

Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

exam – **Embedded Software** – TI2726-B
January 29, 2016 13.30 - 15.00

This exam (6 pages) consists of 60 True/False questions.
Your score will be computed as: $\max(0, \frac{\#correct}{60} - \frac{1}{2}) \times 2 \times 9 + 1$
It is **not** allowed to consult the book, handouts, or any other notes.

Instructions for filling in the answer sheet:

- You may use a **pencil** (erasures are allowed) or a **pen** (blue or black, **no** red, **no** strike outs).
 - Fill in the boxes **completely**.
 - Answer **all** questions; there is no penalty for guessing.
 - Do not forget to fill in your **Name** and **Student Number**, and to **sign** the form.
-

The following abbreviations are assumed to be known:

- FQS (Function Queue Scheduling)
- ISR (Interrupt Service Routine)
- RR (Round Robin)
- RRI (Round Robin with Interrupts)
- RTOS (Real-Time Operating System)

One system clock tick = 10 ms (unless stated otherwise).

We make use of the following definitions:

```
void delay(int ms) {
    !! do some CPU computation to the number of ms milliseconds
}

void putchar(char c) {
    while (!! UART tx buffer not empty)
        ;

    !! send c to UART tx buffer
}

void puts(char *s) {
    !! write string s using putchar
}
```

1. Embedded programming is more difficult than “classical” programming because of the lack of support for recursion. **false**
2. A defining characteristic of embedded systems is the need for predictable timing behavior. **true**
3. The **Embedded software crisis** refers to the “year 2000” bug. **false**
4. Finite State Machines can be coded in VHDL.
- An advantage of doing so is that it results in a fast and predictable process executing on dedicated hardware. **true**
5. Despite advances in software engineering practices, as a rule of thumb, embedded software contains 1-10 bugs per thousand lines of code. **true**
6. The Underground Tank Monitoring System is a classic example of an embedded system in that it involves input (sensors/buttons), output (display/printer) and real-time constraints. **true**
7. Because embedded software engages the physical world, it has to embrace time and other non-functional properties, which requires a view that is significantly different from the prevailing abstractions in computation. **true**
8. An interrupt is a synchronous signal from hardware to indicate the need for processor attention. **false**
9. The C language is centered around the `int` data type that represents the canonical machine word.
- As such the size of an `int` is architecture dependent. **true**
10.

```
typedef void *(* resolve)(void *old, void *new);
```


The definition above declares `resolve` as a pointer to a function that takes two arguments of type `void *` and returns a void pointer as result. **true**
11. GDB is programming tool that aids memory debugging by executing a program in a safe environment. **false**
12. Finite State Machines can be coded in a number of ways in C.
- In the table-based solution, every state is encoded as a separate function. **false**
13.

```
int main(void)
{
    int c;
    statefp state = start;
    while((c = getchar()) != EOF) {
        state = (statefp) (*state)(c);
    }
    return 0;
}
```


The above loop drives the FSM until all characters from the standard input have been processed. **true**
14. Specifying the type of `statefp` is difficult in C because it is recursive and types cannot be referenced before being fully defined. **true**

- 15. Using interrupts improves task response time. **false**
- 16. An interrupt service routine should save the context upon entrance. **true**
- 17. A low-priority ISR can be interrupted by a high-priority task. **false**
- 18. Since disabling interrupts increases interrupt latency, several alternative methods have been developed for dealing with shared data. The Alternating Buffers method is suited for handing data from an ISR to a task.

```

static int tempA[2], tempB[2];
static bool useA = TRUE;

void interrupt readTemp() {
    if (useA) {
        tempA[0]= ...;
        tempA[1]= ...;
    } else {
        tempB[0]= ...;
        tempB[1]= ...;
    }
    useA = !useA;
}

void main(void) {
    while (TRUE) {
        if (useA)
            if (tempB[0]!=tempB[1]) ... ;
        else
            if (tempA[0]!=tempA[1]) ... ;
    }
}

```

The above code with toggling the `useA` flag in the ISR is correct. **false**

- 19. An interrupt vector contains the address of an ISR. **true**
- 20. Disabling interrupts guarantees atomicity of the code until the interrupts are enabled again. **true**
- 21. **Priority inversion** occurs when the `volatile` and `static` keywords are wrongly used inside a task or interrupt. **false**
- 22. The worst-case latency for servicing an interrupt is a combination of factors, including the time taken for higher priority tasks. **false**
- 23. The number of interrupts is limited by the number of GPIO pins on the processor. **false**
- 24. Mutual exclusive access can also be accomplished by disabling interrupts, which has the advantage of faster context switching compared to using RTOS primitives like semaphores and mutexes. **false**
- 25. The shared-data problem can be solved through using the `volatile` keyword. **false**

