Jonathan W. Valvano March 2, 2012, 10:00 to 10:50am

(10) Question 0. Your crib sheet will be graded on content and correctness.

(4) Question 1. The LM3S8962 has eight different ways the ADC can be configured to trigger.

Software start (controller)	Timer
GPIO external trigger	Analog comparator
3 PWM signals	Continuous sampling

(6) Question 2. For the robot you will have four IR distance sensors
Any sequencer other than seq3 can be used. Specify the sequence to be ADC0, ADC1, ADC2 then ADC3. Program it to END on ADC3, and request an ADC ISR after sampling ADC3

```
ADC0_SSCTL0_R  = 0x00006000;
ADC0_SSMUX0_R = 0x00003210;   // any order is ok
```

(5) Question 3. It is possible to perform a memory read cycle fetching an op code at the very same instant as it performs a memory write cycle pushing data onto the stack because these operations occur on separate buses. The Cortex M3 is a Harvard architecture. It does have a pipeline, which does allow for simultaneous op code fetch and instruction decode. However, a pipeline neither prevents or allows simultaneous op code and data access.

(5) Question 4. R13 is the stack pointer. There is a PSP and a MSP to allow the operating system to run more securely. The user tasks can run with the PSP and the OS functions can run with the MSP.

(12) Question 5. Select the best term from chapter 4 that describes each definition.

Part **a)** Bounded waiting
Part **b)** Path expression
Part **c)** Rate monotonic scheduler
Part **d)** Hook
Part **e)** Certification
Part **f)** Priority inversion

(8) Question 6. Consider the low pass filter function.

Part a) This implementation is not reentrant because of the read modify write to global **`y`**.

Part b) **`x`** is passed in R0.

Part c) LR contains the function return address during the execution of the function.

(10) Question 7. Deadlock cannot occur when using a monitor for thread synchronization because it does not allow **hold and wait**. When a thread waits inside a monitor, it releases the lock. It can not enter more than one monitor at a time. I.e., once a thread enters a monitor, it does not return until that particular task is complete.

(15) Question 8. Consider three foreground threads that I want to run with your OS.

Part a) My OS would crash if the user thread finishes because of a stack underflow.

Part b) This is fine as long as there is at least one free TCB. The time to complete all of **`t2`** is less than 2ms, so the first execution will kill before the second instance launches.

Part c) With two calls to **`OS_AddThread(&t3)`**, this will run out of TCBs. The second **`OS_Kill`** never occurs. This will use up all the TCBs because there are two calls to **`OS_AddThread`** for every one **`OS_Kill`**. Each execution creates two more. So 1 execution creates 2, 2 executions create 4, 4 executions create 8… until you run out of TCBs.

There was a typo on the original exam. With two calls to **`OS_AddThread(&t2)`**, this will run not out of TCBs. Exactly two thread t2s will be created, resulting in two copies of part b).

(25) Question 9. There is a positive logic switch attached to Port D bit 6.

Part a) Describe changes if any you wish to make to the initialization code (changes in bold)

```
#define PD6  (*((volatile unsigned long *)0x40007100))
void (*PD6Task)(void);   // user function on rising edge of PD6
unsigned long static LastPD6; // previous value
//******** OSAddPD6Task  ********
// add a background task to run on rise of PD6, priority 3
// Inputs: pointer to a void/void background function
void OS_AddPD6Task(void(*task)(void)){
  SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOD; // activate port D
  PD6Task = task;             // user function
  GPIO_PORTD_DIR_R &= ~0x40;  // make PD6 in
  GPIO_PORTD_DEN_R |= 0x40;   // enable digital I/O on PD6
  GPIO_PORTD_IS_R &= ~0x40;   // PD6 is edge-sensitive
  GPIO_PORTD_IBE_R |= 0x40;   // PD6 is both edges
  GPIO_PORTD_IEV_R |= 0x40;   // PD6 rising edge event
  GPIO_PORTD_ICR_R = 0x40;    // clear flag6
  GPIO_PORTD_IM_R |= 0x40;    // enable interrupt on PD6
  GPIO_PORTD_PUR_R |= 0x40;   // PD6 does not have pullup
  LastPD6 = PD6;
  NVIC_PRI0_R = (NVIC_PRI0_R&0x00FFFFFF)|(3<<29); // 3, bits 31-29
  NVIC_EN0_R |= NVIC_EN0_INT3;// enable interrupt 3 in NVIC
}
```

Part b) Rewrite the ISR to handle the bounce while still minimizing latency.

```
void static DebounceTask(void){
  OS_Sleep(2);   // foreground sleeping, must run within 50ms
  LastPD6 = PD6; // read while it is not bouncing
  GPIO_PORTD_ICR_R = 0x40;    // clear flag6
  GPIO_PORTD_IM_R |= 0x40;    // enable interrupt on PD6
  OS_Kill();
}
void GPIOPortD_Handler(void){
  if(LastPD6 == 0){  // if previous was low, this is rising edge
    (*PD6Task)();    // execute user task
  }
  GPIO_PORTD_IM_R &= ~0x40;   // disarm interrupt on PD6
  OS_AddThread(&DebounceTask);
}
```