SOTR 24/25 Zephyr project

User-level Static Table-Based Scheduler Framework for Zephyr

1 Introduction

The aim of this project is to implement a Static Table-Based Scheduler (STBS) Framework for Zephyr. To reduce the complexity of the challenge, the scheduler is fully implemented at the user-level, i.e., resorting to tasks, without modifying the Zephyr kernel. While this simpler approach incurs in some performance issues and limits some functions, approaching the problem by modifying the Zephyr kernel code is far more complex and out of the scope of this course.

To verify and validate the implementation of the scheduler, it will be built a test application/use-case. This application is composed of a set of simple tasks, that realize an IoT digital I/O module, and communicates with the PC via a (virtual) serial port.

2 Specification

2.1 For the Static Table-Based Scheduler

The static table-based scheduler should implement an API that allows tasks to register their attributes and synchronize properly. You can consider functions such as:

- STBS_Init(tick_ms, max_tasks)
  - Inits the STBS system, including creating eventual system tasks, initializing variables, etc.
- STBS_Start()
  - Starts the STBS scheduler
- STBS_Stop()
  - Stops the STBS scheduler
- STBS_AddTask(period_ticks, task_id)
  - Adds a task to the STBS scheduler, with a period of period_ticks. task_id is a suitable identifier (its nature depends on the method used to control the activation of the tasks)
- STBS_RemoveTask(task_id)
  - Removes a task identified by task_id from the table
- STBS_WaitPeriod()

- Used inside a task body, terminates a task's job and makes the task wait for its next activation, triggered by the STBS scheduler.

Please note that the API above is just an illustration, to help you understanding what to implement. There are many other functions that can (and should) be implemented, for additional functionality or even for debugging purposes (e.g. print the contents of the table, print the number of activations of a task, …). Moreover, the examples of the API above may not include all necessary input/output arguments, which, to a large extent, depend on your particular approach (and there are a few good ones).

2.2 For the use-case implementation

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)

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.

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

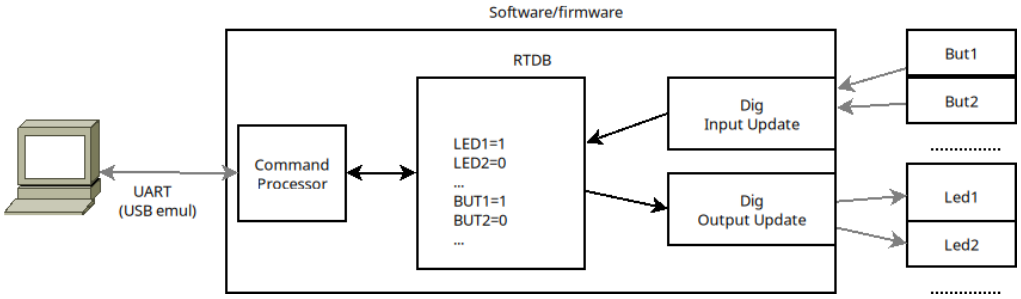

Figure 1: Smart I/O system architecture overview

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

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, ...)

Note 1: the emphasis of this work is on structuring the software according with the real-time model and using the STBS. 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.

3 Schedule and deliverables

- 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
- 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 (user-level scheduler and its API, tasks, real-time database, …)
    - 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 (both for the scheduler and use-case), overhead of the scheduler and a discussion of the global system schedulability
- Zip or tar file with the full NCS project. I must be able to modify the code and compile the application in my PC.

4 Annex A

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

Introduction

This section 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:

- ‘P’: PC

- ‘M’: Microcontroller

Command + Payload

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

- ‘O’: Set the value of one of the digital outputs (Leds)
  - Payload: ‘1’...’4’ (Led #, 1 byte) + {‘0’,’1’} (On/Off, one byte)
- ‘A’: 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)
- ‘I’: Read the value of the digital inputs
  - Payload: empty
- ‘E’: Read the value of the digital outputs
  - Payload: empty
- ‘i’: Digital inputs values
  - Payload: {‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’}+{‘0’,’1’} (On/Off, one byte, But 1 to But 4, left to right)
- ‘e’: 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)
- ‘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: Set led 1 to ON.

- **PC → UC: !PO11#**
    - 'P' (from PC) + 'O' (set one led) + '1' (led 1)+ '1' (turn on). Checksum variable is 80+79+49+49=257, thus send '2'+'5'+'7' as checksum.

- **UC→PC !1Z01236#**
  - ‘M’ (from UC) + ‘Z’ (Ack frame) + ‘O’ (command received)+ ‘1’ (no errors). Checksum variable is 77+90+79+49=295, thus send ‘2’+’9’+’5’ as checksum.