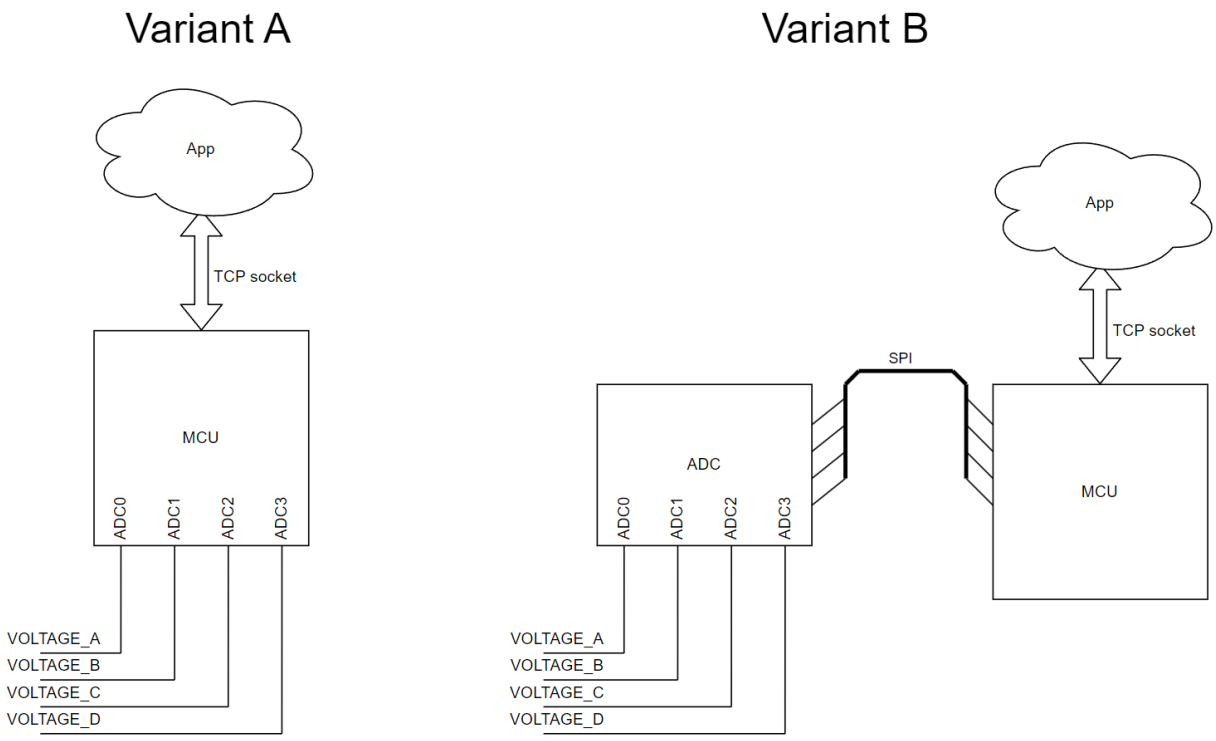
Q1. Microcontrollers

1. Consider a system, where there are two variants of hardware:
    a. low-cost Variant A, where VOLTAGE_[A..D] is measured using MCU internal 12-bit ADC. This ADC uses 3.3 V reference voltage.
    b. high-cost Variant B, where VOLTAGE_[A..D] is measured using external SPI 32-bit ADC. This ADC uses 5 V reference voltage.
2. There is only one firmware for both board variants and a single branch. The firmware needs to be compiled separately for each board Variant.
3. VOLTAGE[A..D] can range from 0 V to 3 V.
4. Please provide *adc_measurement_task.c* and *adc_measurement_task.h* files implementing the requirements listed below.



