Leonardo Fusser (1946995) January 30th, 2022

Week 1 & 2 HW

Embedded Operating Systems

Serge Hould

Multitasking with a scheduler HW:

Q1) Software pattern identification:

- Software pattern A: *round robin infinite super loop pattern*.
- Software pattern B: *multitasking using a pre-emptive OS pattern*.
- Software pattern C: *multitasking using a co-operative FSM pattern*.

Q2) OS component that switches between processes:

- A scheduler. The OS scheduler decides which processes (tasks) get running time and constantly swaps in and swaps out processes in order to delegate running time for the processes on the system.

Q3)

a) Task priorities determine which order the tasks run, the OS scheduler determines when they run, and it is up to the task itself to determine how long it needs to run (otherwise the OS scheduler is involved).

b) The programmer usually has no responsibilities.

Q4) Multitasking system that protects against other task failures:

- A pre-emptive OS protects against other task failures. This is because in a pre-emptive OS, each task is usually isolated from each other, so they cannot interact together easily.

Q5) Multitasking system that does not need a communication manager:

- A multitasking co-operative FSM system does not need a communication manager.

Intro to RTOS tasks HW:

Q1) When a high priority task requires the CPU, but a low priority task is currently running:

- If a low priority task is running, and a higher priority task is waiting to run, the OS scheduler will switch out running time from the lower priority task for the higher priority task.

Q2) Tasks in a RTOS vs tasks in a co-operative multitasking system:

- In a RTOS, a scheduler determines which tasks gets running time and in a co-operative multitasking system, there is no scheduler involved, and the tasks are expected to give up running time when they don't need to run.

Q3) A task with a priority of 4 and an ISR with a priority of 1, the highest priority level goes to:

- The ISR with a priority of 1. This is because ISRs have greater hardware priorities than tasks which have software priorities.

Q4a) The only true statement:

- D. The system will schedule Task1 to run at the end of the current time slice.

Q4b) The only true statement:

- C. The system will schedule Task3 to run at the end of the current time slice.

Q4c) The only true statement:

- E. The system will run ISR A right away.

Q5) Factor that decides a task to become unblocked:

- Some common factors that make a task transition from blocked to ready is:
    - A needed resource(s) for the task has become available.
    - An event has occurred.
    - Waited long enough.

Q6) When a task runs:

The task decides to enter the blocked state if it deems that it is appropriate to do so. A common reason why a task may enter the blocked state is that it needs a resource that has not yet become available.

a) The task also decides to transition from the blocked state to the ready state if it is appropriate to do so. Some of the common reasons that this may occur have been stated previously.

Q7) When a task is in the blocked state:

- A scheduler cannot force a task to transition from the blocked state to the running state. It is up for the task to decide when it is appropriate to transition from the blocked state to the ready state. On the other hand, the scheduler will only decide when the task should go from the ready state to the running state.

Q9) For the task in listing 1:

- It is impossible for the task in listing 1 to ever enter into the blocked state. This is due to the nature of the crude NULL loop. When a task has a crude NULL loop, it has no need to enter the blocked state, since it has all the resources needed for it to run.

Q10) For the task in listing 1:

- The task in listing 1 doesn't leave much room for spare processing capacity since it is always in the running state. The percentage of time consumed for the task is close to 100%. This is due to the nature of the crude NULL loop (as explained above).

Q11) If two tasks are created similar to the task in listing 1 with different priorities:

- The percentage of time that is consumed in vTask1 is 0% and the percentage of time that is consumed in vTask2 is close to 100%. This is because vTask2 has a higher priority (3) than the priority for vTask1 (2). This is also because vTask2 is using a crude NULL loop. As explained before, since vTask2 does not need to wait for any resources, it will always be in the running state, and it will not leave any spare processing time for the system to use.

