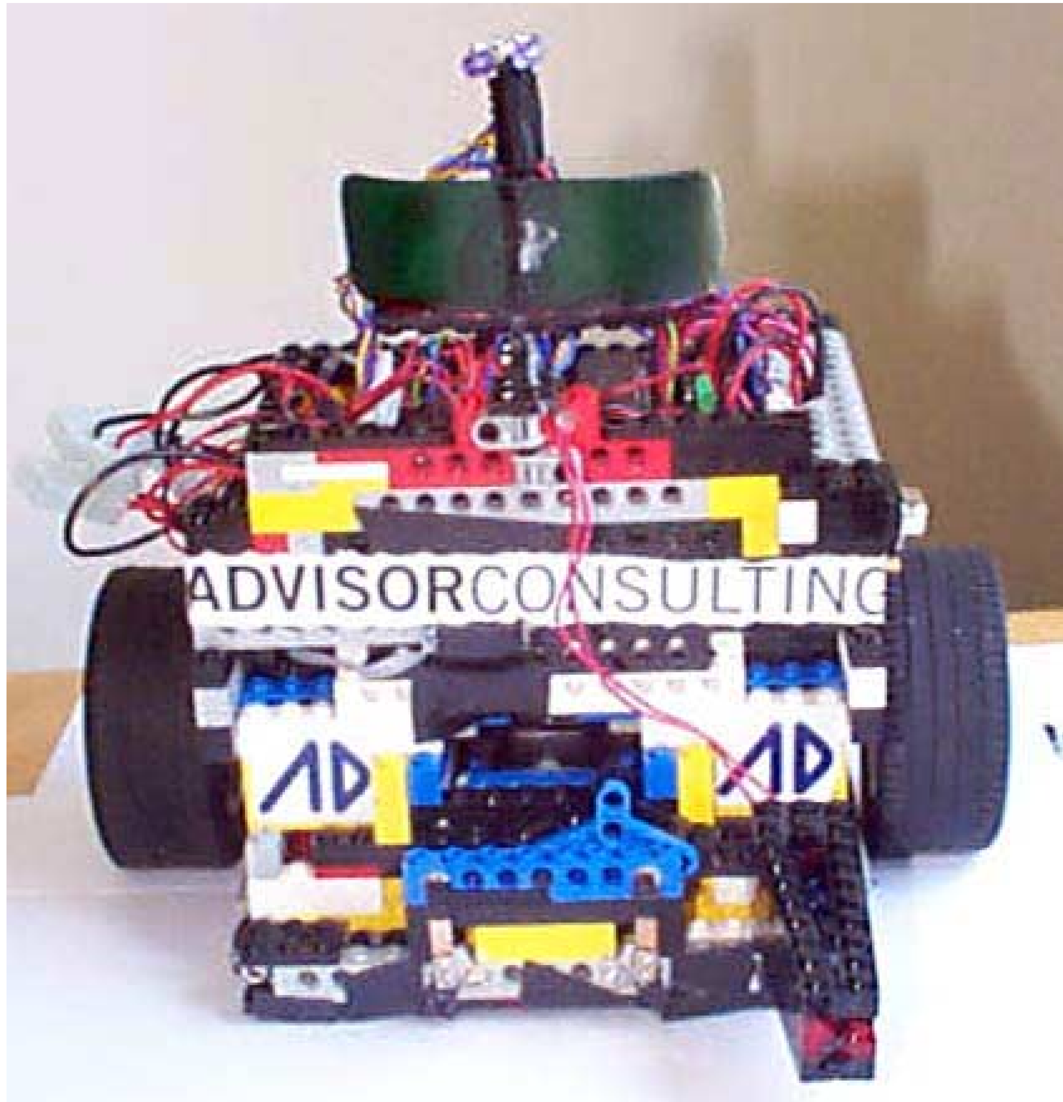
2D1426 Robotics and Autonomous Systems

Project Report

Group 5: “Ad Hoc”.



Frontal view of the vicious “Ad Hoc” robot

Team members:

Heijkenskjöld, Hannes `<d96-hhe@nada.kth.se>`
Kalén, Martin `<d96-mka@nada.kth.se>`
Malmesjö, Stefan `<d96-sma@nada.kth.se>`
Hoc, Ad

Abstract

This is a report of the project for the course *2D1426 Robotics and Autonomous Systems.* It describes the ideas and the design of the *Ad Hoc* robot. It covers the main design goals, implementation and results.

The goal of the robot was to play table hockey, and above all to be able to score a goal. This was actually accomplished during the qualification round for a table hockey tournament. However, the robot never managed to score during any of the actual games. It was built of Lego bricks, and controlled by a micro controller.

Table of Contents

The task at hand 1
Design 1
  Sensing and sensory processing 2
    IR 2
    Reflex detectors 2
    Micro switches 3
  Locomotion and navigation 3
  Manipulation 3
  Integration of the above 4
Software implementation 5
  Behaviors 5
  Behavior collaboration 5
Programming environment, computer hardware, and electronics 6
  Programming environment 6
  Computer hardware 6
  Electronics 6
    IR-detectors 6
    Reflex detectors 6
    Micro switches 6
    Motor wiring 6
    Stick 7
    DIP switches 7
    Bootloader 7
Results 7
Conclusions 7
Appendix 1 – Parts List 8
Appendix 2 – References 9
Appendix 3 – Circuit Diagrams 10
  Motherboard 10
Appendix 4 – Source Code 11
    Adhoc.c 11
    Behavior.h 16
    Behavior.c 18
    Bumplib.h 24
    Bumplib.c 25
    Ir_servo.h 26
    Ir_servo.c 27
    Pwmlib.h 30
    Pwmlib.c 30
    Stdfunc.h 32

The task at hand

The task of the project was to build and program a robot that could play table hockey with special rules, and that fulfilled constraints on the construction. The robot was to be built from Lego bricks and meccano, with the addition of whatever bits and pieces the constructors thought necessary. The teams were assigned two 12 V DC motors, one 6 V motor and an RC servo, which were used for locomotion and for manipulating the stick. The robot was controlled by a small 8-bit micro controller (see Appendix 1 – Parts List; PIC16F877 and [4]). To find its way on the hockey field, the robot could be equipped with a variety of sensors, such as Infra Red light sensors, reflex detectors and micro switches.

The constraints imposed on the robot were among others the following (for complete rules see [5] and [6, Part III]):

- Diameter of the body was to be less than 250 mm.
- No concave part on the body.
- The stick could be no wider or longer than the puck, and should not hide the latter's IR emitters.
- The stick could have no moving joints outside the robot's body.

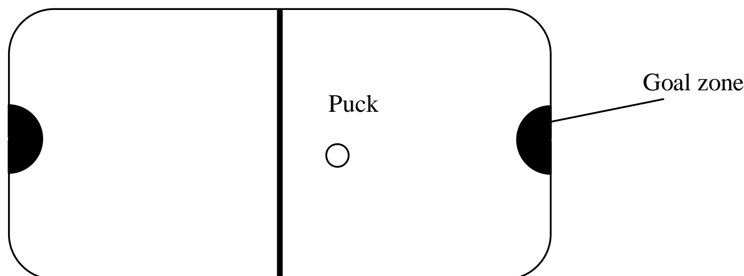


Fig. 1 The hockey field

The hockey field (see Fig. 1) is a white surface, 2.4 x 1.2 m, with a black goal zone in both ends. The robot was not allowed to enter either goal zone, so some kind of sensing of the proximity to the zone was necessary. Both goals emit IR light with different pulse modulations, as does the puck. In order to be able to see each other, the competing robots also radiate IR (see Appendix 1 – Parts List; SFH485P and [3]) with modulation different from that of the puck and the goals.

To score a goal, the puck had to be put within 2 cm of the opponent's goal, without the robot entering the goal zone. Pushing or shooting the puck into the goal could accomplish this. If the robot entered the zone the goal was not counted.

Given these premises, the team designed, built and programmed the *Ad Hoc* robot. This report presents the design ideas, goals and implementation, and the final results and conclusions.

Design

The main goal of the overall design of the robot was to have a small, agile and fast robot, being able to outmaneuver the opponent and scoring a goal by driving the puck into goal with high speed. The team's goal for the robot was to have several behaviors, and then somehow weigh these together in order to achieve a "smart" result. To do this, vector summation was used, where each behavior contributed with one vector.

Fig. 2 Robot side view

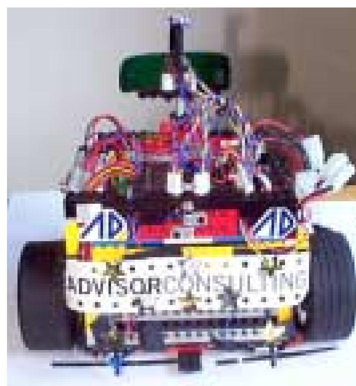


Fig. 3 Robot rear view

Sensing and sensory processing

Three different kinds of sensors were used in the construction; IR detectors (see Appendix 1 – Parts List; TSL261), reflex detectors (see Appendix 1 – Parts List; ITR8307) and micro switches. The input from these devices was used for decision making in navigation. The IR detectors were used to determine direction and distance to the goals, the puck and the opponent (see [2]). The reflex detectors were used to tell whether the robot were in a goal zone or not. The micro switches were used to detect if the robot had run into something. They were also used to determine whether the puck was in the possession of the robot or not. Two different code libraries were used for sensory input, bumplib and ir_servo.

IR

The robot had a separate processor, PIC16F876 (see [4]), which handled both IR signals and the R/C servo. This processor was preprogrammed so it could make out the difference between the four incoming IR signals: the puck, the opponent, the defensive goal and the offensive goal. These four targets continuously sent IR pulses, but they did so with different frequencies, so that they could be separated by the IR-PIC, and the targets distinguished.

Five IR sensors, TSL261 (see [2]), were placed on the robot. Two situated in the front (they were placed in 90° angle), two in the rear (also 90°), and one forward facing sensor in the front. The four ones first mentioned, called "high sensors", were mainly used for estimating the direction to the target – but also the distance from it. The sole front sensor, called "low sensor", was used to measure the distance.

