THE ANSWERS

A: Analysis, B: Bookwork, C: Computed Example, D: Design

1. An ARM Cortex processor uses a periodic timer interrupt of group priority 2 with ISR *pollcount* to poll a single digital input that is driven from an external pulse source. The pulse high and low times are guaranteed to be no less than $1/(2f_p)$, where f_p is the maximum allowed pulse frequency. You may assume that the non-interrupt code running on the processor never disables interrupts and never changes the CPU PRIOMASK register. You may also assume that the timer period T_2 is such as to make CPU utilisation from *pollcount* much smaller than 100%.

   a) Explain the function of group-priority and sub-priority bit fields, and threaded and handler modes, when nested interrupts occur in the Cortex architecture.

   B.
   group priorities are MS bits (3 bits in case of LM4F232). Differing priorities will preempt and allow nesting of interrupts. Differing subpriorities (LS bits) with the same group priority will not preempt each other and therefore not nest, but will prioritise interrupt order.
   When interrupts occur the first interrupt changes the CPU mode to handler mode from threaded mode, subsequent nested interrupts keep handler mode until the last interrupt to exit returns processing to threaded mode. The two modes have different stacks.

   [2]

   b) Ignoring interrupt response, state, giving reasons, the relationship between the period of the timer T_2 and f_p that ensures all pulses are counted by software using this method.

   A.
   In order for a pulse to be observed both low and high parts of the pulse must be sampled at least once. The condition for this is $T_2 \leq f_p/2$.

   [2]

   c) Explain in what manner *late arrival* and *tail chaining* optimisations in the Cortex NVIC can alter the interrupt response of a given interrupt of group priority 2 in the presence of one or more other interrupts.

   A.
   Tail chaining will speed up two consecutive interrupts by jumping directly from one to the next avoiding entry and exit time.
   Late arrival will speed up the response time of an interrupt whenever a lower priority interrupt occurs just before it.

   [4]

   d) Taking interrupt response into account, calculate when and how these optimisations would change interrupt jitter and hence the maximum allowed value of f_p for fixed T_2.

   A.
   The minimum time between samples is T_2- jitter and this determines $1/2f_p$.
   Tail chaining will decrease jitter by the saved time only in the case that there is at least one interrupt of greater priority than *pollcount*. Late arrival will decrease minimum response time, and therefore increase jitter, only in the case that there is at least one interrupt of lower priority than pollcount.

   [4]

   e) Now consider the *pollcount* interrupt running under FreeRTOS with possible changes to `PRIOMASK` in the RTOS. Describe the function of:

   configKERNEL_INTERRUPT_PRIORITY
   configMAX_SYSCALL_INTERRUPT_PRIORITY

How do these two numbers affect *pollcount* jitter and correct operation of the system in the case that the *pollcount* ISR does not contain any RTOS API calls. How would your answer change if the ISR did contain API calls?

B.
configKERNEL_INTERRUPT_PRIORITY is the priority at which the kernel clock tick runs. If <= 2 clock ticks will increase maximum interrupt response of *pollcount* and therefore jitter. If >= 2 tail chaining will decrease minimum interrupt response of *pollcount*. configMAX_SYSCALL_PRIORITY is the priority level at which RTOS critical sections execute from tasks. if <= 2 again max interrupt response is increased, by an amount equal to the maximum length critical section, which itself depends on what RTOS API calls are made. If > 2 there is no direct affect on interrupt response, however also in this case, if *pollcount* contains API calls, there will be no protection for critical sections and therefore correct operation of the system is not guaranteed.

[4]

f) Discuss whether selection of the FreeRTOS tick period and T_2 could be used to reduce *pollcount* jitter, and if possible what would be the likely benefit.

D.

- By choosing tick period synchronous with T_2 (a multiple or sub-multiple) and starting the T_2 counter with the right offset the two interrupts could be separated avoiding any jitter increase.
- This would also reduce deadlines for the two interrupts. Since both are low CPU utilisation this is probably not a problem.
- It is easy to ensure that the priority of *pollcount* is higher than anything else in the RTOS if there are no API calls in *pollcount*. This case separating the two interrupts gives benefit equal to the time saved by the tail chaining optimisation.
- If there are API calls then correctness requires that *pollcount* is delayed by critical sections in the RTOS code. Those will likely be shorter than the clock tick, hence there is still some benefit.

