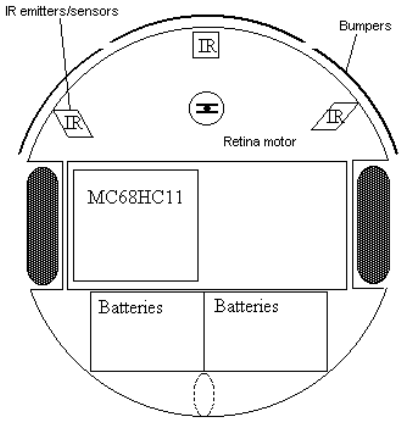
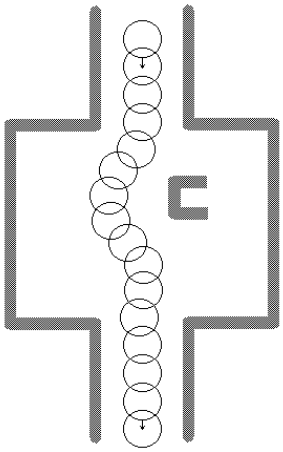
University de Aveiro

Electronics and Telecommunications Department

MOBILE ROBOTICS

"WHISPER - A mobile robot platform for surveillance purposes"





Students: António Branco & Pedro Kulzer
Course: Mobile Robotics
Professors: Keith Doty & Scott Jantz

21rd July, 1995

CONTENTS

ABSTRACT

The robot discussed in this report is intended to be used as a surveillance robot and programmed to avoid obstacles, follow light or follow movement using basic optic flow concepts. An adaptive obstacle avoidance behavior, allows the robot to adapt to new environments, as well as enabling it to explore into large or small rooms. It is not intended that the robot covers the entire room when passing through it; it is only needed that it can wander about the rooms freely and in a coarse manner.

Using both the avoidance and follow light behavior, allows the robot to move into brighter areas, enabling it to detect rooms with light. This presents some problems concerning unforeseen reflections.

Using both the avoidance and follow movement behavior, allows the robot to move into non static bright areas, enabling it to detect dynamic bright areas where a possible intruder may be present.

EXECUTIVE SUMMARY

The robot described in this report, intends to serve as a surveillance robot only. Its main behaviors are *following light*, *following movement* and *avoiding obstacles*. The robot uses IR sensors to avoid obstacles, together with an adaptive algorithm for adjusting each sensor's threshold. There are three contact switches (bumpers) at the front of the robot, which are used for collision detection. This happens due to blind spots or to behavioral responses relative to the adaptive behavioral algorithm approach which allows the robot to get bolder when it finds itself almost trapped. It's intended that the robot, wanders around the rooms and adapts to new environments, because doing so it can monitor any movement or light source and sound the alarm for an intruder. We do not want at all to have the robot covering the entire room or anything like it, we just want it to "search intruders from the distance", efficiently. The light sensing is done by using a spinning motor's retina, which gives a light image of the surrounding environment. The movement detection is done using very basic optic flow concepts like detecting simple brightness changes above a certain threshold, by comparing two consecutive images from the robot's image retina. The obstacle avoidance behavior, does well when changing from one environment to a different one, for example when going from a highly reflective environment to a less reflective one or the opposite, or when going through doors and narrow passages in general. Nevertheless the robot can get "claustrophobic", when it is put in a very confined environment, but because this is a particular environment, not likely to happen in a surveillance patrol, we find this emergent functionality understandable. All that would happen is that the robot gets quite confused and keeps bumping on the walls, eventually getting out of the situation after some time. Nevertheless, it is extremely difficult get this robot trapped because its adaptive threshold tends to keep it away from narrow passages.

It was essential to add noise to the decision of the direction to turn to, when the robot had an obstacle right in front of it, because without it the robot sometimes got locked in closed paths. It was also essential to maintain this direction until the situation changes, or else it would not ever decide where to turn to.

We accomplished smooth changes in the motors speed, using dynamics (attractors) for desired speed for both motors, as well as adding up the contributions of the several sensors instead of using a *winner-takes-all* method. Furthermore, this addition depends on the relative strengths of each sensor's contribution. For the threshold adaptation we used a similar dynamics as well.

Using both the avoidance and follow light behavior, allows the robot to move into brighter areas, enabling to detect rooms with light. This way the robot can go to areas where there may be intruders, fire or forgotten lights on. The robot functions well with these two behaviors, going from darker areas to brighter ones and keeping itself there. Nevertheless, we had to put ourselves in the robot's place, because

it sees unforeseen reflections on walls and many light sources, which may lead us to think that the robot is not behaving well, when in fact it's only behaving according to its own sensing. We tend to see what is on our level of vision and forget these things quite easily.

Using both the avoidance and follow movement behavior, allows the robot to move into non static bright areas, enabling it to detect dynamic bright areas, for example enabling the robot to detect moving shadows, moving light sources, opening of doors, etc. To combine these two behaviors, we had to use them one at a time. This had to be done because the optic flow algorithm didn't compensate for the robot's movement, so we had to stop the robot in intervals of 5 seconds (disabling the avoidance behavior), in order to compute a correct optic flow and acting accordingly.

As for the report itself, we describe in a very superficial way all the details that are not very important for the project itself (e.g. microcontroller hardware and setup, wiring for Sharp sensors, etc.) since it would be redundant, whereas we give a detailed description about the extra work and circuits made to get the robot to do what it was supposed to.

INTRODUCTION

For this surveillance robot, the main purpose is only to wander around the corridors, in order to detect intruders, using a light or movement following behavior. It is not desired to have the robot following walls too closely or having all the space covered, since the main purpose is just to “look around” for motion or light sources, keeping as far as possible from all walls. The robot keeps itself faraway from walls by having the capability of adapting its behaviors, in order to be able to enter and explore narrow spaces and to adapt easily to differently reflective environments.

Earlier work done with Dr. Gregor Schöner [1], in the field of target acquisition and obstacle avoidance through optic flow of the retina image, inspired us to use some of the basic dynamic equations techniques for the motor speed control and adaptive threshold process. The adaptive threshold was also strongly inspired through another work in neural networks [2], where we studied the mechanisms and dynamics of single neuron learning and habituation.

In surveillance robots, we expect to get an overall behavior which permits an efficient coverage and exploration of the entire space where it should wander about, without the problems of getting indefinitely stuck in particular confusing situations that may occur seldomly.

INTEGRATED SYSTEM

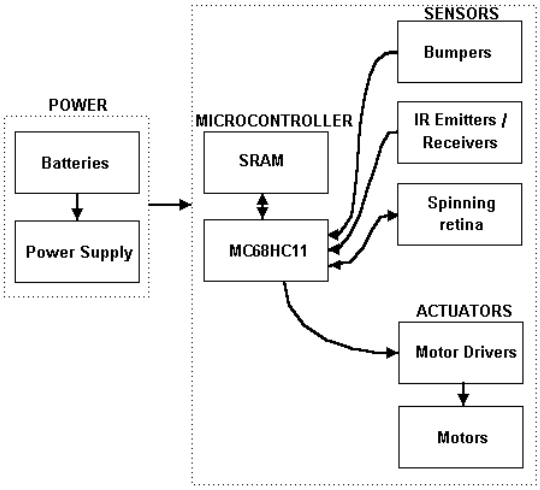


Fig. 1 - Block diagram of the complete system. The batteries feeds the power supply electronics which in turn will give a stabilized voltage to all the microcontroller, actuator and sensor circuitry. Only the motors are directly fed by the batteries.

The IR emitters emit light which is reflected by the surrounding obstacle surfaces. The amount of reflection is received by the IR receivers which communicate it to the microcontroller. With this information, the program can decide what behavior to acquire in order to avoid those obstacles.

In case of collisions against the obstacles due to blind spots or specific behavioral responses, the bumpers transmit this information to the microcontroller which will take the appropriate actions.

There is also a spinning retina, i.e. a motor driven mirror that reflects all directions of an horizontal plane into a phototransistor. This way, we get a linear representation of the surrounding light levels on that plane. This is also used to calculate a very simple optic flow. The corresponding DC motor is controlled by external electronics in a simple fashion.

The microcontroller actuates the motor drivers which in turn will drive the appropriate power to the motors.

The heart of the whole system is the MC68HC11 microcontroller platform. This controller has an external 32Kbyte SRAM mapped into the upper half memory area. The robot is powered by 8 AA NiCd batteries. All the memory, drivers, decoders, etc. circuits were done using classic schematics already available. This is the simpler part of the whole project and does not require further comments. For more details, please refer to the 68HC11 data manuals and technical references [3], [4], aswell the 6.270 robot builder's guide [5].

MOBILE PLATFORM

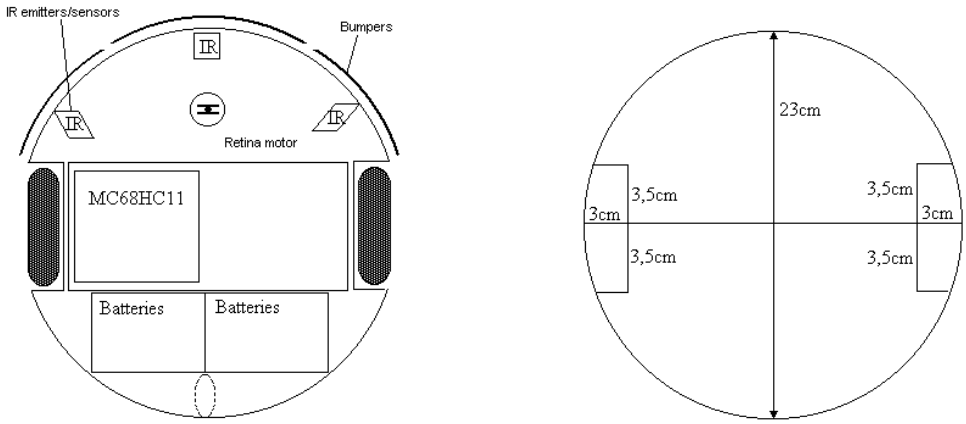


Fig. 2 - Upper view of the wooden mobile platform of the robot. On the left, we can clearly see the two wheels and the caster wheel which support the platform. The shape is round for perfect adaptation to the overall wheels motion and to avoid getting stuck on a sharp edge. This also allows the robot to spin over itself without getting stuck on nearby obstacles. The wheels are inside the outer circle of the structure, to avoid getting stuck by the outstanding wheels. The batteries are put on the rear to exert weight on the caster wheel, leaving at the same time a free front part for sensor applications. The microcontroller board and all the additional electronics are supported by four screws in the center of the platform. All the normal components were used: battery holders, board distance screws, hot glue, etc.

