Lab 3 Bonus Challenge

Implementation (Bump switch reading)

For the implementation of the bump switch reading, the main methods used were HandleCollision() and BumpInt_Init(). Also, to note that the ports used for each bump switch is as follows: P4.7, P4.6, P4.5, P4.3, P4.2, P4.0.

BumpInt_Init(void(*task)(uint8_t))

This method's task is to mainly initialize all the bumper GPIO port (Port 4) as active high input pins. After the initializing phase, we can now read the data produce by our bump switches.

HandleCollision(uint8_t bumpSensor)

Before this method is invoked, a series of task must first be performed. Firstly, when any of the bump switch is hit, ***PORT4_IRQHandler(void)*** will be triggered and ***Bump_Read(void)*** would be called. Next, inside ***Bump_Read(void)*** we will convert the 8-bit result from Port 4. As we only have 6 bump switches, this means that only 6 out of the 8-bits from Port 4 would be used. Port 4 pins are also configured as active high, so when a bump is hit, the respective bit would be 0. As such, my method splits up the 8-bit result into 2 sections after which I would concat them together to produce a 6-bit result, each bit representing each bump switch. As each different bump switch trigger would produce a different 6-bit result, we are able to calculate and determine which bump switch was triggered.

```
if P4.6 was bumped, 8 bit result = 1011 1111
first   = E0    1010 0000 ->(shift right by 2) 0010 1000
second  = 0D    0000 1101 ->(shift right by 1) 0000 0110
first+second+1 ->   0010 1000 | 0000 0110 | 0000 0001 = 0010 1111
```

Figure 1.1, Code summary of 8-bit to 6-bit conversion

With this 6-bit result, ***HandleCollision()*** would reference this result. It would set the global variable of **CollisionData** to the 6-bit result. And, **CollisionFlag** would be set to 1 to signify that a bump hit was detected. When the global variables are updated, the P4 interrupt flag would be reset.

Implementation (Evasive movement)

In this implementation, we would integrate the earlier implementation into the evasive algorithm that was written. The point of this evasive algorithm is so that the robot would know how to react if it's bump switch was triggered. The idea behind this algorithm is for the robot to turn for a certain amount of time accordingly to whichever bump switch was hit. The consideration behind this, is so that the robot is essentially still travelling on a straight path instead of just turning the opposite direction whenever a bump switch is triggered.

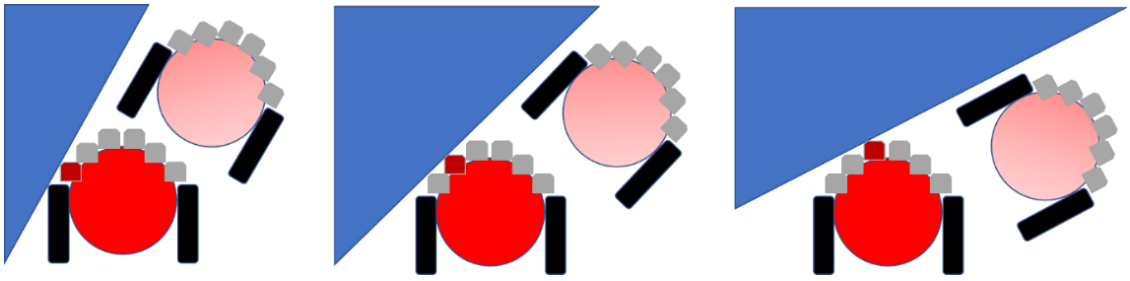

Figure 1.2, Visual representation of evasive movements

Motor_Movt(void)

In this method, it is always checking for a CollisionFlag update. When an update triggers, it will fetch the CollisionData that was previously updated in ***HandleCollision(uint8_t bumpSensor)***. As discussed in our previous implementation, each bump switch is allocated to an individual bit in the 6-bit CollisionData. This means that we can determine which switch was triggered. The method first checks to see if the trigger came from the front 2 bump switches (CollisionData = 51). If this was the case, the robot would reverse and turn ≈180^0 signifying that it had a head-on collision. However, if it was not the front 2 bump switches, the method would check to see if the trigger came from the left or right 3 bump switches. Once checked, we can set specific cases for each triggers, such as turning for a different amount of time.

Bump_switch	CollisionData (Decimal)	Outcome
P4.7, bump 6 (left most)	31	Turn right for 250ms
P4.6, bump 5 (left middle)	47	Turn right for 400ms
P4.5, bump 4 (left front)	55	Turn right for 700ms
P4.3, bump 3 (right front)	59	Turn left for 700ms
P4.2, bump 2 (right middle)	61	Turn left for 400ms
P4.0, bump 1 (right most)	62	Turn left for 250ms
P4.5 & P4.3, bump 4 &3 (left & right front)	51	Turn right for 1400ms

Lab 5

Tachometer

A tachometer is a sensor that is attached to the motors of the robot. The ports of the tachometer connected to the left motor are P8.2 and P9.2. The ports of tachometer connected to the right motor are P10.4 and P10.5.

