1)

a)

Cortex A used for modern smartphones. Cortex R used for airbag controller in a modern car. Cortex M used for a washing machine.

b)

In **Privileged access level**, the processor has **unrestricted access to all system resources** and can use all supported instructions, which is required for performing **low level initialization** (*after reset of the MCU)* of the processor and servicing exceptions (*need access to all hardware resources)*.

In **User access level**, the processor is **restricted from accessing certain system resources**, like the configuration registers, to **protect against user program bugs** and misbehaving code from hanging the system.

c)

Since the processor is in Handler mode, it is handling foreground tasks like exceptions while using the main stack.

Exceptions arise when normal flow of a program needs to be halted temporarily due to an event that requires immediate attention of the processor, which is serviced in the Handler Mode.

d)

The main purpose of defining the **stack size** in a startup file is to **detect and prevent stack overflow** (done by the MCU). Since the **stack is stored contiguously in memory which decrements towards the heap memory**, a large enough stack will overwrite and invalidate data in the heap memory. *(This is very catastrophic for embedded systems where a corruption in memory space can cause troubles for the application it is used in)*

Similar to the stack, the main purpose of defining the heap size is to detect and prevent the heap from overwriting and invalidating data in the stack, as **data is incremented towards it during run-time.**

The exception vector table consists of the reset vector which contains the address of the first instruction it will execute upon startup or reset, hence it is required in a startup file. It also **contains the vector** (address) to every single **exception handler and ISR for the MCUs**. Thus it is needed for servicing interrupts and exceptions as the MCU need to know where to fetch from memory to handle the exceptions.

The instruction format determines each instruction's size, hence the processor needs to know whether 16-bit Thumb instructions or 32-bit ARM instruction is used at startup. ARM Cortex M3 are capable of running at 16 bit or 32 bit instruction format (trading complexity and code sizes in each case). This will also helps with deciding the increment in Program Counter of the MCU.

2)

a)

i)

I would set the Priority Mask (PRIMASK) bit to 1. Refer to the ARM Cortex M3/M4F user manual. [Note that it can only store a 1 bit value]

ii)

Not feasible: I1 < I2 < I3 < I4 in term of priority (4>3>2>1). Thus if we are using Base Mask register, we would not be able to keep I1 active when I4, I3 and I2 are disabled.
[*Disclaimer: Please check exam question again (from NTU Library) as this depends a lot on which interrupt need to be disabled*]

ii)

Priority values of interrupts and exceptions are important in nested interrupt systems, as it determines **which interrupt would be serviced first when multiple interrupts are triggered simultaneously**. As such, higher priority interrupts will almost always be serviced first before lower priority interrupts. It also **determines the urgency of the task to the MCU**. This is important as the real time system must be responsive to some events over others. Thus less latency is wasted on servicing not so important interrupts

For instance, if all 4 interrupts (listed in the question) arrive simultaneously, the highest priority interrupt I4 will be serviced first, followed by the next highest priority interrupt I3 and so on.

The order in which the 4 interrupts will complete are as follows: I4, I3, I2, I1.

b)

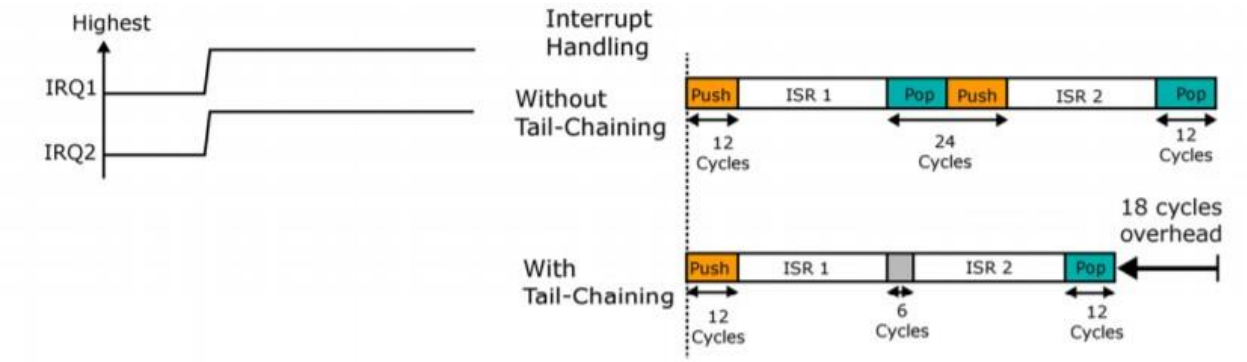

Figure 1. Tail-chaining Mechanism

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.