SENSORS

The following table shows all the available sensors and their properties. We have installed three IR sensors at the front of the robot (at the left, center and right) to provide proximity detection so that the robot can avoid obstacles. We have also installed three contact switches (bumpers) on the front which enable the robot to successful recover from collisions due to blind spots or specific behavioral responses (like boldness increase). There is also a spinning mirror which allows the robot to take sampled retinal images from all the way round itself, for optic flow and light source detection purposes.

Sensor Type	Function	Location	Count
Infrared (IR) Sharp sensors	Proximity detection of obstacles	Front	3
Bumper Contact Switches	Collision Detection against obstacles	Front	3
IR Photo Transistor	Optic Flow	Front	1 (with a motorized spinning mirror)

Tab. 1 - Sensor suite.

INFRARED SHARP SENSORS

There are three IR sensors (left, center and right). These sensors are completed by corresponding IR LED's which are mounted on top of them to illuminate the front of the robot. These 3 IR LED's, each one in the same direction as the corresponding sensor, allow the IR sensors to receive their reflected light due to obstacles in the different directions.

BUMPER CONTACT SWITCHES

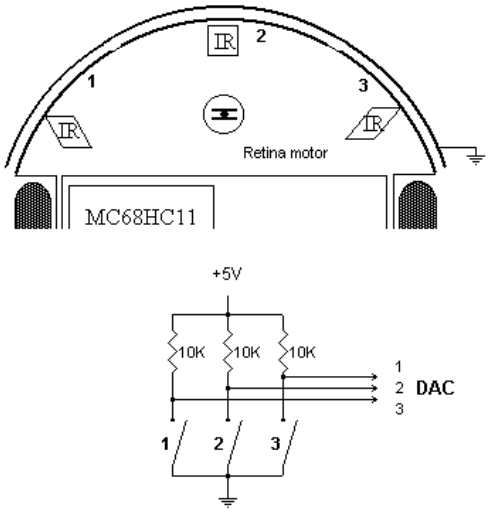


Fig. 3 - This is the mechanical layout of the bumpers. On the front of the platform, we have a strip of aluminum foil which is connected to the common electrical ground. On the inside part, we have three separate aluminum foils which connect each to an analog input of the 68HC11. All these three bumper switches have pull/up resistors which allow two levels of voltage: 5V (logic 1) if no bump and 0V (logic 0) if a bump occurs.

These bumper contacts are intended to aid the obstacle avoidance algorithm in the case where one or more thresholds become too large, which causes the robot to collide against obstacles.

IR PHOTOTRANSISTOR - SPINNING RETINA

This is an IR phototransistor which receives reflected light from a fixed speed rotating mirror. This mirror reflects the light coming from all directions within an horizontal plane at the robot's level of vision, which allows the robot to record a retinal image just like a linear 360º CCD array camera.

FIRST APPROACH - IR SYNCHRONISM

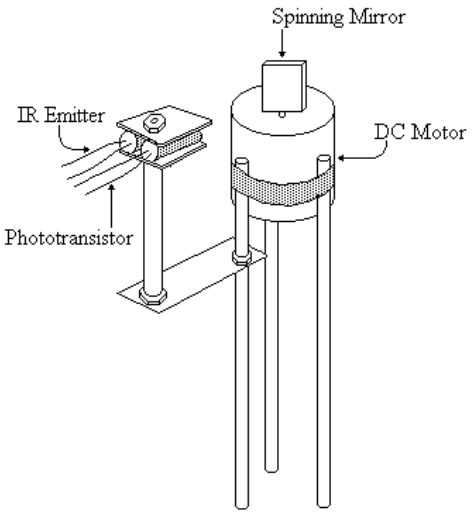


Fig. 4 - Here we can see the details of the linear CCD-like image capturing retina mechanism. The motor spins with a feedback controlled fixed velocity value, which allows then to sample the mirror reflected light onto the phototransistor by means of software. The IR emitter serves for giving a reference high-intensity light spot for the sampling program. Whenever the mirror reflects the IR emitter light directly into the phototransistor, a saturating output voltage is obtained, indicating the corresponding "zero-angle" reference passing point. From this point forth, the sampling program can start to record light values at even spaced angular displacements. Note that the mirror is two-sided, i.e. the resulting retinal image pixels will be "rotating" at twice the speed of that of the motor. Each half-rotation will originate one synchronism pulse and will trigger one complete retinal sampling process. This approach didn't show up to be a very good choice, because the synchronism peak spread over the retina samples inducing false peaks, and creating a distorted images. Nevertheless, it was a good starting point to test the implemented sampling interrupt routines.

The motor must have a constant velocity below the maximum allowable value of the CPU's processing speed, which will be actively controlled by the appropriate electronics. We must also have some circuitry to extract the synchronism impulses and to generate corresponding sample instant pulses. The electronics and the corresponding process and interrupts code needed for this part, is as follows:

ELECTRONICS

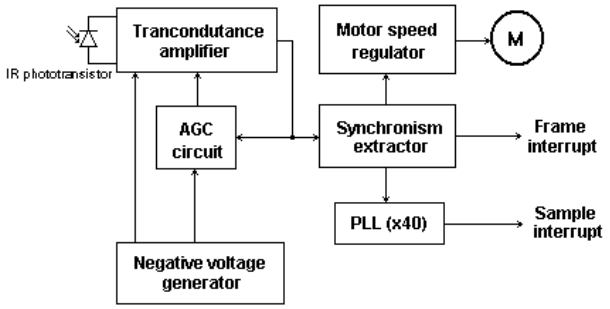


Fig. 5 - Block diagram of the spinning mirror sensor's associated electronic circuitry. This circuitry intends to amplify the IR signal (with automatic gain control to adapt the gain to different mean values of the ambient light), extract the synchronism pulses and to generate the multiple sample instant pulses between those synchronism ones. This last job is well done by a simple digital PLL. The negative voltage generator feeds the amplifiers and the AGE control circuit.

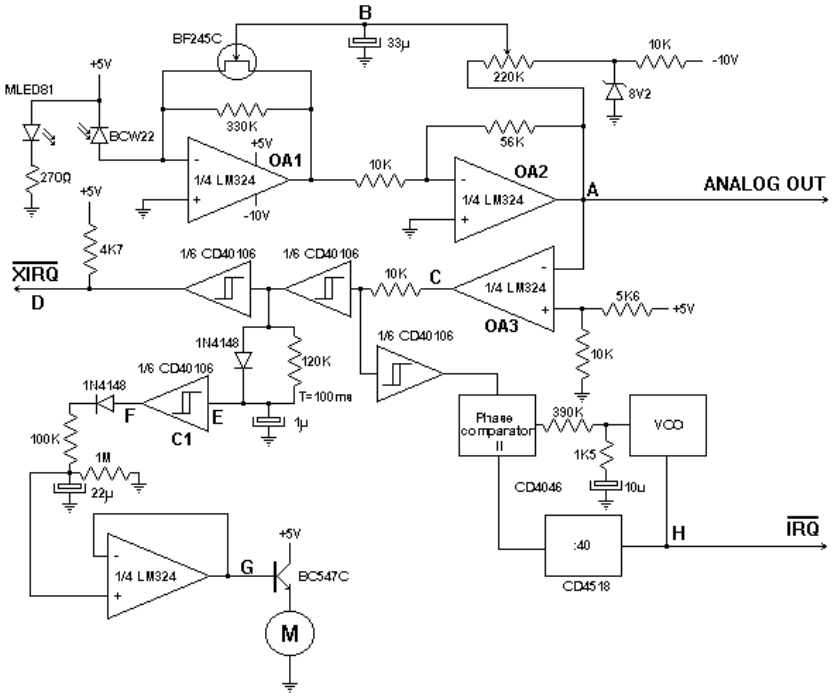


Fig. 6 - This is the circuit that corresponds to the diagram shown before. The IR phototransistor feeds a simple transcondutance amplifier (OA1) to allow a very small input impedance (lower sensitivity to external noise), which has a big resistance value in parallel with a FET transistor. This FET maintains a smaller resistance value, in order to allow an automatic gain control mechanism which holds the output values in some non-saturating range. This enhanced the total dynamic range of the IR light levels on the spinning mirror. The AGE control voltage is obtained at the output of a voltage amplifier (OA2 - an increase on this voltage will permit the FET to conduct more, hence have a smaller resistance, which leads to a smaller transcondutance gain and the output will be lower, and vice-versa). This output is the analog voltage applied to one of the ADC's of the 68HC11. Since this voltage contains the synchronism pulses, we extract those pulses by comparing (OA3) the analog output with a sufficiently big voltage reference. These pulses are going to trigger the non-maskable interrupt XIRQ each time the mirror sweeps 180° around (360° of the image). A PLL will multiply the frequency of these pulses by 40, which will give the interruption rate at the IRQ of the 68HC11, where a software routine samples an ADC value at each one. Hence, we will have 40 samples around 360° of the retinal image which showed to be sufficient for our purposes. Furthermore, Schmitt-Trigger comparator C1 "compares" the incoming frame pulses (one at each 360° retinal image frame) with its own RC delay (120K*1µ ⇒ 100ms) and gives correctional pulses whenever the mirror is spinning slower. It is easy to see that these very narrow correctional pulses will always be present since this circuit forms a closed loop with the motor and those correctional pulses will be the constant "error signal". A heavy low-pass filter feeds a power voltage follower which in turn feeds the motor. Experimental results showed that this circuit holds the motor speed at a constant and very stable value.

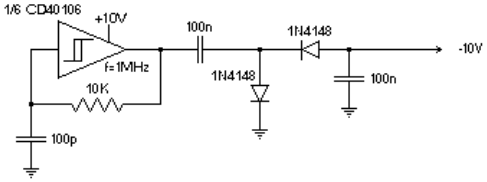


Fig. 7 - This simple circuit generates a -10V (negative) voltage of the 10V battery pack voltage. It is intended to feed the operational amplifiers and the AGE circuit shown in the previous figure. It can only draw a very small current which showed to be more than sufficient to get at least about 7-8V to the circuits which load this circuit.

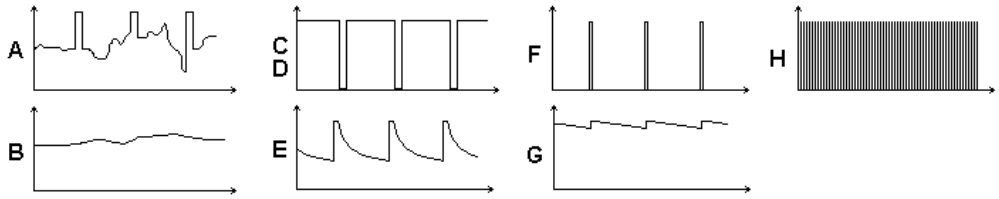


Fig. 8 - Here we can observe the waveforms present at the points of the main circuitry above. (A)-Ambient light signal. (B)-AGE FET gate voltage. (C) & (D)-synchronism pulses. (E)-Pulse width comparison Schmitt-Trigger input. (F)-Correctional motor pulses ("error signal"). (G)-Motor drive voltage. (H)-Sample instants pulses.