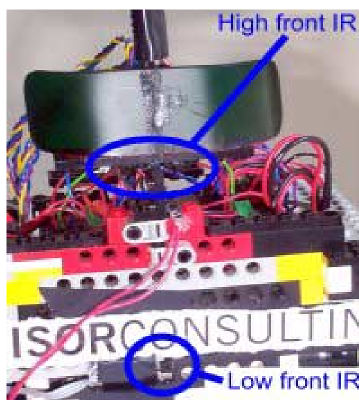


Fig. 4 Front IR sensors

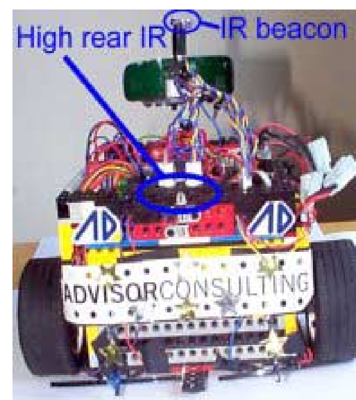


Fig. 5 Rear IR sensors and IR position beacon

Since the sensor values were sensitive to disturbances, and hence had a considerable amount of noise, mean value calculation over the two last readings was used, in order to get better values.

The exact angle to the target was never calculated. Instead, a relative measurement of the IR readings was done. First, the difference between the sum of the two rear sensors and the sum of the two front sensors was calculated. This way it was determined whether the target was in front of or behind the robot. Analogously, it was calculated whether the target was to the left or the right of the robot. The relative difference between the calculated values was used in order to calculate how much to turn each of the wheels. This worked rather well.

Whenever the need for calculating the distance to a target (for example a goal) arose, the way to figure it out was to use the strength of the IR readings. These were calibrated for each individual task, to fit accordingly. It would show that this was harder than originally thought, since the absolute value of the readings varied significantly from run to run.

IR filtering was controlled by the slave micro controller. The only thing that had to be handled in software was the interrupt generated when a complete byte had been sent on the serial line between the two processors (SPI) and to receive the IR data in correct order. This was done in the service routine in the ir_servo library ir_sevice_routine().

Reflex detectors

Under the robot, facing down toward the table, four reflex detectors were placed – one in each corner of the robot. These detectors sent out IR light, and measured how much was reflected from the surface of the hockey field (see parts list and [1]). This reflection was presented to the robot with a value between 0 (white) and 255 (black), after converting the analog signal with the PIC's built-in A/D converter. The reflex detectors were connected to the AN-ports (analog in). The function bump_poll_sensors() in bumplib was used to run the A/D converter to sample all reflex detector channels.

This information was used for estimating whether the robot was going into a goal zone or not, which was not allowed. Since the field had a black stripe for marking the middle of the field, the reflex detectors together with information from the IR sensors were used to estimate if the robot was in a goal zone or merely on the middle field stripe. So if the reflex detectors signaled black surface, and the IR sensors signaled goal proximity, then robot was positioned in a forbidden zone, and hence had to quit going in that direction.

Micro switches

The micro switches were used for tactile sensing. Attached to the micro switches the robot had antennae, which would depress the switches if it ran into something. Seven switches were used; one placed in each corner of the robot, one on the stick, and two in the front of the robot. The two in the front were used for detecting puck possession, and the others were used for obstacle detection. All bumpers and an optional emergency stop were connected to PORTB on the PIC. The function bump_poll_sensors() in bumplib was used to update bumper status, e.g. reading PORTB. As mentioned in the "Reflex detectors" section above, this function also samples reflex detectors.

Locomotion and navigation

There are different ways of moving a robot. The Ad Hoc robot was built with two wheels, which were run independently of each other using two separate 12 V motors. This made it possible for the robot to turn around on the spot if necessary, by running the wheels in opposite directions. In order for the robot to keep its balance, a small, smooth piece of Lego, placed in the front, was used for support.

For navigation the team used a behavioral approach. The idea was to program the robot with several behaviors, active at the same time. They were all to have their own will, and then some coordinator was to combine these wills, and produce some appropriate action for the robot. Each behavior should produce a "wish vector", telling the direction in which the behavior "wanted" to go. This way all the behaviors got to tell their wish, and then vector summation was used to calculate a final direction for the robot. Depending on in which state the robot was (having the puck or not) the different behaviors' wishes were assigned different weight, which were used in the vector summation.

There were five different behaviors in the robot.

- The find puck behavior – always wanted to go toward the puck, unless the puck already was in the robot's possession. If the robot for some reason didn't see the puck, it simply spun around, trying to locate the puck.
- The get puck to goal behavior – always wanted to go toward the offensive goal, as long as the robot had the puck.
- Avoid obstacles behavior – if the robot ran into something, this behavior tried to go in a different direction from the original one. It also stopped the robot from going into a goal zone.
- Avoid opponent behavior – tried to get the robot not to run into the opponent.
- Random behavior – produced a random (small) wish. This was to avoid that the other behaviors would sum up to a no wish, i.e. to wish the robot in exactly opposite directions. A small random vector wouldn't affect the behaviors normally, but in such a case it could add enough to get the robot out of the clash.

All different behaviors were implemented in the behavior module and the wish summation and actual motor speed settings done in the main file, function sequencer().

Manipulation

In order to be able to control the puck, the robot had a stick – seen in the figures below. The stick design was somewhat problematic, since the robot should be able to turn in different directions and still keep the puck (if the robot had it, of course). The team decided that the best way to achieve this was by swinging the stick *over* the puck, not just from the right to the left – which would have made us lose the puck more easily. In order to achieve this, an R/C servo was connected to the stick, to be able to steer it into the appropriate position at all times.

Fig. 6 Stick maneuver from full left to full right

The appropriate position is dependent of the robot's current state. If in possession of the puck (which was determined by combining the information from the front bumpers and the IR readings), the robot should keep the stick on the opposite side from the direction in which the robot was turning. This would make the stick swing virtually all the time, if not suppressed by a turning rate threshold in the software. If the robot were to turn less than this threshold value, the stick swing was skipped. This proved to work rather well; the robot could turn pretty much without loosing the puck.

In the code, the check_club() function in the main program file was used to position the stick on the appropriate side of the puck. If the robot had possession of the puck and it was turning rather fast, the motors were stopped until the stick swing was complete. Otherwise, e.g. if approaching the puck from a distance, there was no need to stop the robot and move the stick all the time, so in this case the function would exit right away. This proved to be rather successful as fast movements from side to side were not carried out at all. The ir_servo lib was used for the actual servo position setting, as servo positioning was also controlled by the slave PIC and sent through the SPI interface used for IR data reception. The function servo_set_club() and the macros CLUB_LEFT and CLUB_RIGHT were used for setting the right position data to transmit on the SPI line.

Integration of the above

To put all this together, a state machine was implemented, which kept the robot in the correct state at all times. There were three states, but more could easily have been added, if needed. The three states were "Have puck", "Don't have puck", and "Score". The "Score" state was rather special, so a completely predefined movement was used for that one. The other two weighed all the behaviors' wishes together, and used vector summation to calculate a final direction vector. In both of these states, the most important behavior was to avoid obstacles (among other things because the robot was not allowed to enter the goal zones), so the weight of this behavior was greater than the sum of the weights of the rest of the behaviors. This way, the avoid obstacles behavior would always get its way. The avoid opponent behavior had a weight that was determined by switches on the motherboard (see section Electronics; DIP switches) with a value ranging from 0 to 3, setting the overall "aggressiveness" of the robot. The random behavior was set to a low weight, so it wouldn't interfere with the others too much. The only use for the random behavior was to avoid livelock, when all the other behaviors' wishes would add up to a zero vector.

- Don't have puck – here the find puck behavior was important, so the weight was set rather high.
- Have puck – now the robot's goal was to get the puck to goal, which is why the weight for that behavior was set high.

The "statemachine" was implemented in the main program file, with behavior activation in the main() function and state control in the decice_state() function.

After the main loop had asked all the behaviors to tell their wishes, the sequencer was told to carry out the order. The sequencer then did the vector summation from all the behaviors' wishes. When it knew which way to go, it decided whether or not to swing the stick, and then it gave speed orders to the two wheels, so that the robot would move in the wanted direction.

The score state was activated whenever the robot was in possession of the puck, was facing toward the opponent's goal, and was close to it. In order for the robot to always try to score, regardless of the opponent's position, a completely separate movement scheme was programmed for this purpose. In order to accomplish this, the robot was set to the maximal speed possible (so the puck would get enough velocity to reach all the way to the goal when the robot stopped at the goal zone), and then kept an eye at the IR readings to know when enough proximity to the goal was achieved. When the robot got close enough – or when the reflex detectors signaled that the robot was in the goal zone – it would stop, and hope that the puck would get into the goal.

Software implementation

Behaviors

Each behavior described in *Locomotion and navigation* is implemented in a function in *behavior.c* (see Appendix 4 – Source Code). These functions calculate a wish vector, indicating in which direction the behavior 'wishes' to go. The behavior algorithms work as follows:

- behavior_get_puck() – This behavior uses the IR readings of the puck to determine if the puck is in front of or behind the robot, and if the puck is to the left or the right of the robot, as described in the section *IR* above. If the readings are too small on all sensors, the puck is not seen, and the wish vector is set to full right spin. If the puck is behind the robot, the y-component of the vector (forward-backward component) is set to zero, because the robot shouldn't back into the puck. When the puck is in front of the robot, the x-component (left-right component) of the wish vector is set to zero if below a certain threshold, in order to avoid an oscillating behavior when closing in on the puck. Otherwise, the largest of the x and y values is set to MAX_WISH, and the other is scaled accordingly.
- behavior_puck_to_goal() – This behavior uses almost the same approach as behavior_get_puck() when calculating where the target is. The main difference is that no consideration is taken as to whether the goal is behind the robot. This is because the vast amount of disturbances from other IR sources, which makes it virtually impossible to find the goal in a reasonable manner when the goal is behind the robot. Instead the behavior assumes that if the goal isn't in front of the robot, it can be found behind the robot, hence the behavior wants to stand and turn in order to try to find the target. On the other hand, if the goal is in front of the robot, the same scaling of the x and y vector components as in the behavior_get_puck() is carried out.
- behavior_avoid_opponent() – This behavior is the opposite of behavior_get_puck(), but when talking about the opponent, not the puck. The wish vector is calculated in the exact same manner, and then the direction is reversed in order to get a repelling conduct. One difference is that if the opponent isn't seen, the behavior doesn't want to turn around to try to find it. Instead a zero wish is produced.
- behavior_avoid_obstacles() – Tries to avoid running into obstacles. To accomplish this, the behavior investigates whether any of the micro switches has been depressed. If this is the case, the behavior wishes to go in the opposite direction from where the depressed switch is situated. The input from the front bumpers is suppressed when carrying the puck, since otherwise the behavior will interpret the puck as an obstacle. More important is the reflex detector inputs, used to avoid running into the goal zones. If any of the reflex detectors indicates that the surface is black, and the IR sensors give a strong goal signal reading, the behavior assumes that the robot is in a goal zone, be it offensive or defensive, and therefore gives a strong wish to go in the reverse direction.
- behavior_random() – produces a small random wish, in order to avoid livelocks. This wish is small enough not to disturb the other behaviors' wishes, but sufficient enough to generate a movement if all the other behaviors' wishes were to sum up to zero.

