ECE3849
D-Term 2021

Real Time Embedded Systems

Module 2 Part 2

Module 2 Part 2 Overview

- Handling Shared Data
  - Disabling Interrupts
  - Priority Inversion
  - Atomic Operations
  - Using Local Variables
  - Common Shared Data Bugs

Shared Data and Resources

- The only way to pass data to or from an ISR is through global variables.
  - This is referred to as inter-thread communication.
  - These globals are called shared data.
  - Data can be shared between ISRs or between the foreground and background tasks.
  - Issues arise when multiple threads try to access the same resource, particularly if one is reading and the other writing.
- Peripherals are classified as shared resources, and treated in the same way as shared data.

Shared Data Problem: Time Example

```
int iHours, iMinutes, iSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++iSeconds >= 60) {
        iSeconds = 0;
        if (++iMinutes >= 60) {
            iMinutes = 0;
            if (++iHours >= 24)
                iHours = 0;
        }
    }
}

long SecondsSinceMidnight(void) {
    return ((iHours * 60) + iMinutes) * 60 + iSeconds;
}
```

Global variables = shared data

Timer ISR called once per second and increments the hours, minutes and seconds.

- Time sequence -hh:mm:ss
  - 03:59:58
  - 03:59:59
  - 04:00:00
  - 04:00:01

Converts time back to seconds.

- Function called whenever needed.

Do we have any control over when the function is called relative to the ISR?

- ?

Any concerns?

- ?

Shared Data Problem: Time Example

```
int iHours, iMinutes, iSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++iSeconds >= 60) {
        iSeconds = 0;
        if (++iMinutes >= 60) {
            iMinutes = 0;
            if (++iHours >= 24)
                iHours = 0;
        }
    }
}

long SecondsSinceMidnight(void) {
    return ((iHours * 60) + iMinutes) * 60 + iSeconds;
}
```

Timer ISR called once per second and increments the hours, minutes and seconds.

- Time sequence -hh:mm:ss

  03:59:58
  03:59:59
  04:00:00
  04:00:01

What if it is 3:59:59 and the interrupt occurs here?

iHours = ?
iMinutes = ?
iSeconds = ?

Do we have a bug: ?
If so, how often will it happen?

Shared Data Fix: Disable Interrupts

```
#include "inc/hw_types.h"
#include "driverlib/interrupt.h"

long SecondsSinceMidnight(void) {
    long lTemp;

    IntMasterDisable();
    lTemp = ((iHours * 60) + iMinutes) * 60 + iSeconds;
    IntMasterEnable();
    return lTemp;
}
```

Disable the Interrupt so ISR can not change values.

Critical section.

Re-enable interrupts after critical section.
Interrupts will resume execution.
ISR will be on a waitlist if they happen during this time.

- It is possible to disable the interrupts during the critical calculation.
  - Care must be taken to minimize the time interrupts are disabled.
  - Why?
- What happens if interrupts are originally disabled?
  - ?

Shared Data : Disable Interrupts Fix

```
#include "inc/hw_types.h"
#include "driverlib/interrupt.h"

long SecondsSinceMidnight(void) {
    tBoolean bIntAlreadyDisabled;
    long lTemp;

    bIntAlreadyDisabled = IntMasterDisable();
    lTemp = ((iHours * 60) + iMinutes) * 60 + iSeconds;
    if (!bIntAlreadyDisabled)
        IntMasterEnable();
    return lTemp;
}
```

IntMasterDisable returns a Boolean that contains the original state.

TRUE: if originally disabled.

FALSE: if originally enabled

Now only enable interrupts if they were originally enabled.

- IntMasterDisable() and IntMasterEnable() are specific to the TI libraries.
  - If you ever wanted to move the code to a different processor. Every instance of them would need to be renamed everywhere in the code.
  - Portability is the ease of which code is moved from one environment to another.
  - Adding the macros below will improve portability as you only need to change one line of code instead of every instance.

```
#define FunctionDisablesInterrupts() tBoolean bIntAlreadyDisabled;
#define EnterCritical() bIntAlreadyDisabled = IntMasterDisable();
#define ExitCritical() if (!bIntAlreadyDisabled) IntMasterEnable();
```

Disabling Interrupts Pros & Cons

- Pros
  - Easy to solve the shared data problem.
    - Just a couple lines of code
    - Doesn’t take long to execute, CPU load not affected to much
- Cons
  - Additional latency is introduced.
  - The longest period of time that interrupts are disabled is added to the latency of every ISR.
    - If other parts of the system have longer disabled periods, the maximum latency will not be increased.