RTOS queues HW:

Q1) For the snippet of code:

- The incorrect queues and semaphore functions are used inside the ISRs. This will cause the system to not behave as expected. Here is a fix using the correct queues and semaphore functions for the ISRs that will meet the user's behavior expectation:

```
void _ISR_NO_PSV _T2Interrupt( void ){
     static int cnt = 0;
     cnt++;
     xQueueSendFromISR(xQueue, &cnt, 0);
     _T2IF = 0;
}
...
void _ISR_NO_PSV _T3Interrupt( void ){
     static int cnt=0, tx=0, rx=0;
     cnt++;
     if(xQueueReceiveFromISR( xQueue1, &rx, 0 ) == errQUEUE_EMPTY){
          tx = rx + cnt;
          xQueueSendFromISR(xQueue, &tx, 0);
     }
     _T3IF = 0;
}
```

Q2)

a) To display the received temperature:

```
QueueHandle_t xQueueLCD;
...
xQueueLCD = xQueueCreate( 5, sizeof(int) );
...
/*Task that receives temperature values and displays them */
void vTaskLCD( void *pvParameters ){
      int temp;
      for( ;; ){
            //Blocking queue
            xQueueReceive( xQueueLCD, &temp, portMAX_DELAY );
            printf("Current temperature: %d\n", &temp);
            //Small delay for printing (if needed)
      }
}
```

b) To display a warning message if the received temperature is over 45 degrees:

```
QueueHandle_t xQueueLCD;
...
xQueueLCD = xQueueCreate( 5, sizeof(int) );
...
/*Task that receives temperature values and displays them */
void vTaskLCD( void *pvParameters ){
      int temp;
      for( ;; ){
            //Blocking queue
            xQueueReceive( xQueueLCD, &temp, portMAX_DELAY );

            if(temp > 45){
                  printf("Warning! High temperature!\n");
            }
            else{
                  printf("Current temperature: %d\n", &temp);
                  //Small delay for printing (if needed)
            }
      }
}
```

Q3)

a) The queue length and size values are not appropriate. This is because at any point of time, the user can enter up to 3 characters in the console. The current queue depth is set to 1, but 4 should be used in order for the system to function correctly. The queue depth should be set to 4 because the null character is included (for string termination).

b) Modification to change a variable named “delay” depending on user input:

```
xQueueCtl = xQueueCreate ( 1, sizeof ( int );
...
void _ISR_NO_PSV _U2RXInterrupt( void ){
   char cChar;
   IFS1bits.U2RXIF = 0;
   while( U2STAbits.URXDA ){
         cChar = U2RXREG;         //Latest ASCII code received
         xQueueSendFromISR( xQueueCtl, &cChar, 0 );

   }

}
...
static void vTaskCtl( void *pvParameters ){
   char rx = 0;
   char str[3] = 0;
   int cnt = 0;
   int xStatus;
   unsigned char delay = 40;
   for( ;; ){
         xStatus = xQueueReceive ( xQueueCtl, &rx, portMAX_DELAY );

         if(xQueueReceive(xQueueCtl, &rx) == 0){
            str[cnt] = rx;
            if(str[cnt] == '\n'){
                  delay = atoi(str);
                  str = 0;
                  cnt = 0;
            }
            cnt++;
         }
         /* 100mS number crunching operations here */
   } //[End loop]
} //[End task]
```

The Idle Task Hook HW:

Q1) For the idle task:

- It is impossible for the idle task to pre-empt another task. This is simply because it has the lowest possible priority than any other task (0). Therefore, as soon as any other task needs to enter the running state, the idle task will give up running time for that task.

Q2) For idle hook function in listing 3:

a) If task in listing 1 is in the same program:
   - The idle hook function in listing 3 will never run due to the nature of the crude NULL loop found in the task in listing 1. The crude NULL loop found in the task in listing 1 will hog the system which will not leave room for spare processing time.

b) If tasks in listing 2 are in the same program:
   - The idle hook function in listing 3 will run from time to time due to the blocking delays found in the tasks in listing 2. The tasks in listing 2 will not hog the system due to the nature of the blocking delays and they will leave room for spare processing time.

Q3) For the listing 4 code:

a) The idle hook:
   - With the way the three tasks are structured and written, there is not much time left for the idle hook to ever run.

b) Improvement to allow the idle hook to run some time:

```
void vTask1( void *pvParameters ){
	for( ;; ){
		if(!_RB0){
			// Do something
		}
		vTaskDelay( 7 / portTICK_RATE_MS );
	}
}
void vTask2( void *pvParameters ){
	for( ;; ){
		if(!_RB1){
			// Do something
		}
		vTaskDelay( 7 / portTICK_RATE_MS );
	}
}
void vTask3( void *pvParameters ){
	for( ;; ){
		// Do some number crunching calculation 1mS
		vTaskDelay( 7 / portTICK_RATE_MS );
	}
}
void vApplicationIdleHook( void ){
	_LATA3 ^=1;
}
```

c) For the priority of vTask3:

- The actual task priority for vTask3 should not be changed. The only way to make vTask3 seem to the user that it has a lower priority is if the blocking delays found in the other tasks are extended for a longer duration.

