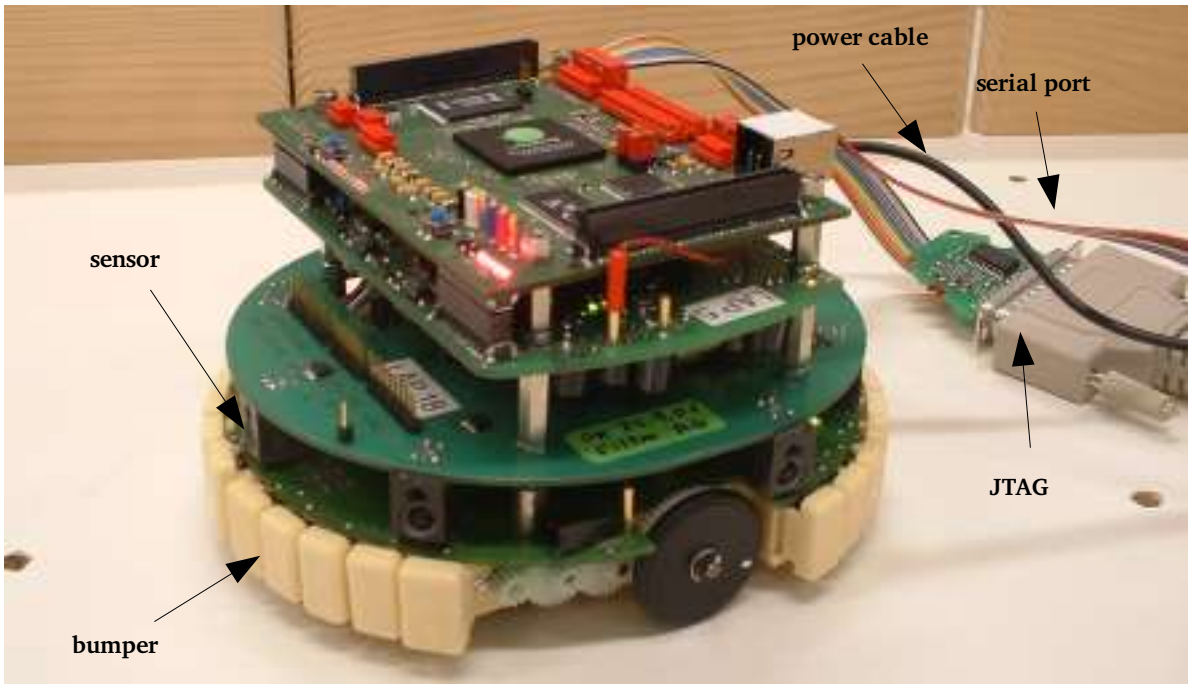
Laboratoire d'Architecture des Processeurs - EPFL
Systèmes Embarqués en temps réels

RokEPXA based Cyclope Robot



Joel Dumas
Alok Menghrajani

February 2004

1.Introduction

The goal of our project is to design a robot that will move around in a maze. We used Cyclope, a RokEPXA based robot developed at the LAP. The RokEPXA board has an interesting hybrid architecture: a programmable FPGA is used to control the robot's fundamental functions (eg sending signals to the motors) while an ARM based general purpose processor is used to program the robot's higher level beahviour (artificial intelligence).

At the end of the project, we tested our robot in a «mini competition», involving Cyclope robots designed by other fellow students. The goal of the competition was to fully explore a maze; and do something (like flash leds or «dance») once the maze was fully explored. The goal was also to explore the maze as quickly as possible. The starting position could be choosen as wanted.

2.The Robot's Anatomy

The Cyclope has the following anatomy:

- 2 wheels, each controlled by a motor. By spining the motors in opposite direction, it is possible to have the robot turn around itself. We can control the duration of the current sent to the motor (enabling us to control the mean tension using only 1 bit)We can also control the direction (forward/reverse).
- 8 infrared sensors, used to estimate the distance to the nearest obstacle.
- 2 optical devices placed one on each wheel, and used to track the rotation of the wheel. This is very useful, since this provides a good estimate of the distance travelled by the robot.
- Several touch sensors (that can be used to detect a collision with a wall, we didn't use them).
- 4 leds that can be used to display whatever we want (useful for debugging purpose).

We used Quartus to do the hardware and software development. We designed the hardware in VHDL, while we wrote the software in C. The robot is programmed using the parallel port (JTAG). The robot can display information via the serial port or on the 8 leds.

3.Hardware Implementation

We implemented the bare minimum in hardware. That is:

- **Motor Controller**
  (See pwm.vhd).
  We used pulse width modulation (PWM) to control the speed of the wheels. The motors use DC, but we can only put VCC or 0[V] on the lines. The trick is that using a pulse, the ratio between «high» and «low» times simulates intermediary tensions, thus the wheels are spining faster or slower. The overall period is fixed (2^{15} clock ticks, at 24 Mhz), and the duty cycle and motor direction are set through a register made available on the avalon bus. (See Figure 1.1).
  The pwm module is instantiated twice, once for each wheel.

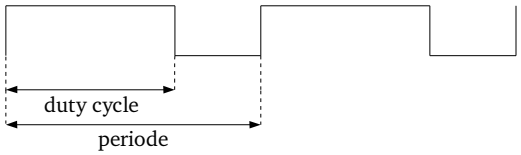


Figure 1.1 – PWM

- **Odometer**
  (See discrvit.vhd)
  The odometer receives two signals that allow us to know in which direction and how fast the wheel is spining (these signals are Gray codes, which means that at most one of them changes at any given time). The input signals are asynchronus, so we first use two levels of flip-flops to synchronize them (the more levels we use, the less chances we have of getting meta-stable. But this also increases the delay between the change of a signal and the response from the system.). We then use a state machine to increment or decrement the distance counter. We also use a timer to calculate the «curent speed», but it turned out that we never used this value.
  The distance and speed can be read by the software in two registers, available through the Avalon bus. One can also write in the distance register, which provides a handy way of resetting it. (See Figure 1.2).