Behavior collaboration

Which behaviors are used is decided by which state the robot is in. The starting state is STATE_NOPUCK. The state machine is implemented in main() in *adhoc.c* (see Appendix 4 – Source Code). In each loop of the state machine decide_state() is run to decide if a new state should be entered, based on the sensor input available to the robot. Also, the weight vector is updated according to the new state. In each state a call to the appropriate behavior functions is carried out, in order to update the wish vector. Then sequencer() is run to combine the wishes from the behaviors.

The state switching logic in decide_state() works as outlined below:

```
if in STATE_SCORE
            → next state is STATE_NOPUCK (just made a goal, "reset" statemachine)
else if the puck is in the robot's possession
            if the robot is within shooting range
                        robot can try to make a goal → next state is STATE_SCORE
            else
                        robot has to go for the goal → next state is STATE_OFFENCE
else (the robot does not have the puck)
            if the puck is in the opponent's goal zone
                        robot unknowingly scored → next state is STATE_SCORED
            else
                        robot has to go for the puck → next state is STATE_NOPUCK
```

The sequencer() uses some heuristic methods for deciding the final input to the robot's motors. The algorithm works as outlined below:

```
if total y-wish is zero
            if total x-wish is zero
                        y=0 and x=0 → stop
            else
                        y=0 but x!=0 → stand and turn
else
            turning rate ::= abs(x-wish/y-wish)
            if (turning rate < 0.6)
                        small angle ≈ no angle → straight ahead
            else if (turning rate < 1.0)
                        medium angle → ahead with slight turn
            else
                        large angle → stand and turn
```

Programming environment, computer hardware, and electronics

Programming environment

The programming environment used throughout development of the robot control software consisted of the following software:

- GNU Emacs text editor (see [7])
- Microchip MPLAB™ Integrated Development Environment (see [8])
- Hi-Tech PIC C Compiler (see [9])
- Hi-Tech PIC Object Linker.

Computer hardware

The computer hardware on the robot consisted of two micro controllers from Microchip; the main processor PIC16F877 (see [4] and Appendix 3 – Circuit Diagrams; Motherboard) and the slave processor PIC16F876 (see [4] and Appendix 3 – Circuit Diagrams; Motherboard).

Electronics

The provided motherboard was extended with wiring and sensory equipment. See Appendix 3 – Circuit Diagrams; Motherboard for a schematic picture of the motherboard wiring.

IR-detectors

The five IR-detectors were connected to the J3-J7 connectors, see section B1, C1 and D1 in the motherboard schematics.

Reflex detectors

The reflex detector transmitters were daisy chained and connected to a driver board, provided by the course administration. The four reflex detector sensors were connected to the analog input pins AN0:3 connected to J15, pins 2-5, see section C5 in the motherboard schematics.

Micro switches

The 7 micro switches used as bumpers were connected to PORTB (RB1:7) connected to J2 (pins 7-1) and ground on J13 (pins 7-1), see section C6 in the motherboard schematics.

Motor wiring

The PIC ports CCP1 and CCP2 were used as PWM output signals to the 12V motors (driving the wheels). The left motor used CCP1, through a wire connected from J16 pin 7 (CCP1) to J8 pin 1 (MOTOR1 enable), and the right motor used CCP2, through a wire connected from J16 pin 6 (CCP2) to J8 pin 3 (MOTOR2 enable) – see sections D5 (CCP) and G5 (MOTOR enable) in the motherboard schematics.

The PIC ports RE1 and RE2 on PORTE were used as direction control signals to the 12V motors. The left motor used RE2, through a wire connected from J15 pin 10 (RE2) to J8 pin 2 (MOTOR1 direction), and the right motor used RE1, through a wire connected from J15 pin 9 (RE1) to J8 pin 4 (MOTOR2 direction) – see sections C5 (RE1,2) and G5 (MOTOR direction) in the motherboard schematics.

The 12V motors were connected to the MOTOR outputs at J11 (left motor as MOTOR1 and right motor as MOTOR2), see section F8 in the motherboard schematics.

Stick

The stick servo was connected to J22 pin 4 (SERVO1 control), J23 pin 7 (Vcc) and J24 pins 7 (GND).

DIP switches

Two of the DIP switches S2, connected to J10 pins 1 and 2, were used by the software to determine the weight for the "avoid opponent" behavior. Wires connected J10 pin 1 and 2 to J21 pins 9 and 11 (PORTD, RD3 and 2) – see sections F2 (S2), D2 (J10) and D7 (J21) in the motherboard schematics.

Bootloader

The Microchip programming environment offers convenient programming of the PIC processor through a serial RS232-interface and a piece of code in the PIC called "bootloader" (supplied with the Hi-Tech PICC Compiler, see [9]). The RS232 interface is not used by the robot during competition and is not covered in this document, see [8] for more information.

Results

It was quite hard to program the robot to do the right things. The base implementation in the state machine and behavior algorithm was rather sturdy, and never had to be changed much from the original idea. The main problem was finding the various threshold values for the sensor readings, e.g. IR signal strength threshold for stopping the motors when scoring. Sometimes the robot had a tendency to stop way too early, and sometimes way too late. This problem was also related to the fact that the robot had its own mind as to which speed it wanted to travel with. Even when the speed wasn't changed in hardware, it varied tremendously from run to run, and it was never clarified why. (One thought was that one of the accumulators was broken with a "memory effect" and stopped charging at low capacity.) It was also hard to make the robot not to think that the puck was the opponent goal. It seemed that the IR signals from the puck and the goal were alike enough to get the robot seriously confused in some situations. A threshold value for when to think that the signals from the goal actually were the goal and not the puck was sought, but this was also a really hard task to accomplish.

Ad Hoc did not do very well in the table hockey tournament. It lost or played even against all opponents. In one match it even broke down and had to be removed from the table for the entire game. For some reason it didn't run the program in the computer. Still no one knows why this happened.

The different behaviors seemed to work together well, although most of the time it was always one behavior that got its way, and the others' wishes were totally suppressed. Maybe with more time at hand, the weights between the wishes could have been calibrated more, to make the behaviors collaborate better.

Conclusions

The domain in which the robot was to operate was a well-defined one: a hockey field, a puck, an opponent and two goals. In this kind of environment, the behavior based approach, with a bunch of wishes and a summation algorithm might not be the best one to use. Maybe more well-defined locomotion schemes for different situations should have been used, not trying to mix different wills together. It proved rather hard to weigh together all the wishes, and in the end the weights were virtually so that only one behavior got all of the saying at each given moment.

When talking about the physical appearance of the robot, the construction was fairly good. The only thing where some problem occurred was the little piece of Lego we used to keep the robot from tipping over. This seemed to get pretty much pressure, and at some times it seemed like it slowed the robot down. Some thoughts as to replace this piece with a castor was put fourth, but that would have made the robot so high off the ground that the stick would have covered the IR transmitters of the puck, which it was not allowed to do, so the smooth piece of Lego was kept instead.

Appendix 1 – Parts List

ITR8307 — Subminiature High Sensitivity Photo Interrupter.
Manufacturer: Everlight. **Part no**: ITR8307.

PIC16F877 — 40-pin 8-Bit CMOS FLASH Micro controller.
Manufacturer: Microchip Technology Inc. **Part no**: PIC16F877.

SFH485P — GaAlAs Infrared Emitter.
Manufacturer: Siemens. **Part no**: SFH485P.

TSL261 — IR Light-to-Voltage Optical Sensor.
Manufacturer: Texas Advanced Optoelectronic Solutions (TAOS). **Part no**: TSL261.

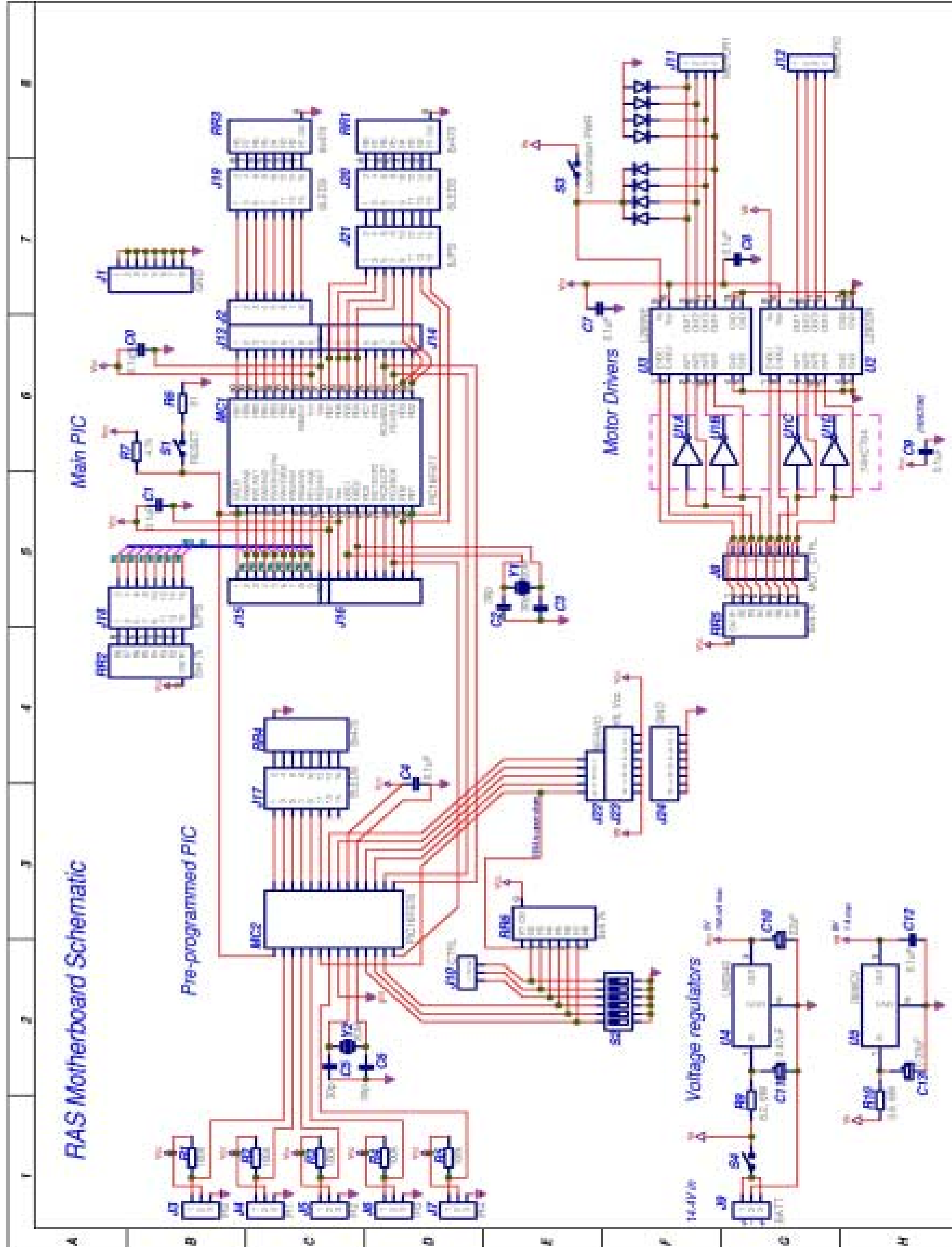
HL149 — 12V 10:1 Motor
Manufacturer: Micromotors. **Part no**: HL149.