Implementation (Tachometer)

For the implementation of the tachometer, the methods used were TimerA3Capture_Init(), PeriodMeasure0(), PeriodMeasure2().

TimerA3Capture_Init(void(*task0)(uint16_t time), void(*task2)(uint16_t time))

This method's task is to initialise the tachometer ports (P10.4 and P8.2) as input pins and configure TimerA3 to trigger interrupts at the rising edges of P10.4 and P8.2 pins. After the initialising phase, we can now read the motor speed.

PeriodMeasure0(uint16_t time)

This method's task is to measure the period of P10.4 (right tachometer). Before this method is invoked, a series of task must first be performed. Firstly, at the rising edge of P10.4, ***TA3_0_IRQHandler(void)*** will be triggered and the value of TIMER_A3 CCR0 would be passed into this function. The value that is passed in minus the previous value (***uint16_t First0***) passed in would be the period of the right tachometer. After the calculation, the passed in value would be set as the "old" measurement for the next calculation. Finally, it updates the global variable (***uint16_t Period0***) for the right tachometer measurement.

PeriodMeasure2(uint16_t time)

This method's task is to measure the period of P8.2 (left tachometer). Before this method is invoked, a series of task must first be performed. Firstly, at the rising edge of P8.2,

TA3_N_IRQHandler(void) will be triggered and the value of TIMER_A3 CCR2 would be passed into this function. The value that is passed in minus the previous value (***uint16_t First2***) passed in would be the period of the left tachometer. After the calculation, the passed in value would be set as the "old" measurement for the next calculation. Finally, it updates the global variable (***uint16_t Period2***) for the left tachometer measurement.

Enhancement / Special Features

The special feature that I did is integrating the IR Sensor, interrupt-enable bump switch functions and the motor movement functions. The logic behind creating this feature is to test whether the robot can avoid any incoming obstacle using the IR sensor readings and react accordingly. As there are some limitations to how IR sensors detect obstacles. Such as having obstacles that are too low for the IR sensors to detect, the bump switch function would then help to resolve such limitations.

Implementation (IR sensor)

For the implementation of IR sensor, the methods mainly used were SensorRead_ISR() and ADC0_InitSWTriggerCh17_12_16().

ADC0_InitSWTriggerCh17_12_16(void)

This method initialises the IR sensors ports to their respective channels. Left sensor P9.1 is configured to analog mode on Channel 16. Middle sensor P4.1 is configured to analog mode on Channel 12. Right sensor P9.0 is configured to analog mode on Channel 17.

SensorRead_ISR(void)

In this method, the raw ADC values of the 3 channels are passed into a low pass filter. The filter attempts to remove some of the noises and improve precision. The respective low pass filter methods are as follows: ***LPF_Calc(uint32_t newdata)*** for Channel 17 values, ***LPF_Calc2(uint32_t newdata)*** for Channel 12 values, ***LPF_Calc3(uint32_t newdata)*** for Channel 16 values.

After this, the filtered digital signals are passed into ***LeftConvert(int32_t nl)***, ***CenterConvert(int32_t nc)*** and ***RightConvert(int32_t nr)*** to produce the estimated distance measured by each IR sensor. Inside each of these covert methods, it will process the ADC values by parsing it through some filters. The filters are derived via a piecewise algorithm which gives us a more accurate conversion of the ADC values to mm.

Implementation (Enhancement / Special Feature / Integration)

Before the start of our routine, the method would initialize the necessary ports for our IR sensors and bump switches. After this, the program execution will enter our routine.

At the start, the robot would first scan if there are any obstacles directly in front of it. The robot would only consider the obstacle a “threat” when it is within 4cm to 20cm of its proximity.

Due to the limitations of the IR sensor, the lowest distance it can detect accurately is approximately 4cm. As the robot is constantly moving, the rate of the IR scanner has troubles keeping up with its reading, hence the window is widened to 20cm. The average distance that the robot would detect an obstacle is about 10cm.

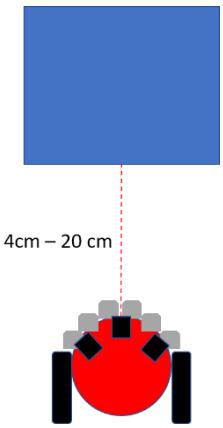

Figure 2.1 IR Detection

If an obstacle were to be detected, the robot would start turning left slowly. At this point, the robot would start to scan with its 3 IR sensors. The condition for the robot to detect a clear path is when the left and middle IR sensors detect a 10cm clearance and the right IR sensor detects a 6cm clearance. The reason for the right IR sensor to have a lower clearance distance is because of how the robot is rotating. As previously mentioned, the robot is rotating left, this in turns makes the distance of the robot from the right to the obstacle closer to each other. If the right IR sensor clearance was the same as the others, it may cause the robot to turn more than needed. As the main goal is to ensure the robot makes minimal redundant moves.