SECOND APPROACH (ACTUALLY USED)- SHAFT ENCODER SYNCHRONISM

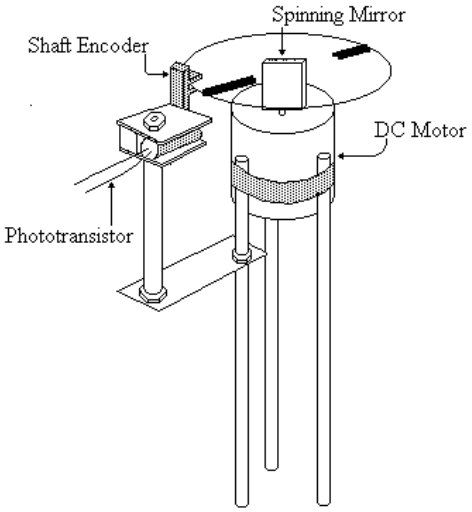


Fig. 9 - In this second approach, a shaft encoder was used to obtain the synchronism pulses independently from the ambient light retinal image. In other words, the synchronism circuit is independent of the acquisition one, enabling a good image of the surrounding environment without any interference from the other circuit. A quick view at the oscilloscope signal showed that the retinal image was much different than the one obtained with the first approach. Now, the signal was clean of false peaks and noise and showed a nice and sensitive relation to the outside light levels and directions. This was the final and chosen approach. Further experimental results (behavioral responses) showed that this approach gave much better and expected responses of the robot, whereas there were major problems with this aspect. Another problem that the first approach had, was that the synchronism extraction circuit would fail at too high levels of ambient light aswell for strong peaks of concentrated light. This was because the AGE either tries to compensate the high level of ambient light, therefore lowering the synchronism IR below threshold levels, or tries to enhance low ambient light level, therefore saturating at concentrated light beams.

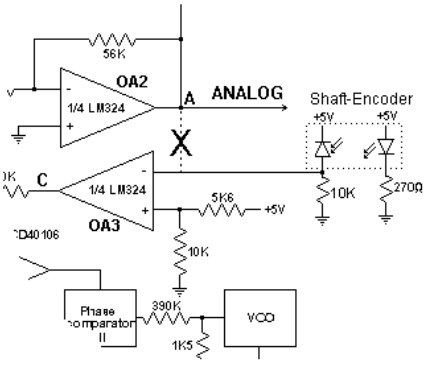


Fig. 10 - To use the new shaft-encoder synchronism source, we must cut the previous wire that fed the synchronism extracting circuit and attach a wire to the shaft-encoder IR photo-transistor. This way, a very small change in the circuitry allows us to finally use a virtually failure-free synchronism mechanism. The only drawback of this, is the fact that we have now an increased mechanical complexity at the mirror level, but it sure was worth the effort.

INTERRUPT DRIVEN SAMPLING SUBROUTINES

Now we are going to present and explain the code associated with the synchronizing and sampling processes which achieve the goal of recording an array of samples of the retinal image. The code was 100% written in assembly language to achieve maximum speed of execution and minimize interference.

```
*A/D Converter Addresses
PORT      EQU $6
ADR1      EQU $1031
ADCTL     EQU $1030
OPTION    EQU $1039

*Array Addresses
INDEX     EQU $A800
ARRAY     EQU $A801

*Interrupt vectors
IRQ_VECT  EQU $FFF2
XIRQ_VECT EQU $FFF4
```

Cod. 1 - This is the part of constant definitions. These constants are needed to make the following assembly code more readable. PORT is the port number we want to read samples from. ADR1 is the address where we are going to read the ADC sample values from. ADCTL is an ADC control register to enable and program it. OPTION is the option register of the 68HC11. INDEX is where the count of read samples is stored. ARRAY is where the maximum 40 retinal samples are stored. IRQ_VECT and XIRQ_VECT are the addresses of the vector table of the 68HC11 in the expanded mode.

```
          ORG $9000

IRQ_INTR  SEI
          LDD #NULL_IRQ
          STD IRQ_VECT
          LDD #NULL_XIRQ
          STD XIRQ_VECT
          LDAA #$10
          TAP
NULL_IRQ  RTI
```

Cod. 2 - This is the initialize subroutine, it first disables further IRQ until the RTI, using the instruction SEI. It updates the interrupt vector table with a null IRQ and XIRQ subroutine which consist only of an RTI. Finally it enables XIRQ interrupts clearing the X bit to zero in the condition code register, with a TAP instruction.

```
          ORG $9800

IRQ_INTR  SEI                 Disable maskable interrupts
          LDAA INDEX          If INDEX>39 then exit
          LDAB #39
          CBA
          BGT NULL_IRQ

*Get analog from port 6 and store in acumulator A
          LDAA #$80           enable ADC converter in OPTION
          STAA OPTION
          LDAA #PORT          Load ADC PORT number
          STAA ADCTL          Store PORT in A/D Control/Status register

READADCTL LDAA ADCTL          Read ADCTL
          ANDA #128
          BEQ  READADCTL      Test if result Valid
          LDAA ADR1           Get ADC result

*Get Array Index and store in B, final address in X
          LDX  #ARRAY         X=ARRAY
          LDAB INDEX          B=(INDEX)
          ABX                 X=X+B
          STAA 0,X            (X)=A

          INC INDEX           Increment (INDEX)
NULL_IRQ  RTI
```

Cod. 3 - This is the IRQ subroutine, which is responsible for sampling the ADC values. It first disables new IRQ until the RTI using the SEI instruction. It then checks if the number of samples are greater than 40. If the index is less than 39 it reads the ADC only after the result is valid. Finally it stores the result in the image array.

```
                ORG $A000

XIRQ_INTR       LDD IRQ_VECT      Is IRQ already enabled?
                CPD #IRQ_INTR
                BEQ DISAB_INTR  If yes, disable IRQ and XIRQ
                LDAA #0             If no, enable IRQ and reset INDEX=0
                STAA INDEX
                LDD #IRQ_INTR
                STD IRQ_VECT
                RTI
DISAB_INTR      LDD #NULL_IRQ     Disable IRQ and XIRQ
                STD IRQ_VECT
                LDD #NULL_XIRQ
                STD XIRQ_VECT
NULL_XIRQ       RTI
```

Cod. 4 - This is the XIRQ subroutine, which is responsible for starting and ending the sampling of the ADC. It starts sampling after the first synchronism signal induces a call to the XIRQ subroutine ($A000), it clears the index and updates the vector table with the address for the IRQ sampling subroutine. In other words, it enables the IRQ interrupt. If IRQ subroutine was already enabled, it stops this sampling interrupt IRQ by updating the vector table with the null subroutine for the IRQ and XIRQ, which again consist only of the instruction RTI. To start another sampling, one must update the vector table with the address for the XIRQ subroutine from a outside process (this is accomplished by the retina sensing process which runs in IC and will be further shown in the next section).

IMAGE SAMPLING PROCESS

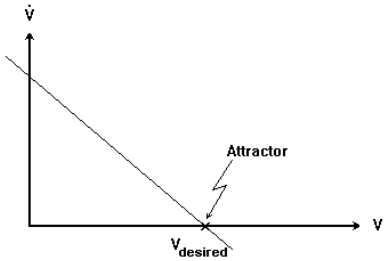
```
void RetinaSense()
{
  int I,Count;
  float Slope;

  while(1)
  {
    kill_process(PIDs[0]);          Kill all killable processes
    kill_process(PIDs[1]);
    kill_process(PIDs[2]);
    kill_process(PIDs[3]);
    kill_process(PIDs[4]);
    pokeword(0xFFF4,0xA000);    Enable XIRQ main interrupt
    msleep(250L);                    Wait sampling to complete
    Count=peek(0xA800);           Get actual samples read
    Slope=(float)Count/40.0;       Normalization line
    for (I=0;I<40;I++)               Normalize samples length to 40
      Retina[I]=peek(0xA801+Floor(Slope*(float)I));
    PIDs[0]=start_process(NosesSense());              Restart processes
    PIDs[1]=start_process(BumpersSense());
    PIDs[2]=start_process(AvoidObstacles(),50);
    PIDs[3]=start_process(FollowLight(),50);
    PIDs[4]=start_process(MotorActivation(),50);
    Wait(500);                  Wait some time until next sampling turn
    defer();
  };
}
```

Cod. 5 - First, all processes except this one are killed to avoid highly undesirable interruptions of the interrupt routines which sample the retina image one complete mirror turn (180^o) each time. After a whole turn of the motorized mirror (disable IC command-line process by means of sleeping for 200ms), the retinal image is normalized in length (if the retinal image has less than 40 samples due to some error of the PLL, the retinal image is stretched to 40 samples where the holes are filled with neighboring sample values) and transferred to an array called *retina*. At last, all processes are restarted in the same order as the initial start (this particular order will be further explained later). Just after 800ms will this process sample the retina again, to allow motion detection with optic flow. Note that the whole interruption process is triggered by a pokeword which changes the null XIRQ interrupt subroutine to the real one. At the end, this process waits for some time before starting with the next sampling turn, to get different retinal images for optic flow processing.

ACTUATION

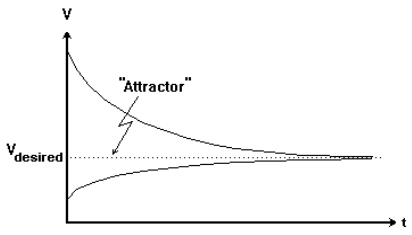
There is only one type of actuation, namely the motor speeds. There are two bi-directional DC motor drivers which feed the desired power into the motors independently according to the commands given by the microcontroller. Each motor has a speed range of -100.0% to 100.0%, where these values correspond to the maximum backward and forward speeds yielded by the motors. The power dosage mechanism is achieved by a classic PWM modulation technique usually used for DC motor drives.



Grf. 1 - General form of the dynamics used for controlling each motor's speed. We set the attractor to have the desired final speed for the motor and the dynamics forces slowly the real speed to achieve that value. This guarantees some controlled smoothness in motion changes.

The corresponding equation that yields this dynamic behavior is the following:

$$\frac{dV}{dt} = \dot{V} = V_{desired} - V \qquad \text{Eq. 1}$$



Grf. 2 - This is the resulting temporal response for the motor's speed, due to the dynamics shown previously. The speed approaches the desired attractor value describing an exponential curve.

This type of dynamics will be further emphasized when we reach the adaptive threshold mechanism yet to be described, where we also derive the programmable way of doing these things.