Requirements:

1. Implement a module, which exports `void AdcMeasurementTask(void)` function. This function should be implemented in a separate file called *adc_measurement_task.c*. Firmware architect will use this function and it will be called periodically, every 100 milliseconds.
2. The job to be performed by `AdcMeasurementTask` function:
    a. send VOLTAGE_A to the Application every 100 ms

    b. send VOLTAGE_B to the Application every 500 ms
    c. send VOLTAGE_C to the Application every 2200 ms
    d. send VOLTAGE_D to the Application every 3700 ms
3. To get the voltage values either from internal MCU ADC or SPI ADC, use `MCU_ADC_GetAdcData` and `SPI_ADC_GetAdcData` functions respectively. These functions are exported in *mcu_adc.h* and *spi_adc.h* files. For the purpose of this task, assume that these functions do not block, and return the current value of raw ADC data immediately.
4. To send the voltage values to the Application, use `SendVoltage` function exported in *voltage_sender.h* file. For the purpose of this task, assume, that this function does not block and it takes care of all the jobs needed to send the voltage value to the application.
5. If there is a need to initialize the module, `void AdcMeasurementTask_Init(void)` function can be provided as well. It will be called once, before `AdcMeasurementTask` is ran for the first time.

The prototypes of all required functions are provided below:

MCU ADC driver functions (*mcu_adc.h* file):

```c
typedef enum _channel_t
{
    ADC_CHAN_0,
    ADC_CHAN_1,
    ADC_CHAN_2,
    ADC_CHAN_3,
} channel_t;

/*
 * Returns raw data from given ADC channel.
 * This ADC is 12-bit ADC and Vref = 3.3 V
 */
uint16_t MCU_ADC_GetAdcData(channel_t channel);
```

SPI ADC driver functions (*spi_adc.h* file):

```c
typedef enum _channel_t
{
    ADC_CHAN_0,
    ADC_CHAN_1,
    ADC_CHAN_2,
    ADC_CHAN_3,
```

```
} channel_t;

/*
 * Returns raw data from given ADC channel.
 * This ADC is 32-bit ADC and Vref = 5 V
 */
uint32_t SPI_ADC_GetAdcData(channel_t channel);
```

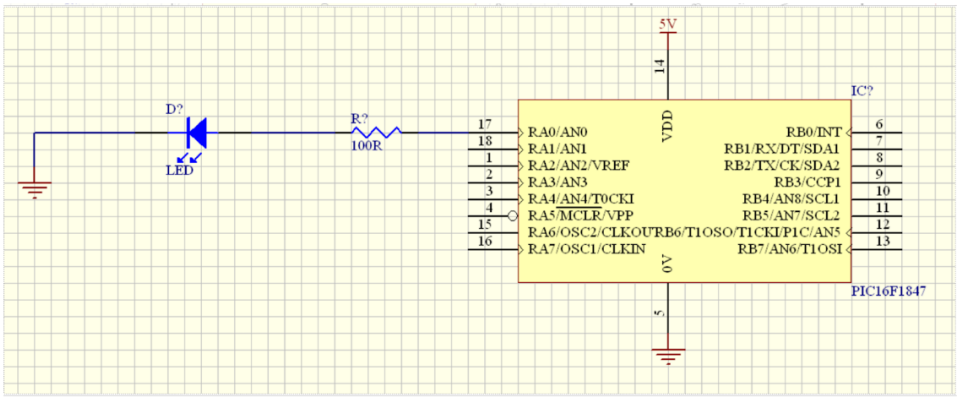
Functions for sending out the voltage values to the Application (*voltage_sender.h* file):

```
typedef enum _voltage_t
{
    VOLTAGE_A,
    VOLTAGE_B,
    VOLTAGE_C,
    VOLTAGE_D,
} voltage_t;

/*
 * Sends voltage value in to the Application via TCP socket.
 * @param voltage_type Voltage Type
 * @param value_mV Voltage value in millivolts [mV]
 */
void SendVoltage(voltage_t voltage_type, uint32_t value_mV);
```

