**CG2271 Real Time Operating Systems**

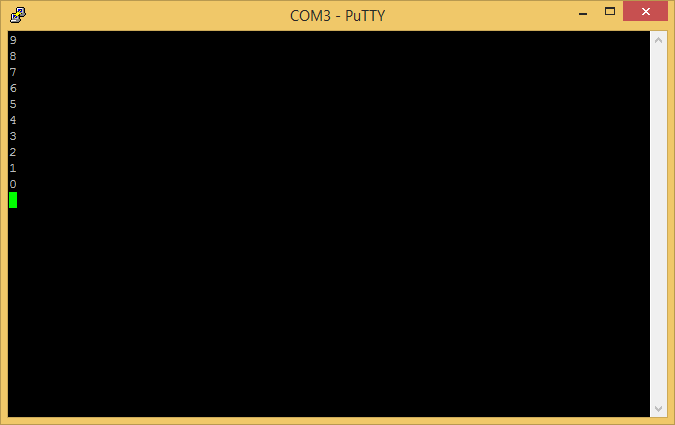
**Lab 3 - Real Time Architectures II**

**Answer Book**

|  |  |
| --- | --- |
| **Name: Ang Kah Min, Kelvin** | **Matric No: A0111890** |
| **Name: Sitti Maryam Binte Rashid Ridza** | **Matric No: A0112675** |

Question 1 (2 marks)

The following is the output of the terminal program.



The values are printed one after another in reverse from 9 to 0 with a 500ms delay between them.

Question 2 (3 marks)

The numbers are not printed in the same order they were enqueued. (They were enqueued in ascending order, while printing was in descending order.)

The enq() function takes in a priority queue created by makeQueue(), an item to add, and a priority number.

Lower number denotes higher priority. Since the value 0 is inserted with a priority of 9, with the subsequent values inserted with ascending priority (descending priority number), each new number is always enqueued at the front.

Therefore, the numbers are dequeued in reverse order (descending numbers in ascending priority).

Question 3 ( 4 marks)

The keyword (void \*) means that the function accepts a pointer to a variable of the void type.

This essentially means that it is a pointer to data of unspecified type, which is useful for simulating polymorphic behaviors.

The (void \*) type is analogical to the Object type which is used in Java to denote a universal / top-level parent type (though the underlying concepts are different).

By using the (void \*) type, we can allow the queue to work with any data type, which we can typecast appropriately when dequeuing.

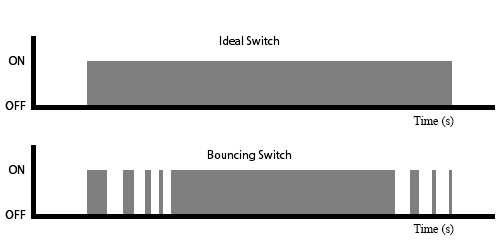
Question 4 (4 marks)

As mentioned in Question 3, the TPrioQEntry struct (used as the list node in this implementation) can point to data of any type by casting them as (void \*). However, the type is not stored with the data. Therefore, when dequeuing, we will retrieve a (void) type as well.

We need to typecast this to the appropriate data type before we can make use of them.

Question 5 (3 marks)

Switch bounding is caused by imperfections of mechanical switches. Whenever a switch is closed or opened, there can be tiny vibrations in the contacts which can result in a bouncing signal which looks like this:



This may be interpreted by the circuit as a series of ONs and OFFs instead of a single transition from ON to OFF.

In applications like turning a lamp ON or OFF, this may not be a significant problem. However, in sensitive applications, it may cause serious problems. For example, a bouncing button meant to trigger a single interrupt can end up causing the ISR to run multiple times.

Question 6 (6 marks)

The debounce function keeps track of the last time a function is called, and returns false if it is called less than 500ms ago.

Each function in the implementation has a separate long value keeping track of the last time it was called with reference to the number of milliseconds since the system has started up.

By comparing the timer variable with the current time (using the millis()) function, it can determine if sufficient time has passed for the button press to be considered legitimate.

If 500ms has passed since the last press, the timer variable is updated with the current time, and the function returns true. Otherwise, nothing is done and the timer returns false.

We can use a conditional if-then block to only perform the ISR functions when the function returns true.

Question 7 (5 marks)

We have to first create a valid queue. In the program, there is a file scope pointer called \*queue which points to the queue. This queue is initialized in the setup() function.

In order to enqueue an item, we need to use the following function:

void enq(TPrioQueue \*queue, void \*item, int priority);

For this to work, we need to specify the queue as the first object, cast the item into (void \*) and pass it as the second parameter, and specify a priority number (where lower number is higher priority).

When dequeuing, we first check if there is anything to dequeue using the qlen() function. If there is something to dequeue, we make use of the following function:

void \*deq(TPrioQueue \*queue);

This would return an item of type (void \*) which we would need to manually typecast to the correct type before using it.

