Lab 4 Report

Tobias L & Özgun M (Group 5)

May 24, 2016

The purpose of this lab was to get a better understanding of how a POSIX message queue works and to get better comprehension of the utilization of different combination of tasks/threads. The main task was to connect a model car consisting of 2 motors and a MDB (Motor Driver Board) to the Raspberry Pi and make it run according to a set of instructions.

1 Motor design

To control the motors, one must provide the program with different parts. One of the parts is a function that takes a motor struct as an argument, but in this case, also speed as well as in which direction the motor should run. The PWM function from previous labs is utilized.

```
typedef enum // MotorSide labels which wheel a motor controls
{
    RIGHT,
    LEFT
} MotorSide;

typedef struct
{
    MotorSide side;
    GPIO direction_pin_fwd; // GPIO pins that controls the running ↩
        direction of the motor
    GPIO direction_pin_back;
    PWM pwm_pin;                // This pwm controls the speed
} Motor;

// motor_activate pulses the motor with a given speed and direction.
void motor_activate(Motor* motor, float speed, int direction)
{
    if (direction)
    {
        onOff(motor->direction_pin_fwd, ON);
        onOff(motor->direction_pin_back, OFF);
    }
    else
    {
        onOff(motor->direction_pin_fwd, OFF);
        onOff(motor->direction_pin_back, ON);
    }
    pwmPulse(motor->pwm_pin, speed / 100);
}
```

2 Implement the Car

This section will briefly describe the different threads and functions that were used to solve the implementation of the car. The using of different components from the C POSIX library including mQueue, pThread will be explained.

2.1 Thread function: input_thread

This thread reads from 4 GPIO buttons and puts a specific message for each button in a POSIX message queue. The buttons are read from ~10 times per second by using the nanosleep function to prevent the buttons to "bounce". When a message has been sent, it is also saved in a variable. This variable is checked when another button press has been read, so that the same message just can be sent once in a row. Now if a button is held down, it still only sends one message thanks to this. There is a special case for the green button which should act in different ways depending on for how long it was pressed. The function below was made for making this possible.

```
char motor_direction(GPIO* button)
{
    struct timespec ts1, ts2;
    clock_gettime(0, &ts1);
    clock_gettime(0, &ts2);

    while (!readIn(*button))
    {
        clock_gettime(0, &ts2);

        int diffs = ts2.tv_sec - ts1.tv_sec;    // If the button is ↩
            held for more than 2 seconds, reverse message will be sent
        if (diffs > 2)
            return 's'; // Reverse message that is put into the ↩
                message queue
    }
    return 'g'; // Accelerate message
}
```

2.2 Thread function: motor_manager_thread

The motor_manager_thread reads from the POSIX message queue. Depending on which message is read from the queue, the corresponding state for the motor is applied. The only exception is for the emergency message which has high priority, but this has almost no effect in the end. There should always at most be only one message in the queue because the input_thread was designed this way. The state of the motors are stored in global enums of the types that are shown below.

```
typedef enum
{
    NEUTRAL,
    TWENTY_FIVE,
    FIFTY,
    SEVENTY_FIVE,
    HUNDRED,
    REVERSE,
    EMERGENCY_STOP
} MotorStates;

typedef enum
{
    RIGHT_ACTIVE,
    LEFT_ACTIVE,
    BOTH_ACTIVE
} ActiveMotors;
```

2.3 Thread function: motor_thread

This thread is used for each pthread that runs a motor. It runs the function motor_speed that checks the state of the current motors instructions which is set by the motor_manager_thread (The global motor state enums).

2.4 Thread function: motor_led_thread

The motor_led_thread runs the function motor_led_decider which activates the appropriate led depending on the state of the motors (The global motor state enums).