Final Project: Implementation and experimental verification of a Cyclic Asynchronous Buffer (CAB) real-time communication mechanism

Bernardo Falé[1][93331] and Diogo Maduro[2][80233]

University of Aveiro, Aveiro, Portugal

1 Introduction

In this project it was proposed that the students should implement and develop a CAB mechanism and to verify its operation by building a simple application. The students were able to achieve the proposed challenge and will explain the details about it below. It should be said that since the demonstration we changed the data reception task period and the usage of the CAB (Saving images in the CAB). All the work and code is available in our team's repository at `https://github.com/bernardofalle/SOTR`, so the sustained development of the work can be verified.

2 CAB Implementation

The first code written was the CAB, and we sought to implement it in the best way possible, meaning that we followed Buttazzo's paper and its definitions/declarations. We also made it in a way that is generic, which means that a CAB is possible to implement in any primitive that the client needs, as long as the number of messages and the size of that primitive are well chosen.

Basically, our CAB structure consists of `N_TASKS + 1` buffers, and one pointer to the head position, and this pointers holds, at all times, the address of the most recent message put in the CAB. This structure also holds a link counter array that helps our CAB to realize its purpose; While a buffer is owned by one or more tasks, it can not be reserved. We also initialize a mutex that's capable of synchronizing all the CAB operations; Denoting that it's not possible to have race conditions in our implementation.

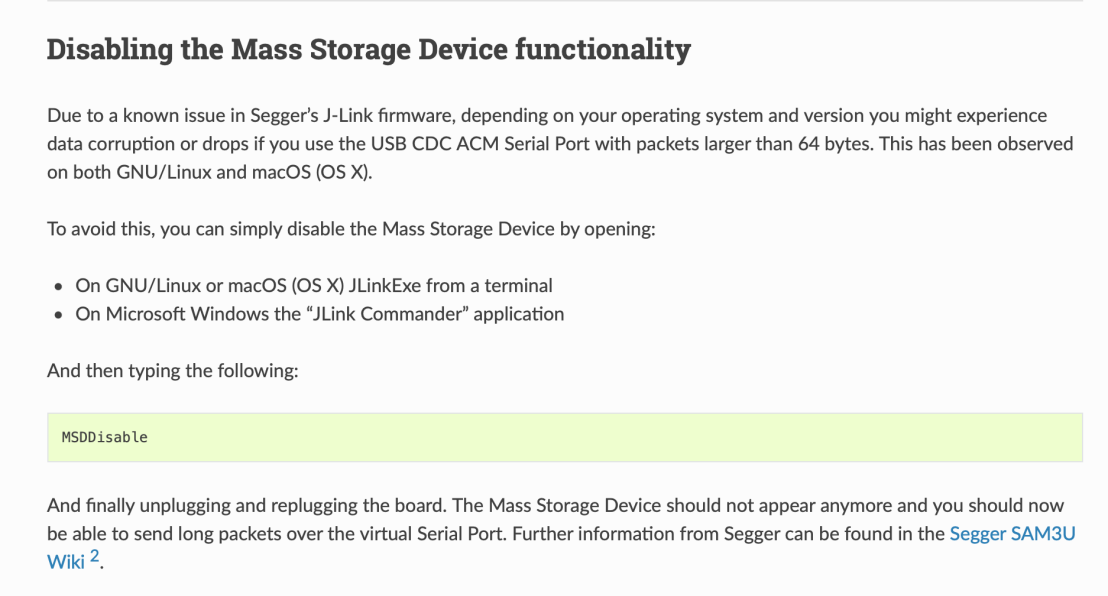
Our operation are all based in the same idea : Memory management. We were able to create operations that follow simple formulas. If the buffer is free of ownership and is not the current head, it can be reserved.

We calculate the buffer address by obtaining the initial pool address, and iterating through the pool. Due to the mutex present, no address can be obtained by different tasks. After reserving the address, we can put the message in the CAB, this works by resetting the value in the link counter, and this tells that the buffer is no longer in use and is the latest message. Get and unget follow similar formulas, getting the head and increasing the link counter, and decreasing the link counter, respectively.

