Interrupt Processing III

ENCE361 Embedded Systems 1

Course Coordinator: Ciaran Moore (ciaran.moore@Canterbury.ac.nz)

Lecturer: Le Yang (le.yang@canterbury.ac.nz)

Department of Electrical and Computer Engineering

Where we’re going today

- **More on shared-data problem**
- Atomicity and volatility
- Resource sharing and embedded system design
- Homework

Shared Data Problem Revisited

- Interrupts are asynchronous
  - Interrupt service routine (ISR) does not have input parameters and does not return anything
- Global and static variables can both be used for communications between foreground programs (ISRs) and background programs
- Shared data problem may occur when
  - The program accessing global variables is interrupted
  - Global/static variables are modified by the associated ISR

Example A (1)

```
static int iTemperatures[2];

void interrupt vReadTemperatures (void)
{
    iTemperatures[0] = !! read in value from hardware
    iTemperatures[1] = !! read in value from hardware
}

void main (void)
{
    int iTemp0, iTemp1;

    while (TRUE)
    {
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        if (iTemp0 != iTemp1)
            !! Set off howling alarm;
    }
}
```

// Nuclear reactor temperature readings from 2 sensors !!

// ISR for getting 2 temperature readings

// Background program

What if an interrupt occurs here? False alarms!

// Alarm if two readings are not identical

From Page 93 in D. E. Simon's book "An Embedded Software Primer"

Example A (2)

```
static int iTemperatures[2];
void interrupt vReadTeperatures (void)
{
   iTemperatures[0] = !! read in value from hardware
   iTemperatures[1] = !! read in value from hardware
}

void main (void)
{
   int iTemp0, iTemp1;
   while (TRUE)
   {
      disable (); // Disable interrupts while we use the array
      iTemp0 = iTemperatures[0];
      iTemp1 = iTemperatures[1];
      enable ();

      if (iTemp0 != iTemp1)
         !! Set off howling alarm;
   }
}
```

Critical Section requiring undisturbed access to shared variables

From Page 96 in D. E. Simon's book "An Embedded Software Primer"

Example B (1)

```
static int iSeconds, iMinutes, iHours;        // Global variables: seconds, minutes, hours

void interrupt vUpdateTime (void)              // ISR for keeping track of time in the format Hour : Min : Second
{
    ++iSeconds;                                // Interrupts occur every second
    if (iSeconds >= 60)
    {
        iSeconds = 0;
        ++iMinutes;
        if (iMinutes >= 60)
        {
            iMinutes = 0;
            ++iHours;
            if (iHours >= 24)
                iHours = 0;
        }
    }

    !! Do whatever needs to be done to the hardware
}

long lSecondsSinceMidnight (void)              // Function that accesses global variables (c.f. slide 18 in last lecture)
{
    return ( (((iHours * 60) + iMinutes) * 60) + iSeconds);     Critical section
}
```

What if it is interrupted when iMinutes = 59 and iSeconds = 59?
(see Homework problem 1)

From Page 98 in D. E. Simon's book "An Embedded Software Primer"

Example B (2)

- Original code

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

All interrupts are always enabled after lSecondsSinceMidNight is executed

Call to enable() may mistakenly enable interrupts, leading to interference

- An improved/better program

```
long lSecondsSinceMidnight (void)
{
    long lReturnVal;

    disable ();

    lReturnVal =
        (((iHours * 60) + iMinutes) * 60) + iSeconds;

    enable ();

    return (lReturnVal);
}
```


From Page 99 in D. E. Simon's book "An Embedded Software Primer"

Where we’re going today

- More on shared-data problem
- **Atomicity and volatility**
- Resource sharing and embedded system design
- Homework

Atomicity

- Atomicity
  - A part of a program is said to be atomic if it cannot be interrupted
    - E.g., use IntMasterDisable() or it can be executed in a single processor clock cycle
- Shared data problem
  - The code that uses the shared data is *not* atomic
    - Could occur in foreground and/or background programs
- Critical section
  - Part of the program that must be atomic in order for the system to work properly

Example 2 Revisited

- Original code

```
long lSecondsSinceMidnight (void)
{
   long lReturnVal;

   disable ();

   lReturnVal =
      (((iHours * 60) + iMinutes) * 60) + iSeconds;

   enable ();

   return (lReturnVal);
}
```

- An further improved program

```
long lSecondsSinceMidnight (void)
{
   long lReturnVal;
   BOOL fInterruptStateOld; /* Interrupts already disabled? */

   fInterruptStateOld = disable ();

   lReturnVal =
      (((iHours * 60) + iMinutes) * 60) + iSeconds;
   /* Restore interrupts to previous state */
   if (fInterruptStateOld)
      enable ();

   return (lReturnVal);
}
```