Q2. Schematics

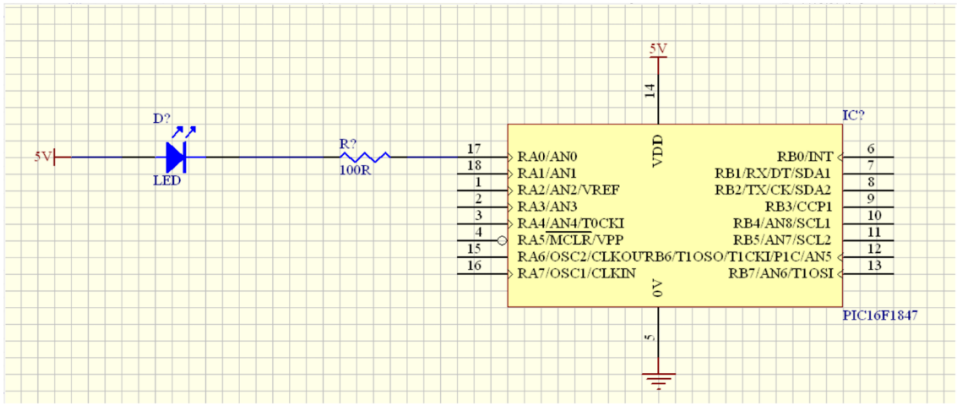
Which method is preferred, why? What are the specific features in both cases?

[Vcc --> LED --> resistor --> IO port]



or

[IO port --> LED --> resistor --> GND] (see diagrams)?



Q3. C-Coding

A. What are the problems with dynamic memory allocation in embedded systems?

B. Is it a valid declaration on a machine which is not 16 bit? Give an explanation. Propose proper declaration.

```
unsigned int null = 0;
unsigned int complement2zero_by_ones = 0xFFFF; // 1' s complement to zero
```

C. What does the following code output and why?

```
#include "stdio.h"

int main (void)
{
long int a = -1;
unsigned int b = 1;

printf ("%zu\n", sizeof (a));
printf ("%zu\n", sizeof (b));

if (a>b)
puts ("a");
else
puts ("b");

return 0;

}
```

D. Which is faster?

```
int A[N][N];
for (j = 0; j < N; j++)
for (i = 0; i < N; i++)
printf ("%u", A[i][j ]);
```

or

```
int A[N][N];
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
printf ("%u", A[i][j ]);
```

E. What will be printed out?

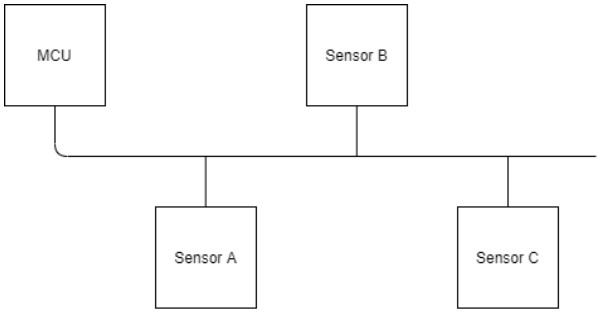
```
#include <stdio.h>
 int main ( void )
 {
 float x = 9/7;
 printf ("%f\n", x);
 return 0;
 }
```

F. What will be printed out?

```c
#include<stdio.h>

void inc(char & array)
{
    array++;
    *(&array + 1) = 'P';
    (*(&array + 1))++;
    char * ptr = &array;
    *(ptr + 2) = 'N';
    char *ptr2 = ptr;
    *ptr2 = 'R';
    (*ptr2)++;

}

int main(void)
{
    char array[] = {'A','G','V',0};//{0x41, 0x47, 86, 0};
    inc(array[0]);
    printf("%s\n", array);
    return 0;
}
```

Q4. Practical Task

Consider the following I2C Bus:



The MCU supplier provides an I2C library and the following interface for writing:

```
I2C_Write(uint8_t address, uint8_t* data, size_t data_length, void (*callback)(bool))
```

This function is non-blocking and returns immediately. The “callback” parameter is a pointer to a function which is called when the write operation is finished.

Example callback function:

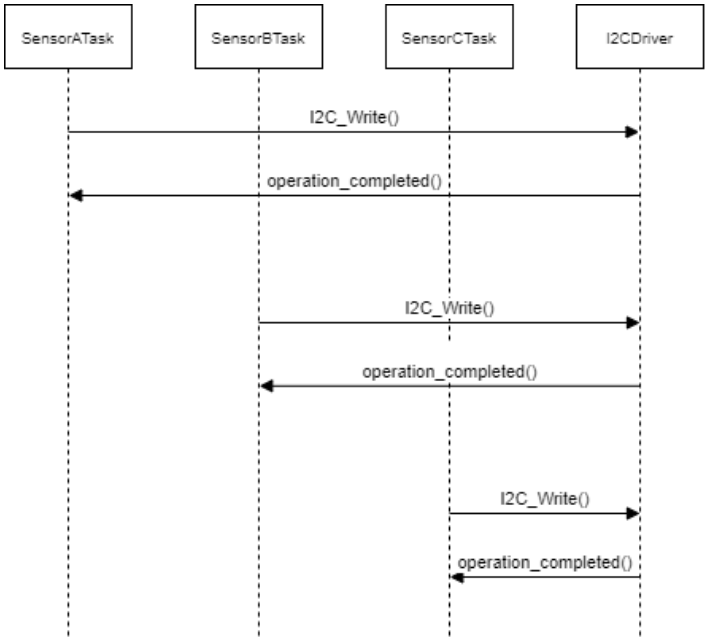
```
void SensorAWritingDoneCallback(bool writing_status_ok)
{
   if(writing_status_ok)
   {
      state = WRITING_DONE;
      invoke_task(SensorATask);
   }
   else
   {
      handle_writing_error();
   }
}
```

Note that I2C_Write() can be called again only, if a previous operation has been completed (i.e. the callback has been called). Violating this requirement makes the I2C driver behavior unpredictable.

The system runs 3 RTOS tasks:

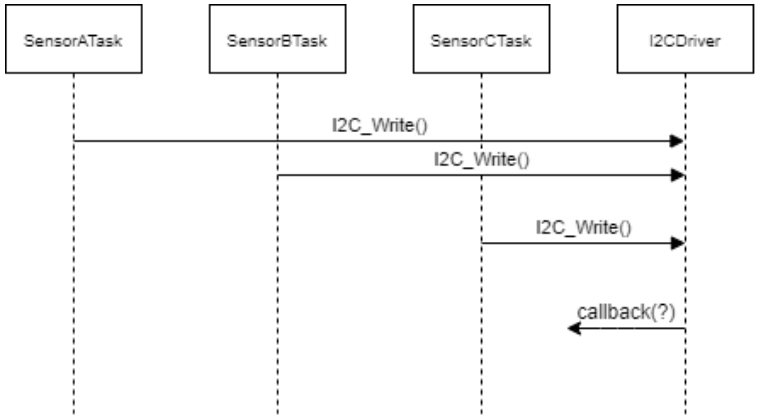
- SensorATask – priority 2 (lower)
- SensorBTask – priority 2 (lower)
- SensorCTask – priority 1 (higher)

After calling I2C_Write(), the I2C driver performs all necessary operations, and as soon as the operation is completed, it calls the callback passed in the I2C_Write() function. The simplest flow would look like in the sequence diagram below:



The problem:

SensorATask writes a big amount of data and it takes some time until the operation is finished, and the callback is called. SensorBTask is executed very frequently and we can face an issue, when it calls I2C_Write(), before a previous operation requested by SensorATask is finished. SensorCTask is high priority task and it is invoked upon interrupt which may happen at any time. If any lower priority task is trying to use I2C interface at this time, we may face data corruption. This bad scenario is shown on the sequence diagram below. After calling I2C_Write() multiple times without waiting for callbacks, the Driver in the end may call the callback or not, as the behavior is unpredictable. Even if it is called, we do not know which operation it refers to.



Please provide the solution which solves the problem described above. You can describe the solution using words only, or UML diagrams, or C code, or pseudocode. Please keep in mind the following assumptions:

- SensorATask, SensorBTask, and SensorBTask are independent and each of them is maintained by different developer.
- Each developer needs to request I2C write operation at any time and receive operation completion callback as soon as possible. This operation should be as simple as calling original I2C_Write() and the user does not need to be aware of the fact that other software components may use the bus at the same time.
- Assume, that task priorities may change and they can preempt each other.
- Assume that in future, the system designer may add more sensors and there is a need to create additional RTOS tasks which also needs to use I2C.
- Assume that some of these tasks may have strict timing requirements and have to be invoked e.g. every 100 milliseconds without any delay.
- MCU vendor issues silicon errata which says that I2C hardware sometimes does not set the bit in register indicating that operation is completed, which may lead to situation, when after using I2C_Write() no callback is called by the I2C Driver.