# Exam Instructions

The exam is similar to a regular exam in terms of contents. What differs is how the exam is taken, namely:

- how each exercise shows up
- what are the input forms
- what are you allowed/not allowed to do.

With respect to the last item, you can browse the Internet/the course material, the only thing we ask is that you *do not collaborate with other students*. For the first two items, we use the features provided by Inspera.

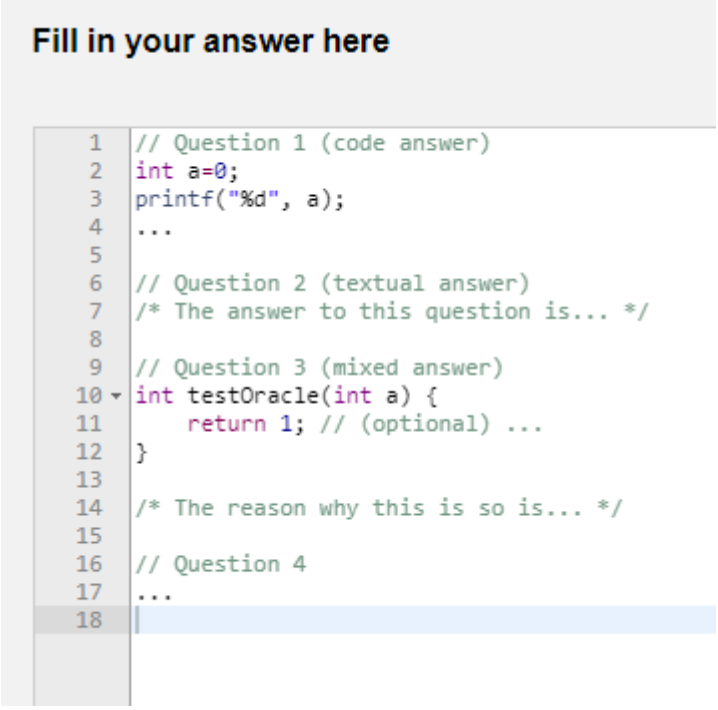
Each exercise (of which there are 3) appears on the left of your screen, while on the right you will find answer form(s). On this page we include exercise 1, as an example, shown on the left panel. There is no answer form so you cannot give answers on this page. If you click on the exercise panel you should be able to scroll up and down from the beginning to the end of the exercise. In case the text on the panel appears illegible or you encounter formatting problems, disable auto-zoom and select a zoom option which gives you the best readability. Also useful are options for resizing the panel. These are highlighted by red arrows on the image below.


Before proceeding any further, ensure you find the settings with which all the text/code of Exercise 1 is readable. On a small screen it is possible that you still encounter issues with scrolling. Hence we also make the full exam available for download as .pdf from the link below. In case you use this option, you can minimize the left panel displaying exercise.

Exam download link

Exercise 1 provides multiple-choice questions with single answers. For each question there is a corresponding form which allows you to choose exactly one answer. Old rules apply, a correct answer awards you 2 points, an incorrect answer losses you 1 point. If unsure, it is (often) better to leave the question unanswered.

Exercises 2 and 3 have specialized input forms which recognize the C programming language, providing syntax highlighting and automatic indentation. We thought using such forms made sense since most questions require writing C code. Unfortunately, limitations of the platform mean there is only one form per exercise. Hence, we expect you to deliver answers as shown in the example below.




Concretely, answers are clearly delimited by line comments containing the corresponding question number ("// Question 1 "). Code is written directly into the form. To help indentation, you may want to copy the context (e.g. the method signature, or the switch block) where the code is supposed to fit in. Explanations are provided inside block (/* multi-line explanation */) or line comments (// single-line explanation). If you think it enhances your answer, you may add comments to the code itself.

Should you have doubts regarding questions, you can contact me at paul.fiterau_brostean@it.uu.se . I will try to reply within one hour.

Good luck!

# 1 Exercise 1: Multiple choice questions (20 points)

Select one answer (alternative) for each of the questions in Exercise 1. 2 points are given for each correct answer, 1 point is subtracted for each wrong answer (the total number of points achieved in this exercise is at least 0).

