# SOTR 23/24 Zephyr project

**“Smart” I/O module for the Industrial Internet of Things (IIoT) using Zephyr**

## Introduction

The aim of this project is to implement a simple Input/Output Sensor/Actuator module, which is a common component of IoT and IIoT applications.

The embedded I/O module is based on the Nordic nRF52840 Devkit, using the Nordic NCS toolchain, which includes the Zephyr RTOS. The firmware shall be structured according to the real-time model.

It should be remarked that the “smart” adjective appears here in the (nowadays) conventional sense (i.e. network-enabled node with local processing capabilities), not meaning that it incorporates AI functionalities.

In a realistic scenario, the wireless radio of the DevKit would be used (e.g. via BLE) to connect the I/O node to a hub/controller. However, adding a wireless protocol is not trivial (it is not “rocket science”, but it takes a few hundreds of lines of code and a several hours of work to learn how to use it, and it is out of the scope of this course), so the interface to the I/O module is made via UART. A simple UART-based communication protocol, specified in Annex A , shall be used to interface a PC and the I/O module. Note that if the firmware is properly implemented (i.e. following a modular architecture), moving from the UART interface to a wireless protocol such as BLE would not impact on the structure and functionality of the software.

## Specification

The smart I/O module comprises the following inputs and outputs:

- 4 digital inputs, that match the buttons of the devkit (But 1 to But 4).
- 4 digital outputs, that match the Leds of the devkit (Led 1 to Led 4)
- 1 temperature sensor (TC74, connected via I2C)

> **NOTE**: The devkit supply (VDD) is 3V. Feeding an input pin with a voltage higher than 3V can destroy the devkit. Use the internal power supply (VDD and GND at connector P1) to power the TC74 CI. It is safe to do this as this device consumes a small amount of current.

External (originated in the PC) read and write operations are carried out asynchronously. That is, when the external computing device (the PC) sets an output value or reads an input value, it accesses a Real-Time Database (RTDB). This RTDB is, concurrently, accessed by internal real-time

tasks that keep it synchronized with the I/O interfaces, according to the real-time model. The digital inputs and outputs are synchronized with the RTDB every 100 ms, while the temperature is sampled every second.

Refer to Figure 1 for an overview of the overall system architecture.

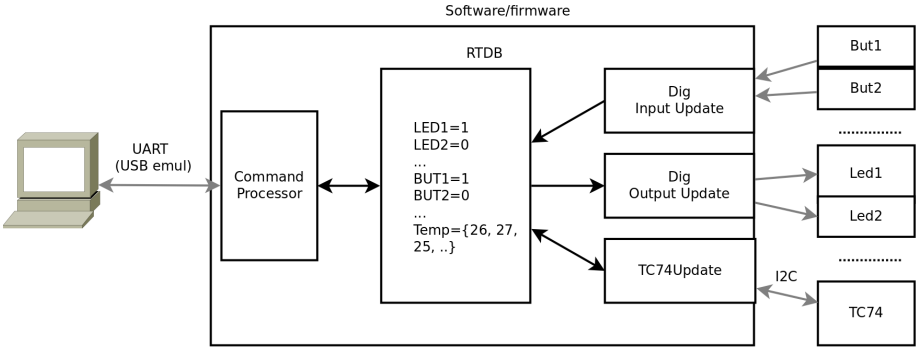


*Figure 1: Smart I/O system architecture overview*

The UART interface should allow the following operations:

- Set the value of each one of the digital outputs (individually)
- Set the value of all the digital outputs (all at once, atomic operation)
- Read the value of the digital inputs
- Read the newest value of temperature
- Read the past 20 values of temperature
- Read the max and min temperature
- Reset the temperature history (including max and min)

As mentioned above, the interface with the “outside world” is made through the UART. A protocol, specified in Annex A , defines commands that allow to carry out the operations listed above. The protocol works on “clear text mode”, that is, only printable characters are used, so the interaction can be made via a terminal (e.g. minicom, PUTTY, ...)

Tasks should have suitable activation modes and use adequate IPC mechanisms.

Additional functionality can be added to the project and shall be valued. In that case, discuss it previously with me, before starting the corresponding development.

**Note 1:** the emphasis of this work is on structuring the software according with the real-time model. A solution using one loop with all functionality inside it will be evaluated with 0 (zero), even if “it works”.

**Note 2:** the global quality of the solution is relevant. This includes e.g. documenting the code and using suitable activation methods and synchronization protocols for tasks.

## Deliverables

- Report, pdf format, up to 10 pages, submitted via eLearning, with:
  - Page 1:
    - Identification of the course, project and authors (student ID number and name)
  - Remaining pages:
    - Discussion of the global system architecture (please include diagrams and text) of the system components (tasks, real-time database, device-drivers, …)
    - Presentation and discussion of tasks, namely its characteristics (e.g. nature of activation, priority, …), execution patterns and relevant events
    - Real-time characterization of the tasks and a discussion of the system schedulability
- Zip or tar file with the full NCS project. I must be able to compile the application in my PC.

# Schedule

- Application:
  - Presented and demonstrated at the last practical class of the course;
  - Submitted via eLearning until the beginning of the last practical class;
- The report can be submitted up to one week before the written exam;
  - Submission via eLearning

# Annex A

**Specification for a Communication Protocol to Connect a PC to a Microcontroller Using a UART interface with Explicit Acknowledgment Messages**

**Introduction**

This document outlines the specification for a communication protocol to connect a PC to a microcontroller (UC) using the Universal Asynchronous Receiver-Transmitter (UART) interface, emulated via USB in this case. The protocol is designed to be simple and reliable, and it can be used to transmit data between the PC and the microcontroller in both directions. The protocol includes explicit acknowledgment messages to ensure reliable data transmission.