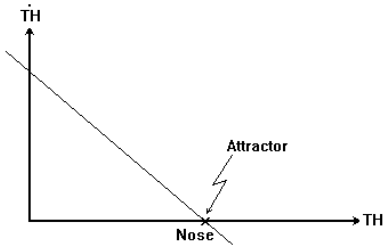
```
void MotorActivation()
{
  while (1)
  {
    if ((DV_Left_OA!=0.0) || (DV_Right_OA!=0.0))
    {
      V_Left=V_Left+(0.1*DV_Left_OA);
      V_Right=V_Right+(0.1*DV_Right_OA);
    }
    else
      {
        V_Left=V_Left+(0.1*DV_Left_LF);
        V_Right=V_Right+(0.1*DV_Right_LF);
      };
    motor(0,V_Left);
    motor(1,V_Right);
    defer();
  };
}
```

Cod. 6 - This process is the last one of the process order of execution, as it will be further explained later. The global variables suffixed by OA come from the Obstacle Avoidance process whereas the LF ones come from the Light Following process (or the alternative motion following one). As it will be emphasized in the next section, only one of the above behaviors are given to the motors. If there is no obstacle avoidance currently happening (DV_Left_OA=0.0 and DV_Right_OA=0.0), then the motors receive speed changes produced by the light following behavior. Otherwise, only the obstacle avoidance takes action. Since the robot is only going to wander about large spaces for a high percentage of the time, it is expected that the light following or motion following behaviors have a sufficient share of time to act. Mixing all together didn't show up to work satisfactorily.

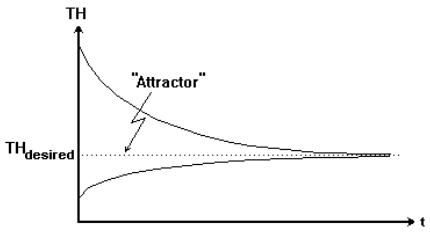
BEHAVIORS

ADAPTIVE THRESHOLD

To achieve a more efficient operation of these sensors, we introduced an automatic calibration mechanism, included in the program code which retrieves the sensed values. This corrects different illumination conditions as well eventual device tolerances between sensors. The most simple mechanism (the one we implemented) consists in always driving each sensor's threshold to the mean values of the corresponding sensor. So we will have 3 thresholds, one for each Sharp sensor. This enables the mechanism to adapt each threshold to its corresponding sensor. The adaptation mechanism consists of a very simple dynamics which can be effectively seen as a weighted mean.



Grf. 3 - This is the dynamics used for the adaptive threshold calculation, being of the same form as the one used for motor speeds. The desired value for the final threshold is always the nose's received value.



Grf. 4 - This is the resulting temporal response for a particular threshold's value, due to the dynamics shown previously. This value approaches the desired attractor value describing an exponential curve.

This dynamics can be programmed with a simple weighted average, which can be shown as follows. We want the following dynamics:

$$\dot{TH} = \frac{dTH}{dt} = Nose - TH \qquad \text{Eq. 2}$$

To achieve this, we must make approximations to get an incremental programmable solution to the differential equation:

$$dTH \approx TH_{new} - TH_{old} \qquad \text{Eq. 3}$$

$$TH_{new} - TH_{old} \approx \left(Nose - TH_{old}\right)dt \qquad \text{Eq. 4}$$

$$TH_{new} \approx \left(1 - dt\right)\cdot TH_{old} + dt \cdot Nose \qquad \text{Eq. 5}$$

Admitting that dt is a finite time increment (say 0.01), we'll have the following approximation:

$$TH = 0.99TH + 0.01Nose \qquad \text{Eq. 6}$$

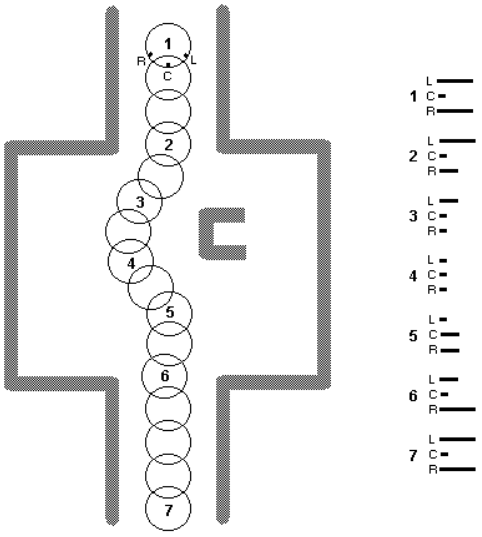
This is a weighted average of the previous threshold and the present sensor value, whose final value leads obviously to the sensor. The largest the value of the chosen time step, the fastest the threshold will be attracted towards its final value. The value of 0.015 showed to be a good experimental choice, which balances somehow the trade-off between fast adaptation (for fast get-through among crowding obstacles) and slow adaptation (for not forgetting so fast the present obstacle density).

The code needed for the periodic record of the sensors' values and for the threshold adaptation mechanism, is implemented as a process in IC[1]:

[1] Interactive C

```
void NosesSense()
{ int I;
  while(1)
  {
    for (I=0;I<=2;I++)
    {
      Noses[I]=analog(I);
      Thresholds[I]=(0.97*Thresholds[I])+(0.03*(float)Noses[I]);
    }; defer();
  };
}
```

Cod. 7 - The array *Noses* receives the current Sharp nose sensors' values while the corresponding threshold array positions are updated accordingly by means of an average (97% last value + 3% new value). Note the last *defer()* call which immediately passes control to the next process without any further time consumption and loop repetition.



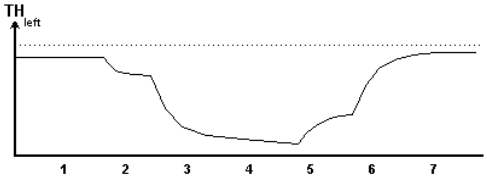


Fig. 11 - On the left side, we observe the theoretic roughly approximated threshold values' adaptations for an example robot path. As long as the robot remains along a tunnel, the lateral thresholds are very high and the robot drives ahead. The center threshold is very low since this sensor does not see anything in front of it yet. As soon as the robot approaches the obstacle in the middle of the room and goes into it, the left threshold remains high but the right one begins to lower. Since the right threshold lowered, the robot sees the rights walls sooner and avoids them adequately soon enough. This is the important feature we talked about earlier, provided by this adaptation scheme: it allows the robot to wander efficiently through a room without having to get unnecessary close to walls which would increase the path length and time consumption (at least its better than using a fixed threshold). On the other hand, it allows the robot to enter small openings as well.

Although this is not a strictly a behavior, it is a useful mechanism that actively influences the obstacle avoidance behavior. When the robot finds itself free of obstacles, or if it has enough space to wander freely, the threshold will decrease to a minimum value. When this happens, the robot will stay far away from walls, moving itself through the middle of the room. Remember that this is a surveillance robot, with visual sensing mechanism, so it doesn't need to cover all the space in the room, it just needs to have overall view, in order to detect movement.

When a robot faces a narrowing space it will increase the threshold, in order to see less obstacles and try to go through. For example, when passing through doors, after period of indecision, the robot will pass through the middle.

If the robot finds itself too enclosed by obstacles, it will further try to go through them, by increasing the threshold. If the robot is really trapped by obstacles, it will continue to increase the threshold trying to find a way through the obstacles, eventually leading to a collision.

There is an independent threshold for each sensor direction, which equally independent behavioral adaptation.

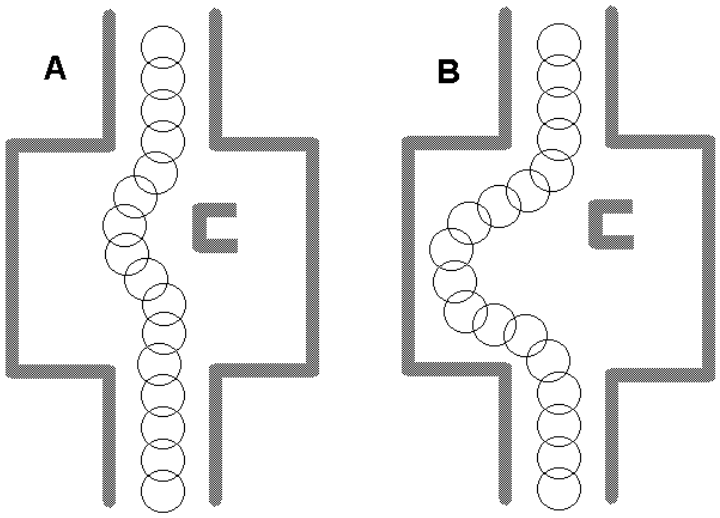


Fig. 12 - Illustration of the better surveillance performance of the approach in A. Here, we use the adaptive threshold where the robot sees far walls and corrects its path accordingly sooner. The path in B is the normally taken one for the "myope" robot. The key to the desired behavior is to let the robot always in the most sensitive state possible in each moment, desensitizing just when needed (this is accomplished by the adaptive threshold).

OBSTACLE AVOIDANCE BEHAVIOR

We implemented a low level obstacle avoidance mechanism, which consists simply in inspecting regularly, if the reflections due to distance in certain direction is above a threshold. The action taken depends directly on the direction on which the obstacle proximity was detected.

The code was again implemented as an independent IC process:

```
void AvoidObstacles()
{
  while(1)
  {
    float DV_Left_L=0.0,DV_Left_R=0.0,DV_Left_C=0.0;          Initialize all speed differences to zero
    float DV_Right_L=0.0,DV_Right_R=0.0,DV_Right_C=0.0;
    float Weight_L=0.0,Weight_C=0.0,Weight_R=0.0;             Initialize all direction weghts to zero

    if ((float)Noses[0]>(Thresholds[0]+5.0))       /* If left nose senses major reflections, then set speed corrections */
    {
      DV_Left_L=0.0-V_Left;
      DV_Right_L=-40.0-V_Right;
      Weight_L=(float)Noses[0]-Thresholds[0];
    };
    if ((float)Noses[1]>(Thresholds[1]+5.0))       /* Centre nose */
    {
      if (TurningCenter==0)                        /* Turn with a constant random value */
      {
        Rand=Random();
        TurningCenter=1;
      };
      DV_Left_C=((40.0*Rand)-V_Left);
      DV_Right_C=((-40.0*Rand)-V_Right);
      Weight_C=(float)Noses[1]-Thresholds[1];
      }
    else
      TurningCenter=0;
    if ((float)Noses[2]>(Thresholds[2]+5.0))        /* Right nose */
    {
      DV_Left_R=-40.0-V_Left;
      DV_Right_R=0.0-V_Right;
      Weight_R=(float)Noses[2]-Thresholds[2];
    };

    /*Obstacle Avoidance*/
    if ((Weight_L+Weight_C+Weight_R)>0.0)    /* If some obstacle sensed, then actuate on the global motor variables  by
                                                means of an average of the different directions sensed*/
    {
       DV_Left_OA=((DV_Left_L*Weight_L)+(DV_Left_C*Weight_C)+(DV_Left_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
       DV_Right_OA=((DV_Right_L*Weight_L)+(DV_Right_C*Weight_C)+(DV_Right_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
    }
    else              /* If no obstacle, then signal it with zeroes */
      {
        DV_Left_OA=0.0;
        DV_Right_OA=0.0;
      };
    defer();
  };
}
```

Cod. 8 - Each direction contributes with a weight that depends on the relative strength of each direction's sensed reflections. The sum of these weighted and normalized contributions will set the correctional speed differences for the motor speeds by means of global variables that are used by the motor actuation process.