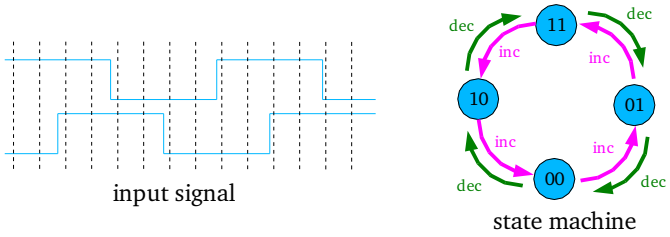


Figure 1.2 – Discrvit

- **IR Sensors**
  The IR sensors are connected to an A/D converter, which we can command through a SPI bus. This is controlled by the software, no extra control hardware was added. We also used two signals that control the LEDs and which sensor is curently connected to the A/D converter.. Each sensor has a LED and reads the amount of light that bouncs back from the walls. In order to eliminate ambient light, we perform each reading twice, once with the LED on and once without, and compute the difference.

4.A Little Maze Theory

There are different types of mazes, each having different properties. We knew that the maze would have the following properties:

- 2D (no steps or ramps)
- orthogonal grid based (walls are the same length and are either horizontal or vertical)
- closed 5x5 maze
- can have inaccessible areas
- each cell has at least one wall (which means: there can't be any 4-way junction)

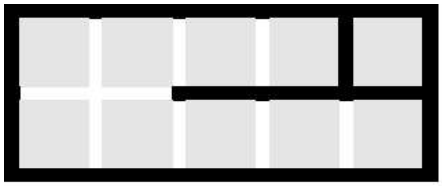
5.Software Implementation

The first part of the software design was to get the robot moving properly (movement primitives). The sensors return values that grow exponentially (not exactly since there is a saturation point, but you get the idea) as the walls get closer. Instead of trying to determine the exact curve, we just used reference values. Our motor and odometer don't use the same units of mesure, so we also used reference values to calibrate them. Since we wanted high precision, we always used the motors at half speed (we could have increased the speed once everything worked, but we didn't get enough time).
We always performed 3 readings on the sensors, and took the mean. This way, we had more stable readings.
We didn't need to calibrate the two wheels (by reading the odometer after setting their speeds). Some student had to do this (and some even implemented hardware solutions). We were probably lucky to have exactly the same motors on our robot (and even if it is not exactly the case, small errors can be corrected by our «move forward» algorithm, see below).

- We wrote a function *move_forward* which will move the robot until it reaches the next intersection (either missing walls on the sides or a wall in front, it could be just a corner). This way, whenever the robot stops, we have reference points (either an empty space on the sides or a wall in front) to reposition ourselves and calculate how many cells we have covered.Since we recalibrate the robot each time, the precision is good enough. We also use the left and right walls to stay in the center of the alleys (we use linear recalibration, so if the robot gets too close to a wall it will immediatly correct itself, while still performing small corrections when near the center).
  One of the problems we ran into was the robot starting to recalibrate itself just before the end of a wall (and then losing it's direction). Linear recalibration fixed this problem, as by the end of a wall, the robot should always be in the center of the cell.
  Note: we also took special consideration to detect the absence of walls on the sides, as this can happen when the robot starts to move (during the first half- cell).
- We used three different funtions to turn left, right and 180°. We used three seperate functions so we could calibrate them individually. We also noticed that the U-turn required a slightly larger value (probably because of the way the processor reads the odometer value in a loop).

For navigating the maze, we wrote two algorithms:
- follow the left wall:
  It is very simple, but it doesn't guarantee that all cells will be visited (it can loop arround islands) and it can't tell when it is done.
- a more complex algorithm:
  The idea is to map the maze and go to unvisited cells until none are remaining.
  The C structure used to represent the maze was a 2*size_x+1 by 2*size_y+1 array of char. The (odd, odd) indexes represent cells. The (even, even) indexes are ignored. The (odd, even) and (even, odd) indexes represent walls. (See Figure 1.3).

Figure 1.3 – Internal maze representation



The walls can be in 3 states: unknown, set or clear. The cells can be in 4 states: unvisited, visited, inaccessible, accessible. The difference between accessible and unvisited is that the path to get to an accessible cell is known, whereas the path to an unvisited cell is unknown- it could even not exist (accessible cells are neighbours to visited or accesible cells, whose seperating wall is marked as clear). Before taking a move decision, we first perform graph analysis:

- Any cells that has 3 walls marked clear has the 4th wall marked set.
- Every cell that is unvisited is marked as inaccessible.
- Every cell that is a neighbour of a visited or accessible cell and has the wall set to clear is marked as accessible.
- Every cell that is a neighbour of an accessible cell but who's wall state is unknown is marked as unvisited.
- The remaining cells are marked as inaccessible.

After this analysis is done, the best accessible cells are searched for (See Figure 1.4), taking into account distance and number of surrounding walls (we want to first visit dead ends). Once this cell is found (it's path is determined at the same time), the robot is instructed to rotate and move forward (the robot doesn't actually go to the chosen cell, it simply moves towards it and recomputes everything, enabling it to change it's decision about which cell to visit if it finds a better one on its way.

As the robot moves, the information its sensors receive are used to mark the map's walls. Our algorithm is able to detect when the robot is lost (if it wants to change the state of a wall in an incoherent way). When this happens, we fall back to the simpler «*follow the left wall*» algorithm.

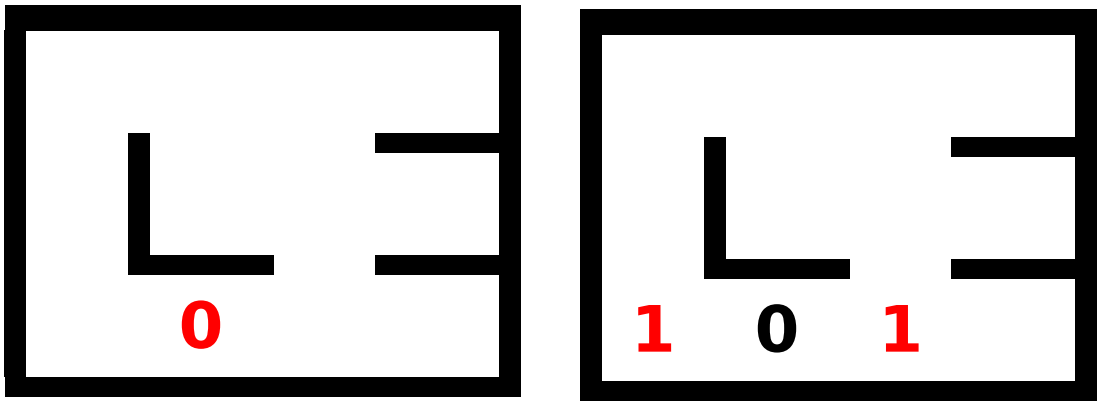
One of the requirements of this algorithm is to know the starting position and orientation. This can be changed by defining a larger grid, but then the robot will take less optimal decisions (it's a trade- off).

