1)

a)

First statement: 0x015a8000 is more than 8-bit and hence **can't be fit in the immediate filed of the instruction**. The processor will generate a LDR instruction with PC relative addressing where the immediate value is stored in a literal pool (offset away from the current PC value – current instruction). **This instruction will read the value from literal pool and load it into the register**.

Second statement: the immediate value 0x05700000 **can be right rotated to become 0x57 which is within 8-bit**. Thus, the processor will execute the statement as a MOV instruction with rotation of the immediate value.

Third statement: 0x1a8 = 110101000_2. Thus, **when right rotated it can be constructed by an 8 bit immediate value**. Therefore, like statement 2, MOV instruction with right rotation will be executed by the processor.

b)

Cortex M supports 2 operating modes: **Handler and Thread mode**. The processor is in Thread mode when it is executing Background tasks as well as upon reset. The handler mode is activated when the processor is executing Foreground tasks such as exception handling. During both Thread and Handler mode, the processor might be running at **privileged access level** which allow unrestricted access to all system resources and instruction set. In contrast, **user access level** which restrict system resources are only available during Thread mode.

On reset, the processor will be operating in Thread mode at privileged level. This is for the various initialisation of the processor before stable operation can be achieved. By default, Main Stack is used by the processor

Once the initialisation is completed, the main program can now be fetching and executed by the processor. This is done in Thread mode at user access level (to protect from buggy user program)

If exception occurs, processor will switch from thread mode to handler mode and raise to privileged access level for exception handling. The Main stack is also used (need to switch if previously the processor is using Process Stack in Thread mode).

*Note that processor may use **Main Stack or Process Stac**k in Thread mode and this depends on program to program.*

Once the exception handler finishes its execution, context is returned and the process now return to thread mode, running at same privilege level and using the same stack as before exception occurs.

Refer to the figure below for a clearer flow.

c)

Tail chaining and Late Arrival are 2 mechanisms implemented for the efficient handling of exceptions by Cortex M processors.

Tail chaining is to speed up the interrupt servicing of one interrupt after another by skipping the overhead of state saving and restoration between the consecutive interrupts. If another interrupt is pending after the completion of an ISR, **the stack pop** (context restoration to before entry of ISR) and **hence push is skipped**. This is unnecessary context restore and saving. At the same time, the processor will **immediately fetches the vector of the next pending interrupt** (of highest priority). This reduce the waiting time to 6 cycles before the next ISR can be executed.

Late arrival is to ensure efficient handling of high priority interrupt by using pre-empting. If a higher interrupt/ exception occurs during the state saving of another lower priority interrupt, the **existing saving of context will continue**. However, the vector of the higher priority interrupt will be started, saving the need to discard, restore and re-save the context of processor again. Once finished with ISR of higher priority interrupt, tail chaining will be applied for the remaining lower priority pending interrupt.

2)

a)

By **AAPCS (Arm Architecture Procedure Call Standard)**, the first four arguments n1, n2, c1, n3 will be passed into the function using R0-R3 registers. The last argument n4 will be passed using the stack.

Function passing register will be more efficient than the stack. Also returning function multiple values may sometimes be needed. Thus it is advice to use pointers for more efficient passing of arguments.

b)

i) *Design problems will have multiple answers, this is a suggested answers based on the author's assumption. During exam, just assume and write down because you have no time to think everything through.*

