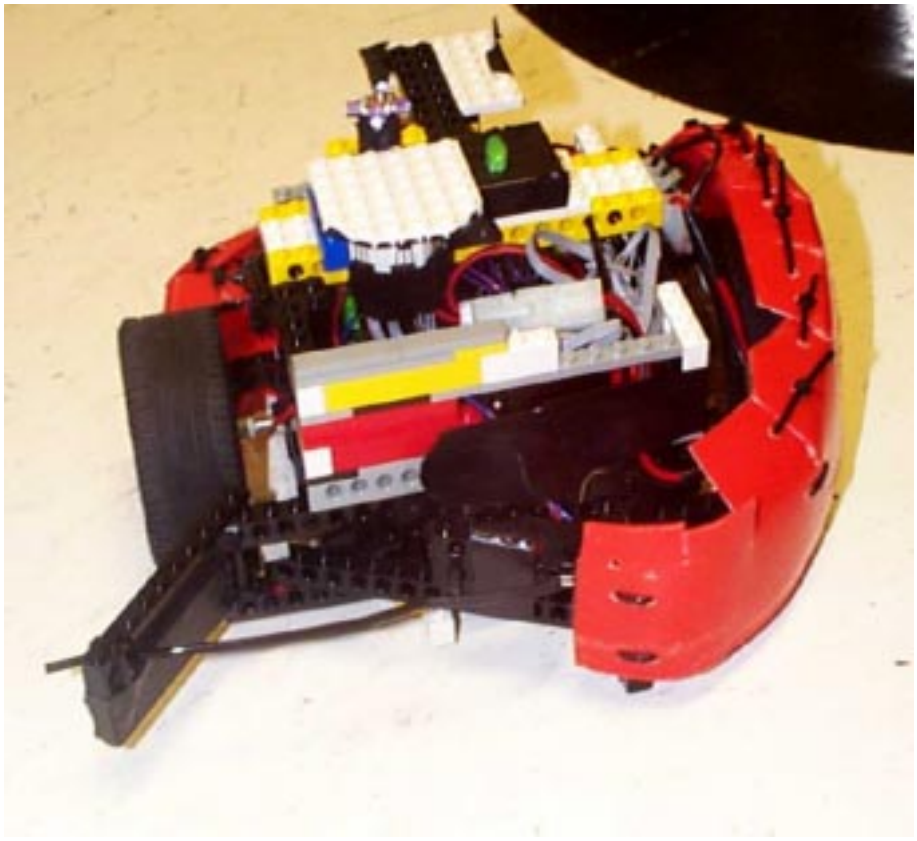
2D1426 Robotics and Autonomous Systems

Project work: Hink
or how to Build a Winning Robot



Johnny Bigert
d95-jbi@nada.kth.se

Andreas Berndt
f95-anb@nada.kth.se

Markus Markusson
f96-mma@nada.kth.se

5th September 2000

Abstract

As a part of the examination in the course 2D1426 Robotics and Autonomous Systems at the KTH, all students were assigned to construct a robot in groups of three. The robot was built to participate in a robot hockey tournament with rules set up within the course. Our implementation, called the Hink, was never defeated and won the tournament.

We were convinced that simplicity and robustness were winning concepts, which led us to use only one sensor of contact type aside of five IR sensors. We will explain why we chose to use the robot's differential steering to shoot the puck, as opposed to using a movable stick. We will also explain how the sensor positioning was used to produce a smooth locomotion and how the robot program implements a reactive behavior.

Contents

1 Background

In the course 2D1426 Robotics an Autonomous Systems at the KTH, the students were assigned to build and program a robot in groups of three. The robots should play against each other in a hockey tournament. The rink size is about 1*2 m, where the robots played one against one. This led to a finite size of maximum 25 cm in diameter of the robots. There is more rules for the design of the robots and how they should behave in a match. A complete list with all rules can be found at `http://www.nada.kth.se/kurser/kth/2D1426/rules1.gif` and `http://www.nada.kth.se/kurser/kth/2D1426/rules2.gif`

The puck and goal emit infrared light, and with photo sensors the robots can detect them. To be able to detect the opponent, each robot emit infrared light too. Each group had access to two 7.2 volts rechargeable accumulators, two 12 volts DC motors, one motherboard with two PIC microcontrollers and a few IR sensors. Other materials like electrical cables, micro switches, lego to build with, wheels etc. where to be shared between all groups. The purpose of the course is to realize how a autonomous system works and the difficulties with it.