6. Conclusion

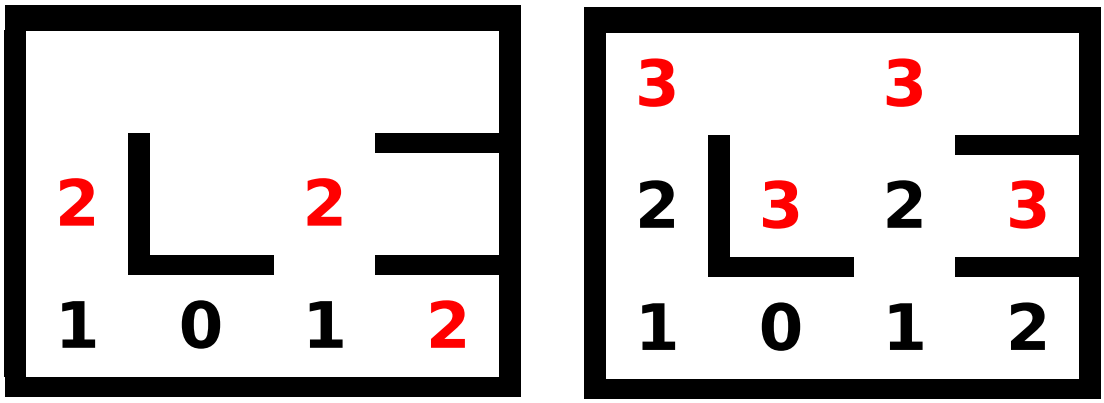
Unfortunately we had a lot of problems the day of the contest, including: an odometer that stopped to work (due to a connection that broke), the sensors reported different values than during our preparation, most notably the front sensor (we had to quickly hack a fix for this). Even though we didn't have enough time to solve all these last- minute problems, our robot performed remarkably well, being 1 second short from the 1st place in the first maze, and missing only one cell in the second (which only one robot managed to explore entirely). But what we are most proud of, is the way it's moving: it's very accurate, and it always stays in the middle of the cells.

Had we got more time, we would have improved the following:

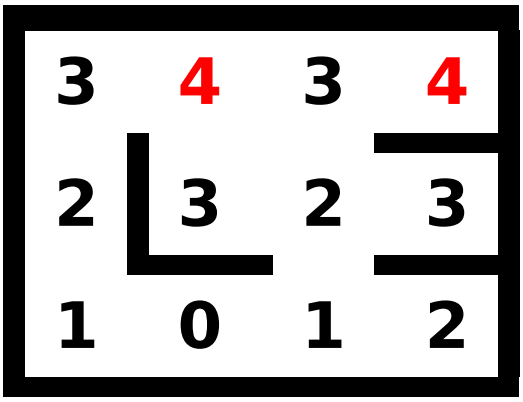
- Write some code to test the routing algorithm (we did all our tests by running our code on the robot which isn't always the best way to develop, because the «write-test-fix» process is slow (it takes about 1 minute to compile the code, few seconds to transfer it to the robot, another few seconds to test and the output the robot can give is very limited).
- Dynamic calibration based on current environment. This is something very easy to do, and very helpful, if not mandatory (we did by hand).
- Hardware self-test (we had so many problems that this would definitely be useful).
- We could improve the routing algorithm by moving along walls whenever possible (when the robot has lateral walls it makes much less mistakes).
- I wish we could have worked on camera based navigation (which would provide us with a whole new set of challenges but would enable us to have a much better robot, as we could discover larger parts of the maze at a time).
- Better error recovery. Right now we simply fallback to the follow the left wall algorithm, we could do something better.
- One problem with our current way of doing rotations, is that we don't have any reference points. So performing several rotations in a row can be a problem. It is very difficult to perform good rotations without losing too much time. We tried some different solutions, and we think the only solution would be to have an external sensor (compass or cheap gyroscope). This would really increase the reliability.



iteration 0 *iteration 1*



iteration 2 *iteration 3*



iteration 4

Figure 1.4 – How the algorithm finds the shortest path by value propagation.

7.VHDL source

pwm.vhd

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity pwm is
  port (
    signal clk       : in std_logic;
    signal reset     : in std_logic;

    -- Avalon bus interface
    signal chipselect : in std_logic;
    signal address    : in std_logic_vector(1 downto 0);
    signal read       : in std_logic;
    signal write      : in std_logic;
    signal readdata   : out std_logic_vector(15 downto 0);
    signal writedata  : in std_logic_vector(15 downto 0);

    -- PWM channel output
    signal pwm_pulse   : out std_logic;
    signal pwm_dir     : out std_logic
    );

end pwm;

architecture synth of pwm is
      signal speed, timer: std_logic_vector(14 downto 0);
begin
      process (chipselect, write, reset)
      begin
            if reset = '1' then
                  pwm_dir <= '0'; speed <= (others => '0');
            elsif clk'event and clk = '1' then
                  if chipselect = '1' and write = '1' then
                        if address = "00" then
                              speed <= writedata(14 downto 0);
```

```
                        pwm_dir <= writedata(15);
                  end if;
            end if;
      end if;
end process;

process(clk, reset)
begin
      if reset = '1' then
            timer <= (others => '0');
      elsif clk'event and clk = '1' then
                  timer <= timer + (14 => '1', 13 downto 0 => '0');
      end if;
end process;

process (chipselect, write, timer, speed, reset)
begin
      if clk'event and clk = '1' then
            if chipselect = '1' and write = '1' then
                  if timer >= writedata(14 downto 0) then
                        pwm_pulse <= '0';
                  else
                        pwm_pulse <= '1';
                  end if;
            else
                  if (reset = '1') or (timer = (14 downto 0 => '0')) then
                        pwm_pulse <= '1';
                  end if;
                  if timer = speed then
                        pwm_pulse <= '0';
                  end if;
            end if;
      end if;
end process;
end synth;
```

---

discvit.vhd