Leonardo Fusser (1946995) February 6th, 2022

Week 3 HW

Embedded Operating Systems

Serge Hould

Shared Data and Semaphores HW:

Q1) For the following two functions:

- One of the two functions are not re-entrant. The "`vCountErrors`" function is not re-entrant because it relies on a global variable called "`cErrors`" that is non-atomic. On the other hand, the "`strlen`" function is considered a re-entrant function because it does not rely on any global variables, and it also does not rely on other non-re-entrant functions for it to work.

Q2) Improvement to avoid forgotten mutex semaphore release problems:

```
#define STEP1 10
#define STEP2 20
long cnt = 0;    //Global shared variable.
void vTask1(){
      setCnt(STEP1);   //Replaces mutex semaphore used before.
}
void vTask2(){
      setCnt(STEP2);   //Replaces mutex semaphore used before.
}
void setCnt(int temp_step){
      /* Mutex semaphore used once here to avoid any mistakes. */
      xSemaphoreTake(xMutexCnt, portMAX_DELAY);   //Takes semaphore.
      cnt += temp_step;
      xSemaphoreGive(xMutexCnt);                  //Gives semaphore.
}
```

Q3) Solution to fix problems:

```
long iRecordCount;
void increment_records (int iCount){
      set_iRecordCount(iCount, 1);     //Sets and increments.
}
void decrement_records (int iCount){
      set_iRecordCount(iCount, 0);     //Sets and decrements.
}
void set_iRecordCount(int temp_iCount, int operation){
      /* Mutex semaphore used once here to avoid any mistakes. */
      xSemaphoreTake(xMutex, portMAX_DELAY);//Takes semaphore.
      if (operation == 1){                  //Increment operation.
            iRecordCount += temp_iCount;    //Increments.
      }
      else{                                 //Decrement operation.
            iRecordCount -= temp_iCount;    //Decrements.
      }
      xSemaphoreGive(xMutex);               //Gives semaphore.
}
```

Q4.1) For each of the following situations:

a) Task M and N continuously share and update a non-native variable:
   - Mutex semaphore shared-data protection mechanism is the best choice for this application. This is because this type of shared-data protection mechanism will place any other task that tries to access the semaphore when it is in use into a blocked state, and in turn protects and synchronizes the non-native variable. The semaphore will be passed around to the other tasks to allow them to update the non-native variable when it is deemed appropriate.

b) Task P shares a single non-native variable with an ISR:
   - The best shared-data protection mechanism for this situation is if the ISR is disabled and re-enabled inside the task. This is because this type of shared-data protection mechanism will prevent the ISR from causing data corruption. ISRs have greater priorities than tasks, so they can easily cause data corruption. When the non-native variable is being updated from inside the task, the ISR is disabled beforehand and re-enabled once the non-native variable has been updated.

Q4.2) Improvement to protect a common resource:

```
void vTask1(){
	while(1){
		print_console(1);	//Prints a menu using a common resource.
	}
}
void vTask2(){
	while(1){
		print_console(0);	//Prints a menu using a common resource.
	}
}
void print_console(int choice){
	/* Mutex semaphore used once here to avoid any mistakes. */
	xSemaphoreTake(mutex_UART2, portMAX_DELAY); //Takes semaphore.
	if (choice == 1){	//Prints vTask1 menu with delay.
		vTaskDelay(1000 / portTICK_RATE_MS);
		fprintf2(C_UART2,"\r\n1111111111111111111111111111111");
		fprintf2(C_UART2,"\r\This is an example of ");
		fprintf2(C_UART2,"\r\Menu being displayed every second");
		fprintf2(C_UART2,"\r\n1111111111111111111111111111111\r\n");
	}
	else{			//Prints vTask2 menu with delay.
		vTaskDelay(2000 / portTICK_RATE_MS);
		fprintf2(C_UART2,"\r\n2222222222222222222222222222222");
		fprintf2(C_UART2,"\r\This is an example of ");
		fprintf2(C_UART2,"\r\Menu being displayed every 2 seconds");
		fprintf2(C_UART2,"\r\n2222222222222222222222222222222\r\n");
	}
	xSemaphoreGive(mutex_UART2);		//Gives semaphore.
}
```

