SOTR 22/23 – Final project

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

1. Introduction

Communication between hard and soft real-time tasks is a critical aspect in real-time systems, because blocking on shared resources or waiting for events or messages to arrive may introduce significant, and in some cases unbounded, delays. Therefore, synchronous interactions should be avoided among time-critical tasks, unless they are strictly predictable and time bounded.

CABs (Cyclic Asynchronous Buffers) are a communication mechanism concept designed to allow communication between periodic activities, such as control loops and sensory acquisition processes ([1],[2]). CABs provide a one-to-many communication channel that holds, at any instant, the latest message inserted in its buffer. Messages are not consumed by receiving processes, being instead maintained in the CAB until a new message arrives. This way, a receiving process will always find data in the buffer and gets the latest version, eliminating any sort of synchronization delays.

There are a few considerations that need to be take into consideration. E.g. a buffer can only be updated/overwritten when no reader owns it, the required number of buffers depends on the number of writers/readers, etc. References [1] and [2] provide all the necessary details.

The challenge proposed in this project is to build a CAB mechanism for the Zephyr RTOS and develop a simple application to verify its proper operation.

2. Specification of the work

As mentioned above, the main objective of this project is developing a CAB library for the Zephyr RTOS. Both a library and a test application shall be developed.

2.1. Implementation of the library

It is required to implement a software module/library, composed by header and code files named “cab.c” and “cab.h”.

The interface functions are the ones presented in [1]:

- Cab data structure
    - cab *cab-id;

- Create/initialize a CAB
  - cab-id = open-cab(name, num, dim, first) ;
- Get a pointer to a free CAB buffer (to add data)
  - bufgointer = reserve(cab-id);
- Add the buffer to the CAB
  - put-mes (bufgointer, cab-id) ;
- Get a pointer to the latest data on the CAB (to read data)
  - mesgointer = get-mes (cab-id) ;
- Release the CAB buffer, after reading and processing all data on it
  - unget (mesgointer, cab-id) ;

Details about these functions, including e.g. the purpose and type of the arguments and return variables, can be found in [1].

Other functions can be added to the library, if found necessary, but they should be internal. That is, the interface functions available to an application that uses the CAB library should be only the ones listed above.

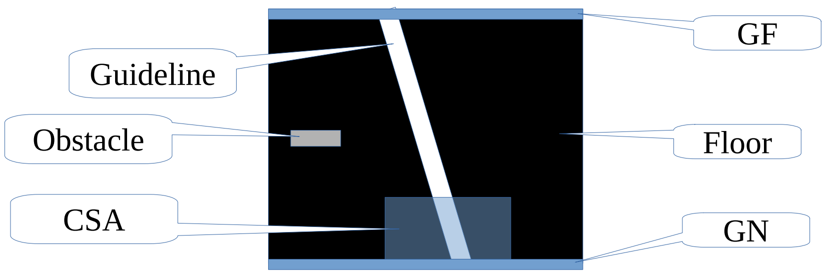
2.2. Testing application

The correctness of the library should be properly assessed. To this end it shall be developed an application that mimics an industrial application.

For this project it is proposed to develop a module for a guidance system of an autonomous robot. The module comprises a serial video-camera (emulated by a PC) that periodically sends an image to the microcontroller. To simplify the analysis algorithms, the image is quite simple, being composed of:

- Floor: black pixels;
- Guideline: white pixels, with a width of 1 pixel;
- Obstacles: gray pixels (at least two pixels, side-by-side in a single row).

Refer to Drawing 1 for a visual description of the image elements.



Drawing 1: Schematic representation of the image

It is assumed that the image was previously processed and is presented in a “birds eye” view, i.e., it looks that the image is taken from above, resembling a map, without optical or perspective distortion. Moreover, the images are also assumed to be denoised (i.e. all floor pixels are “pure” black, all guideline pixels are pure “white”, etc.).

