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

\begin{tabular}{l}
Instructions for filling in the answer sheet: \\
- You may use a \textbf{pencil} (erasures are allowed) or a \textbf{pen} (blue or black, \textbf{no red}, \textbf{no strike outs}). \\
- Fill in the boxes \textbf{completely}. \\
- Answer \textbf{all} questions; there is no penalty for guessing. \\
- Do not forget to fill in your \textbf{Name} and \textbf{Student Number}, and to \textbf{sign} the form. \\
\end{tabular}

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:

```c
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. A defining characteristic of embedded systems is the need for large volumes of scale.  
   true/false

2. 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/false

3. 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/false

4. Embedded programming is more difficult than “classical” programming because of the event-based programming model.  
   true/false

5. Interrupts cannot only be generated by hardware, but also by software.  
   - A software interrupt is a synchronous signal to indicate the need for a change in the execution flow.  
   true/false

6. An embedded program can be coded as a finite state machine.  
   - When for every state S the number of incoming transitions (arcs) equals the number of outgoing transitions (arcs), the code is free of deadlocks.  
   true/false

7. 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/false

8. The C language is centered around the int data type, which is defined to hold 32-bit integral numbers.  
   true/false

9. Arrays in C are basically syntactic sugar for pointers, and notation may be mixed freely.  
   ```c
   int array[100];
   int *ptr = array;
   
   ptr = 17;
   array[0]++;
   assert(array[0] == *ptr);
   ```  
   - the above assert will hold.  
   true/false

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/false

11. Memory allocated by the malloc() function is located on the call stack at the high end of the address space.  
    true/false

12. Finite State Machines can be coded in a number of ways in C.  
    - In the function-based solution, every state is encoded as a separate function.  
    true/false

13. GDB is programming tool that provides controlled execution of an executable.  
    - it also provides post mortem inspection when a core file is generated.  
    true/false

14. An interrupt service routine should restore the context upon entrance.  
    true/false
15. Using interrupts avoid wasting time in polling loops for external events  
   true/false

16. To guarantee atomicity critical sections must be disabled.  
   true/false

17. An interrupt vector points to a table with interrupt routines.  
   true/false

18. When a processor is powered up, the state of the interrupt controller needs to be initialized  
   before the RTOS can be invoked.  
   true/false

19. 
   ```
   static int iSeconds, iMinutes;
   void interrupt vUpdateTime(void)
   {
     ++iSeconds;
     if (iSeconds>=60) {
       iSeconds=0;
       ++iMinutes;
     }
   }
   long lSeconds(void)
   {
     disable();
     int now = iMinutes*60+iSeconds;
     enable();
     return(now);
   }
   
   The above pseudo code correctly dis-/enables the interrupts to solve the shared-data  
   problem.  
   true/false
   ```

20. Given the following pseudo code, which reads the current values of 3 different buttons  
    and acts accordingly. The 3 buttons are all mapped to bits 0..2 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 main (void) {
     while (1) {
       if (buttons & 0x01) f1();
       delay(1000);
       if (buttons & 0x02 ) f2();
       delay(1000);
       if (buttons & 0x04 ) f3();
     }
   }
   
   This code is an example of an RR architecture.  
   true/false
   ```

21. When none of the buttons have been pressed, the longest time that button #2 must be  
    pressed to activate f2() once is 2 seconds.  
    true/false

22. When the system is in an arbitrary state, button #1 must be pressed at most 8 seconds to  
    activate f1().  
    true/false
23. Since disabling interrupts increases interrupt latency, several alternative methods have been developed for dealing with shared data.
   - The Alternating Buffers technique can be used between two “communicating” tasks of equal priority. true/false

24. **Priority inversion** requires a minimum of 3 tasks of different priority and 1 semaphore to occur. true/false

25. On 8-bit processors the number of interrupt priorities is limited to $2^8$. true/false

26. Given is the following RTOS (pseudo) code with priority $T_1 > T_2$.

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

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

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

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

27. The function $f()$ is reentrant true/false

28. With an RR architecture, the handling of I/O devices occurs in a fixed order. true/false

29. An FQS architecture supports priority-based task scheduling. true/false

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

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

32. The **primary** shortcoming of an RRI architecture is that all tasks have the same priority. true/false

33. When detecting a car crash an airbag should not be inflated instantly.
   - An RR architecture provides functionality to support such delayed actions. true/false

34. An ISR can signal a task by operating a semaphore. true/false

35. A function can be made reentrant by means of a critical section, but then it may no longer be called by an ISR. true/false
36. In an RTOS, tasks can be in state BLOCKED, READY or RUNNING.
   - A task can transition directly from READY to BLOCKED.  
     true/false

37. A reentrant function may only be used by one task at a time  
    true/false

38. A program running on an RTOS may create tasks dynamically at runtime.
    - the program ends once main() and all spawned tasks have finished.  
      true/false

39. The 'alternating buffers' technique addresses the shared-data problem by having the
    RTOS control when to switch between buffers.  
    true/false

40. In the implementation of the OS_Pend() primitive, the RTOS first switches the state of
    the current task to BLOCKED, and then looks for a task in the READY queue.
    - if the READY queue is empty the processor may be put into sleep mode to save energy
      when idling.  
      true/false

41. A semaphore used for condition synchronization must be initialized to 1.  
    true/false

42. int f (int x) {
    disable_int();

    !! read some global variables
    !! do some processing, call some functions
    !! write some global variables

    enable_int();
}

Function f() disables/enables interrupts to address the shared-data problem.
- However, when f() calls itself recursively, it is no longer reentrant.  
  true/false

43. Tasks may call the OS_pend() routine, but not the OS_post() routine.  
    true/false

44. The accuracy of a OSTimeDly() depends on the frequency of the periodic timer used
    by the OS.
    - the higher the frequency, the lower the accuracy.  
      true/false

45. The heartbeat timer is a single hardware timer an RTOS is using to verify that the system
    is still progressing (i.e. not deadlocked).  
    true/false

46. 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/false

47. The advantage of pipes over queues is that messages/items can be of variable length.  
    true/false

48. 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.  
    true/false

49. 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

50. An advantage of using tasks is that it allows for better data encapsulation.  
    true/false
51. A key principle of RTOS-based design is that short interrupt routines are needed for a responsive system. **true/false**

52. Printing from an ISR is to be avoided except when the RTOS provides a reentrant primitive to do so. **true/false**

53. Time-slicing should be avoided in an RTOS because it introduces the shared-data problem. **true/false**

54. A semaphore S used by task A must be initialized before A is created. **true/false**

55. Tasks should have different priorities to prevent the RTOS selecting the wrong task. **true/false**

56. 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 host platform. **true/false**

57. Debugging through scripting test scenarios is difficult when the target platform is unavailable. **true/false**

58. Although the assert macro is a useful debugging aid, it can only be used on embedded devices with a display. **true/false**

59. A large study of outdoor sensor-network deployments [Beutel:2009] has shown that the most underestimated problem has been securing the power supply of the sensor nodes. **true/false**

60. When debugging code for a distributed sensor network, collecting the (debug) output of the nodes can be arranged in different ways.
   - **offline** sniffing requires logging facilities on the sniffer nodes. **true/false**