Question 1:
**Select one alternative:**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

Question 2:
**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4
- Alternative 5

Question 3:
**Select one alternative**

- Alternative 1
- Alternative 2

Question 4:

**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

Question 5:

**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

Question 6:

**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

Question 7:

**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

Question 8:

**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4
- Alternative 5

Question 9:
**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3

Question 10:
**Select one alternative**

- Alternative 1
- Alternative 2
- Alternative 3
- Alternative 4

---

Maximum marks: 20

# 2 Exercise 2: RTOS Programming and Concurrency (20 points)

**Fill in your answer for Questions 1 through 3**

```
1
```

Maximum marks: 20

# 3 Exercise 3: Testing and Coverage (20 points)

**Fill in your answers for Questions 1 through 5.**

```
1
```

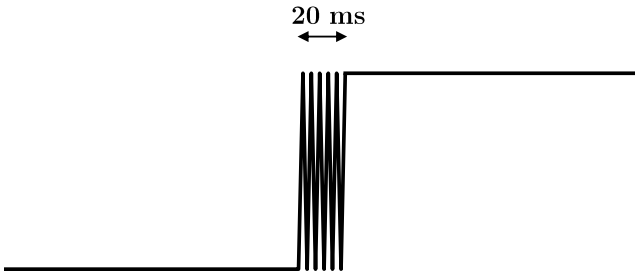
Maximum marks: 20

**Document 1**
Attached

## Exercise 1 Multiple choice questions (20p)

1. An embedded programmer implements a switch signal listener using the function "pollSignalTask" shown bellow. The function adds two events - RISING and FALLING - when detecting rising and falling edges, respectively. The current level of the signal is given by the macro "SIGNAL", which can be either 0 (low) or 1 (high). The function polls the signal every 10 ms and also applies a 6 ms delay for debouncing. Finally, the function uses "state" to keep track of the level of the signal. **(2p)**

```
  int RISING=1, FALLING=0;
void pollSignalTask(void *params) {
  int state = 0, level;
  portTickType wakeTime;
  xLastWakeTime = xTaskGetTickCount();
  for (;;) {
    level = SIGNAL;
    if (level != state) {
      vTaskDelay(6 / portTICK_RATE_MS);
      if (level != state) {
        xQueueSend(eventQueue,
        level? &RISING : &FALLING, portMAX_DELAY);
        state = level;
      }
    }
    vTaskDelayUntil(&wakeTime, 10 / portTICK_RATE_MS);
  }
}
```




Assume that toggling the switch generated the above signal, which takes 20 ms to stabilize to level 1. Also assume the value of "state" was 0 before the switch toggle. What is the maximum number of events that can be generated by this signal?

**1:** 1 **2:** 2 **3:** 3 **4:** 4

2. Suppose three tasks are started, two with priority 1, another with a higher priority 2. Each task calls the method "update" 1000 times without delays and then terminates. **(2p)**

```
void update() {
  x++;
}
```

Suppose the increment operation "++" over a variable $v$ is implemented using the single register $R$ by three atomic instructions:

1. load value of $v$ to $R$
2. add 1 to $R$
3. store content of $R$ to $v$

What is the minimum value shared global variable $x$ can take after the tasks have finished executing? Assume that $x$ is initially 0, and that the scheduler can intervene at any point in the execution. Hint: make sure to consider the given task priorities in your evaluation, as they affect the result. The first atomic instruction has the potential to undo the increment of the second.

**1:** 1000 **2:** 1001 **3:** 2000 **4:** 2001 **5:** 3000

3. On an ARM CORTEX M3 micro-controller, to implement an interrupt service routine it is necessary to introduce a binary semaphore. **(2p)**

**1:** Correct **2:** Incorrect

4. Suppose a structure ("struct") with $l$ int fields, $m$ short fields and $n$ char fields, where $l > m + n$. What is the maximum amount of memory an element of this structure can consume on a 32-bit self-aligned architecture? Consider all possible orderings of the fields. **(2p)**

**1:** $7 * l + 3 * m + n$ bytes
**2:** $4 * l + 2 * m + n$ bytes
**3:** $4 * (l + m + n)$ bytes
**4:** None of the above

5. Suppose "FMUL" is a macro which multiplies three 4-bit signed fixed-point numbers with significand exponent $P = 1$ (i.e., the numbers can be represented in the form $m \cdot 2^{-1}$, where $m$ is a signed 4-bit integer). **(2p)**

```
#define FMUL(x, y, z)      ((x) * (y)) * (z)
```

Which application of this macro to values 0.5, 1.5 and 2.0 does not cause overflows or rounding errors.

