# Handling robot whiskers by interrupt

## Introduction

There are several ways to handle the whiskers of the xBot, or any other robot with two whiskers. We use Arduino environment, but as you know Arduino is built on top of C. Using the C instructions and not the *Arduino functions* allows to be much more efficient, and there are not so many new things to understand.

## Declarations

Whiskers are active low, wired on pins 2 and 3.

```
   #define WL  2
   #define WR  3
```

The set-up will define these two pins as input
If one need to take a decision if the left whisker is depressed, the usual way is to read the whisker's status in a variable, and then test it:

```
   int  statusWL ;
```

And in the program

```
  statusWL = digitalRead (WL) ;
  if (statusWL == 0)  // whiskers are active low
    { do this and this ; }
```

There is indeed no need to define a `statusWL` variable, and the `!` sign is useful to invert a condition. This mean that in place of `if (statusWL == 0)` one can write

```
  if (digitalRead (WL) == 0)
```
or
```
  if (!digitalRead (WL) == 1)
```
or the simplest
```
  if (!digitalRead (WL))
```

## Functional naming

A next improvement for clarity is to define names for the whiskers state "pressed".

```
  #define WhiskLOn  !digitalRead (WL)  // on mean depressed
  #define WhiskROn  !digitalRead (WR)
```

Better names are

```
  #define ObstacleLeft  !digitalRead (WL)
  #define ObstacleRight !digitalRead (WR)
```

In the program loop, you will read

```
  if (ObstacleLeft)
    { do this and this ;  }
```

This is the way programs should be written. There is a definitions part that take care of the hardware (pin number, active low or high). The program must only refer to names that express the functionality, and ignore hardware details.

## C declarations

`digitalRead ();` is an Arduino function you will not find with other C compiler, if you use a different processor, or an Arduino AVR board with AVR studio.
Pins 2 and 3 are on PORTD/INPB, bits 2 and 3 on Arduino compatible boards. One can write

```
  #define ObstacleLeft !bitRead (PIND,2)
```

and there is no change in the program. Note that the `bitRead` is 10 times faster than the digitalRead (0.5 us in place of 5 us) and it takes 100 less bytes in memory.

## Debounce

Switches bounce a random amount of time, maximum 20ms. This is not a problem usually with whiskers, because the action starts at the first bounce and lasts long enough for the whisjer being back in place. With switches, it im important to filter the bounces, using a delay of 20ms between readings.

## The need for interrupts

Think to a program in which the task of the robot is to move toward an objective. The program is concerned about handling a lot of informations and if you ask it to check every 20ms if there is an obstacle (or less if the robot is moving fast. Think to a blind person, all its attention is moving his cane. We are lucky to look at the scenary and if there is an obstacle, we switch to an avoidance task and continue our way. What we need with our robot is to have the main program being interrupted when there is an obstacle.

## Interrupt - Arduino

On any microcontroller, there are pins that can trigger an interrupt. That is the program will be interrupted, an adequate task will be executed, and then the program will resume.
Arduino has a mechanism to handle interrupts from pins 2 an 3, and other pins depending on the board. It is very confusing, sometimes one need to refer to interrupt number, 0 and 1 associated to pins 2 and 3 (or 3 and 2!), sometimes to pin number (check on the Web). We handle here the Due software behaviour. A complex program is inserted with your code when you write `attachInterrupt(interrupt source, function, mode).`

`interrupt source` : Arduino Due ***pin***, other boards the *number* of the interrupt

`function` : the function to call when the interrupt occurs; this function must take no parameters and return nothing.

`mode` : defines if the interrupt should be triggered by `LOW CHANGE RISING FALLING` action.

Lets take an example to use that tricky Arduino “facility”. When the left whisker is touched, that is when the signal goes to low, the robot must back-up and turn, this is done by the procedure DoObtacleLeft, when LOW. Easy to program but notice you should not put a delay () in that procedure if you do not want to disturb the main program job.

What will be seen in the program is

```
int  WL 2  // Due
void setup()
{
  pinMode . . .
  attachInterrupt(WL, DoObstacleLeft, LOW);
}
void DoObstacleLeft ()
{
   // Back, etc
}
void loop()
{
   // just ignore the whiskers and their beaviour
}
```

One important point if you are not familiar with C and interrupts, the variables used in the `DoObstacleLeft` procedure must include the prefix `volatile`, this causes the compiler to use RAM instead of a storage register.

## Interrupt – C

If you can read AVR documentation, programmining the INT0 and INT1 on pin 2 and 3 is not so complex and will be much more efficient than the `attachInterrupt` procedure.
We propose here another approach that use the timer2 to handle a cascade of interrupts and be independent on the processor and Arduino. Every processor has a timer and program examples to trigger an interrupt every several micro or milliseconds.
You have to read and test [www.didel.com/C/Interruptions.pdf](www.didel.com/C/Interruptions.pdf) . It provides a frame work to take care of the whiskers and behaviour every 20ms, just by calling our function `DoObstacle` ();

What must be done every 20ms is
- test if a whisker is active
- if yes, activate a flag

- if the flag is active, start a state machine that will define the avoidance procedure
- deactivate the flag

If you are not familiar with `switch case` instruction and state machines, see www.didel.com/C/Instructions2.pdf or search on the Web.

Interrupts must always take a minimum of time, this mean they must be programmed in C and not use Arduino functions. If you need to back-up for 1 second, you start the motor back, you count up to 50 in the 20ms interrupt before you stop the motor. Then you specify the next `case`. It is easy to program tricky behaviours, e.g. if you have a whisker in the back, when you count going back, you can test that whisker is touched and start another case.
Lets do the example corresponding to the state diagram right. Cases are numbered by the value given to the `state` variable.

```
  volatile byte del20 ;  // max 5 seconds
  byte state = 1 ;
  // adjust obst avoidance timings here
  #define TimeBack = 50 ; 1 sec
  #define TimeTurn = 20 ; 0,4 sec

switch (state)
{
case 1 :
  if (DoObstacleLeft) state = 10;
  if (DoObstacleRight) state = 20;
  break;
case 10 :
    del20 = TimeBack;  GoBack () ;
    state = 11 ; break;
case 11 :
    del20-- ;
    if (!del20)
    state = 12 ; break;
case 12 :
    del20 = TimeTurn;  TurnRight () ;
    state = 13 ; break;
case 13 :
    del20-- ;
    if (!del20) state = 14 ; break;
case 14 :
    GoForward () ;
    state = 1 ; break;
case 20:
    . . . similar as before
}  //end switch
```

You have understood that `GoForward (), GoBack (),TurnRight () ; TurnLft() ;` are functions you have defined according to your knowledge of motor control. It could be a single function with two signed parameters.

The next step is to test this module and give it the shape of an included file or a procedure. It will be called every 20ms by the interrupt, we can test in a loop with 20 ms delay. This allows to test also what has to be put in the set-up.

The last step is to insert the module in the interrupt procedure, with an empty loop as the main program.

Step by step, you have built a reliable obstacle avoidance module you can easily improve by adding cases, and you can work on the main program, knowing it will only be interrupted for 10-20 microseconds every 100 microseconds or 20ms depending on your interrupt routine.

mw+jdn 1308