Q5) For the following two interrupts:

- There is a shared data problem. The issue is that a global variable called "of" is being used inside of the two ISRs. Since "_IC1Interrupt" is at priority level 2 and that "_T2Interrupt" is at priority level 1, "_IC1Interrupt" can interrupt whatever "_T2Interrupt" is doing at any given point in time, resulting in data corruption for the global variable "of". A simple fix to this issue would be to disable the "_IC1Interrupt" inside of the "_T2Interrupt" when it is trying to modify the global variable "of" to avoid data corruption. This fix is shown below:

```
long of;
void _ISR _IC1Interrupt( void ){
      static unsigned int start,final;
      unsigned long ulTicks;
      final=IC1BUF;
      ulTicks = ((long)of*0xFFFF)+final-start;
      of=0;
      start=final;
      _IC1IF = 0;
}

void _ISR _T2Interrupt( void ){
      _INTIC1IE = 0;  //Disables _IC1Interrupt.
      of++;
      _INTIC1IE = 1;  //Re-enables _IC1Interrupt.
      _T2IF = 0;
}
```

Q6) Populated snippet of code:

```
void vTaskA(void){
      while(1){
            xSemaphoreGive(xBinSema);                    //Non-blocking.
      }
}
void vTaskB(){
      while(1){
            xSemaphoreTake(xBinSema, portMAX_DELAY);    //Blocking.
            _LATA0 ^= 1;                                 //Toggles a LED.
      }
}
```

Q7) Populated snippet of code:

```
int count;
void _ISR _T2Interrupt(void){
      count += 1;                                   //Counts.
      if(count == 100){
            count = 0;                              //Resets count.
            /* Non-blocking. */
            xSemaphoreGiveFromISR(xBinSema, 0);     //Deffering.
      }
      _T2IF=0;
}

void vTaskT2Handler(){
      while(1){
            /* Blocking. */
            xSemaphoreTake(xBinSema, 1000/portTICK_RATE_MS); //1Hz delay.
            _LATA0 ^= 1;          //Toggles a LED regardless of semaphore.
      }
}
```

s

Q8) For the following two ISRs:

- There are a few issues present with how the ISRs are structured. The first issue is that the scope of the variable "cnt" is limited to "_T2Interrupt", so "INT1Interrupt" cannot execute the line "cnt += 10" because it does not know what "cnt" is. The second issue is that even if the scope of the variable "cnt" is changed so that both "_T2Interrupt" and "INT1Interrupt" know what it is, there is a new issue: possibility of data corruption because of "_T2Interrupt", since the variable "cnt" is not protected. Finally, the last issue is that although the variable "cnt" is also protected inside of "INT1Interrupt", mutex semaphores cannot be used inside of any ISRs, since they will cause the system to halt and the "xQueueSendFromISR" API should be used inside of "_T2Interrupt" instead of the "xQueueSend" API. For each of the two ISRs, each one should disable the other ISR while they are updating the "cnt" variable and then re-enable them once completed. The fixes to the mentioned problems are implemented below:

```
static int cnt = 0;   //"cnt" is now a global variable.
void _ISR_NO_PSV _T2Interrupt( void ){
      _INT1IE = 0;                          //Disables INT1Interrupt ISR.
      cnt++;
      _INT1IE = 1;                          //Re-enables INT1Interrupt ISR.
      xQueueSendFromISR(xQueue1, cnt , 0);  //New queue API for ISR.
      _T2IF = 0;
}

void _ISR_NO_PSV INT1Interrupt(){
      _T2IE = 0;       //Disables _T2Interrupt ISR.
      cnt += 10;
      _T2IE = 1;       //Re-enables _T2Interrupt ISR.
      _INT1IF=0;
}
```