3 Image processing tasks

After the CAB implementation, we started to dig on the image processing tasks. We created four tasks, the Near Obstacle Detection (NOD) and Output, which were critical to the robot's safety and created with prority = 1, the Orientation and Position (OAP), which was executed at a second priority level and created with priority = 2, and finally, the Obstacle Counting (OBSC), which was only made available for statistical purposes and had the least priority of all the tasks. This tasks are similiar, in the way that all should be executed after the reception of an image (in order) by taking a semaphore, and each one executes its image processing function by getting the latest CAB image and setting its values for the output task, which is also given a semaphore to execute by every task, after being executed.

After this we started working on the task that should receive the images dynamically; To implement this we used the `UART ASYNC API` offered by the Zephyr RTOS, this was especially hard, since we could only send 127 bytes each time we had communication between the two devices. After some inconclusive work, we noticed that the Zephyr Dev Team made available in the docs a bug (https://docs.zephyrproject.org/3.2.0/develop/flash_debug/nordic_segger.html) present in the JLink firmware, which disallowed the exchange of large data through UART in our development kit.



Disabling the Mass Storage Device functionality

Due to a known issue in Segger's J-Link firmware, depending on your operating system and version you might experience data corruption or drops if you use the USB CDC ACM Serial Port with packets larger than 64 bytes. This has been observed on both GNU/Linux and macOS (OS X).

To avoid this, you can simply disable the Mass Storage Device by opening:

- On GNU/Linux or macOS (OS X) JLinkExe from a terminal
- On Microsoft Windows the "JLink Commander" application

And then typing the following:

```
MSDDisable
```

And finally unplugging and replugging the board. The Mass Storage Device should not appear anymore and you should now be able to send long packets over the virtual Serial Port. Further information from Segger can be found in the Segger SAM3U Wiki [2].

Fig. 1: Bug disallowing using serial port

Our code only works if this bug is solved, and the solution is given in Fig.1. To conclude, after the UART receives all bytes, the reception task puts it into the CAB and releases the semaphores for the other tasks.

4 Temporal characterization

To find the needed periods we started by analyzing each task, and what scenarios should work to find their WCET. It is also important to say that we measured the following values by making use of the clock cycles and the Zephyr functions that make possible the conversion of cycles to microseconds; We believe that this is the best way to measure time in our micro-controller.

In the NOD task, it should be known that the worst case scenario is when an obstacle is only found in the last position of the CSA upper bound, because in our case, we read the CSA bottoms to tops `(WCET = 580us)`. In the OAP task, the worst case scenario should be when a guideline is found in the last position of the first and last row; Nonetheless, this was not proven in testing, this could be due to the calculation of the angle, or the position. We verified that the WCET was found when our guideline was on the last position of one row, and in the middle of the

other row `(WCET = 423us)`. Since in the OBSC task, we always read every position of the array, the WCET would be O(n), where n is the length of the array `(WCET ± 3500us)`. In general, the execution time of a task that only performs print statements will depend on factors such as the speed of the processor, the efficiency of the code or the type and speed of the output device. After repeating several experiences of the output task, we defined that `WCET = 25381us)`. After collecting this values, we sought to define the periods of the download and upload image tasks; Since the NOD and OUTPUT tasks were the most critical, we defined the period of the data reception task as the sum of the WCET of the critical tasks.

5 Test procedures and Analysis

Finally, we were only able to develop this application because we took the needed steps for its iterative design. We built this small app by following simple rules made available by the faculty, and that, by the way, were really useful. Firstly, we tested our CAB implementation in the PC, by creating a simple sequential program using the structure's operations; This actually worked fine, and we immediately focused to implement it on Zephyr. We only made changes to it later when we re-designed it to be "primitive generic". We did the same for the implementation of the image processing algorithms; They were simple to implement and easy to migrate to the development kit. After testing them with a sample image stored in an array, we were able to mark that job as done. The complicated part became the last iteration of the work, the development of the real-time UART driver. We did not implement the "peer-to-peer" protocol suggested by the teacher between the PC and the MCU, nevertheless, we had some problems building our own solution. Firstly, and like it was mentioned before, we tried other approaches before finding the bug present in Fig.1 in the Zephyr documentation; This includes, dividing the image in 127 byte chunks, alternating through the Zephyr UART Api's and check what worked and what not (interrupt api, async api, poll api). After resolving the bug we found that the ASYNC Api was best, and everything became much more clearer after that. We were able to test different images to get different outputs, and we also were able to make use of the UART "seemingly" continuous reception. Finally, we had trouble saving the image on the CAB structure, and after a lot of thinking we were able to reach the conclusion that we did not allocate enough HEAP memory that we should.