REAL-TIME EMBEDDED SYSTEMS PROJECT

CSE_411

Menna Mostafa Naguib	19P5078
Mazen Tarek Hanafy	19P8142
Mirna Ihab Ramzy	19P6793
Amir Mamdouh Nassif	19P5021
Ragui Chawkat Naguib	20P1355

MAY 13, 2023

DR. SHERIF HAMMAD

Eng. Mohamed Tarek

Contents

Figures

1.0 Code

1.1 Main function:

The main function starts by initializing the ports and pins for the system through the Port_init() function.

Then two queues are created for sending motor control commands: the xAutomaticMotorCommandQueue and the xManualMotorCommandQueue, which can each hold one integer value.

A binary semaphore is created using vSemaphoreCreateBinary(), which will be used to synchronize the tasks.

Next, four tasks are created using xTaskCreate():

vUserInterface task: This task handles the user interface of the system and has a priority of 1.

vAutomaticMotorTask task: This task controls the motor in automatic mode and has a priority of 2.

vManualMotorTask task: This task controls the motor in manual mode and has a priority of 2.

vObstacleHandlerTask task: This task handles obstacle detection and avoidance and has a priority of 3.

Finally, the FreeRTOS scheduler is started using vTaskStartScheduler(). The program will block here and execute the tasks based on their priorities. If there is a failure to allocate enough memory from the heap, the program will be stuck in the infinite loop at the end of the main function.

```c
int main()
{


  Port_init();
  Motor_Control(MOTOR_CCW);
  Motor_Control(MOTOR_STOP);
  vSemaphoreCreateBinary( xBinarySemaphore );

  xAutomaticMotorCommandQueue = xQueueCreate( 1, sizeof( int ) );
  xManualMotorCommandQueue = xQueueCreate( 1, sizeof( int ) );

  //GPIO_PORTB_DATA_R = 0x40;

  /* Create you Tasks Here using xTaskCreate()  */

  xTaskCreate(vUserInterface, "User Interface Task",240, NULL, 1, NULL);
  xTaskCreate(vAutomaticMotorTask, "Motor Task",240, NULL, 2, NULL);
  xTaskCreate(vManualMotorTask, "Motor Task",240, NULL, 2, NULL);
  xTaskCreate(vObstacleHandlerTask,"Obstacle Handler task",240,NULL,3,NULL);

  // Startup of the FreeRTOS scheduler.  The program should block here.
  vTaskStartScheduler();

  // The following line should never be reached.
  //Failure to allocate enough memory from the heap could be a reason.
  for (;;);

}
```

Figure 1- main function

1.2 Port_init function:

This code initializes the GPIO (General Purpose Input/Output) ports B, D, and E of a microcontroller by setting up various registers for controlling the direction and configuration of the GPIO pins.

First, the code enables the clock for the GPIO port B, waits for the clock to be ready, and then locks and sets the Commit Register for port B, and enables pins 5,6 and 7 as output.

Figure 2-Port_init function

```c
void Port_init() {
    // Enable the GPIO port B clock
    SYSCTL_RCGCGPIO_R |= 0x02;
    while((SYSCTL_PRGPIO_R & 0x02) == 0);
    GPIO_PORTB_LOCK_R = 0x4C4F434B;
    GPIO_PORTB_CR_R = 0xFF;
    GPIO_PORTB_PDR_R |= 0xFF;
    GPIO_PORTB_DIR_R = 0xE0;
    GPIO_PORTB_DEN_R = 0xFF;

    SYSCTL_RCGCGPIO_R |= 0x10;
    while((SYSCTL_PRGPIO_R & 0x10) == 0);
    GPIO_PORTE_LOCK_R = 0x4C4F434B;
    GPIO_PORTE_CR_R = 0xFF;
    GPIO_PORTE_PDR_R |= 0xFF;
    GPIO_PORTE_DIR_R = 0x00;
    GPIO_PORTE_DEN_R = 0xFF;


    SYSCTL_RCGCGPIO_R |= 0x08;
    while((SYSCTL_PRGPIO_R & 0x08) == 0);
    GPIO_PORTD_LOCK_R = 0x4C4F434B;
    GPIO_PORTD_CR_R = 0xFF;
    GPIO_PORTD_PDR_R |= 0xFF;
    GPIO_PORTD_DIR_R = 0x00;
    GPIO_PORTD_DEN_R = 0xFF;

}
```