Q9) Modified stepper motor control code to incorporate motor CCW functionality:

```
#define PB_CW PORTDbits.RD7
#define PB_CCW PORTDbits.RD6
int stepper_state [4] = {0b1100, 0b1001, 0b0011, 0b0110};

static void vTaskStepperCW(){
      int steps;
      while(1){
            steps = 6;
            xSemaphoreTake(xBinSemaCW, portMAX_DELAY);  //Takes semaphore.
            xSemaphoreTake(xMutexSema, portMAX_DELAY);  //Takes semaphore.
            while(steps--){
                  printf("Stepping CW..\n");
                  if(cnt >= 4)cnt = 0;                  //Forwards.
                  LATA = stepper_state[cnt++];
                  vTaskDelay(100/portTICK_RATE_MS);
            }
            xSemaphoreGive(xMutexSema);                 //Gives semaphore.
      } //while(1)
}
```

...*continues on next page*...

```
static void vTaskStepperCCW(){
	int steps;
	while(1){
		steps = 6;
		xSemaphoreTake(xBinSemaCCW, portMAX_DELAY); //Takes semaphore.
		xSemaphoreTake(xMutexSema, portMAX_DELAY);  //Takes semaphore.
		while(steps--){
			printf("Stepping CCW..\n");
			if(cnt-- < 0)cnt = 3;                  //Backwards.
			LATA = stepper_state[cnt];
			vTaskDelay(100/portTICK_RATE_MS);
		}
		xSemaphoreGive(xMutexSema);                //Gives semaphore.
	} //while(1)
}

static void vTaskPB(){
	while(1){
		vTaskDelay(10/portTICK_RATE_MS);
		if(PB_CW == 0){
			xSemaphoreGive(xBinSemaCW);
			printf("PB_CW pressed\n");
		}
		else if(PB_CCW == 0){
			xSemaphoreGive(xBinSemaCCW);
			printf("PB_CCW pressed\n");
		}
	} //while(1)
}
```

Leonardo Fusser (1946995) February 12th, 2022

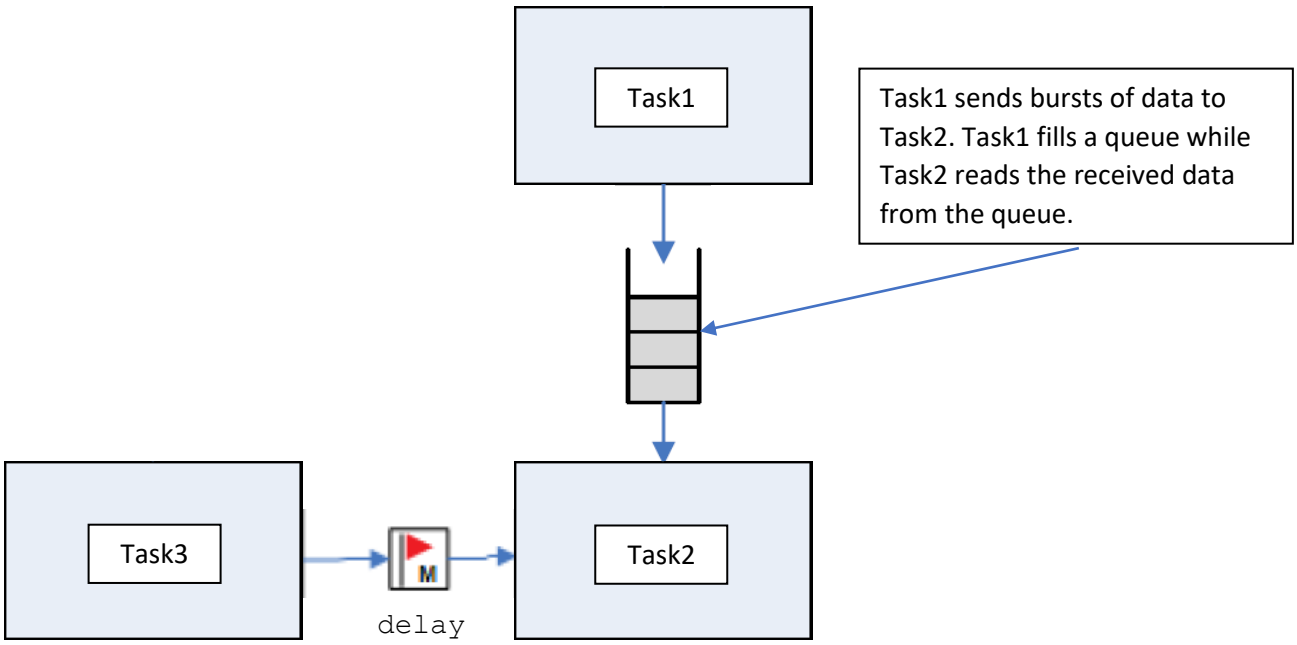
Week 4 HW

Embedded Operating Systems

Serge Hould

Recommended task structure & Encapsulation HW:

Q1)



Q2) For the previous question, the variable "`delay`" is:

- Not updated instantly. Before it can be updated, it has to wait for Task2 to receive something from Task1 (waits until queue is not empty). When Task2 finally receives data from Task1, there are further delays because of the mutexes used to safely update the variable "`delay`".

Q3) For a task that has to wait for two or more queues:

- There is a significant impact on system performance. If a task has to wait for two or more queues, it might block for too long on either one and it may also cause the system to grind to a halt as a result.

Q4) Listing 7 encapsulated into a new file called "`taskLCD.c`":

```
/* Private static queue declaration. */
static QueueHandle_t xQueueLCD = NULL;

/* Private static function declaration. */
static void vTaskLCD(void *pvParameters);

/* Private function definition. */
static void vTaskLCD(void *pvParameters){
	int pressure;
	for( ;; ){
		/* Blocking queue. */
		xQueueReceive( xQueueLCD, &pressure, portMAX_DELAY );
		sprint(buf, "Pressure: %d kPa ", pressure);
		fprintf2(C_LCD, buf);
	}
}

/* Public interface function definitions. */
void vInitTaskLCD(void){
	xQueueLCD = xQueueCreate ( 5, sizeof (int) );
}

void vStartTaskLCD(void){
	xTaskCreate( vTaskLCD, (char* ) "TLCD", 240, NULL, 2, NULL );
}

void vSendQueueLCD(int temp_tx){
	xQueueSendToBack(xQueueLCD, &temp_tx, 0);
}
```

Week 5 HW

Embedded Operating Systems

Serge Hould

Pre-emptive OS Programming HW:

Q1)

[Listing 1 modification]

```
static void LEDTask(void){
      for( ;; ){                                          //Infinite loop.
            LATD6 = LATD6^1;                              //Blinks.
            /* 500mS blocking delay. */
            vTaskDelay(DELAY_500mS/portTICK_RATE_MS);
      }//End infinite loop.
}//End task.
```

[Listing 3 modification]

```
static void serialTxTask(void){
	int cnt = 0, tx_bit, tx_byte;
	for( ;; ){   //Infinite loop.
		switch(state){
		...
		case SM_STARTB:
			TX_PIN = START_BIT;
			/* 2mS blocking delay. */
			vTaskDelay(MILLISECONDS_2/portTICK_RATE_MS);
			state = SM_SEND_BYTE;
			break;
		case SM_SEND_BYTE:
			while(cnt != 8){
				/* 2mS blocking delay. */
				vTaskDelay(MILLISECONDS_2/portTICK_RATE_MS);
				tx_bit = tx_byte >> cnt;
				tx_bit &= MASK;
				TX_PIN = tx_bit;     //Asserts the TX pin.
				cnt++;
			}
			cnt = 0;
			state = SM_STOPB;
			break;
		case SM_STOPB:
			TX_PIN = STOP_BIT;
			/* 2mS blocking delay. */
			vTaskDelay(MILLISECONDS_2/portTICK_RATE_MS);
			state = SM_WAIT;
			break;
		}//End switch state.
	}//End infinite loop.
}//End task.
```

Leonardo Fusser (1946995) March 5th, 2022

Week 7 HW

Embedded Operating Systems

Serge Hould

Counting semaphores HW:

Q1) Console output:

```
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.

Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
...
```

Q2) Console output:

```
Task3 accessing
Task2 accessing
Task1 access denied
Task3 giving up
Task2 giving up
Task1 accessing
Task3 accessing
Task2 access denied
Task1 giving up
...
```