Appendix 2 – References

1. Data sheet for ITR8307. ***Subminiature High Sensitivity Photo Interrupter***. Everlight.
2. Data sheet for TSL261. ***IR Light-to-Voltage Optical Sensor***. Texas Instruments.
3. Data sheet for SFH485P. ***GaAlAs Infrared Emitter***. Siemens.
4. Data sheet for PIC16F87X. ***40-pin8-Bit CMOS FLASH Micro controller***. Microchip.
5. Competition rules. ***Updated competition rules part 1 and 2***. `http://www.nada.kth.se/kurser/kth/2D1426/rules1.gif`, `http://www.nada.kth.se/kurser/kth/2D1426/rules2.gif`. Bratt, Mattias.
6. Course Notes for the Robotics and Autonomous Systems course 2000. ***Course Notes I-IX***. Bratt, Mattias.
7. The GNU Project. ***GNU Software***. `http://www.gnu.org/`.
8. MPLAB Manual. ***MPLAB***™ ***IDE, Simulator, Editor – User's Guide***. Microchip.
9. PIC C Manual. ***PIC C ANSI C Compiler – User's Guide***. Hi-Tech.

Appendix 3 – Circuit Diagrams

Motherboard



Appendix 4 – Source Code

Adhoc.c

```
/*
 * adhoc - main program
 */
#include <pic.h>
#include <stdlib.h>
#include <math.h>
#include <sys.h>
#include "behavior.h"
#include "pwmlib.h"
#include "ir_servo.h"
#include "bumplib.h"

// Possible states
#define STATE_NOPUCK   3
#define STATE_OFFENCE  2
#define STATE_SCORE    1
#define STATE_SCORED   0

// The state we're in
bank1 char state = STATE_NOPUCK;

// Motor speed and wish constants
#define MAX_SPEED              110
#define MAX_TURN_SPEED         127
#define GOAL_SPEED             120
#define GOAL_SPEED_TIME        40000
#define TURN_MULTIPLIER        85
#define CLUB_SWING_MOTOR_SPEED 78
#define CLUB_SWING_TIME        100000
#define SEQUENCER_STRAIGHT_AHEAD_RATIO 0.60 // xyratio below this => drive straight ahead
#define SEQUENCER_FINE_TURNING_RATIO   1.00 // xyratio between this and above => drive and fine tune
#define CLUB_SWING_MOTOR_DIFF  SEQUENCER_STRAIGHT_AHEAD_RATIO * TURN_MULTIPLIER

// The final bearings after weight- and mean-calculations
bank1 long x;
bank1 long y;

// Speed calculation variables
bank1 char   left_speed;
bank2 char   right_speed;
long   w;
bank3 double xyratio;
persistent long goal_speed_time;

// Global iterators...
bank2 int  i;
bank2 char j;
bank2 long loop;


/* emergency_stop
 *  External interrupt has signalled request for fast shutdown!
 */
void emergency_stop(void) {
     pwm_stop_motors();
     servo_set_club(0);

     while (1) {
               ; // deep sleep
     }
}
```

```
/* check_club
 * If we have puck and turning,
 * see to it that the club is in the right position
 *
 * Parameters: left_motor - wanted left motor speed
 *             right_motor - wanted right motor speed
 */
void check_club(char left_motor, char right_motor) {
     static bank2 bit wait_for_swing;

     wait_for_swing = 0;
     if ((left_motor - right_motor > CLUB_SWING_MOTOR_DIFF) &&
               (servo_club_pos == CLUB_RIGHT)) {
                // turning right, need to have club on left side!
                servo_set_club(CLUB_LEFT);
                if (have_puck())
                     wait_for_swing = 1;
     }
     else if ((right_motor - left_motor > CLUB_SWING_MOTOR_DIFF) &&
               (servo_club_pos == CLUB_LEFT)) {
                // turning left, need to have club on right side!
                servo_set_club(CLUB_RIGHT);
                if (have_puck())
                     wait_for_swing = 1;
     }

     if (wait_for_swing) {
                // we have the puck, wait a while so club really gets swinged to other side
                for (loop = 0; loop < CLUB_SWING_TIME; loop++) {
                     if (loop == CLUB_SWING_TIME / 4) { // start motors after half swing time
                               if (servo_club_pos == CLUB_LEFT)
                                         pwm_set_motor_speed(-CLUB_SWING_MOTOR_SPEED,
CLUB_SWING_MOTOR_SPEED);
                               else
                                         pwm_set_motor_speed(CLUB_SWING_MOTOR_SPEED, -
CLUB_SWING_MOTOR_SPEED);
                     }
                     // stop motors early (momentum!)
                     else if (loop == CLUB_SWING_TIME * 3 / 8) {
                               pwm_stop_motors();
                               break;
                     }
                }
     }

}


/* stand_and_turn
 *  y is small or angle is large so we want to stand and turn with speed speed
 */
void stand_and_turn(char speed) {
     if (x > 0) {      // Turn right
                left_speed  =  speed;
                right_speed = -speed;
     }
     else if (x < 0) { //Turn left
                left_speed  = -speed;
                right_speed =  speed;
     }
}
```

```
/* sequencer
 *   weigh all behaviour X- and Y- wishes and calculate mean,
 *   run motors accordingly
 *
 * Params: none
 * Return: sets global x and y
 */
void sequencer(void)
{
     // reset bearings
     x = 0;
     y = 0;
     w = 0;

     for (i=0; i<NUM_BEHAVIORS; i++) {
               // get all behaviors' wishes
               x += behavior_bearing_wish[i][X] * behavior_weight[i];
               y += behavior_bearing_wish[i][Y] * behavior_weight[i];
               // calculate total weight
               w += behavior_weight[i];
     }

     // Calculate the mean wish from all the behaviors
     x /= w;
     y /= w;

     // Now, calculate the motor speeds
     left_speed  = 0;
     right_speed = 0;

     // If no forward wish
     if (y == 0) {
               // If there is a lateral wish
               if (x != 0) {                // Turn
                    stand_and_turn(MAX_TURN_SPEED);
               }
               // else we don't do anything -> stop
     }
     else {
               xyratio = fabs(((float)x)/y);
               /* If we're going really straight,
                * set full speed on both motors to avoid oscillating behavior
                */
               if (fabs(xyratio) < SEQUENCER_STRAIGHT_AHEAD_RATIO) {
                    left_speed =  y >= 0 ? MAX_SPEED: -MAX_SPEED;
                    right_speed = left_speed;
               }
               // If we're going kinda straight, let's go forward and turn somewhat
               else if (fabs(xyratio) < SEQUENCER_FINE_TURNING_RATIO) {
                    /* We don't need to take care of
                     * x == 0, since that makes xyratio == 0,
                     * and both motors will have the same speed */
                    if (x > 0) { // We wanna go to the right...
                              left_speed  = MAX_SPEED;
                              right_speed = (signed char)(MAX_SPEED - xyratio * TURN_MULTIPLIER);
                    }
                    else { // Leftward driving
                              right_speed = MAX_SPEED;
                              left_speed  = (signed char)(MAX_SPEED - xyratio * TURN_MULTIPLIER);
                    }
                    // This is for backwards driving
                    if (y < 0) {
                              right_speed = -right_speed;
                              left_speed  = -left_speed;
                    }
               }
               // If the ratio is larger, stand in one place and turn
               else {
                    stand_and_turn(MAX_TURN_SPEED);
               }
     }

     // Put the club in the right position
     check_club(left_speed, right_speed);

     pwm_set_motor_speed(left_speed, right_speed);
}
```

```
/* Decides what state we should be in based on new readings of sensor input
 */
char decide_state(char state)
{
     static bank1 char new_state;
     static bank1 char avoid_opp_weight;

     // update environment info
     avoid_opp_weight = (RD3 << 1) | RD2; // avoid opponent weight
     bump_poll_sensors();                 // update bumpers
     ir_wait_for_data();                  // update IR
     for (i = 0; i < 5; i++) {
               for (j = 0; j < 3; j++) {
                    ir_temp_data[i][j] = ir_data[i][j];
               }
     }
     ir_wait_for_data();  // new IR for average calculations
     for (i = 0; i < 5; i++) {
               for (j = 0; j < 3; j++) {
                    ir_avg_data[i][j] = (ir_temp_data[i][j] + ir_data[i][j]) / 2;
               }
     }

     if (state == STATE_SCORE) {
               new_state = STATE_NOPUCK;
     }
     else if (have_puck()) {
               // to STATE_OFFENCE or STATE_SCORE if nearby opponent goal
               if (within_shooting_range()) {
                    new_state = STATE_SCORE;
                    /* no weight adjustment needed for scoring,
                     * this will soon be done in STATE_NOPUCK
                           */
               }
               else {
                    new_state = STATE_OFFENCE;
                    behavior_weight[BEHAVIOR_GET_PUCK] = 0;
                    behavior_weight[BEHAVIOR_PUCK_TO_GOAL] = 4;
                    behavior_weight[BEHAVIOR_AVOID_OBSTACLES] = 6+avoid_opp_weight;
                    behavior_weight[BEHAVIOR_AVOID_OPPONENT] = avoid_opp_weight;
                    behavior_weight[BEHAVIOR_RANDOM] = 1;
               }
     }
     else if (puck_in_zone()){ // No puck but puck within goal zone -> STATE_SCORE
               new_state = STATE_SCORED;
     }
     else { // No puck
               new_state = STATE_NOPUCK;
               behavior_weight[BEHAVIOR_GET_PUCK] = 4;
               behavior_weight[BEHAVIOR_PUCK_TO_GOAL] = 0;

               behavior_weight[BEHAVIOR_AVOID_OBSTACLES] = 6+avoid_opp_weight;
               behavior_weight[BEHAVIOR_AVOID_OPPONENT] = avoid_opp_weight;
               behavior_weight[BEHAVIOR_RANDOM] = 1;
     }

     if (new_state != state) {
               /* if changing state,
                * we want to clear the vector
                * (to avoid disabled behaviors' wishes from summing up)
                */
               for (i=0; i < NUM_BEHAVIORS; i++)
               {
                    behavior_bearing_wish[i][X] = 0;
                    behavior_bearing_wish[i][Y] = 0;
               }
     }
     return new_state;
}
```

```
/* We think we have scored a goal!! Do a noticeable gesture
 */
void scored(void) {
     // Back up
     pwm_set_motor_speed(-MAX_SPEED, -MAX_SPEED);
     for(loop = 0; loop < 150000; loop++);

     // Stop, wave the club and flash some lights
     pwm_set_motor_speed(0,0);
     for(i = 0; i < 2; i++) {
               servo_set_club(CLUB_RIGHT);
               PORTD = 0b01010101;
               for (loop = 0; loop < CLUB_SWING_TIME; loop++);
               PORTD = 0b10101010;
               servo_set_club(CLUB_LEFT);
               for (loop = 0; loop < CLUB_SWING_TIME; loop++);
     }
     PORTD = 0;
}

/* Try to score a goal by driving the puck into the goal zone at high speed
 */
void score(void)
{
     // Goal proximity reached, full speed ahead and SCORE!
     pwm_set_motor_speed(GOAL_SPEED, GOAL_SPEED);

     for (loop = 0; loop < GOAL_SPEED_TIME; loop++) {
               bump_poll_sensors();
               if (bump_reflexes != 0 ||
                    ir_avg_data[IR_FRONT_LOW][OFFENSIVE_GOAL] > IR_ZONE_LOWSENS_THRESHOLD) {
                    // reflex detectors signals goal zone or IR signal is very high -> STOP!!!
                    break;
               }
     }
     pwm_set_motor_speed(-MAX_SPEED, -MAX_SPEED);
     scored();
}

void main(void)
{
     // Use PORTD LEDs for status indication and debugging, RD3..2 for avoid opponent weight
     TRISD = 0b00001100;
     PORTD = 0;

     // Init all low-level libraries
     di();
     pwm_init();
     pwm_set_motor_speed(0, 0); // Stop motors
     pwm_left_comp  = 1.0;      // Motor downscaling factors
     pwm_right_comp = 0.98;

     bump_init();
     INTE=1;

     ir_init();
     PEIE=1;

     ei();

     ir_wait_for_data();   // wait for at least one complete IR-reading
     servo_set_club(CLUB_RIGHT);
```

```
     // Statemachine
     while (1)
     {
               state = decide_state(state);
               switch (state) {
               case STATE_NOPUCK:
                    behavior_get_puck();
                    behavior_avoid_opponent();
                    behavior_random();
                    behavior_avoid_obstacles();
                    sequencer();
                    break;
               case STATE_OFFENCE:
                    behavior_puck_to_goal();
                    behavior_avoid_opponent();
                    behavior_avoid_obstacles();
                    behavior_random();
                    sequencer();
                    break;
               case STATE_SCORE:
                    // Score and signal!
                    score();
                    break;
               case STATE_SCORED:
                    // Ooops, we thought we didn't have the puck but it's a clear goal - sinal!
                    scored();
                    break;
               default:  // Something is wrong, let's reset state...
                    state = STATE_NOPUCK;
                    break;
               }
     }
}

void interrupt ISR(void)
{
     if (INTF) {
               INTF = 0;
               emergency_stop();
     }
     else if (SSPIF)
               ir_service_routine();
}
```