Similar steps are followed for ports D and E, with the difference being that all pins on these ports are set to inputs, and their digital enable register is set to enable all pins as digital inputs/outputs.

Overall, this code initializes the GPIO ports B, D, and E for use with external devices by setting the direction and configuration of their pins.

1.3 Motor_Control function:

This is a function named Motor_Control that takes an argument dir of type uint8_t, which is an unsigned 8-bit integer. The purpose of this function is to control a motor's direction of rotation by setting the appropriate pins of the microcontroller. Inside the function, a switch statement is used to select the appropriate motor control action based on the value of dir. There are three cases: MOTOR_CCW, MOTOR_CW, and MOTOR_STOP. In the MOTOR_CCW case, the pins are set to rotate the motor counter-clockwise. The EN pin is set high, which enables the motor. Pin 7 is set low and pin 6 is set high, which sets the motor direction to counter clock wise. In the MOTOR_CW case, the pins are set to rotate the motor clockwise. The EN pin is set high, which enables the motor. Pin 7 is set high and pin 6 is set low, which sets the motor direction to clock wise. In the MOTOR_STOP case, the pins are set to stop the motor. The EN pin is set low, which disables the motor. Pin 7 and pin 6 are both set low, which sets the motor direction to stop.

```
void Motor_Control(uint8_t dir)
{
    switch(dir)
    {
                case MOTOR_CCW:
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 5, 1); //Motor EN
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 7, 0);
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 6, 1);
        break;
                case MOTOR_CW:
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 5, 1); //Motor EN
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 7, 1);
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 6, 0);
        break;
                case MOTOR_STOP:
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 5, 0); //Motor EN
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 7, 0);
                    DIO_WritePin(&GPIO_PORTB_DATA_R, 6, 0);
        break;
    }
}
```

Figure 3-Motor_Control function

1.4 vObstacleHandlerTask function

This is a task function named vObstacleHandlerTask that takes a single parameter of type void *. This function starts by calling xSemaphoreTake() to obtain the xBinarySemaphore semaphore with a timeout of 0 (i.e., do not block if the semaphore is not available). The purpose of this semaphore is to allow this task to be triggered by other tasks when an obstacle is detected.

The task then enters an infinite loop, waiting for the xBinarySemaphore semaphore to be given at first when it enters the task without having semaphore it will be blocked. Once the semaphore is given, the task executes a series of instructions to control the motor. Specifically, it stops the motor, waits for some time, runs the motor in the counter-clockwise direction, waits for some time, stops the motor again. This sequence of motor instructions is intended to be a response to detecting an obstacle in the environment. For example, another task may detect an obstacle and signal the xBinarySemaphore semaphore to trigger this task to respond appropriately.

```
void vObstacleHandlerTask(void *pvParameters){

  xSemaphoreTake(xBinarySemaphore, 0);

  for(;;){

    xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
    Motor_Control(MOTOR_STOP);
    for(int i =0 ;i<1000;i++);
    Motor_Control(MOTOR_CCW);
    for(int i =0 ;i<1000;i++);
    Motor_Control(MOTOR_STOP);
    //for(int i =0 ;i<1000;i++);
    //Motor_Control(MOTOR_STOP);

    //
  }

}
```