**1:** FMUL$(0.5, 2.0, 1.5)$
**2:** FMUL$(1.5, 2.0, 0.5)$
**3:** FMUL$(1.5, 0.5, 2.0)$
**4:** None of the above

6. Which formula correctly captures the following specification: **(2p)**

   "The traffic light should flash red whenever there is a car nearby".

   **1:** $flashRed \vee carNearby$
   **2:** $(\neg flashRed \vee carNearby) \wedge (flashRed \vee \neg carNearby)$
   **3:** $\neg flashRed \vee carNearby$
   **4:** $flashRed \vee \neg carNearby$

7. The minimum number of test cases necessary to achieve MC/DC coverage for the decision $(a \wedge b) \vee (a \wedge c)$ is: **(2p)**

   **1:** 3 **2:** 5 **3:** 4
   **4:** MC/DC is not achievable for this decision

8. Suppose a programs fails for inputs which contain both 0 and 3 (e.g. it fails for $\{0, 2, 3\}$ but not for $\{0, 1, 2\}$). The following "ddMin" implementation is used to shrink the failing inputs. **(2p)**

```
int[] ddMin(Test t, int[] input, int nrChunks){
  for (int i = 0; i < nrChunks; i++){
    int[] withoutChunk = removeChunk(input,nrChunks,i);
    if (!t(withoutChunk)) {
      return ddMin(t,withoutChunk,max(nrChunks-1,2));
    }
  }
  if (nrChucks == strlen(input)) return input;
  return ddMin(t,input,min(2*nrChucks,strlen(input)));
}
```

   Determine the number of times the program is executed (i.e. the number of times function "t" is invoked) during the shrinking of $\{0, 1, 2, 3\}$. Assume "nrChunks" is 2 in the initial "ddMin" call.

   **1:** 4 **2:** 5 **3:** 7 **4:** 8 **5:** other

9. The formula $((\neg p \vee \neg q) \wedge (q \vee r)) \rightarrow (\neg p \vee r)$ is **(2p)**

   **1:** valid **2:** unsatisfiable **3:** satisfiable but not valid

10. A program can be seen as a transition system whose states are identified by a control location (e.g. node in CFG) and the values the program's variables take upon reaching that location. For the program below the states are of form $< location, a, b >$, with $< 1, 0, 0 >$ being the initial state (we assume $a$ and $b$ are initially 0). Control locations are marked by numbers placed before each instruction. **(2p)**

```
1: int a = 12;
2: int b = 8;
3: while ( b != 0 ) {
4:   a = a % b;
5:   swap(a, b);
   }
6: return a;
```

Identify which of the following states cannot be reached by executing the program. Assume "swap" is a single instruction which swaps the values of two variables.

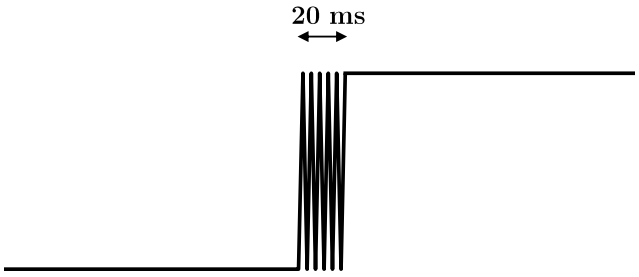
**1:** $< 4, 12, 8 >$ **2:** $< 5, 0, 4 >$ **3:** $< 6, 4, 0 >$ **4:** $< 6, 6, 0 >$

