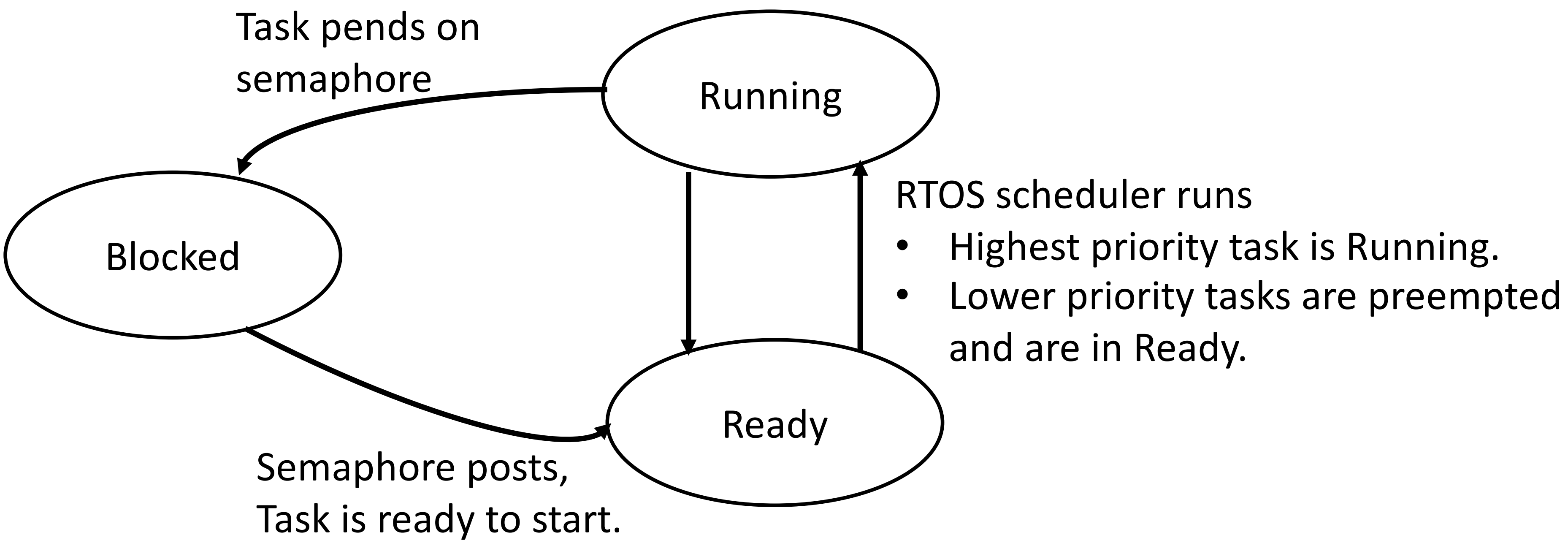
If two things need to run at once, what happens?

- ?

RTOS: Controls Priority

- Each task has a state that is monitored by the RTOS
  - If a task is in its pend() function, it has a blocked state.
  - If a task a has received a signal and has work but a higher priority thread is already running it enters the ready state.
  - If a task is executing on the CPU, it is running.
  - ISRs are never in blocked state.
- The RTOS is responsible for prioritizing event1 and event2 functionality.
  - ISR's always have higher priority than tasks.
  - If both tasks are not blocked,
    - The RTOS selects the higher priority task placing it in running state.
    - It preempt the lower priority task placing it in ready state.

RTOS: Task State Transitions



- RTOS has a scheduler to determine which task should run.
  - Highest priority task is selected that is ready to run.
  - There is no time slicing like in a traditional OS.
  - There maybe starvation just like with non-RTOS scheduling.
  - Most RTOS do not support more sophisticated algorithms than RMS.

RTOS: Scheduler

- Handling of tasks with same priority depends on RTOS.
    - Some disallow same priority tasks.
    - Some will time-slice them.
    - Some will run each until it blocks.
- If no threads are running, RTOS runs an idle thread.
    - Can contain user-supplied non real-time code.
    - Idle thread can not block itself or unblock other tasks.
    - Can be used to collect statistics like CPU utilization.
    - Can enter low-power mode.
- Highest priority task latency and recovery time is increased.
    - Max task latency = execution time of all ISRs + time to schedule the task + context switch time.
    - Max recovery time = time to find highest priority task + context switch time + return from interrupt.
- ISRs latency is increased, during task scheduling because interrupts are disabled.

RTOS: Interrupt Service Routines

- TI-RTOS hardware interrupts (Hwi) call the interrupt service routines (ISRS).
- When exiting from an ISR, the RTOS runs the scheduler.
  - If the ISR preempted a lower priority ISR, the task scheduler does not run but instead it returns to the ISR.
  - This is because ISR are always higher priority than tasks. The task is then delayed until all the ISRs are exited.
- ISR may return to a different task than the one it interrupted.
  - Caused by posting to a higher priority semaphore then the one that was interrupting.
- ISRs are not allowed to block.
  - They can not wait on semaphores, messages queues or other blocking objects.
  - Most synchronization functions offer non-blocking versions that can be used in ISRs. They may return an error code instead of blocking.

RTOS: Interrupt Service Routines

- TI-RTOS “zero-latency interrupts”.
  - ISR called directly by the hardware interrupt mechanisms.
  - They are the highest priority interrupt.
  - They do not communicate with the RTOS and thus can not unblock tasks.
  - They are always enabled even when scheduler is running.
  - Some RTOS only support this type of interrupt.
- Variations for less capable RTOS.
  - In TI-RTOS, interrupts are routed to the RTOS and then the RTOS calls the ISR.
  - Therefore explicit RTOS calls are not required on entry and exit of the ISR. (not always the case with other RTOS).
  - May ignore interrupt prioritization features and implement them in software.