1 mark for each point or equivalent.

[4]

Question 1. has a total of 20 marks.

Answers

2. a) Figure 2.1 shows a subroutine `swap` which implements an atomic swap operation using the new `ldrex` and `strex` instructions.

i) Explain how this code works, giving an execution trace of its simultaneous overlapping execution on two different CPUs.

A.
the ldrex/strex if not interrupted by another overlapping ldrex/strex will correctly perform the swap.
if two such overlap the first strex will succeed and implement a swap. The second strex will fail, causing a branch back to swap and a repeat ldrex/strex which will now succeed.

(1) ldrex
(2) ldrex
(1) strex ; succeeds
(2) strex ; fails
(2) cmp
(2) bne
(2) ldrex
(2) strex ; succeeds

[2]

ii) In the case of overlapping calls to `swap` from different CPUs can the system stay in livelock? Justify your answer.

A.
When multiple CPUs are in the swap busy-wait loop, one strex must succeed every loop iteration, thus reducing the number of waiters. Hence livelock is not possible.

[2]

iii) Give two advantages of `ldrex/strex` over an atomic swap machine instruction such as `swp` when implementing synchronisation in an RTOS running on multiple CPUs.

B.
The memory bus is not locked during the atomic operation, which allows better use of the memory
A more general read modify write can be implemented, for example with a decrement operation for a semaphore, not just a swap.

[2]

b) Figure 2.2 shows the CPU length and period of a set of 101 tasks implementing jobs and running on a single CPU core in an RTOS application. Tasks have priority equal to their number, where higher priorities are larger numbers, and have pre-emptive priority scheduling. There is no blocking other than that caused by the use of a semaphore. The critical section length is shown where the semaphore is used. Each task using the semaphore has one critical section of the specified length.

i) Which tasks suffer blocking due to the semaphore?

C.
1-100 use the semaphore. Therefore tasks from 2-100 can suffer blocking from a lower priority task critical section.

[2]

ii) Determine the maximum blocking for each task. Explain this result.

C.
For task 1 there is no blocking.
For task 2 there is one critical section of task 1.
For tasks 3-100 this is extended by priority inversion. The total blocking for task n is 1 critical section, plus $(12-4)(n-2)$ time units.

[4]

iii) State what extended rate monotonic analysis says about the schedulability of this system both with and without the use of Priority Inheritance Protocol in the semaphore.

C.
The utilisation here even without blocking is 100*0.6%+20%=80% - well above the RMA limit for 101 tasks which is $\log_e 2 = 69\%$. Therefore the system is not guaranteed schedulable although this may still be possible. With blocking from the semaphore the extra utilisation under extended RMA increases, much more for the case without PIP than with, but this does not alter the answer.

[2]

iv) Determine the schedulability of this system using the maximum completion time method.

C.
Task 101 is much faster than the others and can be included by multiplying other CPU utilisation by 1/0.8. It is highest priority and always schedulable since it cannot be blocked. The other tasks can occupy the CPU continuously (including Task 1 as above) for a time of at most 100*12/0.8 = 1500. This is less than the shortest deadline for these tasks (2000) and therefore all tasks are schedulable. Note that the semaphore blocking has no affect on this argument.

[3]

v) Discuss the merits of Priority Threshold Scheduling and Priority Inheritance Protocol in this system.

A.
PIP has no use here since it only affects schedulability which is fine without PIP. In fact PIP does not significantly alter overall schedulability since the MCT of every task1-100 will be either delayed or blocked by every other such task, the overall result is the same.
PTS would allow the 100 tasks to run with fewer task switches.
Also (potentially, if PTS is implemented as co-routines) it would reduce stack use of these tasks by a factor of 100.

[3]

Question 2. has a total of 20 marks.

```
; atomic swap of r0 with mem[r1]
swap  ldrex  r2,[r1]
      strex  r3, r0, [r1]
      cmp    r3, #0
      bne    swap            ; branch if r3 != 0
      mov    r0, r2          ; r0 := r2
      bx     lr              ; return
```

Figure 2.1: Implementation of atomic swap using `ldrex/strex`.

Task	CPU length	Period	Semaphore critical section length
101	1	5	n/a
n where $1 \leq n \leq 100$	12	$2100 - n$	4

Figure 2.2: 101 Tasks. Note that CPU length includes critical section.

Answers