// Returns TRUE if interrupts were previously enabled

// Re-enable interrupts if they were previously enabled

From Page 100 in D. E. Simon's book "An Embedded Software Primer"

Globally Disable Interrupts in TivaWare

- TivaWave API function: tBoolean IntMasterDisable(void)
  - Page 349, TivaWare Peripheral Driver Library Users Manual.pdf
  - Prevents the processor from receiving interrupts, except for the set of interrupts enabled in NVIC
  - Returns TRUE if interrupts were already disabled or FALSE if interrupts were initially enabled
  - Include “hw_types.h” first and then include “interrupt.h”

Volatility: Use of volatile (1)

```
static long int lSecondsToday;              // Static variable for foreground/background communication

void interrupt vUpdateTime (void)           // ISR for keeping track of time
{
    .
    .
    .
    ++lSecondsToday;                        // Increase lSecondsToday by one
    if (lSecondsToday == 60L * 60L * 24L)
        lSecondsToday = 0L;                 // Reset lSecondsToday
    .
    .
    .
}

long lSecondsSinceMidnight (void)           // Read global variable twice to make sure no interrupts
{                                           occur between two reading operations.
    long lReturn;

    /* When we read the same value twice, it must be good. */
    lReturn = lSecondsToday;                // Copy the value of lSecondsToday
    while (lReturn != lSecondsToday)
        lReturn = lSecondsToday;            // Copy the value of lSecondsToday

    return (lReturn);
}
```


From Page 102 in D. E. Simon’s book “An Embedded Software Primer”

Volatility: Use of volatile (2)

- A potential problem from compiler optimization
  - lSecondsToday may be realized as a register variable
  - In this case, compiler may think the while loop is not needed at all!!

```
long lSecondsSinceMidnight (void)
{
    long lReturn;

    /* When we read the same value twice, it must be good. */
    lReturn = lSecondsToday;
    while (lReturn != lSecondsToday)
        lReturn = lSecondsToday;

    return (lReturn);
}
```

- Solution: change the declaraction of lSecondsToday into

Type qualifier → volatile static long int lSecondsToday

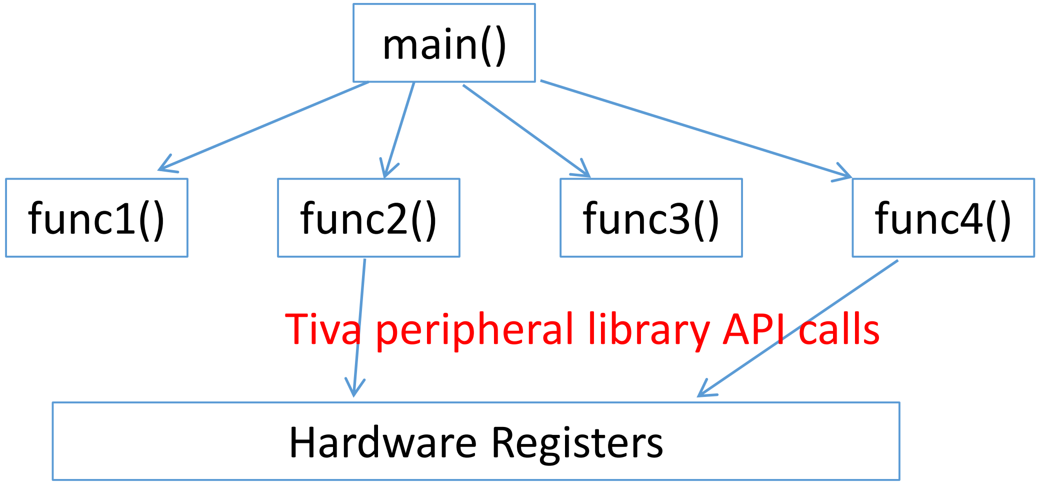
- This tells compiler that the variable may be changed unexpectedly
  - Must be realized as a memory variable