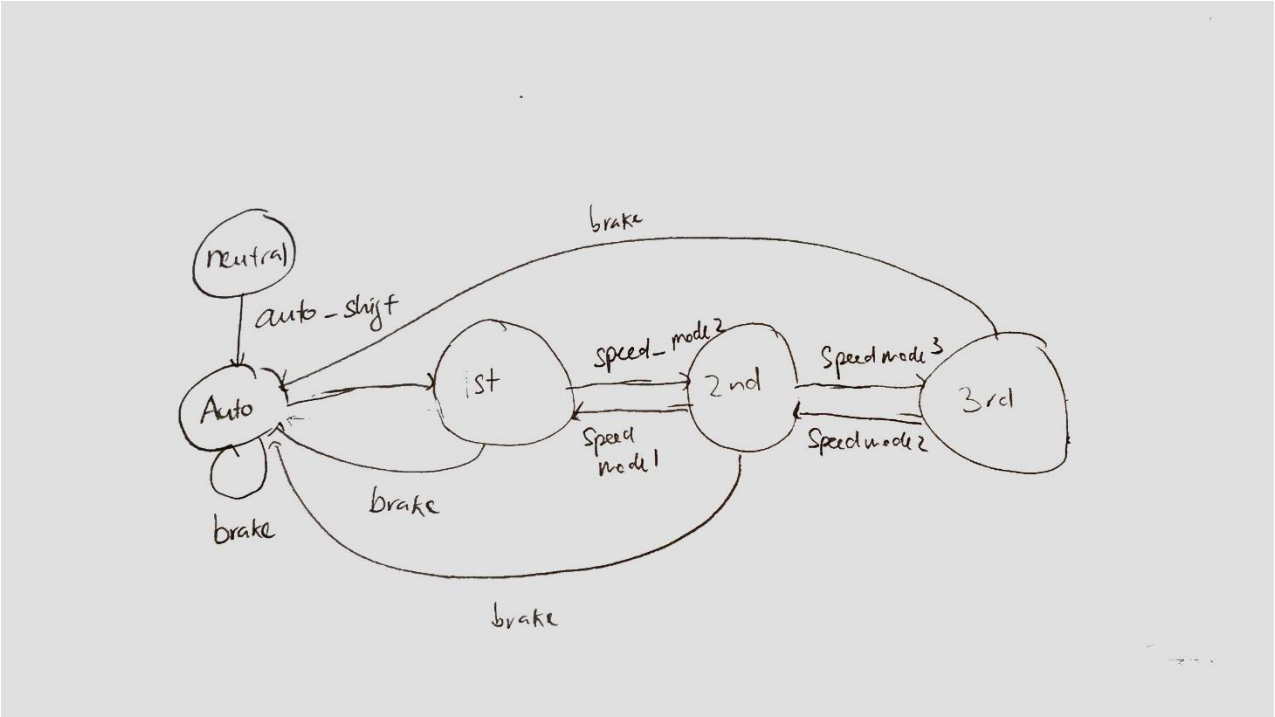

Figure 1 FSM for the gear shift

Speed_mode2 is the event where the speed is from 15 to less than 40. Speed_mode3 is the event where the speed is above 40. Auto_shift is the event of user's shifting the gear from neutral to automatic and brake is the event when break is applied. Assumption is that break will go to a complete stop of the vehicle (when in 1st 2nd or 3rd state) => same effect as applying break during Auto state.

ii)

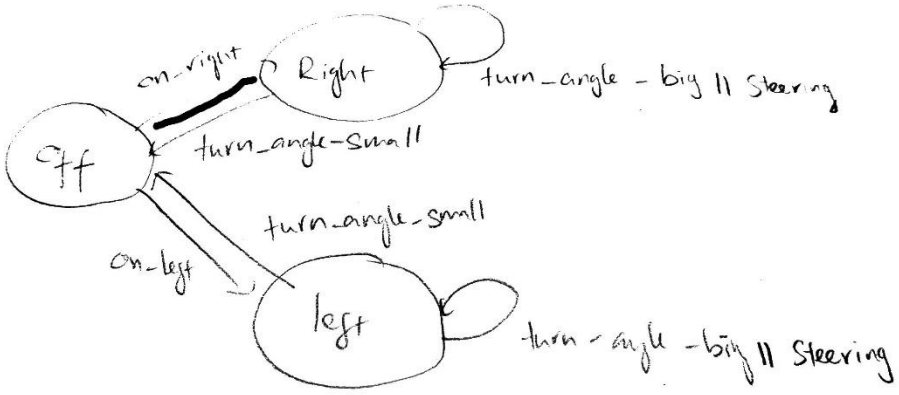

Figure 2 FSM for turning signal

Once again, multiple answer to do this.

Off is the state of the turning signal when it is off, Right and Left are the respective state of the turn signal (turning left or right). Lastly turn_angle_big and turn_angle_small are the events/flags of the wheel steering angle (more than and less than 5 degree respectively). Steering is the event where the driver engage the steering wheel actively. Once finished the turn, the wheel will not be engaged (by my assumption). Thus the light should only be off if steering wheel is not engaged and the turn angle of the wheel is less than 5 degree.

iii)

Note: Implementing the FSM can be done with RTOS using the multitasking and scheduling capabilities of the OS. However, this is out of the scope for our current syllabus. Here we assume a bare-metal approach to designing the FSM. That means there is no RTOS running on the MCU board for now.