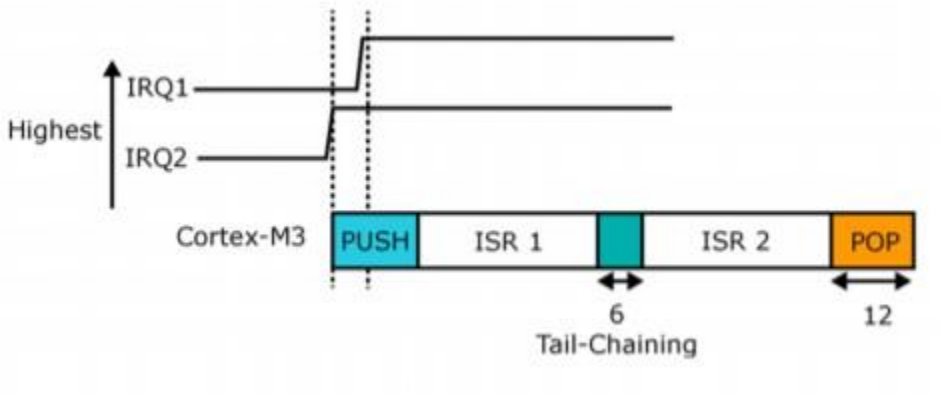

Figure 2. Late Arrival Mechanism

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.

c)

i) Using only 16-bit instructions.

ii) Using only 32-bit instructions.
iii) Using a mix of 16-bit and 32-bit instructions.

3)

a)

Using code optimization techniques such as **loop unrolling**, which reduces the number of loops by increasing the code size, and **inlining**, which moves a small and frequently used function directly into its caller, will **increase the code size but improves an application's performance by removing or reducing the overhead of branching.**

Other code optimization techniques such as **variable assignment** and **loop counting**, which converts a countup loop to a countdown loop, would **reduce code size and improves an application's performance by reducing the total number of instructions it needs to fetch**, hence reducing overall execution time.

Some optimization techniques such as the use of cache and register may cause variables modified by ISRs to be unchanged, thus causing unexpected behavior in the system.

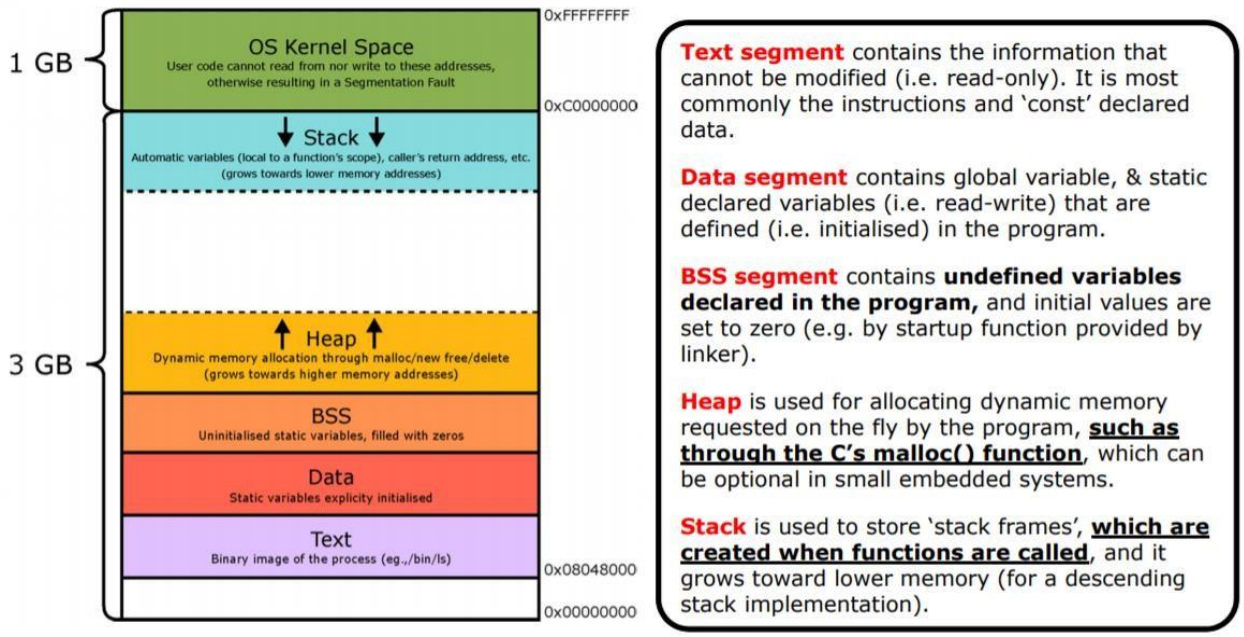

Figure 3. Memory space diagram (ignore OS Kernel Space)