Q3) Complete snippet of code:

```
...
/* Counting semaphore declaration. */
SemaphoreHandle_t xCountSema;


/* PB macro. */
#define PB    PORTDbits.RD7

/* Stepper motor states. */
int stepper_state[4] = {0b110, 0b1001, 0b0011, 0b0110};
...
/* Main. */
int main(void){
       ...
       xCountSema = xSemaphoreCreateCounting(6, 0);   //Init counting semaphore.
       ...
} //End of main.
...
/* vTaskPB. */
static void vTaskPB(void){
       while(1){
              if(PB == 0){                            //When PB is pressed...
                     printf("PB pressed!\n");         //Prints.
                     for(int i = 0; i < 6; i++){      //Releases 6 count semaphores.
                            xSemaphoreGive(xCountSema, 0);
                     }
              }
              vTaskDelay(10/portTICK_RATE_MS);        //Gives slack time.
       } //End of mini-infinite loop.
} //End of vTaskPB.
...
```

...continues on next page...

```
/* vTaskHandler. */
static void vTaskHandler(void){
	int cnt;                                                //Step count.
	while(1){
		xSemaphoreTake(xCountSema, portMAX_DELAY);      //Reduces count sema by 1.
		printf("Stepping CW...\n");                     //Prints.
		if(cnt >= 4)cnt = 0;                            //Forwards.
		LATA = stepper_state[cnt++];                    //Steps.
		vTaskDelay(500/portTICK_RATE_MS);               //500mS per step.
	} //End of mini-infinite loop.
} //End of vTaskHandler.
...
```

Leonardo Fusser (1946995) April 3rd, 2022

Week 11 HW

Embedded Operating Systems

Serge Hould

Intro to Real-Time Systems HW:

Q1) What is the name of a system in which the response time is guaranteed?

- A system in which the response time is guaranteed is a deterministic real-time system (a.k.a: a hard real-time system). It is a system that will respond to any events within a bounded amount of time.

Q2) Two essential characteristics of real-time systems:

- Real-time systems must produce correct computational results and that these computations that are performed must conclude within a predefined period (usually between milliseconds and microseconds). In other words, real-time systems must perform exceptionally well computationally and behaviorally.

Q3) What is the name of a system where the level of tolerance for a missed deadline is extremely small or zero tolerance?

- The system where the level of tolerance for a missed deadline is extremely small or zero tolerance is a hard real-time system.

Q4) For the given timing chart (assuming a time-base of 1mS per-division and first dotted line to the left is 1mS):

a) Response time:
   - 7mS.

b) Process time:
   - 3mS.

c) Latency time:
   - 4mS.

d) Period time:
   - 8mS.

Q5) For the given timing charts, say if the systems are deterministic:

a) Deterministic. Guaranteed response time is respected.
b) Non-deterministic. Guaranteed response time not respected.
c) Deterministic. Guaranteed response time is respected.

Q6) Classify the following systems:

System	Type of system	Delay
Industrial oven thermostat using a µController.	Hard real-time system.	Between minutes and hours.
Game like angry bird using an Android OS.	Not a real-time system.	N/A
Gas level LCD display using a µController.	Soft real-time system.	Between milliseconds and seconds.
Missile guidance system using a RTOS.	Hard real-time system.	Between microseconds and milliseconds.
Trader must transfer money between two accounts within exactly two days of the trade execution using a special PCs.	Hard real-time system.	Between hours and days.

Leonardo Fusser (1946995) April 3rd, 2022

Priority Inversion HW:

Q1) Give the reason(s) why a mutex semaphore is superior to a simple semaphore for preventing priority inversion:

- The main reason why a mutex semaphore is much better than a simple semaphore like a binary semaphore is because mutex semaphores have a smart feature called priority inheritance. These mutex semaphores are able to temporarily change the priority of a task to avoid causing major delays in the system. Due to this nature, priority inversion is nearly impossible with mutex semaphores.

Q2) Say whether the given snippets can potentially have a priority inversion issue. If there is an issue, explain how to prevent it:

- There is a priority inversion issue with one of the given snippets. The issue lies in `vTask_Med()`. The problem is that `vTask_Med()` pre-empts `vTask_Low()` for an unbound amount of time, which in turn holds up `vTask_High()`. Here, the system will be switching from `vTask_Low()` to `vTask_Med()` after `vTask_Low()` has taken the mutex. Since `vTask_High()` needs access to the mutex before it can proceed, it will be held up since `vTask_Low()` would have not released the mutex it took (CPU allowed for `vTask_Med()` to run, not `vTask_Low()`).