It also worth noting that there are multiple ways to implement the FSM still. We may do it with function pointer (which nobody knows unless you are very very interested in the embedded industry). And switch statement is another way to do it. We shall stick with the switch as it is easier and more practical during stressful exam.

Our assumption: No RTOS support, running on bare-metal system. The main loop of our embedded program will consist of two switch cases representing the separating FSM system. These two systems can be think as tasks, being executing by multitasking (sharing time with each other).

No context saving happens (as not a full RTOS) and also inter-task communication (if any) shall be done over global variables

Main Loop (1):

```
void main(){
      stateGear = Neutral;
      stateSignal = Off;
      while(1){
            gear_task();
            signal_task();
      }
}
```

Declaration of variables (2):

```
Int Neutral, Auto, First, Second, Third;
Int speed_mode1, speed_mode2, speed_mode3, break, autoshift;
```

```
Int stateGear, eventGear;

Int Off, Right, Left;

Int on_right, on_left, steering, turn_angle_big,
turn_angle_small;
```

Declaration of Tasks (*Some state and events are omitted for time interest in exam)*(3):

```
Void gear_task(){
      Switch(stateGear){
            Case neutral:
                  If (eventGear == autoshift)
                        stateGear = Auto;
                  break;
            ...
            }
}
Void signal_task(){
      Switch(stateSignal){
            Case Off:
                  If (eventSignal == OnLeft)
                        stateGear = Left;
                  else if (eventSignal == OnRight)
                        stateGear = Right;
                  break;
            ...
            }
}
```

Once done with defining the components, we can put all into a C file with the order (2), (3) and (1)

3)

a)

The three considerations are **certification, portability and scalability**.

Certification: Commercial OS is likely to be certified by mandatory and safety standards in targeted application areas. Thus, opting for a commercial OS might help the product to get easier approval as the OS has already complied with the standards (saving times and efforts too).

Portability is the availability of the selected OS in different processor platform. Commercial OS with higher portability can allow software to be port from one platform to another easily, reducing the team time for development and testing in extending their products to another hardware platform.

Scalability: Ease of extending software to accommodate larger/higher system requirement. A commercial OS supports more features that can be easily enabled thus help with the roadmap and scaling of product line of the team.

b)

CPU util = 100% - % time in idle tasks. The idle task is when the CPU has the least load (because it is the first task being created by the OS) and it is used to calculate the baseline of the CPU when it is less used. This baseline can then be used to estimate (in relative to itself) the utilisation of the CPU per other task (with load).

More info at: https://www.embedded.com/how-to-calculate-cpu-utilization/

Also read the manual for micro-C OS, regarding launcher task and OSStart() API call.

c)

Note: Deferred Posting is deprecated in the current release of Micro C OS. Also, look for bolded as main idea.

Upon finish servicing an interrupt, the ISR has 2 methods of posting to the relevant tasks

Direct posting method involves the **ISR posting semaphores, message or flags** (synchronisation components) **directly to the tasks**. As these **components may be posted my many ISRs, we need to put them in an critical sections and protected by disabling interrupt before posting them.** The task being posted will be ready for execution and may or may not pre-empted (continue executing the interrupt task or maybe pre-empted by a higher priority task).

Deferred posting methods involves an **interrupt queue where ISRs adds their synchronisation components to the queue when they need posting of these components to communicate with the interrupted tasks.** Upon completing the ISR, the interrupt queue handler task will always be invoked, extract the post commands from the queue (with disabling of interrupt). **The queue handler will then** re-enable interrupt and lock scheduler (to prevent any pre-emption during posting and to protect critical sections/ synchro components) and **begin the sequence of posting**. Once done, scheduler is unlocked to determine the next task to be performed (after the queue handler)

Deferred Posting has lower interrupt disable period (only during the unloading of the interrupt queue), but higher task latency. Thus it is useful for **more responsive system** (faster response to interrupts needed)

d)

Multiple answers possible. Here is one

Refer to the diagram below:

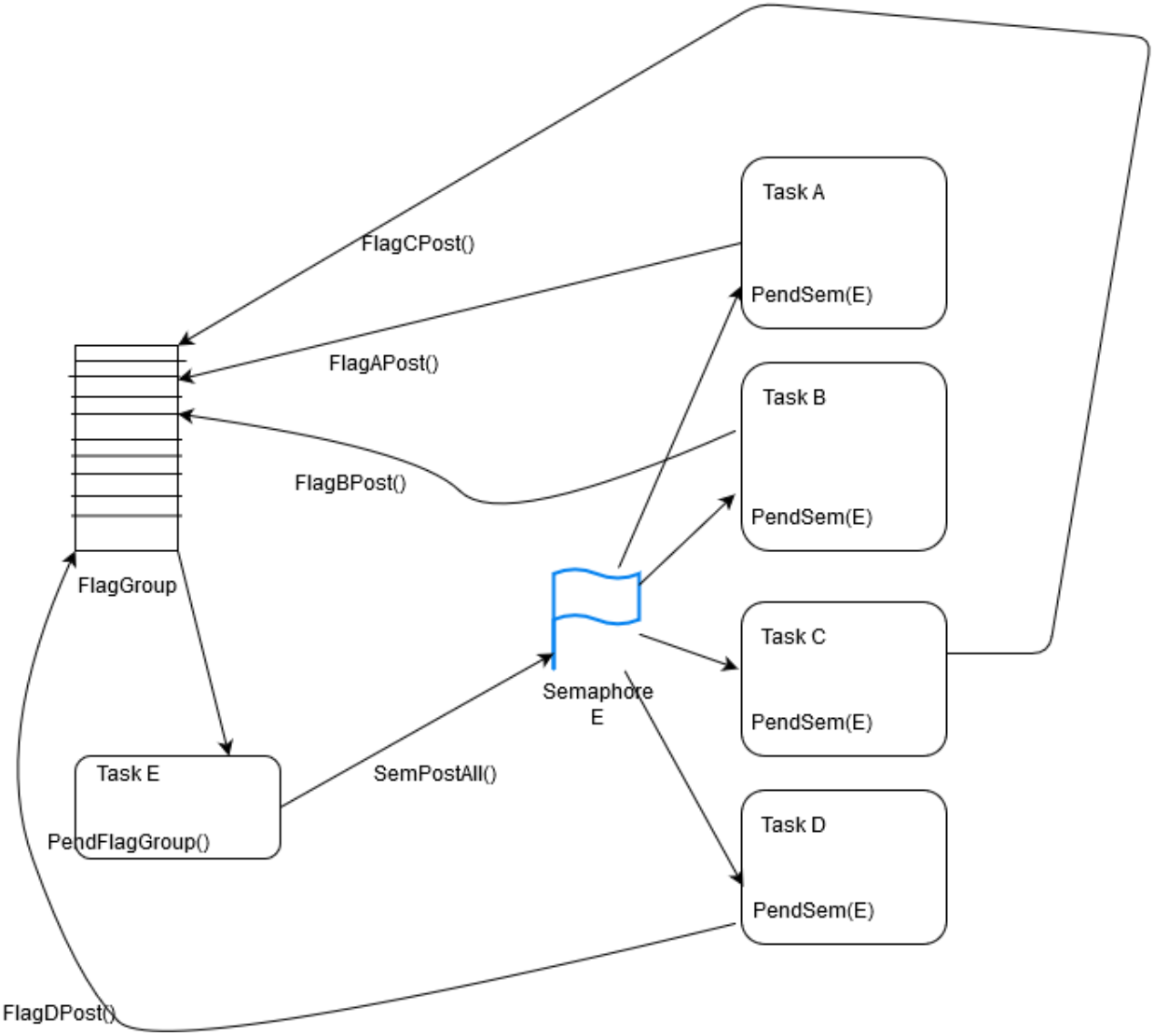

Figure 3. Architecture proposed

Task A,B,C,D,E represents the modules Mod-A,B,C,D,E respectively.

Task E can only start when Task A,B,C,D are done producing. Thus once each of those task finish their execution, they will post a flag (A,B,C,D) in the flag group structure. This flag group is in

turn pended by the Task E. Thus only once all the flags are posted, then the Task E can continue execution (producing the chemical).

Once finish producing the composing ingredients, Task A-D must wait for the production in E to finish before producing the next batch of ingredients. Therefore, they must pend for semaphore E immediately after posting their respective flags. Once finished producing, Task E will do a SemPostAll() call (or posting to all the semaphores pending to its semaphore). Thus all Task A, B, C and D can now produce the next batch.

