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

exam – **Embedded Software** – TI2726-B
January 29, 2018 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:

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

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. true/false

2. A defining characteristic of embedded systems is the restricted, or complete lack, of a user interface. true/false

3. Several models of computation for embedded systems are described in [Lee:2002].
- The ROS software (used in the practicals) is a prime example of the publish-and-subscribe model. true/false

4. The Underground Tank Monitoring System is a somewhat contrived example of an embedded system as it involves input (sensors/buttons) and output (display/printer), but lacks real-time constraints and resource limitations. true/false

5. Despite advances in software engineering practices, as a rule of thumb, embedded software contains 1-10 bugs per thousand lines of code. true/false

6. Hardware interrupts can be disabled; software interrupts cannot. true/false

7. An embedded program can be coded as a finite state machine; the number of incoming transitions (arcs) into a state S must equal the number of outgoing transitions (arcs). true/false

8. Finite State Machines can be coded in a number of ways in C.
- In the table-based solution, every transition (arc) is encoded as a separate function. true/false

9.

```
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 the end state is reached. true/false

10. Unlike recursive data structures, recursive function types cannot be properly defined in C and require kludges like `void` pointers and type casts. true/false

11. The C language does not contain a built-in type to represent booleans.
- in control flow statements, expressions evaluating to 0 are regarded as logically False. true/false

12.

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

The first pair of parenthesis in the definition above is for clarity (stressing a function pointer is involved) and can be left out without changing the meaning. true/false

13. Valgrind is programming tool that provides controlled execution, as well as post mortem inspection of an executable. true/false

14. The worst-case latency for servicing an interrupt is a combination of factors, including the time taken for higher priority interrupts. true/false

15.

```
static int iSeconds, iMinutes;
void interrupt vUpdateTime(void)
{
   ++iSeconds;
   if (iSeconds>=60) {
      iSeconds=0;
      ++iMinutes;
   }
}
long lSeconds(void)
{
   disable();
   return (iMinutes*60+iSeconds);
   enable();
}
```

Despite disabling interrupts the above pseudo code fails to solve the shared-data problem. true/false

16. An interrupt vector table contains the addresses of the interrupt service routines. true/false

17. An interrupt can **not** be serviced faster than the time needed to save the context of code running on the processor. true/false

18. Critical sections can be guarded by disabling and enabling interrupts.
- interrupts arriving during such a critical section are buffered and handled upon exit. true/false

19. 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);
      OSTimeDly(1);
   }
}

void T2(void) {
   while (1) {
      OS_Pend(sem2); // event #2 may unblock any time
      f(-1);
      OSTimeDly(3);
   }
}

void f(int i) {
   OS_Pend(mutex);
   counter = counter + i ; // modify some global counter
   OS_Post(mutex);
}
```

This code suffers from a data sharing problem. true/false

20. If the order of events is 1, 2, 1, 2, 1 and they occur within 10 ms from each other, then the final value of the counter will be increased by 1. true/false

21. The function `f()` is reentrant true/false

22. The shared-data problem can be solved through enabling interrupts. true/false

23. A **deadly embrace** requires a minimum of 3 tasks of different priority and 1 semaphore to occur. true/false

24. When a processor is powered up, interrupts are disabled until further notice. true/false

25. While interrupts are disabled atomicity is guaranteed even when calling a non-reentrant funcion. true/false

26. Shared variables marked `volatile` guarantee atomic access. true/false

27. Using interrupts improves system response time. true/false

28. The **primary** shortcoming of an RRI architecture is that it is more complex than RR. true/false

29. An RTOS architecture supports priority-based ISRs. true/false

30. With an FQS architecture, the worst response time of a task includes the time taken by the longest task in the system. true/false

31. With an RTOS every task needs its own stack. true/false

32. An RR architecture is most robust to code changes. true/false

33. With an RTOS it is impossible to make direct use of harware timers. true/false

34. In an RTOS, tasks can be in state BLOCKED, READY or RUNNING.
- a task starts in the state RUNNING. true/false

35. An ISR may change a task's status from BLOCKED to READY. true/false

36. A high-priority task must **not** invoke an RTOS function that may block. true/false

37. When using an RTOS signaling between ISRs and tasks must be done by calling appropriate RTOS primitives. true/false

38. A program running on an RTOS may create tasks dynamically at runtime.
- the number of tasks is limited by the number of priority levels supported. true/false