Figure 4-vObstacleHandlerTask function

1.5 vAutomaticMotorTask function

The task receives commands through a queue and controls a motor to perform specific actions based on the received command. The task first reads the status of the obstacle button and initializes xReceivedStructure, limitswitch, and reads lock button It then waits to receive a command from a queue. If the queue is empty it blocks, otherwise, if the lock button is pressed,

only the driver can control the window, so we check if the parameter sent to the queue is 1 this means that we gave order to close the window, but first we check in an infinite loop if there was an obstacle detected it gives the semaphore so it immediately goes to the vObstacleHandlerTask function to be executed, else, we send command to the motor to be enabled in the clock wise direction (window is going up), until the limit switch is pressed the motor is stopped. If the xReceivedStructure = 2 that means only the driver can open the window so we send command for the motor to be enabled in the counter clock wise direction (window is going down), until the limit switch is pressed the motor is stopped. In case the lock button is not pressed then the window could be controlled either by the driver or the passenger. If the xReceivedStructure = 1 or 3 then this means that the window is going up in the same manner as described before, and if the xReceivedStructure = 2 or 4 then the window is going down like mentioned before.

```
void vAutomaticMotorTask(void *pvParameters)
{

  while(1) {
    ObstacleButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,2);

    int xReceivedStructure;


    int limitswitch = 0;

    lockbutton = DIO_ReadPin(&GPIO_PORTD_DATA_R,1);


    int xValue = xQueueReceive(xAutomaticMotorCommandQueue, &xReceivedStructure, portMAX_DELAY );

    if (lockbutton == 1) {

    if ( (xReceivedStructure == 1)) {  //automatic driver up
      while(1){
          if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
            xSemaphoreGive(xBinarySemaphore);
            break;
          // give semaphore
          }
        Motor_Control(MOTOR_CW);
        for(int i =0 ;i<1000;i++);
        Motor_Control(MOTOR_CW);
        if(DIO_ReadPin(&GPIO_PORTB_DATA_R,2)==1){ //limit up
        Motor_Control(MOTOR_STOP);
        //xSemaphoreGive(xBinarySemaphore); // was obstacle
        break;
```

Figure 5- vAutomaticMotorTask function part 1

```
    if (xReceivedStructure == 2)  { //automatic driver down

      while(1){

        Motor_Control(MOTOR_CCW);
        for(int i =0 ;i<1000;i++);
        Motor_Control(MOTOR_CCW);

        if(DIO_ReadPin(&GPIO_PORTB_DATA_R,2)==1){ //limit up
          Motor_Control(MOTOR_STOP);
          break;
        }

      }
    }

  }

    else if (lockbutton == 0) {

      if ( (xReceivedStructure == 1) || (xReceivedStructure == 3) ) {  //automatic window up (driver or passenger)

        while(1){
          if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
          xSemaphoreGive(xBinarySemaphore);
            break;
          // give semaphore
        }
          Motor_Control(MOTOR_CW);
          for(int i =0 ;i<1000;i++);
          Motor_Control(MOTOR_CW);
```

Figure 6- vAutomaticMotorTask part 2

```
          if(DIO_ReadPin(&GPIO_PORTB_DATA_R,2)==1){ //limit up
          Motor_Control(MOTOR_STOP);
            //xSemaphoreGive(xBinarySemaphore); // was obstacle
            break;
          }
        }
      }

      else if ((xReceivedStructure == 2) || (xReceivedStructure == 4))   {  //window down (driver or passenger)
        //Motor_Control(MOTOR_CCW);
        while(1){
        //  for(int i =0 ;i<1000;i++);

          Motor_Control(MOTOR_CCW);
          for(int i =0 ;i<1000;i++);
          Motor_Control(MOTOR_CCW);
          if(DIO_ReadPin(&GPIO_PORTE_DATA_R,5)==1){ //limit up
          Motor_Control(MOTOR_STOP);
            //xSemaphoreGive(xBinarySemaphore); // was obstacle
            break;
          }

        }

      }

    }
  }
}
```

Figure 7- vAutomaticMotorTask part 3

1.6 vManualMotorTask function

The function vManualMotorTask is the main task function, which executes in an infinite loop. The task reads the status of two buttons connected to the microcontroller pins: the ObstacleButton and the lockbutton. Then, the task waits for a command to be received from a queue (xManualMotorCommandQueue). The task blocks on the xQueueReceive function until a command is received. When a command is received, the task performs the corresponding motor control operation based on the value of the received command.

