Possible answers for the Embedded Software (TDDI11) exam from 2020-08-19.

The max total in the exam is 35. Thresholds have been updated accordingly.

Assume D1 = 3 and D2 = 1.

1. Question 1:
   a. UARTs transmit data serially. The transmitter checks the UART is ready to transmit, then writes the data (here D2=1=0b00000001) to the transmit buffer of the UART. The transmitting UART then sends a start bit over the transmit pin (tx) and then sends one bit at a time, for instance 1 then seven 0s. It will then signal the host it is ready to transmit more. The receiving UART monitor its rx pin. When it sees the start bit it starts sampling at predetermined intervals and reconstruct the serially transmitted bits into the byte 0b00000001. It then signals its host data is ready to be fetched from its shift register.
   b. A system with hard real time guarantees is correct if none of the deadlines is missed. This would for instance be relevant for ECUs controlling the breaks of a car: having a low average response time should not come at the price of sometimes missing the deadlines. Soft-real time guarantees allow deadlines to be sometimes missed. The focus is on the "quality of service" as in the average response time. This would be relevant for a media decoder which might miss some deadlines but where the average response time can be adopted as a metric for correctness.
   c. The D1:th (here 3:rd) most significant bit is the bit number 3 starting from the most significant bit (bit 0). (D1+2)th is the bit number 5. Solutions starting from the other end are also accepted. In terms of shifting: the 3:rd most significant bit is the 28^{th} bit (starting from 0 for the least significant bit) and the 5^{th} most significant bit is the 26^{th} bit (also starting from 0 for the least significant bit).

```
int myCheck(int x){
    int thirdMSB = ((x & (1 << 28)) >> 28);
    int fifthMSB = ((x & (1 << 26)) >> 26);
    return  thirdMSB == fifthMSB;
}
```

   d. These are two approaches to managing large projects with several teams. The "Over-the-wall" approach divides the work among several teams: e.g., marketing, design, manufacturing, etc. Each team is responsible for "its part". It has to wait for the output of the previous team in order to produce its part and deliver it to the next team. This sequentialization can lead to waiting queues, to poor information sharing among the teams, it encourages each team to feel responsible for its part, but it does not encourage responsibilities for the whole project. Concurrent engineering instead encourages having cross-functional teams to facilitate information sharing and to detect potential problems as early as possible. This also comes with an integrated project management taking responsibility for the whole project, not just part of it.
   e. Memory mapped display:
      i. Address of byte at line 3 and column 2 (counting from 0) is: 0xB2000 + 3*10 + 2 = 0xB2000 + 0x1e + 0x2 = 0xB2020
      ii. **addressOf:**

```
char* addressOf(int row, int col){
    char* display=(char*)0xB2000;
    return display + row*10 + col;
```

}

2. **Question 2**
   a. Foo at line 17 corresponds to Interrupt based communication. This is hinted by the usage of STI and IRET. More importantely, there is no loop here: a test is performed at line 23. Depending on the result of the test, a method is called (line 34) or foo is exited. The bar method at line 45 instead keeps performing the test (like 48) until the outcomes allows it to process the data. This corresponds to polling until the data is ready to be processed.
   b. Polling is based on a simple loop that keeps testing a condition (e.g., whether data is ready) until it can perform the action (e.g., process the data). This requires simple controllers without requiring them to be able to handle interrupts. Controllers that handle interrupts can save controller cycles by not wasting them on checking a signal from an I/O that is typically much slower than the controller.
   c. Such conventions do not apply here: the interrupt might occur at anytime and so all modified registers need to be saved and restored. Otherwise, the task that was performed would witness a "non-deterministic" change in the values of its registers. This is not acceptable.

3. Question 3

Task1: C1= (D1%3)+1=1 and T1=3xC1=3
Task2: C2=C1+1=2 and T2=3xC2=6
Task3: C3=C2+1=3 and T3=3xC3=9

a. Utilization = 1/3+2/6+3/9=1.
b. Task 1 because it has the highest rate.
c. A diagram for preemptive RMS could look like:
   0: task-1
   1: task-2
   2: task-2
   3: task-1
   4: task-3
   5: task-3
   6: task-1
   7: task-2
   8: task-2
   9: task-1 <- task-3 misses its deadline

   Preemptive RMS cannot schedule the tasks.
d. A diagram for premptive EDF could look likle (in case of equality, we pick from the queue. This is not optimal for locality and for context-switch over-head):
   0: task-1
   1: task-2
   2: task-2
   3: task-1
   4: task-3
   5: task-3
   6: task-1

7: task-3
8: task-2
9: task-1
10: task-2
11: task-3
12: task-1
13: task-2
14: task-2
15: task-1
16: task-3
17: task-3
18: repeat from 0

Yes. The tasks can be scheduled with preemptive EDF.

4. Question 4

d=(D2 % 3) + 3 = 4.

State:	in: 0	in: 1
s0 (initial)	out: 0 / goto: s0	out: 0 / goto s1
s1	out: 0 / goto: s0	out: 0 / goto s2
s2	out: 0 / goto: s0	out: 0 / goto s3
s3	out: 0 / goto: s0	out: 0 / goto s4
s4	out: 1 / goto: s0	out: 0 / goto s1

5. Question 5
   a. That is the return address, i.e, the address in the caller method to which to jump after the callee or the current method returns. This is saved on top of the stack so that the callee can itself call other methods (possibly recursively). After such called methods return, it is important to find the address of the caller.
   b. ebp is used to set up the frame of the current method. It is a copy of the top of stack at the beginning of the method. This allows the method to conveniently access its local variables (also stored on the stack) and the arguments that were passed to it even if the top of stack changes during the execution of the method.
   c. The integer: MSB: 0x3 0x3 0x4 0x1: LSB. In big Endian, we start (i.e. smallest address) with the biggest end. Hence:
      Byte 0x3fffffec (smallest address) contains value 0x3 (most significant byte)
      Byte 0x3fffffed contains value 0x3
      Byte 0x3fffffee contains value 0x4
      Byte 0x3fffffef contains value 0x1.