26. Given the following pseudo code, which reads the current values of 4 different buttons and acts accordingly. The 4 buttons are all mapped to bits 0..3 of the button register. The buttons are already debounced.

```

void f1(void) { delay(1000); }
void f2(void) { delay(2000); }
void f3(void) { delay(3000); }
void f4(void) { delay(4000); }

void main (void) {
    while (1) {
        if (buttons & 0x01) f1();
        if (buttons & 0x02 ) f2();
        if (buttons & 0x04 ) f3();
        if (buttons & 0x08 ) f4();
        delay(1000);
    }
}

```

- This code is an example of an RRI architecture. **false**
27. When the system is in an arbitrary state, button #1 must be pressed at most 10 seconds to activate f1(). **true**
28. The **primary** shortcoming of an RRI architecture is that all tasks have the same priority. **true**
29. An RTOS architecture supports priority-based task scheduling. **true**
30. An RR architecture is robust to code changes. **false/true**
31. The response time to an external event in an FQS architecture is deterministic and depends solely on the length of the ISR. **false**
32. Consider an alarm system that constantly monitors the digital output of several motion detector sensors in a house. If a breach is detected then an intermittent alarm sound is triggered.
- That alarm system can be implemented with an RR architecture. **true**
33. A FQS architecture reduces to an RRI architecture when tasks are serviced in LIFO order. **false**
34. In an RTOS, tasks can be in state BLOCKED, READY or RUNNING.
- a task starts in the state READY. **true**
35. A function can be made reentrant by temporarily disabling interrupts, but then it may no longer be called by an ISR. **true**
36. A task can signal an ISR by operating a semaphore. **false**
37. A reentrant function may use variables only in an atomic way. **false**
38. An ISR may call neither the OS_post () routine, nor the OS_pend () routine. **true/false**
39. A semaphore used for guaranteeing mutual exclusive access to shared resources must be initialized to the number of tasks involved. **false**

40. Even a local variable can introduce a shared data problem when its address escapes the defining function, for example, by storing the address in a global datastructure. **true**

41. A program running on an RTOS may create tasks dynamically at runtime.
- the program ends as soon as the last task terminates. **false**

42. Given is the following RTOS (pseudo) code with priority $T1 > T2$.

```
void T1(void) {
    while (1) {
        OS_Pend(sem1); // event #1 may unblock any time
        f(1);
    }
}

void T2(void) {
    while (1) {
        OS_Pend(sem2); // event #2 may unblock any time
        f(-1);
    }
}

void f(int i) {
    delay(10); // do some computation
    counter = counter + i ; // modify some global counter
    printf("%d\n", counter) ; // print result
}
```

This code suffers from a data sharing problem. **true**

43. If count is set to 15 when event 2 occurs, and event 1 follows 3 ms later, then the first value printed is 14. **false**

44. If the call to delay is replaced with OSTimeDly the output will be different. **true**

45. An RTOS usually provides two types of delay functions: polling-based and timer-based.
- polling-based delays are more efficient as other tasks can run while the caller is waiting for the specified time to pass. **false**

46. The **heartbeat timer** is a single hardware timer an RTOS is using as base for all timings. **true**

47. Assume that one system clock tick = 10 ms.
- Calling the function OSTimeDly(6) causes a delay between 60 and 70 ms. **false**

48. To address the shared-data problem, many RTOSs provide communication primitives like queues, mailboxes, and pipes.
- the basic read/write operations on these primitives are atomic. **true**

49. As the RTOSs is aware of which task is using which semaphore, deadlock can be prevented by delaying the OS_Pend operation of the last runnable task. **false**

50. Tasks in an RTOS are often structured as state machines with states stored in private variables and messages in their queues acting as events. **true**

51. The memory footprint of a program grows linearly with the number of tasks. **true**

52. Printing from an ISR is to be avoided when the RTOS does not provide a reentrant primitive to do so. **true/false**
53. A key principle of RTOS-based design is that the separation of concerns, by splitting code amongst several tasks, improves the overall throughput. **false**
54. Time slicing between tasks of equal priority is to be avoided as the response time of individual tasks is comprised. **true**
55. It is recommended to use just the minimum necessary functionality from an RTOS. **true**
56. A logic analyzer is preferred to an in-circuit emulator because it is easier to use. **true/false**
57. When developing code for an embedded system, the software can be structured into HW-dependent and HW-independent code.
- Doing so makes debugging HW-independent code feasible on the target platform **false**
58. Debugging through scripting test scenarios is difficult when the target platform is unavailable. **false**
59. When debugging code for a distributed sensor network, collecting the (debug) output of the nodes can be arranged in different ways [Beutel:2009].
- A **wireless** testbed requires physical instrumentation (i.e. wiring) of the sensor node. **true**
60. A large study of outdoor sensor-network deployments has shown that the most underestimated problem has been the water-proof packaging of the base station. **false**