Behavior.h

```
/*
 * behavior.h - interface for behavior modules
 */
#ifndef __BEHAVIORS_H
#define __BEHAVIORS_H

// 1st dimension in bearing wishes is behavior
#define BEHAVIOR_GET_PUCK        0
#define BEHAVIOR_PUCK_TO_GOAL    1
#define BEHAVIOR_GO_HOME         2
#define BEHAVIOR_AVOID_OPPONENT  3
#define BEHAVIOR_AVOID_OBSTACLES 4
#define BEHAVIOR_RANDOM          5
// * IMPORTANT * When adding a new behavior,
// increase vector dimension below:
#define NUM_BEHAVIORS            6

// 2nd dimension in bearing wishes is direction
#define X 0
#define Y 1

// Global IR data area for average calculation of puck and goal readings
extern bank3 int ir_temp_data[5][3]; // 5 sensors, 3 16bit readings each
extern bank3 int ir_avg_data[5][3];

// Upper bound for behavior wishes
#define MAX_WISH 127
```

```
// The bearing in which all the behaviors wish to go.
// [x,y] for all behaviors - should be normalized between 1 and MAX_WISH
#define _wishT signed char
extern bank2 _wishT behavior_bearing_wish[NUM_BEHAVIORS][2];

// Behavior weights
#define _weightT char
extern bank2 _weightT behavior_weight[NUM_BEHAVIORS];

/* within_shooting_range
 *  Check if it is time to "shoot" and do the goal gesture
 *
 * Pre-cond: a previous check of puck possesion is assumed
 * Return:   1 if withing shooting range, 0 otherwise
 */
char within_shooting_range(void);

/* puck_in_zone
 *  Check if the puck is in the offensive goal zone
 */
char puck_in_zone(void);

/*
 * have_puck
 *  - check if puck is in front of robot
 *
 * Return: 1 if puck is in position, 0 otherwise
 */
char have_puck(void);

/*
 * puck_lost
 *  - check if we no longer have puck
 *
 * Return: 1 if puck is no longer in position
 */
char puck_lost(void);

/*
 * behavior_get_puck
 *  - go for puck or try to locate puck
 */
void behavior_get_puck(void);

/*
 * behavior_puck_to_goal
 *  - drive with puck towards goal
 */
void behavior_puck_to_goal(void);

/*
 * behavior_go_home
 *  - go to defensive goal after scoring
 */
void behavior_go_home(void);

/* behavior_avoid_opponent
 *  - try to avoid hitting (or get hit by!) the other robot
 */
void behavior_avoid_opponent(void);

/* behavior_avoid_obstacles
 *  - try (very much!!) to avoid hitting hard non-moving evil things!
 *    also avoid goal zones (they're off limits pal!)
 */
void behavior_avoid_obstacles(void);

/* behavior_random
 *  - completely random wishes to avoid freezing in "dead spots"
 */
void behavior_random(void);

#endif /* __BEHAVIORS_H */
```