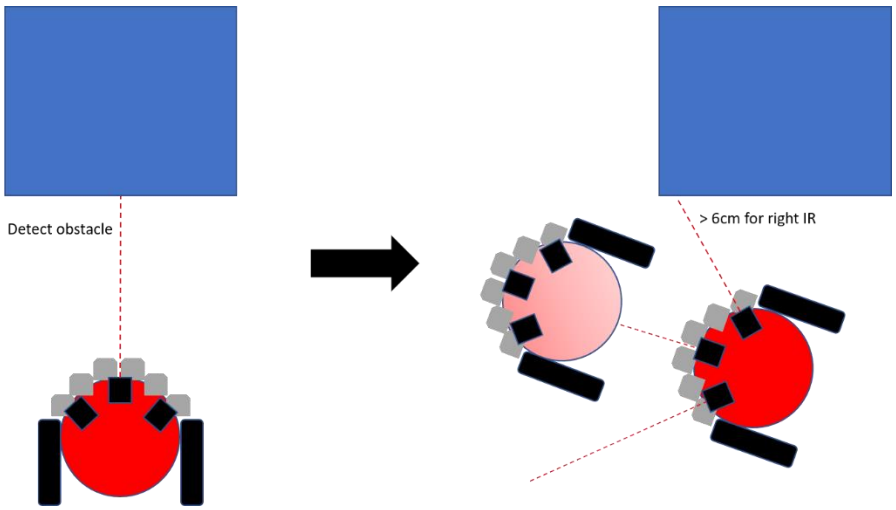

Figure 2.2 Obstacle avoidance routine

As for the 10cm clearance, it is to ensure that the robot has enough clearance to start moving instead of just moving a little and scanning for another output again.

As mentioned previously, the bump switches are only activated when the IR sensors cannot detect the low obstacles. In addition, since the field of vision for the IR sensor is not wide, the bump switches can help omit such blind spots of the IR sensors. The bump switch will react accordingly to the evasive algorithm that was written previously in lab 3.

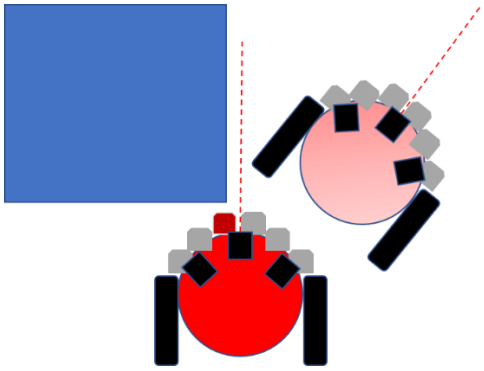

Figure 2.3 bump switch collision detection

In conclusion, this enhancement has integrated some functions to help the robot have a better movement routine. From this implementation, we can see how each function helps one another's limitations. For example, bump switches were used to solve the issues whereby the IR sensor could not detect incoming obstacles that are outside of its detection area. Also, how we can fine-tune the settings for each evasive moment corresponding to each bump switch. To sum up, this movement routine's goal was not only to evade obstacles but also to simulate a mini POC (proof of concept) of a vacuum robot with IR sensors and bump switches.

Appendix

1. 1 Code Snippet

Bump_Read(void)	https://github.com/ychoux/CE2007/blob/master/Bump_Read.c *Attached file is the code snippet for bump read where it converts the 8-bit result to 6-bit result*
MotorMovt(void)	https://github.com/ychoux/CE2007/blob/master/MotorMovt.c *Attached file is the code snippet for evasive algorithm.*
PeriodMeasure0 (uint16_t time)	https://github.com/ychoux/CE2007/blob/master/PeriodMeasure0.c *Attached file is the code snippet of the tachometer calculation.*
SensorRead_ISR(void)	https://github.com/ychoux/CE2007/blob/master/SensorRead_ISR.c *Attached file is the code snippet for the reading of IR sensor port values.*
LPF_Calc (uint32_t newdata)	https://github.com/ychoux/CE2007/blob/master/LPF_Calc.c *Attached file is the code snippet for the filtering of noise and improves precision.*
LeftConvert(int32_t nl)	https://github.com/ychoux/CE2007/blob/master/LeftConvert.c *Attached file is the code snippet for one of the three conversion for ADC to mm.*
Special_Feature()	https://github.com/ychoux/CE2007/blob/master/SpecialFeature.c *Attached file is the code snippet for the implementation of the enhanced feature.*