If the lockbutton is pressed, the task will perform the manual motor operations for the driver. The driver can move the motor up or down by pressing the corresponding button. The motor will continue to move in the corresponding direction until either it reaches the limit switch (DIO_ReadPin(&GPIO_PORTD_DATA_R,6) for up and DIO_ReadPin(&GPIO_PORTD_DATA_R,7) for down) or the ObstacleButton is pressed. If the ObstacleButton is pressed if it is going up, the task gives a semaphore (xBinarySemaphore) to signal the vObstacleHandlerTask function, and the motor stops moving.

If the lockbutton is not pressed, the task will perform the manual motor operations for the passenger or the driver. The passenger/driver can move the motor up or down by pressing the corresponding button. The motor will continue to move in the corresponding direction until either it reaches the limit switch (DIO_ReadPin(&GPIO_PORTB_DATA_R,2) for up and DIO_ReadPin(&GPIO_PORTE_DATA_R,5), for down or the window button, either up or down, is no longer pressed or the ObstacleButton is pressed (if window is going up). If the ObstacleButton is pressed, the task gives a semaphore (xBinarySemaphore) to signal the he vObstacleHandlerTask function, and the motor stops moving. The motor control function is Motor_Control, which takes an argument to specify the direction of the motor (MOTOR_CW for going up for clockwise and MOTOR_CCW for counterclockwise (for going down). The motor control function sets the corresponding pins on the microcontroller to control the direction and speed of the motor. If the lock button is not pressed we will check on each xReceivedStructure and act accordingly. For example, if xReceivedStructure = 1 then the driver press the button to close the window, so we will enable the motor in clockwise direction and check for obstacle until the button is released or the limit switch is pressed. We conducted the same steps according to the xReceivedStructure's value if 2 (manual driver down), if 3 (manual passenger up), if 4 (manual passenger down).Overall, this task controls the movement of a motor based on the commands received from a queue and the status of the buttons connected to the microcontroller.

```
void vManualMotorTask(void *pvParameters) {

  while(1) {
    ObstacleButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,2);
    lockbutton = DIO_ReadPin(&GPIO_PORTD_DATA_R,1);
    int xReceivedStructure;

    int xValue = xQueueReceive(xManualMotorCommandQueue, &xReceivedStructure, portMAX_DELAY );

    if (lockbutton == 1) {
      if ( (xReceivedStructure == 1)) {              //manual driver up
        Motor_Control(MOTOR_STOP);
        while ( (DIO_ReadPin(&GPIO_PORTD_DATA_R,6) == 1) && (DIO_ReadPin(&GPIO_PORTB_DATA_R,2)!=1) )  {
        if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
          xSemaphoreGive(xBinarySemaphore);
            break;
          // give semaphore
        }
        Motor_Control(MOTOR_CW); // delay to let motor be on

        }
        Motor_Control(MOTOR_STOP);
      }

      else if (xReceivedStructure == 2)  {          //manual driver down
          Motor_Control(MOTOR_STOP);
        while ((DIO_ReadPin(&GPIO_PORTD_DATA_R,7) == 1 ) &&(DIO_ReadPin(&GPIO_PORTE_DATA_R,5) != 1)) {

          Motor_Control(MOTOR_CCW);
        }
              Motor_Control(MOTOR_STOP);
        }
```

