Project of Real-Time Programming

POSIX programming of an AADL design

## Expected delivery

This project is not only about programming, but also about general understanding of the CS31 class. At the end, the students are expected to write a report, containing not only the commented source code, but also general explanations about how to program a real-time system, or the end-to-end delays observed.

## System’s description



Figure 1 : controlled process.

We consider the same system as in TD, with some small variations (see Figure 1). Two digital sensors (HLS – High Level Sensor, and LLS – Low Level Sensor) are used in place of the analog sensor to measure the water level. Both are the same kind of sensor, but placed at different positions. They return 1 if they are immersed in the water, 0 if they are not in the water.

The alarm is also considered a digital actuator, as well as the pump command.

Two constants, MS\_L1=70 and MS\_L2=100 will be used to distinguish between the three alarm states.

## Software architecture

Figure 2 is giving the AADL software architecture of the system. It is not the most efficient architecture, but for pedagogical reasons (using most AADL constructions), it is important that the students respect it exactly. The program should have the exact same tasks, and communication objects as in the design. It is not forbidden though to propose in the report, as a discussion, a more efficient design.

There are **four tasks**, **two shared data** allowing to share a data in mutual exclusion, **a synchronization** (***NewData***), and **a mailbox communication** with a mailbox of size one with potential message loss (when a new message is sent to a full mailbox, the previous message is replaced by the new one) between ***CtrlPump*** et ***CmdAlarm***). You can notice that AADL has a graphic representation, but also a textual representation, which is richer than the graphic one. Excerpts of the textual AADL properties related to tasks and communication objects are given on Figure 2.

***AcqMethaneLevel*** is a periodic task. Every time it executes, it reads the methane sensor MS, compares it to MS\_L1 and MS\_L2, in order to compute the alarm state (normal, alarm1, or alarm2). Then it is writing this state in the shared date ***AlarmState***, then triggers the synchronization ***NewData***.

Note: in order to access the sensors, we will use the simulator module, for example, in order to read MS status, we will use ***ReadMS( )***, which is returning the methane level.

***AcqWaterLevel*** is a periodic task. Everytime it executes, it reads the two digital sensors ***LLS*** and ***HLS***, and computes the water state (High or Low). Then it writes this state in the shared data ***WaterState***, then triggers the synchronization ***NewData***.

***CtrlPump*** is a task triggerd by the synchronization ***NewData***: when it is triggered, it reads the two shared data, and computes the pump command as well as the alarm command. It directly sends the pump command to the simulator, but not the alarm command. The alarm command is sent to the ***Alarm*** mailbox.

***CmdAlarm*** is a task waiting for a message in **Alarm** mailbox. Everytime it receives a message, it just applies the received command on the simulator.



Figure 2 : AADL diagram of the controller.

## Files used for this project

What is given:

|  |  |  |
| --- | --- | --- |
| **minepump.tcl** | Simulation of the minepump process. Runs a TCP server listening to the port 4242. Answers to a simple binary protocol: the client (the simulator functions) sends strings ended with a carriage return.  « HLS »: returns a byte with value 0 or 1 depending on the state of the sensor HLS.  « LLS »: returns a byte with value 0 or 1 depending on the state of the sensor LLS.  « MS »: returns a byte with value in 0 .. 255 giving the state of sensor MS.  « Pump cmd »: if cmd = 0 turns the pump off, otherwise, switches it on.  « Alarm cmd »: if cmd = 0 turns the alarm off, otherwise, switches it on. | |
| **simulator.c simulator.o** | Client giving the C functions to communicate with minepump.tcl. Remark: if there is a connection error to the simulator, the program ends. | |
|  | **mymailbox.h and .c** | Same mailbox module as in the TD. |
| **SharedData.h and .c** | SharedData containing an integer. If you want, you can use your own module developed during the TD to replace it. |

## Work to do

* Define MS\_L1 and MS\_L2 as constants with values 70 and 100 respectively.
* Define integer constants for High, Low and Normal, Alarm1, Alarm2.
* Declare two shared data modules for AlarmState and WaterState.
* Declare a mailbox Alarm which will be used by CmdPmp to communicate with CmdAlarm.
* Declare a semaphore NewData which will be used for the synchronization.
* Implement the four tasks
* Create a main function which is:
  + Initializes the simulator (InitSimu),
  + Initializing the two shared data modules,
  + Initializing the semaphore,
  + Initializing the mailbox,
  + Starting the four tasks,
  + Waiting at least one task’s termination.
* Write a report about the project, presenting the commented code, and explaining how AADL elements are programmed.

Remark: for linking, the library to add is -lrt

Testing the program:

* First, run minepump.tcl (you need ActiveTcl to be installed).
* Then, run the program from the Cygwin shell