COLLISION BUMPER

If the robot collides against an obstacle, due to blind spots or the attempt to pass through too closely standing objects, it will sense the contact switches and it will behave in conformity.

It can be readily seen that, if the robot is being hassled too much (too many and too near surrounding obstacles, which somehow trap the robot), it is obviously seeing obstacles very frequently, which causes one or more thresholds to go beyond the maximum value where the obstacle avoidance algorithm no longer reacts. This is seen as a “try to get out of here” at any cost, where the robot gets closer and closer to obstacles, trying to find an opening where it may pass through, since it has nowhere else to go. An extreme case happens exactly at the very moment when the algorithm no longer tells the robot to stay away from obstacles, which causes the robot to collide with them (the robot can only know how close it can get to an obstacle until it collides with it). At this point, the contact switches are actuated and the algorithm receives this extra information. The action taken is very simple and straightforward: the corresponding threshold(s) is(are) reduced by a fixed amount.

The code needed for the periodic inspection of the bumpers’ states and for the thresholds corrections, is implemented as a process in IC:

```
void BumpersSense()
{
  int I;

  while(1)
  {
    Bumpers[0]=analog(4);
    Bumpers[1]=analog(5);
    Bumpers[2]=analog(3);
    for (I=0;I<=2;I++)
      if (Bumpers[I]==0)
        Thresholds[I]=(float)Noses[I]-10.0;
    defer();
  };
}
```

Cod. 9 - As it can readily be seen, whenever there is a bump against an obstacle, the corresponding threshold (on the same side as the bump) will be immediately reduced somewhat below the corresponding nose sensor’s value. This way, we guarantee that, for almost all the cases, the robot will be instantly able to see the obstacle it collided with due to a high threshold which leaded to a temporary blindness. Note the last *defer()* call which immediately exits as in the noses process shown earlier.

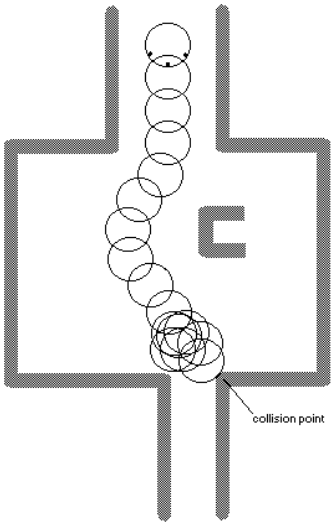


Fig. 13 - This is what happens when the robot tries to get through an opening that is too small. It gets closer and closer ("Lets see what the limits are"...) until it eventually collides against the edge ("Ooops! Definitely too small for me"). It's never too much when we emphasize again the fact that the robot is not able to know the limits on how close it can get to an obstacle without touching it. At the robot's level of "thinking", it just sees reflections, hence it can never know if the magnitude of those reflections is due to a close obstacle or due to a highly reflective but far one. It is inevitable to allow the robot to try out its limits, if we want it to pass certain situations like the one in this picture successfully.

FOLLOW LIGHT

We implemented a follow light behavior, which consist in turning the robot towards the direction which exhibits the highest light value. In order to do so, we divided to robot's retinal image in three parts, front (≈30%), right (≈35%) and left side (≈35%). If the maximum light source is at the front, the robot goes forward. If it is at the left it turns left and if it is at the right it turns right. To combine the avoid obstacle and follow light behavior without getting interference's which could seriously trash each one's task, we decided to give a higher priority to the avoid obstacle behavior. So, when there is an obstacle to be avoided, the robot behaves first by just avoiding it and not sensing any light, and only when there is no obstacle in sight the robot will behave to following the light.

We used an adaptive threshold for filtering light noise and the front blind spot, because in the absence of light sources, the robot would turn into any small light. This way, the threshold keeps the robot going in front into the light source for some time even if it does not see it (blind spot due to the IR photo-transistor) before it starts to be attracted to other smaller lights.

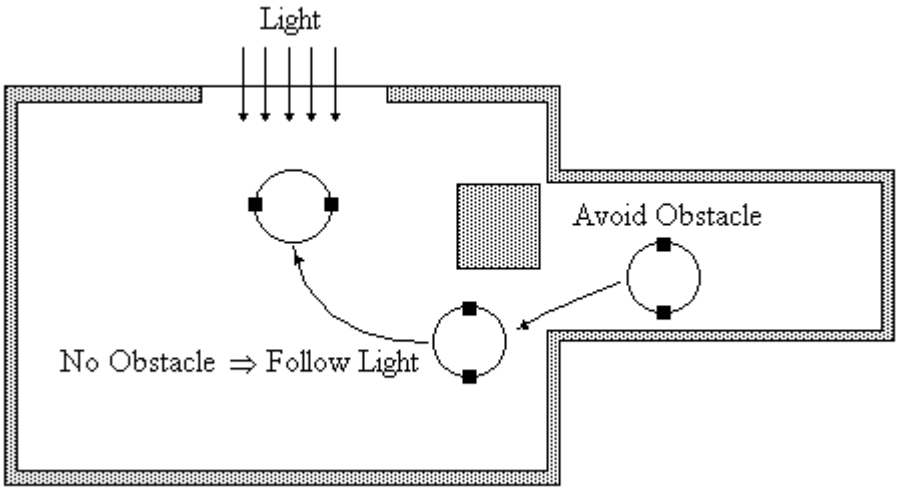


Fig. 14 - Here we can see the behavior of the robot when it encounters an obstacle and a light source. It first surrounds the obstacle, ignoring any light sources. After it does not see the obstacle any more, it starts heading towards the light source. In average, when there are many obstacles, we found out that the robot achieves almost always its goal (if it is not too much trapped).

```
int GetMax()
{
  int I,MaxLight=0,LightDirection=0;

  for (I=0;I<40;I++)
    if (Retina[I]>MaxLight)
    {
      MaxLight=Retina[I];
      LightDirection=I;
    };
  return(LightDirection);
}
```

Cod. 10 - This function returns the maximum light direction as a sample index in the retina array data structure.

```
void FollowLight()
{
  while (1)
  {
    int LightDirection=GetMax();

    DV_Left_LF=80.0-V_Left;
    DV_Right_LF=80.0-V_Right;
    if ((LightDirection>=2) && (LightDirection<15))
    {
      DV_Left_LF=-10.0-V_Left;
      DV_Right_LF=30.0-V_Right;
    }
    else
      if (LightDirection<=28)
      {
        DV_Left_LF=30.0-V_Left;
        DV_Right_LF=-10.0-V_Right;
      };
    defer();
  }
}
```

Cod. 11 - This process first sets a default forward speed for both motors. After that, if there is a light source at either one of the left and right zones, the speed dynamics force the robot to turn towards it. We had to make small corrections to the sample index values which make the frontiers between zones, due to the inaccurateness of the centering of the mechanical mirror+shaft-encoder structure.

FOLLOW MOVEMENT

We implemented a follow movement behavior, which consist in turning the robot towards the direction which exhibits the highest light changing value between to retinal images. In order to do so, we divided to robot's retinal image in three parts, front, right and left side. If the maximum light changing is at the front the robot goes forward, if it is at the left it turns left, if it is at the right it turns right. To combine the avoid obstacle and follow movement behavior we used them separate in time, disabling the avoid obstacle and enabling the follow movement with a 5 seconds interval. So every 5 seconds the robot stops,

computes the optic flow, turns into the highest movement direction and switches back to the avoid obstacle behavior.

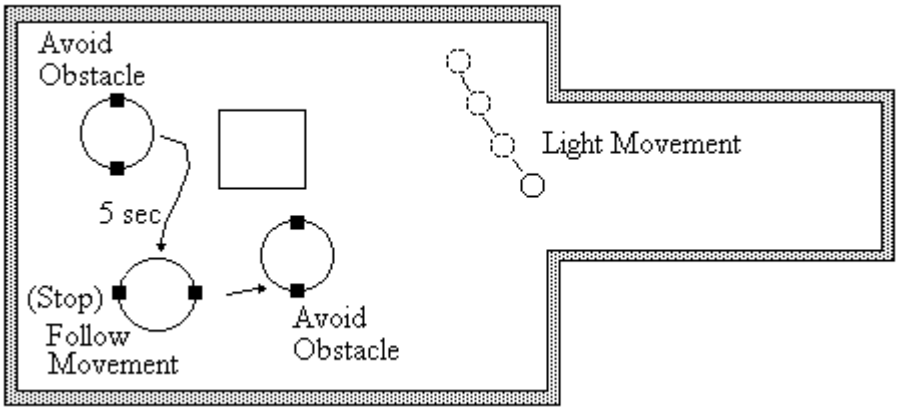


Fig. 15 - In this example we can see how the robot avoids obstacles in the usual way, stops after some time, rotates towards the movement of a light source and restarts the normal obstacle avoidance. Again, on the average, the robot will successfully achieve the motion's location.

```
void RetinaSense()
{
  int I;

  while(1)
  {
    kill_process(PIDs[0]);
    kill_process(PIDs[1]);
    kill_process(PIDs[2]);
    kill_process(PIDs[4]);
    if (ImageNumber>5)
      {
        ImageNumber=0;
        FlowReady=1;
        for (I=1;I<30;I++)                    /* Stop motors smoothly */
        {
          V_Left=V_Left+0.1*(0.0-V_Left);
          V_Right=V_Right+0.1*(0.0-V_Right);
          motor(0,V_Left);
          motor(1,V_Right);
          Wait(10);
        };
        motor(0,0.0);
        motor(1,0.0);
        Wait(200);
        GetRetina(LastRetina);          /* Get first retinal image */
        Wait(800);                      /* Wait for a moment */
        GetRetina(Retina);              /* Get second retinal image */
        OpticFlow();                    /* Compute optic flow */
        FollowMovement();               /* Follow movement if any */
      }
    else
      ImageNumber++;
    PIDs[0]=start_process(NosesSense());
    PIDs[1]=start_process(BumpersSense());
    PIDs[2]=start_process(AvoidObstacles(),50);
    PIDs[4]=start_process(MotorActivation(),50);
    Wait(500);
    defer();
  };
}
```

Cod. 12 - The only changes necessary to the retina sensing process is to stop the robot each time 5 retinal image where recorded and record two images for optic flow computation.

```
int GetMaxFlow()
{
  int I,MaxFlow=0,MotionDirection=0;

  for (I=0;I<40;I++)
    if (FlowRetina[I]>MaxFlow)
    {
      MaxFlow=FlowRetina[I];
      MotionDirection=I;
    };
  if (MaxFlow<20)
    return(0);
  MotionDirection+=5;
  if (MotionDirection>39)
    MotionDirection-=40;
  return(MotionDirection);
}
```

Cod. 13 - The code for maximum flow detection is similar to the one used to get the maximum light direction.

```
void FollowMovement()
{
  int MotionDirection;
  MotionDirection=GetMaxFlow();
  if(MotionDirection<20)
  {
    motor(0,-50.0);
    motor(1,50.0);
    msleep((long)MotionDirection*73L);
  }
  else
  {
    motor(0,50.0);
    motor(1,-50.0);
    msleep((40L-(long)MotionDirection)*73L);
  };
}
```

Cod. 14 - If the maximum motion is at the left, the robot turns left by the exact amount necessary to get itself facing the movement source. After some experiments, we found out the value 73 that sets the necessary sleep time for the motors to rotate the robot by the desired angle.

