SOTR 2022/23 - Final Project

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

Group

- David José Araújo Ferreira - 93444
- Tiago Adónis Hipólito Dias - 88896

CAB

Structure

Type	Variable	Description
char*	name	CAB name
int	num	Number of messages
int	dim	Dimension (in bytes) of messages
void*	first	Pointer to the initial message

Methods

- *CAB *open_cab(name, num, dim, *first)*

  It begins with the allocation of the CAB structure into the memory. Then initialize the fields of the CAB struct and returns a pointer to the CAB struct.

- *void *reserve(*cabId)*

  First of all, check if the CAB is full. If the previous condition is true, return NULL; if not, returns a pointer to the initial message.

- *void put_mes(*cabID, *buffer)*

  Stores the buffer as the initial message in the CAB.

- *void *get_mes(*cabId)*

  It starts to check if the CAB is empty. If this happens, return NULL; if not, return a pointer to the initial message.

- *void unget(*mes_pointer, *cabId)*

  If the pointer points to the initial message, clear the initial message; if not, return an error message.

Image processing

Image structure

Type	Variable	Description
unsigned char[]	data	Array of bytes representing the image

Methods

Most of the image processing is based of the sample code provided, with a few minor modifications.

Processing

Since the images are always squares, the maximum angle will always be 45º and the minimum -45º. With this, and since Zephyr's math library does not support *sin* calculation, we can calculate the angle in radians by multiplying the difference between the line and the center (which can be positive of negative), multiplying it by the radian value of 45º and divide all by half of the image width.

To find obstacle inside the CSA area, a method similar to the already existing one for counting the total number of obstacle is used, but instead of all of the image, only the CSA area of the image is analyzed.

Main

Task and CABs

The code is composed of four task running in simultaneous in different threads. Each task, except for the output, creates and writes to its specific CAB which stores the relevant information the task has produced, this is necessary because the output task will be accessing each one of this CABs to read the available data inside.

For that, each task, in the beginning of its loop, reads the CAB containing the images and processes the available image in the buffer, this will allow each task to always process the lates image inserted in this CAB buffer by the UART part of the code that we will see later.

Bellow is a simple table that correlates each task, its priority, and its behaviour with each CAB.

Task Name	Priority	CAB (reads from)	CAB (write to)
csa_detection	3	*cab_image*	*cab_csa_detection*
orientation_position	2	*cab_image*	*cab_orientation_position*
obstacle_counting	1	*cab_image*	*cab_obstacle_counting*
output_update	3	*cab_csa_detection*, *cab_orientation_position*, *cab_obstacle_counting*	N/A

UART

The UART communication part of the code is heavily based in Zephyr's sample code of the project *echo_bot* which demonstrated a simple behaviour of reading data from the terminal by using the device node *zephyr_shell_uart* and outputting it back to the terminal.

It will wait for a message for a undetermined amount of time and when a new message is detected, it waits fo a *newline (\n)* character which will indicate the end of the message. This communication only allow for ASCII characters to be passed.

When sending new images, which are an array of bytes, each byte will then have to be converted to an ASCII character in order to be transmitted. When received, this array will be used to create a new *Image* struct and inserted in the *cab_image* for the other tasks to process it.

PC side of communication

On the PC side, there are two main files: an image generator, which is capable of calling a single image or a set of images; and the communication file.

Each image is a text file containing hexadecimal values, this can be then read and converted to ASCII characters by the communication method and a newline character concatenated at the end, and sent via serial to the board.

Faults

The major fault of this project, it that the solution developed does not respect the required criteria of the image size being 128x128 bytes. When 128x128 bytes where tested, the program is not able to be *flashed* into the board since it exceeds the SRAM size.

It is know that are ways of avoiding this problem by allocating the images properly in the memory. The fact that this is not implemented is because the problem was only detected after the development of all of the algorithm, since for the development process, smaller images (like 16x16 bytes) were used for simplicity of management, and thus, this particular problem was never observed. Although the project was developed in a way to account for all sizes of images, it was without the knowledge of this limitation of the hardware.

The communication also has flaws since a *handshake* process of communication was not implemented.