The microcontroller executes the following functions (encapsulated as tasks):

- Near obstacle detection: detects the presence of obstacles in the collision sensor area (CSA);
- Obstacle counting: counts the number of obstacles that shown up on the entire image;
- Orientation and position: detects the robot orientation and position with respect to the guideline. To this end it obtains the position of the white line at areas GN (close to the robot) and GF (far from the robot), computing the corresponding angle (in radians). When the guideline is vertical the angle is 0 rad, positive angles are associated with a tilt to the right and negative angles to a tilt to the left. The position is given as a percentage of the guideline position in the GN (0%- left edge, 50%-middle, 100%-right edge);
- Output update: this task prints the current system state (current orientation, if a near obstacle is detected and total obstacles detected).

The more important tasks are the near obstacle detection and the output update. These tasks are critical to the safety of the robot and should be executed at the highest possible rate.

The orientation and position task is a soft real-time task, being executed at a second priority level.

Finally, the data collected by the obstacle counting task is only used for statistical purposes, so this task in non-real-time.

The image characteristics are as follows:

- 128*128 bytes
- Images are stored in raw 8 bit monochrome format: 8 bits per pixel, representing gray tones/intensity. Varies form 0 (black) to 255 (bright white).
- The sensing areas are as follows:
  - GN: first line of the image
  - GF: last line of the image
  - CSA: rectangle comprised of lines 0 to image_size / 2, and columns between image_size / 4 to 3/4* image_size

3. Implementation indications

The implementation of the project should be carried out incrementally, as follows:

- Step 1: Implement the image processing algorithms (guideline detection, near obstacle detection and obstacle counting). These algorithms (i.e. C functions) can (and should) be developed and tested in a PC.
- Step 2: Port the image processing algorithms to the microcontroller. A sample image, stored in an array at compile time, can be used to test if the algorithms are performing correctly.
- Step 3: Implement the tasks and check if they work properly, still based on the sample image stored defined at compile time. Determine the WCET of the image processing tasks.
- Step 4: Add a task that allows to receive images dynamically from the PC. Develop also an application for the PC to download the images periodically. The PC application should read images from a folder labeled "img1.raw" to "img99.raw". If one file is missing the PC application should stop. Determine the maximum rate at which the images can be sent to the microcontroller and processed by the critical tasks and assign the appropriate periods to these tasks.
- Step 5: Additional functionalities will be valued and are a requirement for attaining a grade above 16 values.
  - A few examples:
    - Make the system robust (e.g. handle communication errors with the PC, handle images in which the guideline is not present or in which there are two guidelines, …);
    - Detect situations in which the system is not able to complete the image processing tasks at the desired rate;
    - Add additional image processing algorithms.

4. Final remarks

The specification of this project is made in a "customer-supplier" approach. That is, I'm assuming the role of a non-technical customer that is ordering you a service. As such, the specification of the work is not complete. It is part of your job to identify any aspects that need to be clarified and discuss them with me. These discussions should be made directly with me and not with the other groups. It is perfectly possible (and desirable, in fact) that different groups develop (slightly) different projects.

5. Deliverables

- Compressed file (zip or tgz) with two folders:
  - Microcontroller code: Full folder project. Must compile in any PC with a default NCS installation.
  - PC application (source files and a makefile).
- Report
  - "pdf" format, four pages, single column, 11pt font

- Name and #mec of group members at the first line
- Contents:
  - Brief description of CAB implementation. No “C” code. Just text and eventually a few diagrams.
  - Brief description of the image processing tasks. Again, no “C” code. Just text and eventually flowcharts/activity diagrams.
  - Temporal characterization of the tasks (WCET, period, …) with a suitable discussion and justification. (How was the WCET measured? How were the periods defined? ...)
  - Description of the test procedures, results and its analysis

6. Bibliography:

[1] G. C. Buttazzo, "HARTIK: A real-time kernel for robotics applications" 1993 Proceedings Real-Time Systems Symposium, 1993, pp. 201-205, doi: 10.1109/REAL.1993.393499.

[2] Clark. D.: "HIC: An Operating System for Hierarchies of Servo Loops", Proc. of IEEE Int. Conf. on Rob. and Autom., pp. 1004-1009, 1989.