```
library ieee;
```

```
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity DiscrVit is
   port(
      clk, reset: in std_logic;
      chipselect, read, write: in std_logic;
      address: in std_logic_vector(1 downto 0);
      readdata: out std_logic_vector(15 downto 0);
      writedata: in std_logic_vector(15 downto 0);
      RamboA1, RamboA2, RamboB1, RamboB2: in std_logic);
end DiscrVit;

architecture synth of DiscrVit is
   signal state1, state2, nextState1, nextState2: std_logic_vector(1 downto 0);
   signal add1, add2: std_logic_vector(15 downto 0);
   signal timer: std_logic_vector(19 downto 0);
   signal v1, v2: std_logic_vector(15 downto 0);
   signal vitesse1, vitesse2, dist1, dist2: std_logic_vector(15 downto 0);
   signal ta1, ta2, tb1, tb2: std_logic;
   signal odol_a, odol_b, odor_a, odor_b: std_logic;
begin
   process (clk)
   begin
      if clk'event and clk = '1' then
         ta1 <= RamboA1;   ta2 <= RamboA2;
         tb1 <= RamboB1; tb2 <= RamboB2;
         odol_a <= ta1; odol_b <= tb1;
         odor_a <= ta2; odor_b <= tb2;
      end if;
   end process;

   process (odol_a, odol_b, state1)
   begin
      nextState1 <= odol_a & odol_b;
      add1 <= (others => '0');
      case state1 is
```

```
      when "00" =>
         if nextState1 = "01" then add1 <= (others => '1');
         elsif nextState1 = "10" then add1 <= (0 => '1', others => '0');
         end if;
      when "01" =>
         if nextState1 = "11" then add1 <= (others => '1');
         elsif nextState1 = "00" then add1 <= (0 => '1', others => '0');
         end if;
      when "10" =>
         if nextState1 = "00" then add1 <= (others => '1');
         elsif nextState1 = "11" then add1 <= (0 => '1', others => '0');
         end if;
      when "11" =>
         if nextState1 = "10" then add1 <= (others => '1');
         elsif nextState1 = "01" then add1 <= (0 => '1', others => '0');
         end if;
      when others => null;
   end case;
end process;

process (odor_a, odor_b, state2)
begin
   nextState2 <= odor_a & odor_b;
   add2 <= (others => '0');
   case state2 is
      when "00" =>
         if nextState2 = "01" then add2 <= (others => '1');
         elsif nextState2 = "10" then add2 <= (0 => '1', others => '0');
         end if;
      when "01" =>
         if nextState2 = "11" then add2 <= (others => '1');
         elsif nextState2 = "00" then add2 <= (0 => '1', others => '0');
         end if;
      when "10" =>
         if nextState2 = "00" then add2 <= (others => '1');
         elsif nextState2 = "11" then add2 <= (0 => '1', others => '0');
         end if;
      when "11" =>
```

```
         if nextState2 = "10" then add2 <= (others => '1');
         elsif nextState2 = "01" then add2 <= (0 => '1', others => '0');
         end if;
      when others => null;
   end case;
end process;

process (reset, write, clk)
begin
   if reset = '1' then
      v1 <= (others => '0'); v2 <= (others => '0');
      dist1 <= (others => '0'); dist2 <= (others => '0');
      timer <= (others => '0');
   else
      if clk = '1' and clk'event then
         dist1 <= dist1 + add1;
         dist2 <= dist2 + add2;
         if (chipselect and write) = '1' then
             if address = "00" then
                dist1 <= writedata;
             elsif address = "10" then
                dist2 <= writedata;
             end if;
         end if;
         timer <= timer + (19 => '1', 18 downto 0 => '0');
         if timer = (19 downto 0 => '0') then
            vitesse1 <= v1; vitesse2 <= v2;
            v1 <= (others => '0'); v2 <= (others => '0');
         else
            v1 <= v1 + add1; v2 <= v2 + add2;
         end if;
      end if;
   end if;
end process;

process (clk)
begin
   if clk = '1' and clk'event then
```

```
         state1 <= nextState1;
         state2 <= nextState2;
      end if;
   end process;

   process (chipselect, read, reset)
   begin
      if (chipselect and read) = '1' then
         if address = "00" then
            readdata <= dist1;
         elsif address = "01" then
            readdata <= vitesse1;
         elsif address = "10" then
            readdata <= dist2;
         else
            readdata <= vitesse2;
         end if;
      end if;
   end process;
end synth;
```

8.C source

One of the reasons this source code is ugly is because we coded it in a single evening...

main.c

```
#include <stdio.h>
#include "ARM_Stripe_sdk\inc\excalibur.h"
#include "spi.h"
#include "max1202.h"
#include "spi_max1202.h"
#include "excalibur.h"
#include "led.h"

typedef unsigned int uint;

/* my macros */
#define GET_SPEED(x) ((int)(short)((int*)na_user_Vitesse)[2*(x)+1])
#define GET_SPEED_L     GET_SPEED(0)
#define GET_SPEED_R GET_SPEED(1)
#define SET_SPEED_L(v) (*(short*)na_user_pwm_left = (v));
#define SET_SPEED_R(v) (*(short*)na_user_pwm_right = (v));

#define GET_DIST(x) ((int)(short)((int*)na_user_Vitesse)[2*(x)])
#define GET_DIST_L      GET_DIST(0)
```

```
#define     GET_DIST_R  GET_DIST(1)
#define RESET_DIST  {((int*)na_user_Vitesse)[0] = 0; ((int*)na_user_Vitesse)[2] =
0;}