Behavior.c

```
/*
 * behavior.c - implementation for behavior modules
 */
#include <pic.h>
#include <math.h>
#include "behavior.h"
#include "ir_servo.h"
#include "bumplib.h"
#include "stdlib.h"        // rand()

#define IN_ZONE_FRONT    1
#define IN_ZONE_REAR     2

#define MOMENTUM_COUNTER 8

bank2 _wishT   behavior_bearing_wish[NUM_BEHAVIORS][2];
bank2 _weightT behavior_weight[NUM_BEHAVIORS];

// x,y for use in local calculations in each behavior
static bank2 long x;
static bank2 long y;

bank3 int ir_temp_data[5][3]; // 5 sensors, 3 16bit readings each
bank3 int ir_avg_data[5][3];

// IR readings in four directions
static bank2 long front;
static bank2 long back;
static bank2 long left;
static bank2 long right;

char within_shooting_range(void) {
     static bank2 bit in_range = 0;
     static bank2 bit in_range_last = 0;

     // Check for closeness, and that we're facing the right direction (at least almost)

     /* If one of the high front sensor is facing the goal, we are up to 45 degrees off goal center
      * and the reading is *very* high -> not a good time to shoot. (~ angle check)
      *
      * If the high sensors are not over the goal threshold, we *might* be in position for shooting
      * but we can also be "goal blind" - use front low sensor to check this. (~ proximity check)
      */
     in_range = (ir_avg_data[IR_FRONT_LEFT][OFFENSIVE_GOAL]  < IR_GOAL_HISENS_OVERFLOW &&
                            ir_avg_data[IR_FRONT_RIGHT][OFFENSIVE_GOAL] < IR_GOAL_HISENS_OVERFLOW &&
                            ir_avg_data[IR_FRONT_LOW][OFFENSIVE_GOAL]   > IR_GOAL_LOWSENS_THRESHOLD);

     if (!in_range_last) {
               in_range_last = in_range;
               return 0;
     }
     in_range_last = in_range;
     return in_range;
}

/*
 * Is the puck in the goal zone?
 */
char puck_in_zone(void) {
     return ((ir_avg_data[IR_FRONT_LOW][PUCK] > IR_PUCK_STRONG_SIGNAL * 3 / 4) &&
within_shooting_range());
}
```

```
/*
 * in_goal_zone
 *         are we in a goal zone?
 *
 * Return: 0 if not in any goal zone,
 *         IN_ZONE_FRONT if any of front reflex detectors is in goal zone
 *         IN_ZONE_REAR  if any of back reflex detectors is in goal zone
 */
char in_goal_zone(char goal) {
     if (bump_reflexes == 0)
               return 0;
     else {
               if (ir_avg_data[IR_FRONT_LOW][goal]   > IR_ZONE_LOWSENS_THRESHOLD ||
                   ir_avg_data[IR_FRONT_RIGHT][goal] > IR_ZONE_HISENS_THRESHOLD  ||
                   ir_avg_data[IR_FRONT_LEFT][goal]  > IR_ZONE_HISENS_THRESHOLD)
                    return IN_ZONE_FRONT;
               else if (ir_avg_data[IR_REAR_RIGHT][goal]> IR_ZONE_HISENS_THRESHOLD ||
                             ir_avg_data[IR_REAR_LEFT][goal] > IR_ZONE_HISENS_THRESHOLD)
                    return IN_ZONE_REAR;
               else
                    return 0;
     }
}

/*
 * Do we have the puck?
 */
char have_puck(void)
{
     static bank2 bit have_puck = 0;
     // A bit telling wether we have had the puck or not
     static bank2 bit have_puck_bit = 0;

     if (!have_puck_bit) {
               /* If either of puck bumpers are active and we have a reasonably
                * strong IR-reading, we consider the puck to be in our posession
                */
               have_puck = (ir_avg_data[IR_FRONT_LOW][PUCK] > IR_PUCK_STRONG_SIGNAL) &&
                    (bump_bumper_active(BUMPER_PUCK_LEFT) ||
                          bump_bumper_active(BUMPER_PUCK_RIGHT));
     }
     else { // is puck lost? bumpers will give contact noise, only check IR
               have_puck = ir_avg_data[IR_FRONT_LOW][PUCK] > IR_PUCK_STRONG_SIGNAL;
     }
     have_puck_bit = have_puck;

     RD6 = have_puck;
     return have_puck;
}

/*
 * Calculate a wish to go to either goal
 */
void find_goal(char goal) {
     static bank2 bit stand_and_turn = 0;

     front = ir_avg_data[IR_FRONT_LEFT][goal]  + ir_avg_data[IR_FRONT_RIGHT][goal];
     left  = ir_avg_data[IR_FRONT_LEFT][goal]  + ir_avg_data[IR_REAR_LEFT][goal];
     right = ir_avg_data[IR_FRONT_RIGHT][goal] + ir_avg_data[IR_REAR_RIGHT][goal];

     stand_and_turn = 0;

     if (in_goal_zone(goal) == IN_ZONE_FRONT) {
               // We are in the goal zone and don't want to go any further
               x = 0;
               y = 0;
     }
     else if (ir_avg_data[IR_REAR_RIGHT][goal] > 2 * IR_NOISE_THRESHOLD ||
                   ir_avg_data[IR_REAR_LEFT][goal]  > 2 * IR_NOISE_THRESHOLD)
     {         // Goal is behind us, stand and turn around to "see" it better
               stand_and_turn = 1;
     } else if (front > IR_GOAL_NOT_SEEN_THRESHOLD) {
               // Goal is virtually straight ahead
               // Scale y according to how strong signal is in terms of it's max reading
               y = (int)((float)MAX_WISH * front / IR_GOAL_HISENS_THRESHOLD);
               x = ir_avg_data[IR_FRONT_RIGHT][goal] - ir_avg_data[IR_FRONT_LEFT][goal];
               x = (int)((float)MAX_WISH * x / IR_GOAL_HISENS_OVERFLOW);
               if (x > MAX_WISH)
```

```
                x = MAX_WISH;
            else if (x < -MAX_WISH)
                x = -MAX_WISH;
            if (y > MAX_WISH)
                y = MAX_WISH;
            else if (y < 0)
                y = 0;
    }
    else {
            // Goal is in front, but the angle is big - stand and turn
            stand_and_turn = 1;
    }

    // Avoid writing this code in two places above...
    if (stand_and_turn) {
            y = 0;
            x = right - left;
            if (x == 0) {
                /* If the unlikely event that x=0 is to happen,
                 * don't stop - stand and turn in a "smart"
                 * direction (eg. avoid swinging club)
                 */
                x = servo_club_pos == CLUB_LEFT ? MAX_WISH : -MAX_WISH;
            }
            else {
                x = x > 0 ? MAX_WISH : -MAX_WISH;
            }
    }
}

/*
 * Behavior that produces a whish for going to the puck
 */
void behavior_get_puck(void)
{
    // Do something smart in order to find the puck
    // summarize the front, rear, left and right to get some
    // hint as to where the puck might be
    // if wew have the puck already, find_puck shouldn't look for it anywhere else
    if (have_puck()) {
            x = 0;
            y = 0;
    }
    else {
            front = ir_avg_data[IR_FRONT_LEFT][PUCK]  + ir_avg_data[IR_FRONT_RIGHT][PUCK];
            back  = ir_avg_data[IR_REAR_LEFT][PUCK]   + ir_avg_data[IR_REAR_RIGHT][PUCK];
            left  = ir_avg_data[IR_FRONT_LEFT][PUCK]  + ir_avg_data[IR_REAR_LEFT][PUCK];
            right = ir_avg_data[IR_FRONT_RIGHT][PUCK] + ir_avg_data[IR_REAR_RIGHT][PUCK];

            // Calculate some puck position, relative to mine.
            // +y is forward, and +x is right
            x = right - left;
            y = front - back;

            // Do we really see the puck?
            if (front < 2 * IR_NOISE_THRESHOLD &&
                back  < 2 * IR_NOISE_THRESHOLD &&
                left  < 2 * IR_NOISE_THRESHOLD &&
                right < 2 * IR_NOISE_THRESHOLD) {
                 // Puck is not seen, turn around (right) and search for it
                 x = MAX_WISH;
                 y = 0;
            }
            // This behavior never wants to go backwards... If the puck is behind
            // us, we rather want to turn in the puck's general direction
            else if (y < 0) {
                y = 0;
                x = x >= 0 ? MAX_WISH : -MAX_WISH;
            }
            else {
                /* We see the puck in front of us and
                 * wish to go there very much (=MAX_WISH)
                 * Set the larger of the two values to MAX_WISH, and scale the
                 * other one accordingly
                 */

                // High-pass filter on x to avoid oscillating wishes when approaching
                if (abs(x) < IR_NOISE_THRESHOLD)
```

```
                        x = 0;
            if (abs(x) > abs(y)) {
                        y = (int)((float)y * MAX_WISH / abs(x));
                        x = x >= 0 ? MAX_WISH : -MAX_WISH;
            }
            // y is never negative here

            else if (abs(y) > abs(x)) {
                        x = (int)((float)x *  MAX_WISH / y);
                        y = MAX_WISH;
            }
            else
            {
                        x = x >= 0 ? MAX_WISH : -MAX_WISH;
                        y = MAX_WISH;
            }
        }
    }

    behavior_bearing_wish[BEHAVIOR_GET_PUCK][X] = x;
    behavior_bearing_wish[BEHAVIOR_GET_PUCK][Y] = y;
}

/*
 * Behavior that produces a wish to go towards the opponents goal
 */
void behavior_puck_to_goal(void)
{
    find_goal(OFFENSIVE_GOAL);

    behavior_bearing_wish[BEHAVIOR_PUCK_TO_GOAL][X] = x;
    behavior_bearing_wish[BEHAVIOR_PUCK_TO_GOAL][Y] = y;
}

/*
 * Behavior that produces a wish to go towards our goal
 */
void behavior_go_home(void)
{
    find_goal(DEFENSIVE_GOAL);

    behavior_bearing_wish[BEHAVIOR_GO_HOME][X] = x;
    behavior_bearing_wish[BEHAVIOR_GO_HOME][Y] = y;
}

/*
 * Behavior that produces a wish to move away from the opponent
 */
void behavior_avoid_opponent(void)
{
    // Basically the same as finding the puck, but repelling instead of attracting
    front = ir_data[IR_FRONT_LEFT][OPPONENT] +
              ir_data[IR_FRONT_RIGHT][OPPONENT];
    back  = ir_data[IR_REAR_LEFT][OPPONENT] +
              ir_data[IR_REAR_RIGHT][OPPONENT];
    left  = ir_data[IR_FRONT_LEFT][OPPONENT] +
              ir_data[IR_REAR_LEFT][OPPONENT];
    right = ir_data[IR_FRONT_RIGHT][OPPONENT] +
              ir_data[IR_REAR_RIGHT][OPPONENT];

    // Calculate some opponent position, relative to mine.
    // +y is forward, and +x is right.
    x = right - left;
    y = front - back;

    // Do we really see the opponent?
    If      (front < 2 * IR_OPPONENT_THRESHOLD &&
             back  < 2 * IR_OPPONENT_THRESHOLD &&
             left  < 2 * IR_OPPONENT_THRESHOLD &&
             right < 2 * IR_OPPONENT_THRESHOLD) {
              // opponent is not seen, or is too far away.
              x = 0;
              y = 0;
    }

    /* This behavior never wants to go towards the opponent. If the
     * opponent is in front of us, we want to turn away from him
     * i.e. invert x */
```

```
    else if (y > 0) {
              y = 0;
              x = x >= 0 ? -MAX_WISH : MAX_WISH;
    }
    else {
              /* We see the opponent behind us and
               * wish to go away much (=MAX_WISH)
               * Set the larger of the two values to MAX_WISH, and scale the
               * other one accordingly
               */
              // High-pass filter on x to avoid oscillating wishes
              if (abs(x) < IR_NOISE_THRESHOLD)
                   x = 0;
              if (abs(x) > abs(y)) {
                   y = - (int)((float)y * MAX_WISH / abs(x));
                   x = x >= 0 ? -MAX_WISH : MAX_WISH;
              }
              // y is never positive here
              else if (abs(y) > abs(x)) {
                   x = (int)((float)x * MAX_WISH / y);
                   y = MAX_WISH;
              }
              else
              {
                   x = x >= 0 ? -MAX_WISH : MAX_WISH;
                   y = MAX_WISH;
              }
    }

    behavior_bearing_wish[BEHAVIOR_AVOID_OPPONENT][X] = x;
    behavior_bearing_wish[BEHAVIOR_AVOID_OPPONENT][Y] = y;
}

/*
 * Behavior that produces a wish to go away from obstacles we hit, or to go away from a goal zone
 */
void behavior_avoid_obstacles(void)
{
    static bank2 int counter = MOMENTUM_COUNTER;
    static bank2 char last_y = 0;
    static bank2 char last_x = 0;
    static bank2 int ir_puck_in_front = (int)(IR_PUCK_STRONG_SIGNAL * 3.0 / 4.0);
    x = 0;
    y = 0;

    // This is a simple scheme to back away from any obstacles
    // the robot has bumped into.

    // If hit on the club, *and* carrying the puck, only
    // back up if hit on the club (ignore front right/left bumpers)
    if (bump_bumper_active(BUMPER_CLUB) && have_puck()) {
              x = 0;
              y = -MAX_WISH;
              counter = 0;
    }
    else {
              // If hit from front, go backwards if we don't have puck close up front
              if ((bump_bumper_active(BUMPER_FRONT_LEFT)  ||
                   bump_bumper_active(BUMPER_FRONT_RIGHT) ||
                   bump_bumper_active(BUMPER_CLUB)) &&
                  ir_avg_data[IR_FRONT_LOW][PUCK] < ir_puck_in_front) {
                   y = y - MAX_WISH;
                   counter = 0;
              }
              // If hit from rear, go forward maximum warp speed
              if (bump_bumper_active(BUMPER_REAR_LEFT) ||
                  bump_bumper_active(BUMPER_REAR_RIGHT)) {
                   y = y + MAX_WISH;
                   counter = 0;
              }

    }

    // Check that we're not in any goal zone.
    // This is more important than bumpers, so we override them
    if (in_goal_zone(DEFENSIVE_GOAL) == IN_ZONE_FRONT ||
             in_goal_zone(OFFENSIVE_GOAL) == IN_ZONE_FRONT) {
              y = -MAX_WISH;
```

```
                x = 0;
                counter = 0;
        }
        else if (in_goal_zone(DEFENSIVE_GOAL) == IN_ZONE_REAR ||
                        in_goal_zone(OFFENSIVE_GOAL) == IN_ZONE_REAR) {
                y = MAX_WISH;
                x = 0;
                counter = 0;
        }

        // Set momentum for avoiding obstacles
        if (counter == 0) {
                last_x = x;
                last_y = y;
                counter++;
        }
        // Use momentum MOMENTUM_COUNTER times
        else if (counter < MOMENTUM_COUNTER) {
                x = last_x;
                y = last_y;
                counter++;
        }

        behavior_bearing_wish[BEHAVIOR_AVOID_OBSTACLES][X] = x;
        behavior_bearing_wish[BEHAVIOR_AVOID_OBSTACLES][Y] = y;
}

/*
 * Behavior that produces a random wish to avoid livelocks among the other behviors
 */
void behavior_random(void)
{
        static bank2 long total_ir_reading;
        static bank2 int scalefactor;
        // rand() gives an integer in the range 0..32767

        /* our belief is that driving around is more likely to break
         * a livelock in terms of dead spots in behavior summation than
         * is turning, so we give y-coordinates higher priority
         * To avoid random getting too large when the puck is close
         * (and hence find_puck doesn't want as much) we also let random
         * wish for less movement when close to the puck
         */
        total_ir_reading = ir_avg_data[IR_FRONT_LOW][PUCK] +
                ir_avg_data[IR_FRONT_RIGHT][PUCK] +
                ir_avg_data[IR_FRONT_LEFT][PUCK] +
                ir_avg_data[IR_REAR_LEFT][PUCK] +
                ir_avg_data[IR_REAR_RIGHT][PUCK];
        x = (rand() >> 12) - 7; // -7..8
        y = (rand() >> 11) - 15; // -15..16

        // scale according to ir_readings
        scalefactor = total_ir_reading / ((long)5 * IR_MAX_READING);
        behavior_bearing_wish[BEHAVIOR_RANDOM][Y] = y * scalefactor;
        behavior_bearing_wish[BEHAVIOR_RANDOM][X] = x * scalefactor;
}
```