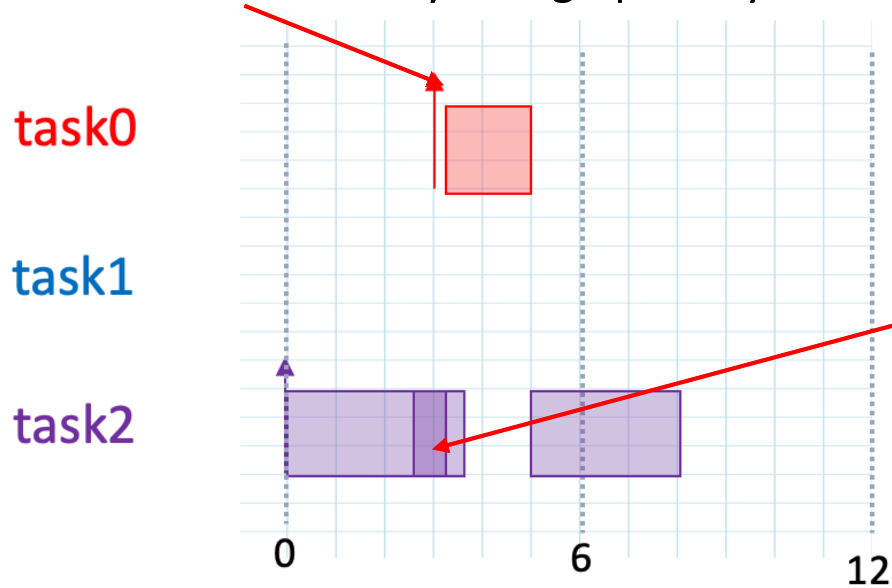
Disabling a Specific Interrupt

```
long SecondsSinceMidnight(void) {
    long lTemp;

    IntDisable(INT_TIMER0A);
    lTemp = ((iHours * 60) + iMinutes) * 60 + iSeconds;
    IntEnable(INT_TIMER0A);
    return lTemp;
}
```

Disable just the interrupt we care about. Example TIMER0A is used to increment Time.

Small increase in latency of high priority task0.

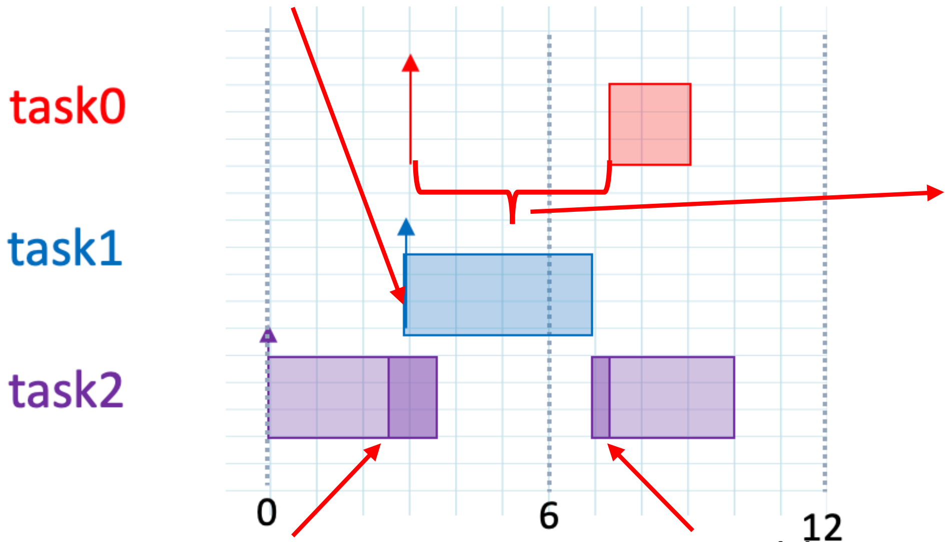


- Expected behavior
  - Task 0 and 2 share TIMER0A.
  - Task2 disables TIMER0A Interrupt.
  - Task0 latency increases by a the small delay of the critical section.