/* robot's constants */
#define NB_SENSORS                    8  /* number of total sensors */
#define SENSOR_READINGS         3  /* number of times to read the sensors */
#define SENSOR_LEFT                   0
#define SENSOR_LEFT_DIAG1       1
#define SENSOR_LEFT_DIAG2       7
#define SENSOR_FRONT            2
#define SENSOR_RIGHT            4
#define SENSOR_RIGHT_DIAG1      3
#define SENSOR_RIGHT_DIAG2      6

#define SENSOR_NO_WALL        250  /* <100 means no wall */
#define SENSOR_WALL                250
#define SENSOR_FRONT_TST     100
#define SENSOR_FRONT_WALL    800

#define DIST_ROTATE               2550
#define DIST_UTURN                2790
#define DIST_FORWARD          3620
#define EPSILON                   1000

/* maze constants */
#define SIZE_X                        5
#define SIZE_Y                        5

#define CELL_UNVISITED       0
#define CELL_VISITED         1
#define CELL_INACCESSIBLE    2
#define CELL_ACCESSIBLE      3

#define WALL_UNKNOWN         4
#define WALL_SET             5
#define WALL_CLEAR           6

/* globals */
unsigned char maze[SIZE_X*2+1][SIZE_Y*2+1];
unsigned char ai[SIZE_X*2+1][SIZE_Y*2+1];
unsigned char path[SIZE_X*2+1][SIZE_Y*2+1];
int robot_x;
int robot_y;
int robot_d;
int errors;

/* prototypes */
void rotate_right(void);
void rotate_left(void);
void uturn(void);
void init_maze(void);
void update_maze(void);
void display_maze(void);
void display_ai(void);
int check_win(void);
void mark_walls(void);
void mark_inaccesible(void);
```

```
int get_newy(int oldy, int d, int s);
int get_newx(int oldx, int d, int s);
void read_sensors(int*);
void recalibrate_with_walls(int *data);
void goto_nearest_accessible(void);
void simple_left_hand(void);

/* maze structure (eg 5x2 => 11x5):
             0 1 2 3 4 5 6 7 8 9 10
      0      ? W ? W ? W ? W ? W ?
      1      W .   .   .   . X I W            N
      2      ?   ?   ? X ? X ? X ?          W   E
      3      W .   .   .   .   . W            S
      4      ? W ? W ? W ? W ? W ?

      (odd, odd) => cell
      (even, even) => don't care
      (even, odd) or (odd, even) => wall
*/

/* this code works well ;)
   todo: improve path finding so that rotations are avoided.
   perhaps: write code to find 'best way' (least risk) to get to
   a given spot.

   maybe: write code to 'cancel' a decision in case the robot
   realizes it has choosen a long path.
*/

int main() {
      int sl, sr;
      int i;
      int sensors[NB_SENSORS];

      robot_x = 1; // in special coordinates !!!
      robot_y = 1;
      robot_d = 2; /* 0=North, 1=East, 2=South, 3=West */

      printf("Welcome to AJO's World.\n");
      led_init();
      init_maze();
      errors = 0;


/*    while(1) {
            read_sensors(sensors);
            for (i=0; i<8; i++) {
                  printf("%d\t", sensors[i]);
            }
            printf("\n");
      }
*/

      /* time to take out jtag */
      for (i=0; i<12000000; i++) ;

      /* AI starts here */
      while (1) {
```

```
            update_maze();
            mark_walls();
            mark_inaccesible();
            if (check_win())
                  break;
//          display_maze();

            goto_nearest_accessible();
            if (errors!=0) {
                  // switch to simpler algorithm if we
                  // get lost !
                  led_value(8);
                  simple_left_hand();
            }
      }

      display_maze();

      printf("done.\n");
      while(1) {
            for (i=0; i<4; i++) {
                  led_blink(i, 500, 1);
            }
            display_maze();
      }
}

// Routines to get new coorindates given the old ones
// a direction and quantity to move.
int get_newx(int oldx, int d, int s) {
      if ((d==0) || (d==2)) {
            return oldx;
      } else if (d==1) {
            return oldx+s;
      } else {
            return oldx-s;
      }
}

int get_newy(int oldy, int d, int s) {
      if ((d==1) || (d==3)) {
            return oldy;
      } else if (d==0) {
            return oldy-s;
      } else {
            return oldy+s;
      }
}

// Find which cell to go next (which is unvisited)
void goto_nearest_accessible(void) {
      int i, j, k, t, min;
      int min_i, min_j, min_walls;
      t = 1;
      for (i=0; i<SIZE_X*2+1; i++)
            for (j=0; j<SIZE_Y*2+1; j++)
                  ai[i][j] = 0xff;
      ai[robot_x][robot_y]=0;
```

```
	// calculate nearest accessible (unvisited)
	while (t==1) {
		t = 0;
		for (i=1; i<SIZE_X*2+1; i+=2) {
			for (j=1; j<SIZE_Y*2+1; j+=2) {
				if (ai[i][j]==0xff)
					continue;
				for (k=0; k<4; k++) {
					if (maze[get_newx(i, k, 1)][get_newy(j, k, 1)]
==WALL_CLEAR) {
						if ((ai[get_newx(i, k, 2)][get_newy(j, k, 2)]
>ai[i][j]+1) ||
							(ai[get_newx(i, k, 2)][get_newy(j, k, 2)]
== 0xff)){
							ai[get_newx(i, k, 2)][get_newy(j, k, 2)]
= ai[i][j]+1;
							path[get_newx(i, k, 2)][get_newy(j, k,
2)] = k;
							t = 1;
						}
					}
				}
			}
		}
	}

	// find min
	// it's the cell with ai[i][j] min and use the most number
	// of walls set in case of tie.
	min = 9999;
	for (j=1; j<SIZE_Y*2+1; j+=2) {
		for (i=1; i<SIZE_X*2+1; i+=2) {
			if (maze[i][j]==CELL_ACCESSIBLE) {
				t = 0;
				for (k=0; k<4; k++) {
					if (maze[get_newx(i, k, 1)][get_newy(j, k, 1)] ==
WALL_SET)
						t++;
				}

				if ((ai[i][j]<min) || ((ai[i][j]==min) && (t>min_walls))) {
					min=ai[i][j];
					min_i = i;
					min_j = j;
					min_walls = t;
				}
			}
		}
	}
	// find path to min_i, min_j
	printf("I WANT TO GO TO: %d, %d\n", min_i, min_j);
	while (min>0) {
		k = path[min_i][min_j];
		min_i = get_newx(min_i, (k+2)%4, 2);
		min_j = get_newy(min_j, (k+2)%4, 2);
		min--;
	}
```