Bumplib.h

```
/****************************************************
 *                                                  *
 * Interface to bumper and reflex detector library *
 *                                                  *
 ****************************************************/
#ifndef __BUMPLIB_H
#define __BUMPLIB_H
#include "stdfunc.h"

/* Global bumper activation flags */
extern bank2 unsigned char bump_bumpers;

/* Global reflex detector activation flags - must be cleared in software */
extern bank2 unsigned char bump_reflexes;

/* Bumper sensor bit positions */
#define BUMPER_REAR_LEFT      7
#define BUMPER_REAR_RIGHT     6
#define BUMPER_FRONT_LEFT     5
#define BUMPER_FRONT_RIGHT    4
#define BUMPER_PUCK_LEFT      3
#define BUMPER_PUCK_RIGHT     2
#define BUMPER_CLUB           1
#define BUMPER_EMERGENCY_STOP 0

/* Reflex detector flag positions */
#define REFLEX_FRONT_RIGHT 0
#define REFLEX_FRONT_LEFT  1
#define REFLEX_REAR_LEFT   2
#define REFLEX_REAR_RIGHT  3

/* "Black" detection threshold */
#define REFLEX_BLACK_THRESHOLD 252

/*
 * bump_init - initialize bumper interrupts etc.
 *
 * Usage:
 *       bump_init();  initialize bumper updating
 *       INTE=1;       external interrupt (on emergency stop) enable
 *       ei();         global interrupt enable
 */
void bump_init(void);

/*
 * bump_bumper_active - check if specified bumper is active
 */
#define bump_bumper_active(X) bittst(bump_bumpers, X)

/*
 * bump_reflex_active - check if specified reflex is active
 */
#define bump_reflex_active(X) bittst(bump_reflexes, X)

/*
 * Updates bump sensors and reflex detectors
 */
void bump_poll_sensors(void);

/*
 * bump_service_routine
 *  - dummy interrupt handler for emergency stop (not handled just updated)
 */
void bump_service_routine(void);


#endif /* #ifndef__BUMPLIB_H */
```

Bumplib.c

```
/*******************************************************
 *                                                     *
 * Implementation of bumper and reflex detector library *
 *                                                     *
 *******************************************************/
#include <pic.h>
#include "stdfunc.h"
#include "bumplib.h"

bank2 unsigned char bump_bumpers;
bank2 unsigned char bump_reflexes;

void bump_init(void)
{
     /* RB7..1  - bumper inputs
      * RB0     - emergency stop
      * AN0..3  - reflex detectors
      */

     RBIE  = 0;    // disable port change interrupt on PORTB
     PORTB = 0xFF; // clear all data on PORTB
     RBPU  = 0;    // Enable PORTB weak pull-ups
     TRISB = 0xFF; // set all RBx as inputs
     INTF  = 0;    // clear external interrupt flag

     TRISA |= 0b00001111; // set RA3..RA0 as inputs, used as ANx
     ADCON1 = 0b10000010; // ADFM=1, right justify A/D result, PCFG3:0 = 0b0010
     // RE2, RE1, RE0 - digital I/O. RA5, RA3, RA2, RA1, RA0 = AN5:0 - analog I/O.
     bitset(ADCON0, 7);   // ADCS1:0 = 0b10 (select Fosc/32 as clock)
     bitclr(ADCON0, 6);
     ADON = 1;            // power up A/D conversion module
}

/*
 * Updates bump sensors and reflex detectors
 */
void bump_poll_sensors(void)
{
     static bank3 long delay;
     static bank3 char channel; // A/D channel
     static bank3 char adres;

     // read bumper status on PORTB
     bump_bumpers = 0xFF - PORTB;

     // sample reflex detector activation status
     for (channel = 0b000; channel <= 0b011; channel++) {
               ADCON0 = (ADCON0 & 0b11000111) | (channel << 3); // set channel bits in ADCON0
               for (delay = 0; delay < 50; delay++)
                    ; // wait Tacq
               ADCON0 |= 0b100; // GO!
               while (ADCON0 & 0b100)
                    ; // wait for DONE (Tad)
               adres = ADRESL;

               if (adres > REFLEX_BLACK_THRESHOLD)
                    bitset(bump_reflexes, channel);
               else
                    bitclr(bump_reflexes, channel);

               for (delay = 0; delay < 50; delay++)
                    ; // wait Tad
     }
}
```

```
/*
 * bump_service_routine
 *  - dummy interrupt handler for emergency stop (not handled just updated)
 */
void bump_service_routine(void) {
     if (INTF) {
                // clear interrupt flag
                INTF = 0;

                // emergency stop must be handled in calling software

                // just read bumper status on PORTB
                bump_bumpers = 0xFF - PORTB;
     }
}
```

Ir_servo.h

```
/****************************************
 *                                      *
 * Interface to IR and servo library    *
 *                                      *
 ****************************************/
#ifndef __IR_SERVO_H
#define __IR_SERVO_H

/* IR sensor index */
#define IR_FRONT_LEFT      0
#define IR_FRONT_RIGHT     1
#define IR_FRONT_LOW       2
#define IR_REAR_LEFT       3
#define IR_REAR_RIGHT      4

/* IR targets */
#define PUCK               0
#define OFFENSIVE_GOAL     1
#define DEFENSIVE_GOAL     2
#define OPPONENT           3
#define UNMODULATED        4

/* ir_data - global IR data area */
extern bank1 int ir_data[5][5];          /* 5 sensors, 5 16bit readings each */

/* Servo index */
#define SERVO_CLUB      0 // club is servo 1, place 0 in vector

/* global servo position area */
extern bank2 int servo_pos[4];

/* Club positions */
#define CLUB_LEFT    -3850
#define CLUB_RIGHT    6650

/* Club set macro
 *
 * usage: servo_set_club(CLUB_LEFT);
 *        servo_set_club(CLUB_RIGHT);
 */
#define servo_set_club(X) servo_pos[SERVO_CLUB]=X

/* Club read macro
 *
 * usage, eg: if (servo_club_pos == CLUB_LEFT)...
 */
#define servo_club_pos servo_pos[SERVO_CLUB]
```

```
/* IR-data constraints */
#define IR_MAX_READING              5500
#define IR_NOISE_THRESHOLD          10
#define IR_PUCK_STRONG_SIGNAL       4800 // used when deciding wheter puck is close up front or not
#define IR_OPPONENT_THRESHOLD       100
#define IR_GOAL_LOWSENS_THRESHOLD   580  // used when deciding whether in shooting range or not
#define IR_ZONE_LOWSENS_THRESHOLD   690
//      -^- used when checking reflex detectors, aborting shot in off. zone and for finding off. goal

#define IR_ZONE_HISENS_THRESHOLD    180  // used when checking reflex detectors
#define IR_GOAL_HISENS_OVERFLOW     300
// hisens high reading, used when finding "straight" offensive goal direction
#define IR_GOAL_HISENS_THRESHOLD    360
// hisens sum high reading, used when scaling distance to offensive goal
#define IR_GOAL_NOT_SEEN_THRESHOLD 20
// hisens sum lower bound for turning and searching for offensive goal (with puck)

/*
 * ir_init - initialize IR communications
 *
 * Usage:
 *       ir_init();  initialize IR-data reception
 *       PEIE=1;     peripheral interrupts enable
 *       ei();       global interrupt enable
 */
void ir_init(void);

/*
 * ir_wait_for_data
 *  - blocking call while waiting for complete IR-data reception
 */
void ir_wait_for_data(void);

/*
 * ir_is_overflow - return 1 if IR overflow, 0 otherwise
 */
int ir_is_overflow(void);

/*
 * ir_is_bad_data - return 1 if bad IR data, 0 otherwise
 */
int ir_is_bad_data(void);

/*
 * ir_service_routine - interrupt service handler for IR interrupt
 */
void ir_service_routine(void);

#endif /* #ifndef__IR_SERVO_H */
```