Integers **a, b, i and sum** are local variables and hence, stored in the stack.

f[] is a local array and hence also stored in the stack.

The 3 parameters **x,y and z** will be passed to the registers when the function is called. [Note: There are fewer than 4 parameters, hence stack not needed, by **AAPCS**]

Static integer **temp** will be stored in the BSS Segment when it is uninitialized and will be stored in the Data Segment once a value is explicitly initialized.

```
Note:  static int temp = 0;           //temp will be stored
in BSS Segment temp =  temp + ((x * y)-z);       //temp
will be stored in Data Segment
```

c)

Multiple answers are possible, just need to assume and draw correctly.

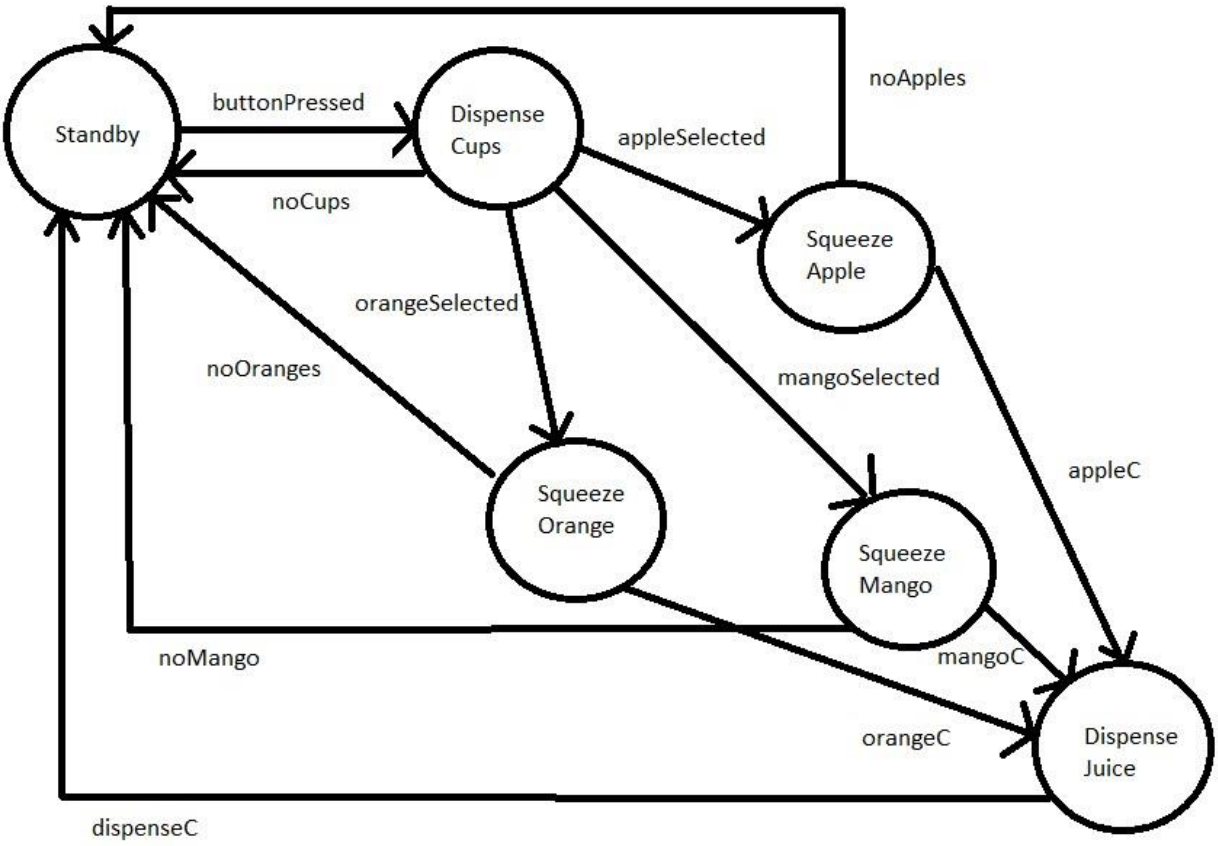

Figure 4 FSM for Tarzan

Assumptions: Only 3 buttons to select desired fruit. Payment method is irrelevant in this example (can simulate the done of payment by buttonPressed event). How the system handle the out of stock problem is it will go back to stanby mode and ask user to repeat again the process (since we don't have payment here, it would be sound enough)