Question 8 (7 marks)

(Only appropriate lines are shown)

TPrioQueue \*queue;

// Do switch debouncing

unsigned long int0time=0, int1time=0;

// Debouncing function. Returns TRUE if this interrupt was not caused by a bouncing switch

int debounce(unsigned long \*debTimer)

{

…

}

// Declares a new type called "funcptr"

typedef void (\*funcptr)(void);

// Flashes LED at pin 7 5 times a 4 Hz

void int0task()

{

…

}

// Flashes LED at pin 6 5 times at 2HZ

void int1task()

{

…

}

void int0ISR()

{

if (debounce(&int0time)) {

enq(queue,(void \*)int0task,0);

}

}

void int1ISR()

{

if (debounce(&int1time)) {

enq(queue,(void \*)int1task,1);

}

}

void setup()

{

queue=makeQueue();

…

attachInterrupt(0,int0ISR,RISING);

attachInterrupt(1,int1ISR,RISING);

}

// Dequeues and calls functions if the queue is not empty

void loop()

{

// If we still have an item to dequeue

if(qlen(queue)>0) {

funcptr fptr = (funcptr)deq(queue);

fptr();

}

}

//

int main(void)

{

init();

setup();

while(1)

{

loop();

if(serialEventRun)

serialEventRun();

}

}

Each time an interrupt is received, the debounce function is used to check if the time between the last call is at least 500ms ago. If it is, it will return true and update the timer to reflect the time it was last called. Otherwise, it will return false.

When an ISR is run, and the debounce function returns true, it enqueues the respective task into the queue using the enq(…) function with the appropriate prorities. For example, when INT0 is received, int0ISR is run (it is attached in setup()) which in turns (after checking for bouncing) enqueues int0task with a priority of 0 (highest priority). The same is done for the other interrupt, with a lower priority of 1.

In the main function, the program repeatedly runs loop() which polls the queue for the next task to run. If there is something in the queue (checked using qlen(…)), it will dequeue it (using deq(…)) and cast it as a funcptr before running it.

In this program, funcptr is a function pointer type definition which takes in parameter of (void).

Question 9 (3 marks)

The one connected to pin 7 (which is associated to INT0) flashes 5 times first, followed by the LED connected to pin 6 (associated to INT1).

When the button connected to INT0 is pressed, the ISR is triggered and int0task is enqueued into the queue. The main loop immediately dequeues it to run before we can press the button connected to INT1 (we probably will not be able to press it fast enough for the priority to matter). The same happens when the button at INT1 is pressed.

Question 10 (3 marks)

This time, the LED at pin 6 (associated with INT1) flashes five times first, followed by the LED at pin 7.

When the button connected to INT1 is pressed, the ISR is triggered and int1task is enqueued into the queue. The main loop immediately dequeues it to run before we can press the button connected to INT0. The same happens when the button at INT0 is pressed.

By the time INT0 is triggered enters the queue, int1task() is already running and will continue running even though int0task() is supposed to be of a higher priority. Since there is no pre-emption mechanism, this behaviour is to be expected.

Question 11 (5 marks)

We pressed the button at INT0 followed by the button at INT1 five times (for a total of 10 button presses). The sequence is observed:

* LED at Pin 6
* LED at Pin 6
* LED at Pin 6
* LED at Pin 6
* LED at Pin 6
* LED at Pin 7
* LED at Pin 7
* LED at Pin 7
* LED at Pin 7
* LED at Pin 7

This is because even though the buttons were pressed in an alternate fashion, the enqueuer function ensured that new int0task()’s is always queued ahead of any int1task()’s due to the higher priority of int0task()’s. New int1task()’s will also always be enqueued to the back of the queue as they have lower priority.

Question 12 (5 marks)

void fastLoop()

{

togglePin6();

}

void slowLoop()

{

if (slowLoopCounter == 4) {

togglePin7();

slowLoopCounter = 0;

} else {

slowLoopCounter++;

}

}

void loop()

{

if (millis() - fastLoopTimer > 99) {

fastLoopTimer = millis();

fastLoop();

slowLoop();

}

}

The loop() function is repeatedly called in the main() function. The loop() function first checks at least 100 milliseconds has passed since the last time fastLoop() was called. If it is indeed so, fastLoopTimer will be set to the current time, and both fastLoop() and slowLoop() will be called.

As a result, both fastLoop() and slowLoop() will be called 10 times per second (ideally), resulting in a calling frequency of 10Hz.

The fastLoop() function calls the togglePin6() function which toggles the output at pin 6. This will result in the blinking of the LED at the rate of 5Hz (which is ON and OFF five times per second).

Even though the slowLoop() is called at a rate of 10Hz, there is an internal counter which only runs togglePin7() every 5 calls. This will result in togglePin7() being called at the rate of 2Hz, which is equivalent to the blinking rate of 1Hz.