**CG2271 Real Time Operating Systems**

**Lab 4**

**Using an RTOS**

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

**Question 1 (3 marks):**

void OSInit(unsigned char numTasks) is the first function that should be present in the main() function of the program. It takes in the maximum number of user tasks expected to run as its argument, which is then incremented by 1 (to account for idle task later on) for initializing the task queue. It also sets up the timer hardware registers to initialize it.

The official description is “**Initializes ArdOS. This should be the first function called before anything else**”.

unsigned int OSCreateTask(int prio, void (\*rptr)(void \*), void \*rarg) takes in a task priority (0 being the highest), a function pointer to the task, as well as a pointer address to the arguments to be passed to the task at startup. This will cause the function to be registered as a task (it is stored in the \_tasks array of tTCB structs, where each tTCB represents a single Task Control Block), while at the same time allocating a stack for it (determined by \_taskStackSize, which is set using OSSetStackSize(unsigned char stackSize)).

The official description is “**Registers a new task. The task will have a stack size determined by OSSetStackSize (default stack size is 50 32-bit words)**”.

void OSRun() will start the 1Khz timer, create an idle task (to be used when all tasks other tasks are BLOCKING), and call void OSSwap() which causes the scheduler to pick a task to run. In this case (OSSCHED\_TYPE OS\_PRIORITY), it will pick the highest priority task.

The official description is “**Starts up the operating system, which will begin executing the highest priority task**”.

**Question 2 (3 marks):**

This part is copied from Question 1:

void OSInit(unsigned char numTasks) is the first function that should be present in the main() function of the program. It takes in the maximum number of user tasks expected to run as its argument, which is then incremented by 1 (to account for idle task later on) for initializing the task queue. It also sets up the timer hardware registers to initialize it.

unsigned int OSCreateTask(int prio, void (\*rptr)(void \*), void \*rarg) takes in a task priority (0 being the highest), a function pointer to the task, as well as a pointer address to the arguments to be passed to the task at startup. This will cause the function to be registered as a task (it is stored in the \_tasks array of tTCB structs, where each tTCB represents a single Task Control Block), while at the same time allocating a stack for it (determined by \_taskStackSize, which is set using OSSetStackSize(unsigned char stackSize)).

**Question 3 (3 marks):**

In simpler architectures, such as (a) Round Robin, (b) Priority Function Queue or (c) Timed Loops, the loop() method is repeatedly called to (a) run tasks, (b) dequeue and run tasks or (c) call the timed loop containing the tasks. In this program, when should the tasks be run is handled by ArdOS’s scheduler so loop() does not need to be implemented.

**Question 4 (3 marks):**

Writing tasks which are not enclosed in an infinite loop resulted in the following observations:

1. When the higher priority task is not enclosed in an infinite loop, the lower priority task runs till it sleeps for the first time, and it stays on indefinitely.
2. When the lower priority task is not enclosed in an infinite loop, system behaviour is as expected.
3. When both tasks are not enclosed in an infinite loop, the same problem mentioned in the first point occurs.

The exact explanation for this behaviour would require debugging in order to reveal the contents of the PC, SP, function queues, task statuses and so on. I will attempt to answer this question from a higher-level point of view.

It is observed that not enclosing the task body in an infinite loop causes the task function to return (exit), resulting in unpredictable behaviour. According to N. Merlot, a running task should never be allowed to return and the only safe way for tasks to stop is for the scheduler to terminate the task or for the task to request for its own termination (using a Destroy scheduler call).

<http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf>

A FreeRTOS implementation manual also states that real-time tasks are typically continuous processes and therefore should be enclosed in infinite loops. Even though infinite loops are typically avoided in most programming situations, it is acceptable in this case as tasks are automatically switched whenever a multitasking opportunity arises. Examples of such opportunities include:

1. When a task goes to sleep.
2. When a task is waiting for an event or input.
3. When a higher priority task becomes ready, pre-empting the currently running task.
4. When the operating scheduler interrupts and forces a context switch after a process has executed longer than its allocated time quanta (Time Division Multiplexing, which is not implemented in ArDOS).

