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

exam – **Embedded Software** – TI2726-B
April 5, 2019 18.30 - 20.00

---

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 only use a **B-pencil** so erasures can be applied to correct mistakes.
- 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**

---

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 event-based programming model. true/false

**2.** A defining characteristic of embedded systems is use of a limited, or even lacking, graphical user interface. true/false

**3.** The **Embedded software crisis** refers to the “millennium” bug. true/false

**4.** An embedded program can be coded as a finite state machine where interrupts trigger state transitions. true/false

**5.** A hardware interrupt is an asynchronous signal to indicate the need for processor attention. true/false

**6.** Several models of computation for embedded systems are described in [Lee:2002].
- Process Networks are primarily used to describe concurrency at the hardware level. true/false

**7.** VHDL is an ideal programming language for embedded systems as its synchronous model of computation supports multi-tasking at the hardware level. true/false

**8.**

```
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

**9.** Valgrind is programming tool that aids memory debugging.
- it does so by executing a program in a safe environment. true/false

**10.** 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/false

**11.** Arrays in C are basically *syntactic sugar* for pointers, and notation may be mixed freely.

```
char hello[] = {'w','o','r','l','d'};
char *ptr = hello;

assert(*ptr == 'w');
```

- the above `assert` holds. true/false

**12.**

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

The above driver loop for a FSM follows a round-robin architecture. true/false

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

**14.** Using interrupts with event-based programming avoids the shared-data problem. true/false

**15.** An interrupt service routine should save the context upon entrance. true/false

**16.** To guarantee atomicity task switching must be disabled. true/false

**17.** 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

**18.** An interrupt service routine must be allocated a dedicated call stack. true/false

**19.** A **deadly embrace** requires a minimum of 2 tasks and 1 semaphore to occur. true/false

**20.** An interrupt vector contains the address of an ISR. true/false

**21.**

```
static volatile int count;

main () {
   ...
   int val = count;
   ...
}
```

Reading the value of the global variable `count` is atomic. true/false

**22.** 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 RR architecture. true/false

**23.** When none of the buttons have been pressed, the longest time that button #3 must be pressed to activate f3() once is 1 second. true/false

**24.** When the system is in an arbitrary state, button 1 must be pressed at most 10 seconds to activate f1(). true/false

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

**26.** A high-priority task can be interrupted by a ISR. true/false

**27.** By design the RR architecture is free of the shared-data problem. true/false

**28.** An RTOS architecture supports priority-based task scheduling. true/false

**29.** With an RTOS, the worst response time of a task includes the time taken by the longest task in the system. true/false

**30.** An RTOS architecture is most robust to code changes. true/false

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

**32.** A reentrant function may use hardware only in an atomic way. true/false

**33.** A task can signal an ISR by operating a semaphore. true/false

**34.** An ISR may call the `OS_post()` routine, provided that the RTOS "knows" that the invocation is by an ISR and not by an ordinary task. true/false

**35.** Even a local variable can introduce a shared data problem when its address escapes the defining function, for example, by returning the address as its result. true/false

**36.** 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
}
```

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

**37.** If `counter` is set to 15 when event 2 occurs, and event 1 follows 3 ms later, then the first value printed is 14. true/false

**38.** If the call to `delay` is replaced with `OSTimeDly` task T2 will not be able to run to completion. true/false

**39.** An RTOS usually provides two types of delay functions: polling-based and timer-based.
- timer-based delays are specified in so-called ticks. true/false

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

**41.** To address the shared-data problem, many RTOSs provide communication primitives like queues, mailboxes, and pipes.
- they have in common that pointers can **not** 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 simple `OS_Pend(), OS_Post()` interface the RTOS cannot know in advance which semaphore(s) will be used by a task. true/false

**44.** Consider the following code fragment:

```
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 FQS architecture. true/false

**45.** Consider lines 1-3 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 'Direct Memory Access'. true/false

**46.** The function `rx_ready()` uses a technique called 'alternating buffers' with the global variable `next_command` pointing `main()` to the buffer that is ready for processing.
- the very first command is passed in `buffer[0]`. true/false

**47.** The code suffers from a (subtle) data sharing bug as both `rx_ready()` and `main()` write to the same global variable `next_command`.
- as a result `main()` may read data before `rx_ready()` has written it to the alternate buffer. true/false

**48.** A second issue with the code is the statement on line 22 that adds a character to the alternate buffer.
- that character may be stored outside the space allocated to the variable `buffer`. true/false

**49.** 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 1 is needed. true/false

**50.** Time-slicing should be avoided in an RTOS because it makes the response time of tasks less predictable. true/false

**51.** A key principle of RTOS-based design is that the separation of concerns; by splitting code amongst several tasks, the memory footprint is reduced. true/false

**52.** A semaphore S used by task A must be initialized by A. true/false

**53.** It is recommended to use just the minimum necessary functionality from an RTOS. true/false

**54.** Printing from an ISR is common practice as no other debugging techniques are available. true/false

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

**56.** Even on embedded devices **without** a display, the assert macro is a useful debugging aid. true/false

**57.** Code coverage tools help in thorough testing.
- a 100% coverage can never be achieved for programs that handle (unknown) user input. true/false

**58.** Debugging through scripting test scenarios has limited use as only one interrupt can be triggered at the exact same time. true/false

**59.** A large study of outdoor sensor-network deployments [Beutel:2009] has shown that the most underestimated problem has been the water-proof packaging of the base station. true/false

**60.** When debugging code for a distributed sensor network, collecting the (debug) output of the nodes can be arranged in different ways.
- Out-of-band collection can handle large volumes of data. true/false