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

40. Assume that one system clock tick = 10 ms.
- Calling the function `OSTimeDly(6)` causes a delay between 50 and 70 ms. true/false

41. To address the shared-data problem, many RTOSs provide communication primitives like queues, mailboxes, and pipes.
- a common advantage is that they allow pointers to be passed from one task to another. true/false

42. A disadvantage of queues over pipes is that messages/items are handled strictly in FIFO order. true/false

43. With the X32 RTOS creating a task amounts to initializing a stack and invoking a context switch to the task's main function.
- This approach provides the possibility to use one stack for multiple (concurrent) tasks and reduce the memory footprint. true/false

44. The **heartbeat timer** is a single hardware timer an RTOS is using to monitor the liveness of the task set involved. true/false

45. Consider the following code fragment:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

extern char *UART_rx_buf;        // copied from <uart.h> for reference
extern char *UART_tx_buf;
extern char *UART_ier;

#define LEN 80
static char *next_command = NULL;

void rx_ready() {
    static char buffer[2][LEN];
    static int toggle = 0;
    static char *command = buffer[0];
    static int cnt = 0;

    char c = *UART_rx_buf;
    if (c == '\n') {
        command[cnt] = '\0';
        next_command = command;
        toggle = 1 - toggle;
        command = buffer[toggle];
        cnt = 0;
    } else {
        command[cnt++] = c;
    }
}

int main() {
    *UART_ier |= 0x3;            // start RX and TX please
    while (1) {
        if (next_command != NULL) {
            if (strcmp(next_command, "exit") == 0) {
                exit(0);
            } else if (strcmp(next_command, "hello") == 0) {
                printf("world\n");
            }
            next_command = NULL;
        }
        ...
    }
}
```

This code is an example of an RRI architecture. true/false

46. Consider lines 5-7 in which some of a UART's registers are declared. This way a UART, or any other peripheral for that matter, can be accessed with normal read/write instructions.
- this mode of operation is called 'memory-mapped I/O'. true/false

47. The function `rx_ready()` uses a technique called 'alternating buffers'.
- From line 13 we can infer that the buffers are allocated on the call stack. true/false

48. The code suffers from a (subtle) data sharing bug as both `rx_ready()` and `main()` write to the same global variable `next_command`.
- in certain cases `rx_ready()` will overwrite buffered data still to be read by `main()`. true/false

49. Removing the write statement on line 39 will not resolve the shared data bug.
- it will cause `main()` to repeat the same command until `rx_ready()` is invoked again. true/false

50. An alternative approach would be to make use of semaphores to support `rx_ready()` passing the next command to `main()`.
- only a single semaphore initialized to 0 is needed. true/false

---

51. Time slicing between tasks of equal priority is to be avoided as it compromises the predictability of their response times. true/false

52. The minimal memory footprint of a program grows linearly with the number of tasks. true/false

53. A semaphore S used by tasks A and B must be initialized by either A or B. true/false

54. An advantage of using tasks is that it allows for better data encapsulation. true/false

55. Tasks should have different priorities to avoid fairness issues imposed by the RTOS. true/false

56. When developing code for an embedded system, the software can de structured into HW-dependent and HW-independent code.
- Doing so makes debugging HW-independent code feasible on the target platform true/false

57. A logic analyzer is preferred to an in-circuit emulator because it can be used with any type of processor. true/false

58. Although the assert macro is a useful debugging aid during program development, it can only be used on the host. true/false

59. A large study of outdoor sensor-network deployments [Beutel:2009] has shown that the water-proof packaging of the base station is key to establishing a reliable connection to the back bone. true/false

60. When debugging code for a distributed sensor network, collecting the (debug) output of the nodes can be arranged in different ways.
- A **wireless** testbed requires **no** physical instrumentation (i.e. wiring) of the sensor nodes. true/false