Ir_servo.c

```
/*******************************************************************
 *                                                                 *
 * Implementation of IR filter communcations- and servo library   *
 *                                                                 *
 *******************************************************************/
#include <pic.h>
#include "stdfunc.h"       // bit manipulation macros etc.
#include "ir_servo.h"

/* global IR data area */
bank1 int ir_data[5][5];        // 5 sensors, 5 16bit readings each

/* global servo position area */
bank2 int servo_pos[4];

/* some useful flags */
static bank3 char spi_flags;
```

```
/* bit defs for spi_flags */
#define GOT_1ST   0        // used by interrupt routine to distinguish two consecutive 0xFF bytes
#define BAD_DATA  1     // set on failure to receive one ten bit set of data for one sensor
                                // or rx overflow, cleared on reception of sensor 0
#define OVERFLOW  2     // set after rx overflow, cleared on reception of sensor 0
#define RECEIVING 3        // 1 when receiving a byte over SPI else 0
#define FULL_SET  4        // set when full 5x5 16 bit readings have been received,
                                // cleared by main program

/*
 * ir_init - initialize IR communications
 *
 * Usage:
 *       ir_init(); initialize IR-data reception
 *       PEIE=1;    peripheral interrupts enable
 *       ei();      global interrupt enable
 */
void ir_init(void) {
              // init SSP module in SPI mode
              SSPCON=0b00000101;         // SPI mode, slave
              bitclr(TRISC,5);           // clear TRISC<5> (SDO is an output)
                                      // TRISC<3><4> are already 1 (clock and SDI are inputs)
              SSPEN=1;                   // enable SSP module

              // enable SSP interrupt
              SSPIF=0;                   // first interrupt clear flag
              SSPIE=1;

              spi_flags=0;               // no spi_flags set
}

/*
 * ir_wait_for_data
 *  - blocking call while waiting for complete IR-data reception
 */
void ir_wait_for_data(void) {
     // wait until flag gets set
     while (!bittst(spi_flags, FULL_SET)) {}
     // and then clear it
     bitclr(spi_flags, FULL_SET);
}

/*
 * ir_is_overflow - return 1 if IR overflow, 0 otherwise
 */
int ir_is_overflow(void) {
     return bittst(spi_flags, OVERFLOW);
}

/*
 * ir_is_bad_data - return 1 if bad IR data, 0 otherwise
 */
int ir_is_bad_data(void) {

     return bittst(spi_flags, BAD_DATA);
}

/*
 * ir_service_routine - interrupt service handler for IR interrupt
 */
void ir_service_routine(void) {
     static bank1 char  currsens=0; // the (last) readings received is (was) for this sensor (0..4)
     static bank1 char  currbyte=0; // the (next) received byte has (will have) (0..7)
     static bank1 char  rxdata;     // temp storage for received byte
     static bank1 char  servo_no;   // the (next) servo signal to send (1..4)
     static bank2 int   servo_data; // temp storage for trasmitted servo byte

     // SSP interrupt
     SSPIF=0;              // clear flag
```

```
    // store received data and...
    rxdata=SSPBUF;
    // ...load next byte to be sent (when *the next* byte is received)
    if (currbyte==10)     // next byte will be 0, send sensor low byte
                            // add 2 to currsens
                            // (has not been updated yet --> +1
                            // requests sensor for next reception --> +1)
              SSPBUF=(currsens < 3)?currsens+2:currsens-3;
    else if (currbyte==0) // next byte is 1, send sensor high byte
              SSPBUF=0;
    else {   // all other bytes are servo positions, high or low byte
                    // send high or low servo byte

              servo_no   = (currbyte-1) / 2;      // next servo number
              servo_data = servo_pos[servo_no];   // get data in vector
              if (currbyte%2 != 0)
                   SSPBUF = servo_data & 0x00FF;  // next byte is even, send low servo pos
              else
                   SSPBUF = (servo_data & 0xFF00) >> 8; // next byte is odd, send high servo pos
    }

    // check for errors
    if (SSPOV)
              bitset(spi_flags, OVERFLOW);

    // check if we got any 0xFF bytes
    // (two 0xFF means start of 12 byte sequence)
    if (rxdata==0xFF && bittst(spi_flags, GOT_1ST)) {  // two FFs in sequence
              // reset SPI
              bitset(spi_flags, RECEIVING);
              bitclr(spi_flags, GOT_1ST);
              currsens++;
              if (currsens>4) {
                   //new 5x5 set starts, clear error flags
                   currsens=0;
                   bitclr(spi_flags, OVERFLOW);
                   bitclr(spi_flags, BAD_DATA);
              }
              // check if last reception was finished
              if (currbyte!=10)
                   bitset(spi_flags, BAD_DATA);
              currbyte=0;
    }
    else {
              if (rxdata==0xFF)
                   bitset(spi_flags, GOT_1ST); // we have received one FF
              else
                   bitclr(spi_flags, GOT_1ST);

              if (bittst(spi_flags, RECEIVING)) {
                   // store received data in appropriate location without average calc.
                   *((char*)(&(ir_data[currsens][0]))+currbyte)=rxdata;
                   currbyte++;
              }
              if (currbyte==10) { //last byte received
                   if (bittst(spi_flags, OVERFLOW))
                             bitset(spi_flags, BAD_DATA);
                   bitclr(spi_flags, RECEIVING); // reception complete
                   // was this the last sensor?
                   if (currsens==4)
                             bitset(spi_flags, FULL_SET);
              }
    }
}
```

Pwmlib.h

```
/************************************
 *                                  *
 * pwmlib - Motor control library   *
 *                                  *
 ************************************/
#ifndef __PWMLIB_H
#define __PWMLIB_H

/* Global data area with (compensated) speeds of motors (-128...+127).
 * Negative values mean backward rotation.
 *
 * Indexes:
 *  0 - left motor compensated speed
 *  1 - right motor compensated speed
 */
extern bank1 signed char pwm_speed[2];

/* Global speed multipliers to compensate for different motors.
 * Set slower motor compensation factor to 1.
 */
extern bank1 float pwm_left_comp;
extern bank1 float pwm_right_comp;

/*
 * pwm_init - initialize PWM control
 */
void pwm_init(void);

/*
 * pwm_stop_motors - set speed 0 on both motors.
 */
void pwm_stop_motors(void);

/*
 * pwm_set_motor_speed - set speeds of motors.
 *
 * Params:
 *  Negative values mean backward rotation.
 *  signed char - left motor speed (-128...+127).
 *  signed char - right motor speed (-128...+127).
 */
void pwm_set_motor_speed(signed char left, signed char right);

#endif /* #ifndef __PWMLIB_H */
```

Pwmlib.c

```
/************************************
 *                                  *
 * pwmlib - Motor control library   *
 *                                  *
 ************************************/
#include <pic.h>
#include "pwmlib.h"

/* Global data area with speeds of motors (-128...+127).
 * Negative values mean backward rotation.
 */
bank1 signed char pwm_speed[2] = {0, 0};

/* Global speed multipliers to compensate for different motors.
 * Set slower motor compensation factor to 1.
 */
bank1 float pwm_left_comp  = 1;
bank1 float pwm_right_comp = 1;
```

```
/*
 * pwm_init - initialize PWM control
 */
void pwm_init(void)
{
     TRISC&=0b11111001;    /* CCP1,2 are outputs */
     TRISE&=0b11111001;    /* RE2,1 (direction pins) are outputs */
     PSPMODE = 0;          /* general purpose I/O on PORTE */
     ADCON1 &= 0b11110000; /* unmask PCFG3:0 */
     ADCON1 |= 0b00000010; /* PCFG3:0 = 0b0010, RE2:0 - digital I/O */

     PR2=0xFF;                           /* low frequency/full resolution */
     CCPR1L=0;                           /* 0 dutycycle */
     CCPR2L=0;                           /* 0 dutycycle */
     T2CON=0b00000111;     /* timer 2 on with prescaler 16
                                            (lowest possible PWM freq.) */
     CCP1CON=0b00001100;/* PWM mode */
     CCP2CON=0b00001100;/* PWM mode */
}

/*
 * setpwm (private) - set the PWM dutycycle
 *
 * Params:
 *  char left  - duty cycle for left motor
 *  char right - duty cycle for right motor
 */
static void setpwm(char left, char right)
{
     CCPR1L=left;      /* set dutycycle */
     CCPR2L=right;     /* set dutycycle */

     /* clear lsb's (8 and 9) iff dutycycle=0 */
     if (left==0)
               CCP1CON&=0b11001111;
     else
               CCP1CON|=0b00110000;
     if (right==0)
               CCP2CON&=0b11001111;
     else
               CCP2CON|=0b00110000;
}

/*
 * setdir (private) - set motor direction
 *
 * Params:
 *  1 = FORWARD, 0 = BACKWARD
 *  char left  - direction for left motor  [FORWARD/BACKWARD]
 *  char right - direction for right motor [FORWARD/BACKWARD]
 */
static void setdir(char left, char right)
{
     RE2 = left;
     RE1 = right;
}

/*
 * s2pwm (private) - convert signed motor values to PWM value
 * (PWM values below 128 have almost no effect
 *  and are excluded from the output set, except for 0)
 *
 * Params:
 *  signed char - signed motor value
 *
 * Return:
 *  char - corresponding PWM value
 */
static char s2pwm(signed char val) {
     if (val>0)
               return (signed int)val+128;
     if (val<0 && val!=-128)
               return 128-(signed int)val;
     if (val==-128)
               return 255;
     return 0;
}
```

```
void pwm_stop_motors(void) {
     /* update global data area */
     pwm_speed[0] = 0;
     pwm_speed[1] = 0;

     /* stop motors */
     CCPR1L=0;      /* set dutycycle */
     CCPR2L=0;      /* set dutycycle */

     /* clear lsb's (8 and 9) */
     CCP1CON&=0b11001111;
     CCP2CON&=0b11001111;
}

/*
 * pwm_set_motor_speed - set speeds of motors.
 *
 * Params:
 *  Negative values mean backward rotation.
 *  signed char - left motor speed (-128...+127).
 *  signed char - right motor speed (-128...+127).
 */
void pwm_set_motor_speed(signed char left, signed char right)
{
     signed char corr_right;
     signed char corr_left;

     corr_right = (signed char)(pwm_right_comp*right);
     corr_left  = (signed char)(pwm_left_comp*left);

     /* update global data area */
     pwm_speed[0] = corr_left;
     pwm_speed[1] = corr_right;

     /* send to motors */
     setpwm(s2pwm(corr_left), s2pwm(corr_right));
     setdir(left>0, right>0);
}
```

Stdfunc.h

```
/*
 * Include file for standard functions
 */
#ifndef __STDFUNC_H
#define __STDFUNC_H

#define bitset(var,bitno) ((var)|=1 << (bitno))
#define bitclr(var,bitno) ((var)&=~(1<<(bitno)))
#define bittst(var,bitno) (((var)&(1<<(bitno)))!=0)

#endif /* #ifndef __STDFUNC_H */
```