USING SEVERAL BEHAVIORS - PROCESS STRUCTURE

The whole robot program was implemented in IC (Interactive C v2.850) in a process structure. Since IC possesses a multitasking kernel which executes processes with a Round-Robin priority strategy, we must sequence the different processes in an adequate manner, as follows:

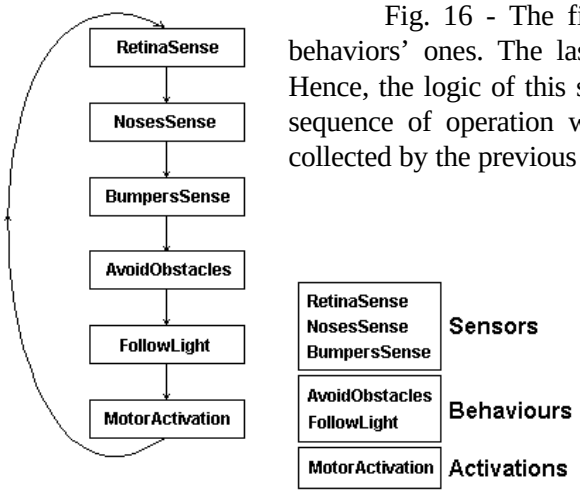


Fig. 16 - The first processes to start are the sensors' ones. Then we have the behaviors' ones. The last process being started is the one which activates the motors. Hence, the logic of this sequence is: sensors-behaviors-activations, which allows a correct sequence of operation where the next group of processes depends always on the data collected by the previous one, which are in this manner readily provided (actualized).

The code that firstly starts all processes is the following:

```
PIDs[0]=start_process(NosesSense());
PIDs[1]=start_process(BumpersSense());
PIDs[2]=start_process(AvoidObstacles(),50);
PIDs[3]=start_process(FollowLight());
PIDs[4]=start_process(MotorActivation());
start_process(RetinaSense());
```

EXPERIMENTAL LAYOUT AND RESULTS

AVOID OBSTACLE BEHAVIOUR EXPERIMENTAL RESULTS

If the robot is wandering through an highly reflective environment, and suddenly the environment changes to a less reflective one, the robot will approximate more to the obstacles. This sometimes leads robot to collide to dark objects, when in the highly reflective environment he was highly adapted to the enclosing objects. The bumpers correct this kind of temporary "blindness". This is what happens to human sight, when for example, going into a dark room after being exposed to bright light.

When going through doors, the robot "sniffs" around for a while, adapting its thresholds to the new situation, passing through it successfully.

When we tried leaving the robot in a small spaced environment, we observed many times that the robot got stuck on cyclic path, which was highly undesired for a surveillance robot. To solve this we introduced random noise in the choice of the direction to take when the center nose is activated. This in turn, caused a initial problem of indecision, because the robot couldn't decide between different successive

choices. We solved this simply by forcing it to take only one decision while the center nose was being activated.

The robot doesn't behave well in a maze or in a permanent too enclosing environment, since it starts feeling trapped and starts trying to get out bumping into the walls, we call this the claustrophobia emergent functionality.

FOLLOW LIGHT BEHAVIOUR EXPERIMENTAL RESULTS

If the robot is left in a dark room, the robot seeks light sources or reflections in walls, eventually coming out from this room and going into the bright rooms. It's interesting to notice that the avoid obstacle keeps working well, when adding-up a new BEHAVIOUR with a lower priority, because the robot keeps avoiding obstacles well and also being able to go into lighter areas.

The reflections in walls sometimes help the robot to go into the light spots, because they function like signs into the light source.

In an experiment we did, we left the robot in a dark room on it's own, relatively trapped among obstacles (bathroom). After a while, the robot was able to get out and wander about into the corridor and finally he found the window that was the main light source but in a completely different room.

We had to be very careful with the timing of the processes, because the processor was used to its limits. We also had to stop all processes and give the needed time for the analog conversion and sampling of the retinal image, which was impossible to do if the IRQ's kept being interrupted by the multitasking environment if the processes were still running at the same time.

FOLLOW MOVEMENT BEHAVIOUR EXPERIMENTAL RESULTS

When exposed to light movement, during the follow movement BEHAVIOUR the robot goes towards it. Nevertheless we found that the retinal values varied to much even though there was no apparent motion in the scenery, so it detects false movements. It behaves well when using an intense light source moving, a moving shadow or a moving reflection of an intense light at sheet of paper or a wall.

We had to stop the robot when computing the optic flow, because we didn't compensate the robot's movement in the calculus of the optic flow.

CONCLUSIONS

The adaptive bumper-aided obstacle avoidance behavior seemed to operate in a desired fashion. The robot does get claustrophobia in confined spaces such as labyrinths, which is explained by the behavioral tendency of the robot getting out or through these situations and as it can't get out easily he gets confused. This is not a fundamental problem, because the surveillance robot is not intended to work in such environment.

The main goal we intended was to have the robot adapting to new environments easily, which worked well, with the help from the bumpers for extreme cases.

The robot works well combining obstacle avoidance and follow light behaviors at the same time, going from darker to brighter areas avoiding the obstacles

The follow movement behavior and obstacle avoidance one were implemented by stopping the robot for computing the optic flow, switching from one behavior to the other, because we didn't implement a compensation for the robot's movement when computing the optic flow. Although this is a constraint, the robot follows movements of light or shadows quite effectively when the robot switched to the follow movement behavior.

For future work it would be interesting to study the possibility of combining both avoiding obstacle and follow movement, but without the switching between them.This would mean having a compensation

for the robot's movement when computing the optic flow. Another solution would be some kind of an adaptive algorithm for movement which would mean that the robot would adapt to the robot's movement.

DOCUMENTATION

[1] A.Branco, R.Costa, P.Kulzer; "Robot obstacle avoidance using a behavioural approach", Non-linear dynamical systems course, University of Aveiro 1995.

[2] P.Kulzer, A.Branco; "Character recognition using neural networks", Final project, University of Aveiro 1994.

[3] "M68HC11 Reference Manual", Motorola

[4] "M68HC11 E series Technical Data", Motorola

[5] "The 6.270 Robot Builder's Guide", MIT

APPENDIX A - MISCELLANEOUS

RANDOM FUNCTION

For the random component added for the central sensor motor speeds selection, we implemented a function returning a random value. This function can only take the following values: -1.0 and 1.0. It consists of a simple inspection on an unused analog input which receives noise from the outside. Extracting only the LSB, we get a 1 or a 0, which can be easely converted to the values specified above.

```
float Random()
{
  return(((float)(analog(7) & 0b00000001)*2.0)-1.0);
}
```

MATH FUNCTIONS

We needed two additional mathematical functions where their names say everything they do:

```
int Floor(float Value)
{
  return (int)Value;
}


int Round(float Value)
{
  if (Value - (float)Floor(Value)>=0.5)
    return (int)Value+1;
    else
      return (int)Value;
}
```