**Question 1**
Attached

## Exercise 1 Multiple choice questions (20p)

1. An embedded programmer implements a switch signal listener using the function "pollSignalTask" shown bellow. The function adds two events - RISING and FALLING - when detecting rising and falling edges, respectively. The current level of the signal is given by the macro "SIGNAL", which can be either 0 (low) or 1 (high). The function polls the signal every 10 ms and also applies a 6 ms delay for debouncing. Finally, the function uses "state" to keep track of the level of the signal. **(2p)**

```
  int RISING=1, FALLING=0;
void pollSignalTask(void *params) {
  int state = 0, level;
  portTickType wakeTime;
  xLastWakeTime = xTaskGetTickCount();
  for (;;) {
    level = SIGNAL;
    if (level != state) {
      vTaskDelay(6 / portTICK_RATE_MS);
      if (level != state) {
        xQueueSend(eventQueue,
        level? &RISING : &FALLING, portMAX_DELAY);
        state = level;
      }
    }
    vTaskDelayUntil(&wakeTime, 10 / portTICK_RATE_MS);
  }
}
```




Assume that toggling the switch generated the above signal, which takes 20 ms to stabilize to level 1. Also assume the value of "state" was 0 before the switch toggle. What is the maximum number of events that can be generated by this signal?

**1:** 1 **2:** 2 **3:** 3 **4:** 4

2. Suppose three tasks are started, two with priority 1, another with a higher priority 2. Each task calls the method "update" 1000 times without delays and then terminates. **(2p)**

```
void update() {
  x++;
}
```

Suppose the increment operation "++" over a variable $v$ is implemented using the single register $R$ by three atomic instructions:

1. load value of $v$ to $R$
2. add 1 to $R$
3. store content of $R$ to $v$

What is the minimum value shared global variable $x$ can take after the tasks have finished executing? Assume that $x$ is initially 0, and that the scheduler can intervene at any point in the execution. Hint: make sure to consider the given task priorities in your evaluation, as they affect the result. The first atomic instruction has the potential to undo the increment of the second.

**1:** 1000 **2:** 1001 **3:** 2000 **4:** 2001 **5:** 3000

3. On an ARM CORTEX M3 micro-controller, to implement an interrupt service routine it is necessary to introduce a binary semaphore. **(2p)**

**1:** Correct **2:** Incorrect

4. Suppose a structure ("struct") with $l$ int fields, $m$ short fields and $n$ char fields, where $l > m + n$. What is the maximum amount of memory an element of this structure can consume on a 32-bit self-aligned architecture? Consider all possible orderings of the fields. **(2p)**

**1:** $7 * l + 3 * m + n$ bytes
**2:** $4 * l + 2 * m + n$ bytes
**3:** $4 * (l + m + n)$ bytes
**4:** None of the above

5. Suppose "FMUL" is a macro which multiplies three 4-bit signed fixed-point numbers with significand exponent $P = 1$ (i.e., the numbers can be represented in the form $m \cdot 2^{-1}$, where $m$ is a signed 4-bit integer). **(2p)**

```
#define FMUL(x, y, z)      ((x) * (y)) * (z)
```

Which application of this macro to values 0.5, 1.5 and 2.0 does not cause overflows or rounding errors.

**1:** FMUL$(0.5, 2.0, 1.5)$
**2:** FMUL$(1.5, 2.0, 0.5)$
**3:** FMUL$(1.5, 0.5, 2.0)$
**4:** None of the above

6. Which formula correctly captures the following specification: **(2p)**

   "The traffic light should flash red whenever there is a car nearby".

   **1:** $flashRed \vee carNearby$
   **2:** $(\neg flashRed \vee carNearby) \wedge (flashRed \vee \neg carNearby)$
   **3:** $\neg flashRed \vee carNearby$
   **4:** $flashRed \vee \neg carNearby$

7. The minimum number of test cases necessary to achieve MC/DC coverage for the decision $(a \wedge b) \vee (a \wedge c)$ is: **(2p)**

   **1:** 3 **2:** 5 **3:** 4
   **4:** MC/DC is not achievable for this decision

8. Suppose a programs fails for inputs which contain both 0 and 3 (e.g. it fails for $\{0, 2, 3\}$ but not for $\{0, 1, 2\}$). The following "ddMin" implementation is used to shrink the failing inputs. **(2p)**

```
int[] ddMin(Test t, int[] input, int nrChunks){
  for (int i = 0; i < nrChunks; i++){
    int[] withoutChunk = removeChunk(input,nrChunks,i);
    if (!t(withoutChunk)) {
      return ddMin(t,withoutChunk,max(nrChunks-1,2));
    }
  }
  if (nrChucks == strlen(input)) return input;
  return ddMin(t,input,min(2*nrChucks,strlen(input)));
}
```

   Determine the number of times the program is executed (i.e. the number of times function "t" is invoked) during the shrinking of $\{0, 1, 2, 3\}$. Assume "nrChunks" is 2 in the initial "ddMin" call.