We need a flag structure and a semaphore structure for this solution.

4)

a)

Use of mutex does not incur an unbounded priority inversion as it has mechanism of priority inheritance. This effectively raises the priority of the lower owner task to the same level as the requestor task (which is higher than the medium priority that is causing unbounded priority in most cases).

Mutex takes longer to process (due to the priority management mechanism). It also is binary which is not ideal for resource management (unlike counting semaphores with multiple instances).

b)

Disabling interrupts makes the task with the critical section to be unresponsive to any interrupts. Thus it is simple and should only be used in short critical section.

Disabling scheduler will also disable pre-emption in scheduling. This means a higher priority task cannot pre-empts the task with critical section. However, interrupt can still happen.

Since disabling scheduler will still yield interrupt to happen during critical sections. Sharing between ISR and task of a common resource need to use disable interrupt (so interrupt and hence ISR cannot modify critical section during the modification by the task)

c)

Convention from LT note: T(Period T, Execution time C, Deadline D)

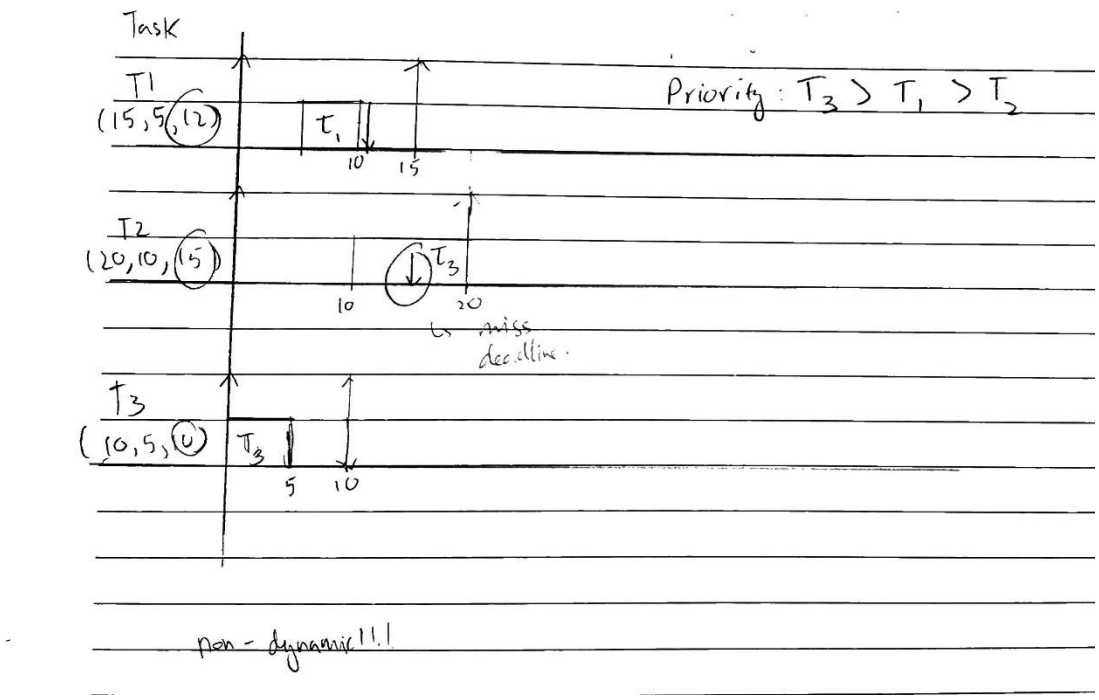

Figure 4 Scheduling Diagram

The deadline monotonic algorithm we used is non-dynamic. That means, the priority of the Tasks based on their deadline is determined at the beginning and will not change through out the course of execution (no pre-emption and change of priority in between). This is for simple study of the algorithm.

Task 3 > Task 1 > Task 2. Task 3 and Task 1 may be completed within the deadline stipulated (shown as downwards arrow in the diagram). Upward arrow is the release of new tasks into the system.

As we can see, the task T2 misses it deadline with the current scheduling scheme. Hence real time requirement of the system is not satisfied.

d)

Only when period and deadline of all tasks are equal. Yes, this is feasible in real life due to the periodic nature of embedded system tasks (fixed sampling rate system). Thus it is natural and easier to use Rate monotonic (because both might be the same).

--End of Answers--

Solver: Do Anh Tu