Figure 8-vManualMotorTask

```
    if (lockbutton == 0) {

      if ( (xReceivedStructure == 1)) {              //manual driver up
        Motor_Control(MOTOR_STOP);
        while ( (DIO_ReadPin(&GPIO_PORTD_DATA_R,6) == 1) && (DIO_ReadPin(&GPIO_PORTB_DATA_R,2)!=1) )  {
        //  Motor_Control(MOTOR_CW);
        if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
          xSemaphoreGive(xBinarySemaphore);
            break;
          // give semaphore
        }
        Motor_Control(MOTOR_CW); // delay to let motor be on

        }
        Motor_Control(MOTOR_STOP);
      }

      else if (xReceivedStructure == 2)  {          //manual driver down
              Motor_Control(MOTOR_STOP);
        while ((DIO_ReadPin(&GPIO_PORTD_DATA_R,7) == 1 ) &&(DIO_ReadPin(&GPIO_PORTE_DATA_R,5) != 1)) {

          Motor_Control(MOTOR_CCW);
          for(int i =0 ;i<1000;i++);
        }
              Motor_Control(MOTOR_STOP);
        }

      else if ( (xReceivedStructure == 3)) {      //manual passenfer up
        Motor_Control(MOTOR_STOP);
        while ( (DIO_ReadPin(&GPIO_PORTB_DATA_R,1) == 1) && (DIO_ReadPin(&GPIO_PORTB_DATA_R,2)!=1) )  {
        //  Motor_Control(MOTOR_CW);
        if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
          xSemaphoreGive(xBinarySemaphore);
```

Figure 9-vManualMotorTask part 2

```
        if( DIO_ReadPin(&GPIO_PORTD_DATA_R,2)==1 ){
          xSemaphoreGive(xBinarySemaphore);
            break;
          // give semaphore
        }
        Motor_Control(MOTOR_CW); // delay to let motor be on
        for(int i =0 ;i<1000;i++);

        }

        Motor_Control(MOTOR_STOP);
      }

      else if (xReceivedStructure == 4)  {          //manual passenger down
            Motor_Control(MOTOR_STOP);
        while ((DIO_ReadPin(&GPIO_PORTB_DATA_R,4) == 1 ) &&(DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1)) {

          Motor_Control(MOTOR_CCW);
          for(int i =0 ;i<1000;i++);
        }
            Motor_Control(MOTOR_STOP);
        }
      }

}
  }
```

Figure 10- vManualMotorTask part 3

1.7 vUserInterface function

This code defines a function called vUserInterface that continuously reads input signals from various buttons and switches and sends corresponding messages to two different message queues, xAutomaticMotorCommandQueue and xManualMotorCommandQueue, which are used to control the direction of movement of motor.

Here's what the code does step by step:

It enters an infinite loop using while(1).

It reads the state of the lock button from pin 1 of the GPIO port D using the DIO_ReadPin function and stores the result in the variable lockbutton.

It reads the state of various other buttons and switches related to manual and automatic operation of motor and stores the results in the corresponding variables ManualDriverUpButton, ManualDriverDownButton, ManualPassengerUpButton, ManualPassengerDownButton, limitswitchUp, limitswitchDown, AutomaticDriverUpButton, AutomaticDriverDownButton, AutomaticPassengerUpButton, and AutomaticPassengerDownButton.

It defines some constants that are used to represent different commands that can be sent to the motor control queues, such as MOTOR_COMMAND_Driver_UP=1, MOTOR_COMMAND_Driver_DOWN=2, MOTOR_COMMAND_Passenger_UP=3, and MOTOR_COMMAND_Passenger_DOWN=4, and are later sent as parameters to the queue.

All the send operations to the queue are with delay 0 because it's block on read as the receiving function is of higher priority, so after we write we immediately read so it will never need to block.

It checks the state of the automatic driver up button and sends a message containing the MOTOR_COMMAND_Driver_UP constant to the xAutomaticMotorCommandQueue if the button is pressed.

It checks the state of the automatic driver down button and sends a message containing the MOTOR_COMMAND_Driver_DOWN constant to the xAutomaticMotorCommandQueue if the button is pressed.