   **1:** 4 **2:** 5 **3:** 7 **4:** 8 **5:** other

9. The formula $((\neg p \vee \neg q) \wedge (q \vee r)) \rightarrow (\neg p \vee r)$ is **(2p)**

   **1:** valid **2:** unsatisfiable **3:** satisfiable but not valid

10. A program can be seen as a transition system whose states are identified by a control location (e.g. node in CFG) and the values the program's variables take upon reaching that location. For the program below the states are of form $< location, a, b >$, with $< 1, 0, 0 >$ being the initial state (we assume $a$ and $b$ are initially 0). Control locations are marked by numbers placed before each instruction. **(2p)**

```
1: int a = 12;
2: int b = 8;
3: while ( b != 0 ) {
4:   a = a % b;
5:   swap(a, b);
   }
6: return a;
```

Identify which of the following states cannot be reached by executing the program. Assume "swap" is a single instruction which swaps the values of two variables.

**1:** $< 4, 12, 8 >$ **2:** $< 5, 0, 4 >$ **3:** $< 6, 4, 0 >$ **4:** $< 6, 6, 0 >$

**Question 2**
Attached

## Exercise 2 RTOS Programming and Concurrency (20p)

An Embedded Systems developer is tasked with developing software for controlling an one-way escalator. The escalator allows swift navigation of people from the lower to the upper floor of a shopping mall. The escalator is powered by an engine. When the engine is turned off, the escalator stops moving and remains stationary until the engine is turned on. While the engine is turned on, the escalator can move at one of two speeds depending on the mode of operation.

There are four modes the escalator can operate in. In running mode, the escalator moves at full speed. This mode is engaged whenever there are people on the escalator. In energy-saving mode, the escalator moves at reduced speed, thus consuming less energy. Energy-saving mode is entered from running mode once all the people have disembarked the escalator and no person has embarked for the last 20 seconds. After spending 120 seconds in energy-saving mode, the escalator enters stopped mode, in which the engine is switched off. This helps further reduce energy consumption. Running mode is resumed as soon as people embark the escalator. The last mode of operation is maintenance mode. While in maintenance mode the engine is switched off, allowing the escalator to undergo necessary repair work. Maintenance mode can only be entered if there are no people on the escalator. For that purpose, mounted at the escalator's entrance is an indicator light, directly controlled by a toggle switch manipulated by an operator. The toggled-on state of the switch corresponds to when the indicator light is on, the toggled-off state when the light is off. The glowing light signals people not to enter the elevator and to use the staircase instead. As a result, the number of people on the escalator gradually decreases until it reaches zero, allowing the escalator to enter maintenance mode. Once repairs are done, the operator toggles off the switch, which turns off the indicator light, causing the escalator to enter stopped mode.

The engine is controlled by GPIOA output pin 0, with 0 turning off the engine and 1 turning it on. Speed is controlled by GPIOA output pin 1: 0 corresponds to reduced speed, 1 to full speed. The switch controlling the indicator light is connected to GPIOB input pin 0: 0 indicates the toggled-off state, 1 the toggled-on state. In order to count the number of people on the escalator, sensors are mounted at its entrance and exit. These sensors are connected to GPIOB input pins 1 and 2, respectively. A person embarking an escalator is identified by a rising edge on pin 1, whereas a person disembarking it is identified by a falling edge on pin 2.