```
	if ((robot_d + 2) % 4 == k) {
		printf("rotate 180\n");
		uturn();
	} else if ((robot_d+1)%4 == k) {
		printf("rotate right\n");
		rotate_right();
	} else if ((robot_d+3)%4 == k) {
		printf("rotate left\n");
		rotate_left();
	}

	t = move_forward();
/*	led_value(t);
	for (i=0; i<1000000; i++);
	*/

	// update info about cells we just passed by,
	// in the maze variable.
	// this code can detect incoherent readings.
	for (i=1; i<=t; i++) {
		int t1, t2;
		t1 = get_newx(robot_x, k, i*2-1);
		t2 = get_newy(robot_y, k, i*2-1);
		if (maze[t1][t2]==WALL_SET)
			errors++;
		maze[t1][t2]=WALL_CLEAR;
	}

	/* must handle case t>1 */
	for (i=1; i<=(t-1); i++) {
		int x, y, t1, t2;
		t1 = get_newx(robot_x, k, i*2);
		t2 = get_newy(robot_y, k, i*2);
		maze[t1][t2]=CELL_VISITED;

		x = get_newx(t1, (k+1)%4, 1);
		y = get_newy(t2, (k+1)%4, 1);
		if (maze[x][y]==WALL_CLEAR)
			errors++;
		maze[x][y]=WALL_SET;

		x = get_newx(t1, (k+3)%4, 1);
		y = get_newy(t2, (k+3)%4, 1);
		if (maze[x][y]==WALL_CLEAR)
			errors++;
		maze[x][y]=WALL_SET;
	}
	robot_d = k;
	robot_x = get_newx(robot_x, k, t*2);
	robot_y = get_newy(robot_y, k, t*2);
}

// if any wall has 3 walls marked as clear, then
// the code will mark the 4th wall as set (and redo the
// calculation again, since changes can propagate).
void mark_walls(void) {
	int i, j, k, n, t;
	for (i=1; i<SIZE_X*2+1; i+=2) {
```

```
            for (j=1; j<SIZE_Y*2+1; j+=2) {
                  // count number of neighbouring walls
                  n = 0;
                  for (k=0; k<4; k++) {
                        if (maze[get_newx(i, k, 1)][get_newy(j, k, 1)]==WALL_CLEAR)
{
                              n++;
                        } else {
                              t=k;
                        }
                  }
                  if (n==3) {
                        maze[get_newx(i, t, 1)][get_newy(j, t, 1)]=WALL_SET;
                  }
            }
      }
}

// you win when you have explored the entire maze.
// ie: all cells are either visited or inaccessible.
int check_win(void) {
      int i, j;
      for (i=1; i<SIZE_X*2+1; i+=2) {
            for (j=1; j<SIZE_Y*2+1; j+=2) {
                  if ((maze[i][j]==CELL_UNVISITED) || (maze[i][j]
==CELL_ACCESSIBLE))
                        return 0;
            }
      }
      return 1;
}

// find which cells are inaccessible, by "flooding" the visited
// cells to form the accessible (you know the wall structure of
// visited cells). From the accessible cells, you can calculate
// the list of unknown cells ("flood" as long as walls are not set)
// all cells that don't get marked are inaccessible.
void mark_inaccesible(void) {
      int i, j, k, t;
      t = 1;
      for (i=1; i<SIZE_X*2+1; i+=2) {
            for (j=1; j<SIZE_Y*2+1; j+=2) {
                  if (maze[i][j]==CELL_UNVISITED) {
                        maze[i][j]=CELL_INACCESSIBLE;
                  }
            }
      }
      while (t==1) {
            t = 0;
            for (i=1; i<SIZE_X*2+1; i+=2) {
                  for (j=1; j<SIZE_Y*2+1; j+=2) {
                        // mark all unvisited cells as inaccessible.
                        // we want to mark cells
                        // that are next to visited seperated by
                        // a wall_clear as accessible.
                        // cells that are inaccessible next to accessible are
                        // marked as unvisited (unknown) (except if already marked
accessible/visited)
```

```
                        // cells that are inaccessible next to unvisited are marked
unvisited.
                        if (maze[i][j]==CELL_VISITED) {
                              for (k=0; k<4; k++) {
                                    if (maze[get_newx(i, k, 1)][get_newy(j, k, 1)]
==WALL_CLEAR) {
                                          if ((maze[get_newx(i, k, 2)][get_newy(j,
k, 2)] != CELL_VISITED) &&
                                              (maze[get_newx(i, k, 2)][get_newy(j,
k, 2)] != CELL_ACCESSIBLE)) {
                                                printf("%d %d causes accessible in
%d %d (%d)\n",
                                                      i, j, get_newx(i, k, 2),
get_newy(j, k, 2),
                                                      maze[get_newx(i, k, 2)]
[get_newy(j, k, 2)]);
                                                maze[get_newx(i, k, 2)][get_newy(j,
k, 2)]=CELL_ACCESSIBLE;
                                                t = 1;
                                          }
                                    }
                              }
                        } else if ((maze[i][j]==CELL_ACCESSIBLE) || (maze[i][j]
==CELL_UNVISITED)) {
                              for (k=0; k<4; k++) {
                                    if (maze[get_newx(i, k, 1)][get_newy(j, k, 1)]!
=WALL_SET) {
                                          if (maze[get_newx(i, k, 2)][get_newy(j,
k, 2)]==CELL_INACCESSIBLE) {
                                                maze[get_newx(i, k, 2)][get_newy(j,
k, 2)]=CELL_UNVISITED;
                                                t = 1;
                                          }
                                    }
                              }
                        }
                  }
            }
      }
}