- Task1 does not share TIMER0A. How will it affect latency?
  - ?

Priority Inversion

- Task0 and Task2 share TIMER0A Interrupt.
  - Task0 will be blocked by Task2 during critical section.
- Task1 does not share TIMER0A
  - Task1 can preempt Task2 during the critical section.



- Latency of high priority task0 increases by the latency of mid priority task1.
- When a lower priority task takes priority over a higher priority task.
  - This is called priority inversion.
- Scheduling analysis no longer valid.

Disable TIMER0A Interrupt

Enable TIMER0A Interrupt

- Pros of Disabling just TIMER0A
  - Less impact in overall system performance.
  - For example, if we are sampling data with an ADC in task1, it is unaffected by the TIMER0A interrupt.
- Cons
  - Increases complexity making it more error prone.
  - Opens the possibility of priority inversion.

Atomic Operations

- What if we simplify the functionality to just deal with seconds?
  - We increment seconds in the ISR and roll over at midnight to 0.
  - Then simply return one value, lSeconds, in the function call.
  - Use a different function to calculate hours, minutes, seconds when needed.

```
long lSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++lSeconds >= 60 * 60 * 24) {
        lSeconds = 0;
    }
}

long SecondsSinceMidnight(void) {
    return lSeconds;
}
```

- lSeconds is a 32-bit value, we have a 32-bit CPU.
  - lSeconds can be read in one clock cycle.
  - Operations that can be completed in a single clock cycle are called atomic operations.
  - There is no shared data problem in this case.
- What if lSeconds was a 64-bit value would we have a problem?
  - ?.

TimerISR: lSeconds Simplification

```
long lSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++lSeconds >= 60 * 60 * 24) {
        lSeconds = 0;
    }
}

long SecondsSinceMidnight(void) {
    return lSeconds;
}
```

- This fix depends both on the data type and the hardware architecture.
  - This makes it hard to port to other CPUs,
  - Easy to make a mistake.
  - Difficult to detect and debug.

- What if SecondsSinceMidnight is called from a higher priority interrupt than TimerISR?
  - ?

Effects of Interrupting TimerISR

```
long lSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++lSeconds >= 60 * 60 * 24) {
        lSeconds = 0;
    }
}

long SecondsSinceMidnight(void) {
    return lSeconds;
}
```

- ++lSeconds
  - Reads lSeconds
  - Modifies lSeconds by incrementing.
  - Writes new value of lSeconds.
- Executes a second write function, to clear the lSeconds value.

- After incrementing, lSeconds == 60*60*24
  - This is not a valid value, not expected in the final result.
  - If the high priority interrupt occurs between the two writes, this value will be returned instead of the expected value of 0.

Local Variable Fix

- Re-writing the TimerISR as shown below
  - Reduces the number of writes to the global variable to only one atomic write.
  - No shared data problem.

```
if (lSeconds + 1 >= 60*60*24)
   lSeconds = 0;
else
   lSeconds++;
```

lSeconds + 1 is stored in a temporary local variable.

lSeconds global variable only written once.

Common Shared Data Bugs

Low-priority thread = main()

high-priority thread = ISR

Operation	Low-priority thread	High-priority thread
Read-modify-write	1. Reads the global variable producing a local copy 2. Modifies the local copy 3. Writes the modified copy to the global variable	Writes to the global variable

If high priority write occurs between steps 1 and 3, data is lost.

Non-atomic read or multiple reads	1. Reads part of the global variable 2. Reads another part of the global variable	Writes to the global variable

If high priority write occurs between steps 1 and 2.

½ of the read value is old and ½ of the read value is new.

Non-atomic write or multiple writes	1. Writes part of the global variable 2. Writes another part of the global variable	Reads the global variable

If high priority read occurs between steps 1 and 2.

½ of the read value is old and ½ of the read value is new.

Write followed by read	1. Writes to the global variable 2. Reads the global variable, expecting the originally written value	Writes to the global variable

If high priority write occurs between steps 1 and 2.

The read value is not the expected value written by the lower priority thread.