  The only way this problem can be resolved easily is if mutex semaphores are used instead of the binary semaphores. The mutex semaphores will prevent the priority inversion from occurring like what was explained above.

Q3) For some RTOS system, the current states of the three tasks are listed in a table. xSema is currently taken by Task C. Why isn't Task A currently running since it has a higher priority than Task B and C? How can this problem be solved or improved?

- The above problem is similar to the previous problem. The reason why Task A is not running, even though it has a higher priority than Task B and C, is because Task C took the mutex and has not released it yet. When Task C initially ran, it took the mutex but was then pre-empted by Task B. Task B runs for an unbound amount of time, which in turn holds up Task A since Task C is not able to release the mutex. A priority inversion problem is happening here (as seen in the previous problem).

  The solution to this problem is the same as in the previous problem. Replacing the binary semaphores with the mutex semaphores will ensure that priority inversion does not take place.

Q4) Find out the problem with the following two tasks if they both are set at priority 1:

- The problem with this scenario is that a one-armed deadly embrace occurs. This is because the two tasks have to wait for one another before they can do anything. Therefore, this is a typical one-armed deadly embrace problem.

Q5) Explain what a one-armed deadly embrace is:

- A one-armed deadly embrace is when two or more tasks are waiting for each other to complete processing data, or even just to produce the data before continuing on.

Leonardo Fusser (1946995) April 10th, 2022

Week 12 HW

Embedded Operating Systems

Serge Hould

Reset module & Watchdog timer HW:

Q1) Name at least 3 sources of reset:

- Some sources of reset include reset instruction (SWR), power-on reset (POR), brown-out reset (BOR), watchdog-timer reset (WDTR) and low-true assertion of the MCLR input on the µC.

Q2) Snippet of code to print "BO-reset" at boot time whenever VDD voltage drops:

```
...
void main(void){
      if(RCONbits.BOR){                  //If brown-out voltage...
            fprintf2(C_LCD, "Hello");    //Print message.
            RCONbits.BOR = 0;            //Clear flag.
      }
      while(1);
}
...
```

Q3) Enable and set the watchdog timer to 16mS:

```
#pragma config WDTPS = PS16      //Sets WDT post-scaler to 16mS.
#pragma config FWDTEN = ON       //Turns on WDT module.
...
void main (void){
      WDTCONbits.ON = 1;         //Enables WDT module.
      while(1);
}
...
```

Q4) For the given snippet of code, the following will occur once power is applied:

- The message "Howdy Y'all" will be displayed once before the WDT resets the system after 2.048 seconds. This is because the WDT post-scaler is set to `PS2048`, and the WDT never gets kicked in the main (cleared). This behavior will happen over and over forever.

Q5a) For the given snippet of code, the following will occur once power is applied:

- The message "Hi" will print only once and the message "Hello" will print many times afterwards before the WDT resets the system after 2.048 seconds. This is because the WDT post-scaler is set to `PS2048`, and the WDT never gets kicked in the main (cleared). This behavior will happen over and over forever.

Q5b) For the given snippet of code, the following will occur once a master reset (MCLR) is applied:

- The behavior is the same as from the previous question.

Q6a) For the given snippet of code, the following will occur once power is applied:

- Initially, the message "POR" will appear upon power-up before the WDT waits after 10 overflows before putting the system into a sleep state. The system constantly gets reset, but when 10 (or more) overflows do occur, the WDT is disabled, and the system enters a sleep state.

Q6b) For the given snippet of code, the following will occur once power is applied:

- Similar to the above problem, the message "POR" will appear upon power-up. The WDT still waits for 10 overflows to occur, but when this is done, a LED will start to blink in a while loop forever instead of putting the system into a sleep state.

Q7) In a pre-emptive RTOS system, the easiest way to implement a watchdog:

- In a pre-emptive RTOS system, the easiest way to implement a watchdog is by adding the feature inside of the idle hook function (kicking the dog inside the idle hook function). This is because the idle hook function should be always running in an RTOS system, and as soon as there is an issue with the system, the idle hook function will not work anymore. Therefore, a watchdog here is the most logical solution.

Q8) A few reasons to add a watchdog to a system:

- A few reasons to add a watchdog to a system might be to prevent tasks from hanging, deadlocks (one-armed deadly embraces) from occurring and to not wait for a reception that will never occur.