<http://www.freertos.org/implementation/a00007.html>

**Question 5 (5 marks):**

void OSSleep(unsigned long millis) takes in the number of milliseconds to sleep as an argument. It will sleep for the specified time before being put back into the priority queue. It may not sleep for exactly the specified time because a higher priority task may be picked as the ISR for timer uses OSPrioSwapFromISR().

The void OSSleep(unsigned long millis) function relies on two important components. The first component is \_sleepTime[\_taskNumber] which is an integer array of remaining sleep time for each task (measured in milliseconds). The second is the \_sleepFlag integer variable which keeps a flag for each task denoting if it is sleeping (1) or not (0). It is a 16-bit variable (Arduino UNO integers are 16-bit values) where the LSB represents task 0 and the MSB represents task 15 (for up to 16 tasks, which is an implicit limit in this implementation).

When void OSSleep(unsigned long millis) is first called, the specified sleep time is first set for the running task in the \_sleepTime[\_running] array. Next, the respective bit in the \_sleepFlag is set to denote that the current task is sleeping. Finally, the corresponding entry in the \_tasks array (which represents the tasks’s TCB) has its status set to \_OS\_BLOCKED. Finally, OSSwap() is called to switch to the next available (READY) task in the queue. The \_running variable is used to track the task number of the running task, which is in turn used to determine the correct array index and bits to set. It is noted that the sleep time for the task is set to millis-1, which is explained in the next section.

The 1KHz timer will interrupt once every millisecond, which is handled by ISR(TIMER2\_OVF\_vect, ISR\_NAKED). The ISR will loop through every task and decrement its corresponding sleep time by 1 if it is non-zero (implying that the task is sleeping).

For tasks that have zero sleep time, it will check the sleep flag for that particular task. If the sleep flag is currently still set for the task, it means that the task just woke up, so the corresponding sleep flag bit is cleared, the task is unblocked (by setting its status in its TCB), and it is enqueued into the function queue. Finally, OSPrioSwapFromISR() is called to switch to the highest priority task in the queue (which may not be the task that just woke up).

In summary, each sleeping task will be blocked and swapped out. It will then have its sleep time decremented millis-1 times (once per millisecond due to the timer interrupts), and then it is put into the ready queue on the next millisecond, resulting in a total of millis milliseconds.

**Question 6 (4 marks):**

void OSTakeSema(TOSSema \*sema) behaves the same way as the generic semaphore operations Pend, Down, Wait, Acquire, Procure or Prolaag.

void OSGiveSema(TOSSema \*sema) is the ArdOS implementation of semaphore operations Post, Up, Signal, Release, Vacate or Vrigave.

A semaphore variable is actually a special struct which includes an integer value and a queue.

When the Give operation is used on the semaphore, it looks in the queue of tasks waiting for that semaphore and picks one to put into the ready queue. The scheduler is then called to swap to the highest priority task. If nobody is currently waiting for the semaphore, the integer variable is incremented by 1.

When the Take operation is used, it first checks for the integer variable. If it is 1 or more, it is decremented by 1. If the semaphore is zero, the running task is added to the semaphore’s queue, put into BLOCKING state and the scheduler is called to swap it out.

The general purpose of semaphores is to act as a flag. If the semaphore is zero then the task cannot proceed (and is put into BLOCKING state). The advantage of semaphores over normal flags is that it can keep track of how many times the semaphore is incremented (if it is not a binary semaphore), which can solve deadlock issues relating to multiple sets to the same flag being discarded (and regarded as a single set) which is observed in the Producer/Consumer problem.