Where we’re going today

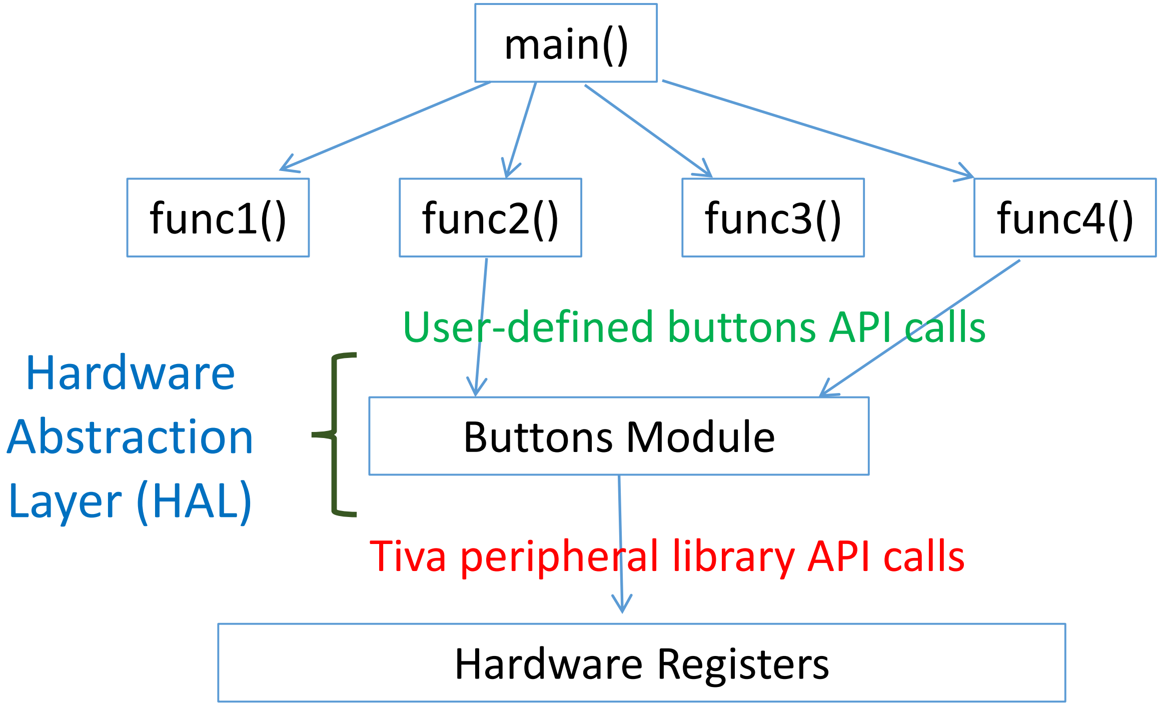
- More on shared-data problem
- Atomicity and volatility
- **Resource sharing and embedded system design**
- Homework

Sharing Peripherals

- Peripherals are shared within a program
  - Any part can write to the registers controlling the peripheral
- Efficient way for sharing peripherals



Every functions make their own calls to Tiva peripheral library API, some of which may be the same → no code reuse!



See buttons4.h and buttons4.c on Learn

Sharing GPIO Ports

- Most GPIO API functions can operate on multiple GPIO pins within a single port (with 8 pins) simultaneously

```
int32_t GPIOPinRead (uint32_t ui32Port, (GPIO_PIN_3 | GPIO_PIN_0))
```

ucPins Parameters

- ucPins parameters specify affected (read/write) pins
  - Other pin values are masked out
  - Pin masking occurs in hardware, allowing to modify individual GPIO pins in a single clock cycle → Atomic bit-banded operation*
    - Typical method resorts to a **read-modify-write** operation to set/clear a single pin, requiring multiple clock cycles (see Homework Problem 3)

*See more in Section 6.7.2 on Page 219 in J. Yiu's book "The Definite Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Embedded System Design (1)

I/O budget

- Interface to each external device
- Interface to each external event
- Means of programming

Resource budget

- Allocation of I/O pins
- Guesstimate of memory requirements (size, type, internal/external)
- Allocation of timers, serial I/O, ADC, etc.
- Need to share resources

Embedded System Design (2)

Task budget (more in next lecture)

- Allocation of interrupts (foreground tasks)
- Identification of background tasks

Scheduling budget (more in future lectures)

- Identification of interrupt priorities
- Identification of background task priorities
- Calculation load
- Choice of scheduling method
- Provision of services, e.g., communication between tasks

Homework

1. Consider Slide 6. If the function **lSecondsSinceMidnight()** starts being executed at the time when the global variables have values **iSeconds** = 59, **iMinutes** = 59 and **iHours** = 3. After an interrupt occurs, what is the biggest difference possible between the value the function returns and the actual number of seconds since midnight?

2. Consider the discussion of the use of the C keyword **volatile** on Slides 12 and 13. The two global variables in the example code for **tick64Test.c** (available on Learn) are declared as **volatile**. Is this necessary? If so, why?

3. Why is there a shared data problem associated with the conventional *read-modify-write* operation used on many microprocessors to write to (i.e., to change) individual bits on I/O ports? Describe a scenario in a program using interrupts where the shared data problem exists.