The following safety requirements have been defined for the system:

**R1:** The engine is turned off only when there are no people on the escalator.

**R2:** The escalator stays in energy-saving mode for a maximum continuous period of 120 seconds.

**R3:** While the engine is on, the speed changes changes only when the escalator's mode of operation changes.

Our Embedded Systems developer has started to implement the system. Soon enough the developer realized the need for two tasks: "controlTask" for implementing the control logic; and a higher priority "peopleTask" for counting the people on the escalator at any one time. Moreover, the developer determined that the control software can be in one of four states, corresponding to the modes the escalator can operate in: (1) STOPPED, when the engine is off, the maintenance switch is toggled off and there are no people on the escalator; (2) ENERGY_SAVING, when the engine is on, the maintenance switch is toggled off, with no people on the escalator; (3) RUNNING, corresponding to the running mode of operation: the engine is on and people are currently using the escalator or used it in the past 20 seconds; (4) MAINTENANCE, when the engine is off and the maintenance switch is toggled on. The state of the system is stored in a global variable "status", and the logic is built around it. The intended shape of the implementation is as follows:

```
#define PERIOD 10


// state variables
enum State {
  STOPPED=0, ENERGY_SAVING=1, RUNNING=2, MAINTENANCE=3,
} status = STOPPED;

int engineOn=0, speed=0;
int switchOn=0, numPeople=0;

void peopleTask(void *params) {
  portTickType wakeTime = xTaskGetTickCount();
  int peopleIn, peopleOut;
  // (1) define local variables

  for (;;) {
    peopleIn = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1);
    peopleOut = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_2);

    // (2) code updating numPeople

    vTaskDelayUntil(&wakeTime, PERIOD / portTICK_RATE_MS);
  }
}
```

```
void controlTask(void *params) {
  portTickType wakeTime = xTaskGetTickCount();
  u32 msCounter = 0; // variable used to measure time
  enum State prevStatus = status;

  for(;;) {
    // read sensors and update counter
    switchOn = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0);
    msCounter += PERIOD;

    // code updating engineOn, speed and status
    switch(status) {
      case STOPPED:
        if (switchOn && numPeople == 0) {
          status = MAINTENANCE;
        } else {
          if (numPeople > 0) {
            status = RUNNING;
            engineOn = 1;
            speed = 1;
            msCounter = 0;
          }
        }
      break;

      case ENERGY_SAVING:
      // (3) the remaining implementation
    }

    // update engineOn/speed signals
    GPIO_WriteBit(GPIOA, GPIO_Pin_0, engineOn);
    GPIO_WriteBit(GPIOA, GPIO_Pin_1, speed);

    vTaskDelayUntil(&wakeTime, PERIOD / portTICK_RATE_MS);
  }
}

int main(void) {
  // some initialisation code, not shown here
  xTaskCreate(controlTask, "controlTask", 100,
              NULL, 1, NULL);
  xTaskCreate(carsTask, "peopleTask", 100,
              NULL, 2, NULL);
  vTaskStartScheduler();
  return 0;
}
```

1. Complete the implementation of the two tasks according to the description above in such a way that it satisfies the requirements **R1**–**R3**. **You should provide the code to be inserted at locations (1), (2) and (3) in functions "controlTask" and "peopleTask". The only statements allowed are assignments (e.g. "a=a+1;"),"if/else,case,break" instructions and, in the case of (1), variable declarations. In particular, use of delay tasks ("vTaskDelay" or "vTaskDelayUntil") will lead to no points being awarded for the respective parts. For the "peopleTask", you can assume no debouncing takes place.** **(11p)**
2. To minimize the likelihood of bugs, the developer has also decided to add a high-priority task, "safetyTask", that checks the requirements **R1**–**R3** at runtime. Implement the requirements by filling location (4). You should use "assert" statements to check if conditions hold. Also, you can use/update the local variables "msSaving" to implement R2, and "prevSpeed" and "prevStatus" to implement R3. You can ignore potential race conditions, and thus, assume that "safetyTask" always checks a consistent state of the system. **(5p)**

```
void safetyTask(void *params) {

  portTickType wakeTime=xTaskGetTickCount();
  int msSaving = 0;
  int prevSpeed = speed;
  enum State prevStatus = status;
  for(;;) {

    // (4) code checking requirements
    // (5) code checking assumptions

    prevSpeed  = speed;
    prevStatus = status;
    vTaskDelayUntil(&wakeTime,
      PERIOD / portTICK_RATE_MS);
  }
}
```