It checks the state of the automatic passenger up button and sends a message containing the MOTOR_COMMAND_Passenger_UP constant to the xAutomaticMotorCommandQueue if the button is pressed.

It checks the state of the automatic passenger down button and sends a message containing the MOTOR_COMMAND_Passenger_DOWN constant to the xAutomaticMotorCommandQueue if the button is pressed.

It checks the state of the manual driver up button and sends a message containing the MOTOR_COMMAND_Driver_UP constant to the xManualMotorCommandQueue if the button is pressed.

It checks the state of the manual driver down button and sends a message containing the MOTOR_COMMAND_Driver_DOWN constant to the xManualMotorCommandQueue if the button is pressed.

It checks the state of the manual passenger up button and sends a message containing the MOTOR_COMMAND_Passenger_UP constant to the xManualMotorCommandQueue if the button is pressed.

It checks the state of the manual passenger down button and sends a message containing the MOTOR_COMMAND_Passenger_DOWN constant to the xManualMotorCommandQueue if the button is pressed.

```
void vUserInterface(void *pvParameters) {


    while(1) {
      lockbutton = DIO_ReadPin(&GPIO_PORTD_DATA_R,1);

      ManualDriverUpButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,6);
      ManualDriverDownButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,7);
      ManualPassengerUpButton = DIO_ReadPin(&GPIO_PORTB_DATA_R,1);
      ManualPassengerDownButton = DIO_ReadPin(&GPIO_PORTB_DATA_R,4);

      limitswitchUp = DIO_ReadPin(&GPIO_PORTB_DATA_R,3);
      limitswitchDown = DIO_ReadPin(&GPIO_PORTE_DATA_R,5);


      AutomaticDriverUpButton = DIO_ReadPin(&GPIO_PORTB_DATA_R,0);
      AutomaticDriverDownButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,3);
      AutomaticPassengerUpButton = DIO_ReadPin(&GPIO_PORTE_DATA_R,2);
      AutomaticPassengerDownButton = DIO_ReadPin(&GPIO_PORTE_DATA_R,3);

      //ObstacleButton = DIO_ReadPin(&GPIO_PORTD_DATA_R,2);

      int MOTOR_COMMAND_Driver_UP = 1;
      int MOTOR_COMMAND_Driver_DOWN = 2;
      int MOTOR_COMMAND_Passenger_UP = 3;
      int MOTOR_COMMAND_Passenger_DOWN = 4;


      if (AutomaticDriverUpButton == 1) {
      xQueueSend(xAutomaticMotorCommandQueue, &MOTOR_COMMAND_Driver_UP,0);
      }
```

Figure 11-vUserInterface

```
      if (AutomaticDriverDownButton == 1) {
      xQueueSend(xAutomaticMotorCommandQueue, &MOTOR_COMMAND_Driver_DOWN, 0);
      }

      if (AutomaticPassengerUpButton == 1) {
      xQueueSend(xAutomaticMotorCommandQueue, &MOTOR_COMMAND_Passenger_UP, 0);
      }

      if (AutomaticPassengerDownButton == 1) {
      xQueueSend(xAutomaticMotorCommandQueue, &MOTOR_COMMAND_Passenger_DOWN, 0);
      }

      if (ManualDriverUpButton == 1) {
      xQueueSend(xManualMotorCommandQueue, &MOTOR_COMMAND_Driver_UP, 0);
      }

      if (ManualDriverDownButton == 1) {
      xQueueSend(xManualMotorCommandQueue, &MOTOR_COMMAND_Driver_DOWN, 0);
      }

      if (ManualPassengerUpButton == 1) {
      xQueueSend(xManualMotorCommandQueue, &MOTOR_COMMAND_Passenger_UP, 0);
      }

      if (ManualPassengerDownButton == 1) {
      xQueueSend(xManualMotorCommandQueue, &MOTOR_COMMAND_Passenger_DOWN, 0);
      }

  }
```

Figure 12-vUserInterface part2

2.0 Flowchart