APPENDICE B - SOURCE CODE

OBSTACLE.C MODULE

This module was first developed as a standalone test for the obstacle avoidance mechanism. It does not follow at 100% all the process and other directives written down in this report.

```
float Noses[3];                 /* Left, Center and Right nose values */
float Bumpers[3];               /* Left, Center and Right bumper values */
float Thresholds[3];            /* Left, Center and Right threshold values */
float DV_Right, DV_Left;        /* Left and Right speed corrections */
float V_Right,V_Left;           /* Current Left and Right speed values */
float Rand;                     /* Random value for center turning */
int   TurningCenter;            /* Flag to hold random value when dynamics turning on center */


/* Random value function that returns either 1.0 or -1.0 */
float Random()
{
  return(((float)(analog(7) & 0b00000001)*2.0)-1.0);
}
```

```
/* Bumpers sensing process */
void BumpersSense()
{
  int I;

  while(1)
  {
    Bumpers[0]=(float)analog(4);          /* Left bumper status */
    Bumpers[1]=(float)analog(5);          /* Center bumper status */
    Bumpers[2]=(float)analog(3);          /* Right bumper status */
    for (I=0;I<=2;I++)
      if (Bumpers[I]==0.0)                /* If bumper activated, then lower corresponding direction's threshold value */
        Thresholds[I]=Noses[I]-10.0;
  };
}


/* Noses sensing process */
void NosesSense()
{
  int I;

  while(1)
    for (I=0;I<=2;I++)
    {
      Noses[I]=(float)analog(I);                                  /* Nose value */
      Thresholds[I]=(0.985*Thresholds[I])+(0.015*Noses[I]);       /* Threshold adaptation through weighted average */
    };
}


/* Obstacle avoidance process */
void AvoidObstacles()
{
  float DV_Left_L,DV_Left_R,DV_Left_C;        /* Left motor contributions from all three directions */
  float DV_Right_L,DV_Right_R,DV_Right_C;     /* Right motor contributions from all three directions */
  float Weight_L,Weight_C,Weight_R;           /* Intensity contributions from all three directions */

  while(1)
  {
    DV_Left_L=0.0;                            /* Initialize all contributions to zero */
    DV_Left_C=0.0;
    DV_Left_R=0.0;
    DV_Right_L=0.0;
    DV_Right_C=0.0;
    DV_Right_R=0.0;
    Weight_L=0.0;
    Weight_C=0.0;
    Weight_R=0.0;
    if (Noses[0]>(Thresholds[0]+5.0))         /* If left nose value higher than threshold, then activate avoidance */
    {
      DV_Left_L=0.0-V_Left;                   /* Left motor speed tends to zero */
      DV_Right_L=-40.0-V_Right;               /* Right motor speed tends to 40% reverse */
      Weight_L=Noses[0]-Thresholds[0];        /* Contribution intensity */
    };
    if (Noses[1]>(Thresholds[1]+5.0))         /* If center nose value higher than threshold, then activate avoidance */
    {
      if (TurningCenter==0)                   /* If not already turning on center, then flag the starting of behavior */
      {
        Rand=Random();                        /* Get random direction */
        TurningCenter=1;
```

```
      };
      DV_Left_C=((30.0*Rand)-V_Left);          /* Left motor speed tends to 30% or -30% */
      DV_Right_C=((-30.0*Rand)-V_Right);       /* Right motor speed tends to -30% or 30% */
      Weight_C=Noses[1]-Thresholds[1];         /* Contribution intensity */
    }
    else
      TurningCenter=0;                         /* Stopped center avoidance behavior, then clear flag */
    if (Noses[2]>(Thresholds[2]+5.0))          /* If right nose value higher than threshold, then activate avoidance */
    {
      DV_Left_R=-40.0-V_Left;                  /* Left motor speed tands to -40% */
      DV_Right_R=0.0-V_Right;                  /* Right motor speed tends to 0% */
      Weight_R=Noses[2]-Thresholds[2];         /* Contribution intensity */
    };
    if ((Weight_L+Weight_C+Weight_R)>0.0)      /* If any avoidance activated at all, then compute speed corrections */
    {
DV_Left=((DV_Left_L*Weight_L)+(DV_Left_C*Weight_C)+(DV_Left_R*Weight_R))/(Weight_L+Weight_C+Weight_R);
DV_Right=((DV_Right_L*Weight_L)+(DV_Right_C*Weight_C)+(DV_Right_R*Weight_R))/(Weight_L+Weight_C+Weig
ht_R);
    }
    else
      {
        DV_Left=100.0-V_Left;                  /* No avoidance triggered, then go straight ahead */
        DV_Right=100.0-V_Right;
      };
    V_Left=V_Left+(0.2*DV_Left);               /* Update motor speeds and actuate motors */
    V_Right=V_Right+(0.2*DV_Right);
    motor(0,V_Left);
    motor(1,V_Right);
  };
}

/* Initialization function */
void Initialize()
{
  V_Left=0.0;                  /* Initialize motor speeds to 0% */
  V_Right=0.0;
  DV_Left=0.0;                 /* Initialize motor speed corrections to zero */
  DV_Right=0.0;
  Thresholds[0]=100.0;         /* Initialize threshold values to an empiric mean ambient value of 100.0 */
  Thresholds[1]=100.0;
  Thresholds[2]=100.0;
  TurningCenter=0;             /* Reset center turning flag */
}


/* Main function */
void main()
{
  Initialize();                /* Initialize global variables */
  poke(0x4000,255);            /* Turn noses IR LED's on */
  start_process(NosesSense()); /* Start processes */
  start_process(BumpersSense());
  start_process(AvoidObstacles());
}
```

FLOW.ASM MODULE

This very important module allows a retinal image sampling cycle to take place, which is started through software from IC as explained in the text.

```
*A/D Converter Addresses
PORT      EQU $6       Analog Port Number
ADR1      EQU $1031    Result of Analog reading
ADCTL     EQU $1030    A/D Control Status
OPTION    EQU $1039    OPTION register

*Array Addresses
INDEX     EQU $A800    Retina sample count
ARRAY     EQU $A801    Retina samples

*Interrupt vectors
IRQ_VECT   EQU $FFF2
XIRQ_VECT  EQU $FFF4


*************************************************************************************************
*IRQ - Interrupt SubRotine initialization
*************************************************************************************************
```

```
        ORG $9000

        SEI                 Disable maskable interrupts
        LDD #NULL_IRQ   Initialize interrupt vectors to NULL
        STD IRQ_VECT
        LDD #NULL_XIRQ
        STD XIRQ_VECT
        LDAA #$10           Enable XIRQ interrupt
        TAP
        RTI
```

```
*****************************************************************************************************
*IRQ - Interrupt SubRotine to fill the array
*array index in INDEX
*array starts at ARRAY
*****************************************************************************************************

        ORG $9800

IRQ_INTR    SEI                 Disable maskable interrupts
            LDAA INDEX      If INDEX>39 then exit
            LDAB #39
            CBA
            BGT NULL_IRQ

*Get analog from port 6 and store in acumulator A
            LDAA #$80       Enable ADC converter in OPTION
            STAA OPTION
            LDAA #PORT          Load ADC PORT number
            STAA ADCTL          Store PORT in A/D Control/Status register
READADCTL  LDAA ADCTL       Read ADCTL
            ANDA #128
            BEQ  READADCTL  Test if result Valid
            LDAA ADR1           Get ADC result

*Get Array Index and store in B, final address in X
            LDX  #ARRAY     X=ARRAY
            LDAB INDEX      B=(INDEX)
            ABX                 X=X+B
            STAA 0,X            (X)=A
```

```
                INC INDEX       Increment (INDEX)
NULL_IRQ    RTI


*************************************************************************************************
*XIRQ - Interrupt SubRotine to reset array index at $A000
*************************************************************************************************

        ORG $A000

XIRQ_INTR       LDD IRQ_VECT    Is IRQ already enabled?
                CPD #IRQ_INTR
                BEQ DISAB_INTR  If yes, then disable IRQ and XIRQ
                LDAA #0         If no, then enable IRQ and reset INDEX=0
                STAA INDEX
                LDD #IRQ_INTR
                STD IRQ_VECT
                RTI
DISAB_INTR      LDD #NULL_IRQ   Disable IRQ and XIRQ
                STD IRQ_VECT
                LDD #NULL_XIRQ
                STD XIRQ_VECT
NULL_XIRQ       RTI
```

FOLLOW.C MODULE

Now we are going to expose the code for the light following processes and functions, which obey to the directives given in the text. There is an extra function called *EnableLED()* and which allows us to turn on three LED's on the robot: RED-Obstacle avoidance behavior currently taking over, YELLOW-Light following behavior, GREEN-Going straight ahead. This debug information was very valuable because it allowed us to observe what the robot was doing at every moment, even in those moments where it showed some strange or confusing behaviors.

```
int   Bumpers[3];
int   Noses[3];
int   Retina[40];                                   /* Retinal image pixels */
float Thresholds[3]={100.0,100.0,100.0};
float DV_Left_OA=0.0, DV_Right_OA=0.0;              /* Obstacle avoidance Left and Right dynamic speed corrections */
float DV_Left_LF=0.0,DV_Right_LF=0.0;               /* Light following Left and Right dynamic speed corrections */
float V_Right=0.0,V_Left=0.0;
int   PIDs[5];                                      /* Process ID's */
float Rand;
int   TurningCenter=0;
int   LED_Flag;
```

```
/* Function which waits for m_second miliseconds before returning */
/* Does not stop other multitasking processes */
void Wait(int m_second)
{
        long stop_time;
        stop_time = mseconds() + (long)m_second;
        while(stop_time > mseconds())
                defer();
}


/* Down round */
int Floor(float Value)
{
  return (int)Value;
}


int Round(float Value)
{
  if (Value - (float)Floor(Value)>=0.5)
    return (int)Value+1;
    else
      return (int)Value;
}


/* Random value got from analog varying input */
float Random()
{
  return(((float)(analog(7) & 0b00000001)*2.0)-1.0);
}
int GREEN_LED=8;
int RED_LED=2;
int YELLOW_LED=4;
/* Enable zero-indexed light */
void EnableLED(int Index)
{
  poke(0x4000,0b11100000+Index);
}


void BumpersSense()
{
  int I;

  while(1)
  {
    Bumpers[0]=analog(4);
    Bumpers[1]=analog(5);
    Bumpers[2]=analog(3);
    for (I=0;I<=2;I++)
      if (Bumpers[I]==0)
        Thresholds[I]=(float)Noses[I]-10.0;
    defer();
  };
}


void NosesSense()
```

```
{
  int I;

  while(1)
  {
    for (I=0;I<=2;I++)
    {
      Noses[I]=analog(I);
      Thresholds[I]=(0.85*Thresholds[I])+(0.15*(float)Noses[I]);
    };
    defer();
  };
}


/* Retina sensing process */
void RetinaSense()
{
  int I,Count;
  float Slope;

  while(1)
  {
    kill_process(PIDs[0]);          /* Kill all killable processes, except this one */
    kill_process(PIDs[1]);
    kill_process(PIDs[2]);
    kill_process(PIDs[3]);
    kill_process(PIDs[4]);
    pokeword(0xFFF4,0xA000);  /* Enable XIRQ interrupt routine which starts the retinal image sampling cycle */
    msleep(250L);                   /* Wait for a whole turn of the motorized mirror to prevent */
                                    /* the interrupts of being interrupted by IC processes */
    Count=peek(0xA800);             /* Get real sample count made */
    Slope=(float)Count/40.0;        /* Normalizing line */
    for (I=0;I<40;I++)                          /* Normalize samples to length of 40 */
      Retina[I]=peek(0xA801+Floor(Slope*(float)I));
    PIDs[0]=start_process(NosesSense());        /* Restart all killed processes */
    PIDs[1]=start_process(BumpersSense());
    PIDs[2]=start_process(AvoidObstacles(),50);
    PIDs[3]=start_process(FollowLight(),50);
    PIDs[4]=start_process(MotorActivation(),50);
    Wait(500);                                  /* Wait for some time before engaging the next sampling cycle */
    defer();
  };
}


void AvoidObstacles()
{
  while(1)
  {
    float DV_Left_L=0.0,DV_Left_R=0.0,DV_Left_C=0.0;
    float DV_Right_L=0.0,DV_Right_R=0.0,DV_Right_C=0.0;
    float Weight_L=0.0,Weight_C=0.0,Weight_R=0.0;

    if ((float)Noses[0]>(Thresholds[0]+5.0))
    {
      DV_Left_L=0.0-V_Left;
      DV_Right_L=-40.0-V_Right;
      Weight_L=(float)Noses[0]-Thresholds[0];
    };
    if ((float)Noses[1]>(Thresholds[1]+5.0))
```

```
    {
      if (TurningCenter==0)
      {
        Rand=Random();
        TurningCenter=1;
      };
      DV_Left_C=((40.0*Rand)-V_Left);
      DV_Right_C=((-40.0*Rand)-V_Right);
      Weight_C=(float)Noses[1]-Thresholds[1];
      }
    else
      TurningCenter=0;
    if ((float)Noses[2]>(Thresholds[2]+5.0))
    {
      DV_Left_R=-40.0-V_Left;
      DV_Right_R=0.0-V_Right;
      Weight_R=(float)Noses[2]-Thresholds[2];
    };
    if ((Weight_L+Weight_C+Weight_R)>0.0)
    {
        DV_Left_OA=((DV_Left_L*Weight_L)+(DV_Left_C*Weight_C)+(DV_Left_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
        DV_Right_OA=((DV_Right_L*Weight_L)+(DV_Right_C*Weight_C)+(DV_Right_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
    }
    else
      {
        DV_Left_OA=0.0;
        DV_Right_OA=0.0;
      };
    defer();
  };
}
/* Function that returns the maximum light pointing direction */
int GetMax()
{
  int I,MaxLight=0,LightDirection=0;

  for (I=0;I<40;I++)
    if (Retina[I]>MaxLight)
    {
      MaxLight=Retina[I];
      LightDirection=I;
    };
  return(LightDirection);
}


/* Light following process */
void FollowLight()
{
  while (1)
  {
    int LightDirection=GetMax();                    /* Get brightest light direction */

    DV_Left_LF=80.0-V_Left;                         /* Default corrections to straight ahead */
    DV_Right_LF=80.0-V_Right;
    LED_Flag=GREEN_LED;
    if ((LightDirection>=2) && (LightDirection<15))  /* If light direction on the left side, then turn left */
    {
      LED_Flag=YELLOW_LED;
```

```
      DV_Left_LF=-10.0-V_Left;
      DV_Right_LF=30.0-V_Right;
    }
    else
      if (LightDirection<=28)                    /* If light direction on the right side, the turn right */
      {
        LED_Flag=YELLOW_LED;
        DV_Left_LF=30.0-V_Left;
        DV_Right_LF=-10.0-V_Right;
      };                                         /* Else, go straight ahead */
    defer();
  }
}


/* Motor actuation function */
void MotorActivation()
{
  while (1)
  {
    if ((DV_Left_OA!=0.0) || (DV_Right_OA!=0.0))  /* If any obstacle avoidance in progress, use its corrections */
    {
      LED_Flag=RED_LED;
      V_Left=V_Left+(0.1*DV_Left_OA);
      V_Right=V_Right+(0.1*DV_Right_OA);
    }
    else                                         /* Else, use the light following corrections */
      {
        V_Left=V_Left+(0.1*DV_Left_LF);
        V_Right=V_Right+(0.1*DV_Right_LF);
      };
    EnableLED(LED_Flag);
    motor(0,V_Left);                             /* Actuate motors */
    motor(1,V_Right);
    defer();
  };
}


void Initialize()
{
  poke(0x4000,0b11111110);       /* Turn on LED's and motorized mirror */
  pokeword(0xFFF2,0x9000);       /* IRQ Interrupt vector */
}


void main()
{
  Initialize();
  PIDs[0]=start_process(NosesSense());
  PIDs[1]=start_process(BumpersSense());
  PIDs[2]=start_process(AvoidObstacles(),50);
  PIDs[3]=start_process(FollowLight(),50);
  PIDs[4]=start_process(MotorActivation(),50);
  start_process(RetinaSense());
}
```