/*
      update the maze to mark the current cell as visited.
      and check the walls.
      this code can also detect incoherent readings.
*/
void update_maze(void) {
      int x, y;
      int sensors[NB_SENSORS];
      maze[robot_x][robot_y] = CELL_VISITED;
      // see what is the current situation
      read_sensors(sensors);

      led_value(0);
      if (sensors[SENSOR_LEFT] > SENSOR_WALL)
            led(0, 1);
      if (sensors[SENSOR_FRONT] > SENSOR_FRONT_TST)
            led(1, 1);
```

```
      if (sensors[SENSOR_RIGHT] > SENSOR_WALL)
            led(2, 1);
//    for (x=0; x<1000000; x++)
      //    ;

      x = get_newx(robot_x, robot_d, 1);
      y = get_newy(robot_y, robot_d, 1);
      if (sensors[SENSOR_FRONT] > SENSOR_FRONT_TST) {
            if (maze[x][y]==WALL_CLEAR) {
                  printf("Error 1\n");
                  errors++;
            }
            maze[x][y]=WALL_SET;
      } else {
            if (maze[x][y]==WALL_SET) {
                  printf("Error 2\n");
                  errors++;
            }
            maze[x][y]=WALL_CLEAR;
      }

      x = get_newx(robot_x, (robot_d+1)%4, 1);
      y = get_newy(robot_y, (robot_d+1)%4, 1);
      if (sensors[SENSOR_RIGHT] > SENSOR_WALL) {
            if (maze[x][y]==WALL_CLEAR) {
                  printf("Error 3\n");
                  errors++;
            }
            maze[x][y]=WALL_SET;
      } else {
            if (maze[x][y]==WALL_SET)  {
                  printf("Error 4\n");
                  errors++;
            }
            maze[x][y]=WALL_CLEAR;
      }

      x = get_newx(robot_x, (robot_d+3)%4, 1);
      y = get_newy(robot_y, (robot_d+3)%4, 1);
      if (sensors[SENSOR_LEFT] > SENSOR_WALL) {
            if (maze[x][y]==WALL_CLEAR) {
                  printf("Error 5\n");
                  errors++;
            }
            maze[x][y]=WALL_SET;
      } else {
            if (maze[x][y]==WALL_SET) {
                  printf("Error 6\n");
                  errors++;
            }
            maze[x][y]=WALL_CLEAR;
      }
}

// simple navigation routine. this code never returns !
// simple follow the wall on the left side.
void simple_left_hand(void) {
      int sensors[NB_SENSORS];
```

```
	while (1) {
		move_forward();
		read_sensors(sensors);
		if (sensors[SENSOR_LEFT] < SENSOR_NO_WALL) {
			rotate_left();
			continue;
		} else if (sensors[SENSOR_FRONT] < SENSOR_FRONT_TST) {
			continue;
		} else if (sensors[SENSOR_RIGHT] < SENSOR_NO_WALL) {
			rotate_right();
			continue;
		}
		rotate_right();
		rotate_right();
	}
}

void init_maze(void) {
	int i, j;
	// set all the walls to unknown state
	for (i=0; i<(SIZE_X*2+1); i++) {
		for (j=0; j<(SIZE_Y*2+1); j++) {
			maze[i][j] = WALL_UNKNOWN;
		}
	}

	// set all the cells to unvisited
	for (i=1; i<SIZE_X*2+1; i+=2) {
		for (j=1; j<SIZE_Y*2+1; j+=2) {
			maze[i][j] = CELL_UNVISITED;
		}
	}

	// We know the outer ring has walls.
	for (i=1; i<SIZE_X*2+1; i+=2) {
		maze[i][0] = WALL_SET;
		maze[i][SIZE_Y*2] = WALL_SET;
	}
	for (j=1; j<SIZE_Y*2+1; j+=2) {
		maze[0][j] = WALL_SET;
		maze[SIZE_X*2][j] = WALL_SET;
	}
}

/* debugging purpose routine */
void display_maze(void) {
	int i, j;
	for (j=0; j<SIZE_Y*2+1; j++) {
		for (i=0; i<SIZE_X*2+1; i++) {
			if ((i%2==0) && (j%2)==0) {
				printf("  ");
			} else if ((i%2==1) && (j%2)==1) {
				printf("%c ", maze[i][j]==CELL_UNVISITED ? '.' :
					(maze[i][j]==CELL_VISITED ? '*' : (maze[i][j]
==CELL_ACCESSIBLE ? 'A' : 'I')));
			} else {
				printf("%c ", maze[i][j]==WALL_UNKNOWN ? '?' :
					(maze[i][j]==WALL_SET ? 'X' : ' '));
```

```
                }
            }
            printf("\n");
      }
      printf("\n");
}

/* debugging purpose routine */
void display_ai(void) {
      int i, j;
      for (j=0; j<SIZE_Y*2+1; j++) {
            for (i=0; i<SIZE_X*2+1; i++) {
                  if ((i%2==0) && (j%2)==0) {
                        printf("  ");
                  } else if ((i%2==1) && (j%2)==1) {
                        if (ai[i][j]!=0xff) {
                              printf("%d ", ai[i][j]);
                        } else {
                              printf("  ");
                        }
                  } else {
                        printf("%c ", maze[i][j]==WALL_UNKNOWN ? '?' :
                              (maze[i][j]==WALL_SET ? 'X' : ' '));
                  }
            }
            printf("\n");
      }
      printf("\n");
}

/* code to get the robot to walk straight */
void recalibrate_with_walls(int *data) {
      if (data[SENSOR_LEFT] > 580) {
            SET_SPEED_L(0x3fff - (data[SENSOR_LEFT]-580)*0x0200/200);
            SET_SPEED_R(0x3fff - (data[SENSOR_LEFT]-580)*0x0800/200);
      } else if (data[SENSOR_LEFT] > 300) {
            SET_SPEED_L(0x3fff - (580-data[SENSOR_LEFT])*0x1000/300);
            SET_SPEED_R(0x3fff - (580-data[SENSOR_LEFT])*0x0200/300);
      } else if (data[SENSOR_RIGHT] > 580) {
            SET_SPEED_R(0x3fff - (data[SENSOR_RIGHT]-580)*0x0200/200);
            SET_SPEED_L(0x3fff - (data[SENSOR_RIGHT]-580)*0x0800/200);
      } else if (data[SENSOR_RIGHT] > 300) {
            SET_SPEED_R(0x3fff - (580-data[SENSOR_RIGHT])*0x1000/300);
            SET_SPEED_L(0x3fff - (580-data[SENSOR_RIGHT])*0x0200/300);
      } else {
            SET_SPEED_R(0x3fff);
            SET_SPEED_L(0x3fff);
      }
}