**Protocol Overview**

The protocol consists of a series of data frames, each of which starts with a synchronization byte (‘!’) and ends with a checksum followed by an end of frame symbol (‘#’). Frames contain a device ID, that identifies the sending node (PC or UC), and allow a set of distinct commands. Each command has a predefined payload. The checksum is calculated using a simple checksum algorithm, based on the full frame contents, except the beginning and end-of-frame delimiters.

**Data Frame Structure**

| Byte | Description | Size (Bytes) / Contents |
|---|---|---|
| 0 | `Synchronization symbol` | 1 / ‘!’ |
| 1 | `Tx Device ID` | 1 / {‘0’,’1’} |
| 2 | `Command ID` | 1 / printable character (see below) |
| 3 | `Payload[0]` | 1 / printable character (see below) |
| 4 | `Payload[1]` | 1 / printable character (see below) |
| ... | ... | ... |
| n-5 | `Payload[k]` | 1 / printable character (see below) |
| n-[4,3,2] | Checksum | 3 / 3 least significant digits of checksum, in ASCII |
| n-1 | End of frame symbol | 1 / ’#’ |

**Tx Device ID**

The Tx Device ID field is used to identify the transmitting device. The following device ID bytes are defined:

- ‘0’: PC

- ‘1’: Microcontroller

**Command + Payload**

Defines the command and payload carried out in the frame. The following commands are defined:

- ‘0’: Set the value of one of the digital outputs (Leds)
  - Payload: ‘1’...’4’ (Led #, 1 byte) + {‘0’,’1’} (On/Off, one byte)
- ‘1’: Set the value of all the digital outputs (atomic operation)
  - Payload: {‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’} (On/Off, one byte, Led 1 to Led 4, left to right)
- ‘2’: Read the value of the digital inputs
  - Payload: empty
- ‘3’: Read the value of the digital outputs
  - Payload: empty
- ‘4’: Read the last value of temperature
  - Payload: empty
- ‘5’: Read the last 20 values of temperature
  - Payload: empty
- ‘6’: Read the max and min temperature
  - Payload: empty
- ‘7’: Reset the temperature history (including max and min)
  - Payload: empty
- ‘A’: Digital inputs values (answer to cmd ‘2’)
  - Payload: {‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’} (On/Off, one byte, But 1 to But 4, left to right)
- ‘B’: Digital output values (answer to cmd ‘3’)
  - Payload: {‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’} (On/Off, one byte, Led 1 to Led 4, left to right)
- ‘C’: Last temperature value (answer to cmd ‘4’)
  - Payload: {‘+’,’-’} + {‘0’...’9’} + {‘0’...’9’} (i.e. signal followed by temperature, integer with two digits)
- ‘D’: Last 20 temperature values (answer to cmd ‘5’)

  - Payload: same as last temperature value, repeated 20 times, newer to older
- 'E': Maximum and minimum temperature (answer to cmd '6')
  - Payload: same as last temperature value, one for maximum and one for minimum (in this order)

- 'Z': Reception acknowledgment
  - Payload: command ID of the received command + error code code (1 byte)
    - Error code: '1' – command received OK
    - Error code '2' – unknown command
    - Error code '3' – checksum error
    - Error code 4 – frame structure error (any error related to the frame structure, such as wrong number of bytes, receiving a synchronization symbol before the termination of the previous frame, etc.)
  - Should be sent immediately after a command is received

Retransmission of a command occurs if the corresponding Reception Ack message is not received in a 5 seconds interval. In real cases this time is much smaller, but we use it like this to facilitate debugging using a terminal

**Checksum Byte**

The checksum byte is calculated using the following algorithm:

```
uint_16 checksum = 0
for i = 1 to n-3
    checksum = checksum + data[i]
```

where `n` is the number of data bytes in the frame, and `data[i]` is the `i`th data byte.

In other words, it is the sum off all bytes except the synchronization and end-of-frame fields (and obviously the checksum itself).

Only the three least significant decimal digits of the checksum are sent. E.g. if the computed checksum is 12345 (in decimal), then the checksum field should be '3'+'4'+'5'.

**Synchronization Byte**

The Synchronization symbol ('!') is used to synchronize the communications. The PC/microcontroller should discard any data received until a synchronization symbol is received.

**End Byte**

The end of frame symbol ('#') is used to indicate the end of the data frame.

**Communication Sequence**

The following is the communication sequence between the PC and the microcontroller:

1. The PC sends a frame to the microcontroller, waits for an acknowledgment and, when suitable, a reply.
2. The microcontroller waits for a frame:
    (a) Once the microcontroller receives a full frame, it checks its validity and sends the corresponding acknowledgment. See errors above to check which verifications are required.
    (b) If the frame is valid, the command is processed. Depending on the command this translates to:
        i. Updating the RTDB, or
        ii. Reading the RTDB and sending a reply frame
            A. In this case wait for an acknowledgment from the PC and retransmit in case of error or omission (5 seconds timeout, as specified above)

**Example: S**et led 1 to ON.

- **PC → UC: !0011194#**
    - ‘0’ (from PC) + ‘0’ (set one led) + ‘1’ (led 1)+ ‘1’ (turn on). Checksum variable is 48+48+49+49=194, thus send ‘1’+’9’+’4’ as checksum.
- **UC → PC !1Z01236#**
    - ‘1’ (from UC) + ‘Z’ (Ack frame) + ‘0’ (command received)+ ‘1’ (no errors). Checksum variable is 49+90+48+49=236, thus send ‘2’+’3’+’6’ as checksum.