3. The developer has noted that there are several inherent environment assumptions that can be implemented and checked for in "safetyTask". Describe two such assumptions in natural language and implement them by filling location (5). Note that no points are awarded for trivial assumptions such as "sensor(s) are working correctly" or "value read is 0 or 1". Also note that both the textual description and the implementation are required to achieve maximum score. **(4p)**

**Question 3**
Attached

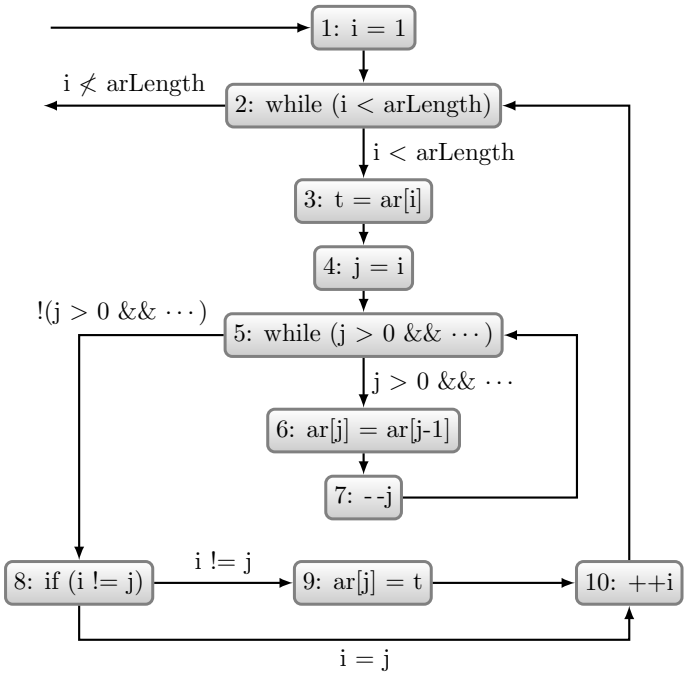
## Exercise 3 Testing and Coverage (20p)

We consider the following function for sorting arrays of integers:

```
void sort(int ar[], int arLength) {
  int i, j, t;
  i = 1;
  while (i < arLength) {
    t = ar[i];
    j = i;
    while (j > 0 && ar[j-1] > t) {
      ar[j] = ar[j-1];
      --j;
    }
    if (i != j)
      ar[j] = t;
    ++i;
  }
}
```

The function receives an array of integers and the length of the array as inputs, and then swaps elements of the array until the array is sorted in ascending order.

The control-flow graph of the function is as follows:



1. We first want to derive a test oracle that checks whether the array "ar" is sorted in ascending order after termination of "sort". **(4p)**
   Write a corresponding test oracle by providing the body of a function

```
int arrayIsSorted(int ar[], int arLength);
```

   that returns 1 if the given array is sorted, 0 otherwise.

2. Write a single minimal test case that achieves full MC/DC coverage for the function "sort", considering the decisions **(6p)**

   i < arLength, j > 0 && ar[j−1] > t, i != j.

   The test case should be minimal with respect to the length of the array, and should take the form

```
int testCase() {
  // create inputs for sort
  // ...

  // call sort
  // ...

  return arrayIsSorted(...);
}
```

   Explain in at most 6 sentences why your test case achieves MC/DC coverage, and why it is minimal.
   **Note:** you may assume that ar[j−1] > t is true if !(j > 0).

3. Write an additional test case showing that decision coverage does not subsume (or imply) MC/DC. Justify your test case in at most 3 sentences. **(3p)**

4. The oracle defined in question 1 is weak, since it does not verify that the array "ar" after sorting (the "post-array") is a permutation of the array before sorting (the "pre-array"). Implement the function "isPermutation" which checks the permutation property by comparing the pre-array ("ar1") with the post-array ("ar2"). The function has the below signature, returning 1 if the post-array is a permutation of the pre-array, and 0 otherwise. **(4p)**

```
int isPermutation(int ar1[], int ar2[], int arLength);
```

5. Provide an improved version of the test case from question 2 which uses both "arrayIsSorted" and "isPermutation". **(3p)**