4)

a)

Real-time operating systems (RTOS) are required for time-critical applications, where behavior of the **system must be reliable and deterministic in execution and meeting time deadlines**. It is especially important in hard real time application like automobile in this case as the **real time deterministic characteristic is highly needed for humans' safety.** Lacking a RTOS may result in fatal error of the system which may cause serious results to the user's or application in use. Thus RTOS is needed for the MCUs of the air-bags as it will decide if the driver's is alive or not when crashed.

b)

When a task wants to access its critical section (*where resources/variables are shared with other tasks or ISRs)*, it will first acquire a mutex before executing its critical section. Any other task that tries to acquire the same mutex will be blocked and put on a pending queue until the owner of the mutex exits the critical section and releases the mutex. The highest priority task in the pending queue will then be able to acquire the mutex (when it is post/unlocked) to execute its critical section. As such, since only one task may own a mutex at any given task, it protects a critical section from being accessed by multiple tasks.

c)

Assumption of non-dynamic priority-based scheduling, meaning there will be no change in priority during runtime.

Shortest Deadline First: Priority – T1 > T2 > T3

Figure 5. Shortest Deadline First Scheduling

⇨ No Deadline missed

Longest Deadline First: Priority – T3 > T2 > T1

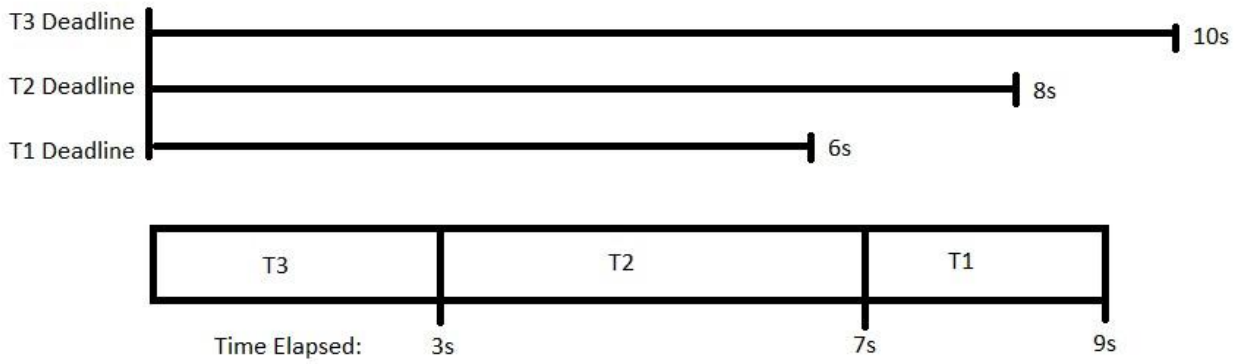

Figure 6. Longest Deadline First Scheduling

⇨ T1's deadline is missed.

d)

To develop this application on MicroC/OS III RTOS:

I would first use **OSInit()** for initialization of the MicroC/OS III on the target architecture as different embedded hardware require slightly different initialisation.

Next, I would use **OSStart()** to start multitasking under MicroC/OS III. Also, this will create a launcher task that we need to have to measure the baseline utilisation of the system. Thus from there we can measure accurately the utilisation of our CPU so we may adjusting our scheduling or multitasking of the 3 tasks accordingly so as not to overload the system.

Finally, I would use **OSTaskCreate()** prior to the start of the multitasking to create 3 tasks and assign the names "P1", "P2" and "P3" as well as to set their respective priority levels of 1, 2, and 3 respectively. This once again is important as we can only afford to have 1 launcher task in the beginning (like discussed above). This will keep the initial flow to be deterministic and fast as we filter out unnecessary overhead from these user tasks.

--End of Answers--

Solver: Wee Soon Lee Aaron