**Question 7 (10 marks):**

OSSema pin6sema, pin7sema;

void flashPin6(void \*param) {

while (1) {

OSTakeSema(&pin6sema);

for (int i=0; i<5; i++) {

digitalWrite(6,HIGH);

OSSleep(250);

digitalWrite(6,LOW);

OSSleep(250);

}

}

}

void flashPin7(void \*param) {

while (1) {

OSTakeSema(&pin7sema);

for (int i=0; i<5; i++) {

digitalWrite(7,HIGH);

OSSleep(250);

digitalWrite(7,LOW);

OSSleep(250);

}

}

}

void int0isr() {

if (debounce(&int0time)) {

OSGiveSema(&pin6sema);

}

}

void int1isr() {

if (debounce(&int1time)) {

OSGiveSema(&pin7sema);

}

}

void setup()

{

OSInit(2);

attachInterrupt(0, int0isr, FALLING);

attachInterrupt(1, int1isr, FALLING);

pinMode(6,OUTPUT);

pinMode(7,OUTPUT);

OSCreateSema(&pin6sema, 0, 0);

OSCreateSema(&pin7sema, 0, 0);

OSCreateTask(0,flashPin6,NULL);

OSCreateTask(1,flashPin7,NULL);

OSRun();

}

The two main tasks in this system are flashPin6 and flashPin7, which are created using OSCreateTask(…). The semaphores used by the tasks, pin6sema and pin7sema, are initialized with OSCreateSema(…). The two tasks repeatedly wait/block on their respective semaphores using OSTakeSema(…) and blinks their corresponding LEDs whenever the semaphore is released using OSGiveSema(…).

The pushbuttons on pins 2 and 3 are then attached to their respective ISRs (with debouncing applied as per previous labs). Pushing the pushbuttons will release their corresponding semaphore, causing tasks waiting/blocking on that semaphore to be executed. These semaphores are initialized to be non-binary, which allows them to count the number of times the semaphores are released, simulating a queue-like effect.

Note that leaving the debouncing duration at the default 500ms actually negates any benefits to having non-binary semaphores as the processes will be waiting for the semaphores at the same rate the semaphores can be released. This program actually behaves the same whether the semaphore is binary or not, so we can actually use binary semaphores here as well with little impact. However, binary ones are used to more closely model this solution to the previous lab where there is a queue.

**Question 8 (4 marks):**

When the button at INT0 is pressed, the LED at pin 6 immediately lights up. After two blinks, pressing the button at INT1 actually causes the LED at pin 7 to immediately light up and blink concurrently with the LED at pin 6. When the LED at pin 6 stops flashing, the LED at pin 7 blinks for another two counts before stopping as well.

Because we are using an RTOS, we can now perform some degree of multitasking. Even though the tasks have different priorities here, it is only in play when both tasks are simultaneously placed in the ready queue. In this case, whenever an action is performed (i.e. turning the GPIO ON or OFF), the task is immediately placed into blocking using OSSleep(…), providing opportunities for multitasking.

For example, immediately after the LED at pin 6 turns off, flashPin6 sleeps for 250ms. If INT1 is pressed during this period, flashPin7 can immediately execute because it is the only process in the ready queue. Immediately after the LED at pin 7 is turned on, flashPin7 sleeps for 250ms, which allows flashPin6 to turn the LED at pin 6 on before going to sleep again. The process repeats until the tasks need to wait for their semaphores again.

Question 9 (4 marks):

The same is observed here as with Question 8, with the roles of INT0 and 1, LED 6 and 7 as well as flashPin6 7 interchanged.

**Question 10 (6 marks):**

The two programs are similar as they both respond to user input through the use of interrupts. However, the way the tasks are executed is different.

In the function queue scheduling version, the ISRs simply add their corresponding function pointers into the priority queue, while the tasks are executed when they are dequeued in the main function.

On the other hand, the ISRs in the RTOS version signals the semaphores, causing blocked tasks to be added into the ready queue. The kernel’s scheduler then decides the when the task should be run.

Since the delay(…) function is a form of CPU busy-waiting, there is no opportunity for multitasking in the function queue scheduling version (other than if an interrupt has occurred). The RTOS version allows multitasking as it uses OSSleep(…) which puts the task into BLOCKING, allowing other tasks to be run.

**Total Marks: / 45**