/* moves forward until we encounter an intersection:
      - if the right or left walls disappear, then the robot
        advances exactly half a cell (recalibration)
      - it uses the left and right wall to walk straight
      - if there is a wall ahead, then the robot will recalibrate
*/
int move_forward(void) {
      int i;
```

```
      int sensors[NB_SENSORS];

      /* Reset odometer */
      RESET_DIST;

      SET_SPEED_L(0x3fff);
      SET_SPEED_R(0x3fff);
      while (1) {
            read_sensors(sensors);
            if ((GET_DIST_L + GET_DIST_R) > (2*DIST_FORWARD)) {
                  break;
            }

            if (sensors[SENSOR_FRONT] > SENSOR_FRONT_TST) {
                  while (sensors[SENSOR_FRONT] < SENSOR_FRONT_WALL) {
                        read_sensors(sensors);
                        recalibrate_with_walls(sensors);
                  }
                  SET_SPEED_L(0);
                  SET_SPEED_R(0);
                  return 1;
//                return (GET_DIST_L + GET_DIST_R + DIST_FORWARD) /
(2*DIST_FORWARD);
            }
            /* recalibrate */
            recalibrate_with_walls(sensors);
      }

      /* Check if we have a hole on the left or right */
      if ((sensors[SENSOR_LEFT] < SENSOR_NO_WALL) || (sensors[SENSOR_RIGHT] <
SENSOR_NO_WALL)) {
            SET_SPEED_L(0);
            SET_SPEED_R(0);
            return 1;
//          return (GET_DIST_L + GET_DIST_R + DIST_FORWARD) / (2*DIST_FORWARD);
      }

      while (1) {
            read_sensors(sensors);
            if (sensors[SENSOR_FRONT] > SENSOR_FRONT_TST) {
                  /* there is a wall in front */
                  while (sensors[SENSOR_FRONT] < SENSOR_FRONT_WALL) {
                        read_sensors(sensors);
                        recalibrate_with_walls(sensors);
                  }
                  SET_SPEED_L(0);
                  SET_SPEED_R(0);
//                return (GET_DIST_L + GET_DIST_R) / (2*DIST_FORWARD);
                  return (GET_DIST_L + GET_DIST_R + EPSILON) / (2*DIST_FORWARD);
            } else if ((sensors[SENSOR_LEFT] < SENSOR_NO_WALL) || (sensors
[SENSOR_RIGHT] < SENSOR_NO_WALL)) {
                  /* we have a hole on the left or right */
                  int tsl = GET_DIST_L;
                  int tsr = GET_DIST_R;
                  /* move half a cell */
                  SET_SPEED_L(0x3fff);
                  SET_SPEED_R(0x3fff);
                  /* compensate for sensor being a little discentered */
```

```
                  while ((GET_DIST_L-tsl) + (GET_DIST_R-tsr) < (DIST_FORWARD+200))
{
                        read_sensors(sensors);
                        if (sensors[SENSOR_FRONT] > SENSOR_FRONT_TST) {
                              /* there is a wall in front */
                              while (sensors[SENSOR_FRONT] < SENSOR_FRONT_WALL) {
                                    read_sensors(sensors);
                                    recalibrate_with_walls(sensors);
                              }
                              SET_SPEED_L(0);
                              SET_SPEED_R(0);
//                            return (GET_DIST_L + GET_DIST_R) / (2*DIST_FORWARD);
                              return (GET_DIST_L + GET_DIST_R + EPSILON) /
(2*DIST_FORWARD);
                        }
                        recalibrate_with_walls(sensors);
                  }
                  SET_SPEED_L(0);
                  SET_SPEED_R(0);
//                return (GET_DIST_L + GET_DIST_R) / (2*DIST_FORWARD);
                  return (GET_DIST_L + GET_DIST_R + EPSILON) / (2*DIST_FORWARD);
            }
            /* recalibrate */
            recalibrate_with_walls(sensors);
      }
}

/* rotation code */
void rotate_right(void) {
      RESET_DIST;
      SET_SPEED_R(0xbfff);
      SET_SPEED_L(0x3fff);
      while ((GET_DIST_L-GET_DIST_R) < DIST_ROTATE)
            ;
      SET_SPEED_R(0);
      SET_SPEED_L(0);
}

void rotate_left(void) {
      RESET_DIST;
      SET_SPEED_L(0xbfff);
      SET_SPEED_R(0x3fff);
      while ((GET_DIST_R-GET_DIST_L) < DIST_ROTATE)
            ;
      SET_SPEED_R(0);
      SET_SPEED_L(0);
}

void uturn(void)
{
      RESET_DIST;
      SET_SPEED_L(0xbfff);
      SET_SPEED_R(0x3fff);
      while ((GET_DIST_R-GET_DIST_L) < 2*DIST_ROTATE)
            ;
      SET_SPEED_R(0);
      SET_SPEED_L(0);
}
```

```
/* read sensors multiple times and return average */
/* this improves the readings stability */
void read_sensors(int* data) {
      int i, j, s, t;
      /* clear array */
      for (i=0; i<NB_SENSORS; i++)
            data[i] = 0;

      for (j=0; j<SENSOR_READINGS; j++) {
            /* reset */
            na_capt_rst->np_piodata = 1; na_capt_rst->np_piodata = 0;
            for (i=0; i<NB_SENSORS; i++) {
                  /* with led */
                  na_capt_clk->np_piodata = 1; na_capt_clk->np_piodata = 0;
                  s = adc_acquire(na_spi_0, 5);
                  /* without led */
                  na_capt_clk->np_piodata = 1; na_capt_clk->np_piodata = 0;
                  t = adc_acquire(na_spi_0, 5);
                  data[i]+=s-t;
            }
      }

      /* calc average */
      for (i=0; i<NB_SENSORS; i++) {
            data[i] = data[i] / SENSOR_READINGS;
      }
}
```