FLOW.C MODULE

This module is very similar to the previous one, having the only major difference on the motion following function. Since this was the most commented module in the report text, it does not require further comments on its code.

```
int   Bumpers[3];
int   Noses[3];
int   Retina[40];
int   LastRetina[40];              /* Previous Retinal Image*/
int   FlowRetina[40];              /* Retinal Optic Flow*/
float Thresholds[3]={100.0,100.0,100.0};
float DV_Left_OA=0.0, DV_Right_OA=0.0;
float DV_Left_LF=0.0,DV_Right_LF=0.0;
float V_Right=0.0,V_Left=0.0;
int   PIDs[5];
float Rand;
int   TurningCenter=0;
int   ImageNumber;                 /* Image Number*/
int   FlowReady;                   /*  Flow Ready for behaviour*/


void Wait(int m_second)
```

```
{
        long stop_time;

        stop_time = mseconds() + (long)m_second;
        while(stop_time > mseconds())
                defer();
}


int Floor(float Value)
{
  return (int)Value;
}


int abs(int Value)
{
  if (Value>0)
    return Value;
  else
    return -Value;
}


int Round(float Value)
{
  if (Value - (float)Floor(Value)>=0.5)
    return (int)Value+1;
    else
      return (int)Value;
}



float Random()
{
  return(((float)(analog(7) & 0b00000001)*2.0)-1.0);
}


int GREEN_LED=8;
int RED_LED=2;
int YELLOW_LED=4;
/* Enable zero-indexed light */
void EnableLED(int Index)
{
  poke(0x4000,0b11100000+Index);
}


void BumpersSense()
{
  int I;

  while(1)
  {
    Bumpers[0]=analog(4);
    Bumpers[1]=analog(5);
    Bumpers[2]=analog(3);
    for (I=0;I<=2;I++)
      if (Bumpers[I]==0)
```

```
        Thresholds[I]=(float)Noses[I]-10.0;
    defer();
  };
}


void NosesSense()
{
  int I;

  while(1)
  {
    for (I=0;I<=2;I++)
    {
      Noses[I]=analog(I);
      Thresholds[I]=(0.8*Thresholds[I])+(0.2*(float)Noses[I]);
    };
    defer();
  };
}


void GetRetina(int TempRetina[])
{
  int I,Count;
  float Slope;

  pokeword(0xFFF4,0xA000);
  msleep(250L);
  Count=peek(0xA800);
  Slope=(float)Count/40.0;
  for (I=0;I<40;I++)
    TempRetina[I]=peek(0xA801+Floor(Slope*(float)I));
}
void RetinaSense()
{
  int I;

  while(1)
  {
    kill_process(PIDs[0]);
    kill_process(PIDs[1]);
    kill_process(PIDs[2]);
    kill_process(PIDs[4]);
    if (ImageNumber>5)              /* If time elapsed, then initiate one motion following cycle */
      {
        ImageNumber=0;
        FlowReady=1;
        for (I=1;I<30;I++)
        {
          V_Left=V_Left+0.1*(0.0-V_Left);
          V_Right=V_Right+0.1*(0.0-V_Right);
          motor(0,V_Left);
          motor(1,V_Right);
          Wait(10);
        };
        motor(0,0.0);
        motor(1,0.0);
        Wait(200);
        GetRetina(LastRetina);
        Wait(800);
```

```
      GetRetina(Retina);
      OpticFlow();
      FollowMovement();
    }
  else
    ImageNumber++;
  PIDs[0]=start_process(NosesSense());
  PIDs[1]=start_process(BumpersSense());
  PIDs[2]=start_process(AvoidObstacles(),50);
  PIDs[4]=start_process(MotorActivation(),50);
  Wait(500);
  defer();
 };
}


/* Very simple optic flow computation */
void OpticFlow()
{
 int I;
 int Flow;

 for (I=0;I<40;I++)
 {
  Flow=Retina[I]-LastRetina[I];
  if (Flow<0)
    Flow=0;
  FlowRetina[I]=Flow;
 };
}




void AvoidObstacles()
{
 while(1)
 {
  float DV_Left_L=0.0,DV_Left_R=0.0,DV_Left_C=0.0;
  float DV_Right_L=0.0,DV_Right_R=0.0,DV_Right_C=0.0;
  float Weight_L=0.0,Weight_C=0.0,Weight_R=0.0;

  if ((float)Noses[0]>(Thresholds[0]+5.0))
  {
    DV_Left_L=0.0-V_Left;
    DV_Right_L=-40.0-V_Right;
    Weight_L=(float)Noses[0]-Thresholds[0];
  };
  if ((float)Noses[1]>(Thresholds[1]+5.0))
  {
    if (TurningCenter==0)
    {
      Rand=Random();
      TurningCenter=1;
    };
    DV_Left_C=((40.0*Rand)-V_Left);
    DV_Right_C=((-40.0*Rand)-V_Right);
    Weight_C=(float)Noses[1]-Thresholds[1];
    }
  else
    TurningCenter=0;
```

```
    if ((float)Noses[2]>(Thresholds[2]+5.0))
    {
      DV_Left_R=-40.0-V_Left;
      DV_Right_R=0.0-V_Right;
      Weight_R=(float)Noses[2]-Thresholds[2];
    };

    /*Obstacle Avoidance*/
    if ((Weight_L+Weight_C+Weight_R)>0.0)
    {
      EnableLED(RED_LED);
      DV_Left_OA=((DV_Left_L*Weight_L)+(DV_Left_C*Weight_C)+(DV_Left_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
      DV_Right_OA=((DV_Right_L*Weight_L)+(DV_Right_C*Weight_C)+(DV_Right_R*Weight_R))
          /(Weight_L+Weight_C+Weight_R);
    }
    else
      {
        EnableLED(GREEN_LED);
        DV_Left_OA=100.0-V_Left;
        DV_Right_OA=100.0-V_Right;
      };
    defer();
  };
}
```

```
int GetMaxFlow()
{
  int I,MaxFlow=0,MotionDirection=0;

  for (I=0;I<40;I++)
    if (FlowRetina[I]>MaxFlow)
    {
      MaxFlow=FlowRetina[I];
      MotionDirection=I;
    };
  if (MaxFlow<20)
    return(0);
  MotionDirection+=5;
  if (MotionDirection>39)
    MotionDirection-=40;
  return(MotionDirection);
}


void FollowMovement()
{
  int MotionDirection;
  MotionDirection=GetMaxFlow();
  EnableLED(YELLOW_LED);
  if(MotionDirection<20)
  {
    motor(0,-50.0);
```

```
    motor(1,50.0);
    msleep((long)MotionDirection*73L);
  }
  else
  {
    motor(0,50.0);
    motor(1,-50.0);
    msleep((40L-(long)MotionDirection)*73L);
  };
}


void MotorActivation()
{
  while (1)
  {
    if ((DV_Left_OA!=0.0) || (DV_Right_OA!=0.0))
    {
      V_Left=V_Left+(0.1*DV_Left_OA);
      V_Right=V_Right+(0.1*DV_Right_OA);
    }
    else
      {
        V_Left=V_Left+(0.1*DV_Left_LF);
        V_Right=V_Right+(0.1*DV_Right_LF);
      };
    motor(0,V_Left);
    motor(1,V_Right);
    defer();
  };
}



void Initialize()
{
  poke(0x4000,0b11100000); /* Turn on LED's and motorized mirror */
  pokeword(0xFFF2,0x9000); /* IRQ Interrupt vector */
  FlowReady=0;
}


void main()
{
  Initialize();
  PIDs[0]=start_process(NosesSense());
  PIDs[1]=start_process(BumpersSense());
  PIDs[2]=start_process(AvoidObstacles(),50);
  PIDs[4]=start_process(MotorActivation(),50);
  start_process(RetinaSense());
}
```

AUTHORS

Pedro Kulzer

António Branco