2 Project Planning

The process of building the robot was preceded by a lot of planning and discussion. We had two primary questions, how to place and use the sensors, and how to lead the puck and score.

2.1 Positioning of the Sensors

Our first concern was with the seeing, how to use our IR-sensors. Our first notion was to make two forward-seeing eyes, each consisting of two sensors in a 90 degree angle, and from this get two angles to the object of interest. Using triangulation we were then planning to determine distances, and from that information build up an internal world representation. This would have led to great possibilities for smart navigation, but we decided that other properties were more important, such as fast reactions, small number of computations, stability and an effective forward vision.

We therefore placed one sensor facing straight forward and two tilted approximately 45 degrees, one at each side as shown in figure 1. Later we also added two in the rear. This gave good properties for our purposes, and the system proved to be very easy to implement.

2.2 Robot Shape

The design of the robot shape was driven partly by the design rules, and partly the need to, after having captured the puck, move forward and turn

with the puck under control. This implied that a v-shaped front would be effective. We realized that this could be accomplished by having an asymmetrical robot, with one side of the robot together with the stick forming the desired shape.

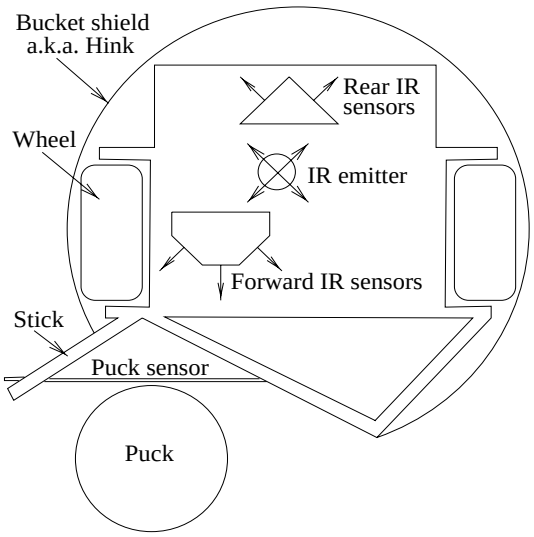


Figure 1: *Schematic of the Hink (swedish for bucket) robot*

2.3 How to Score

We also discussed what tactics could be used to score. The methods discussed will be described here. Our primary concern was to avoid the opponents goal area, since going there while scoring was prohibited by the robot hockey rules.

One was to shoot, using a rotating stick. Others were to shoot by parallel movement of the stick, or by changing the size of the stick (which proved to be against the design rules since it demanded an active joint on the stick), or using a rubber band (also against all rules). A more elegant thought was to place the puck at the goal area border, back up, turn the club forward, and use the stick to push the puck into goal. This seemed to be an insecure and slow method. Additionally, we could be prevented from scoring by the other robot, who could block us from backing up, move etc.

Further we thought of trying to score by breaking at maximum speed just outside the goal area, expecting the puck to continue the necessary distance. This was tried and discarded, and the other ones were ruled out for other reasons. We finally came to the conclusion that turning the whole robot would be the most efficient way to use our resources. This would also be a very fast method, not allowing the opponent to catch up, and would be

the method that could give the puck the greatest speed; a real shot! This also enabled us to stop the robot in a very short distance, due to the robot's circular shape.

3 Overall Strivings

Throughout the design, building and programming of the robot we let ourselves follow some simple guidelines. For instance, we wanted the robot to be robust, both in the the sense that it should not break apart, and in the sense that we would chose a behavior for the robot that could manage many situations over one that might be better in some special cases.

We were also very restrictive in changing something that worked, until it was really necessary. To keep the possible error sources to a minimum we also chose a truly minimalistic line: few sensors, few computations, few moving parts and very few states.

The programming of the robot was indeed made with few states. Except for the special cases of scoring, signaling goal, and break free if stuck, the processor could be reset at any time with no effect on the behavior.

4 Sensor Processing

During the competition, the puck, the defensive goal, the offensive goal and the opponent all carried infrared (IR) emitters, generating square pulses of different frequencies. The robot motherboard included a CO-processor that interpreted the IR input to determine how much came from each source. The CO-processor delivered the data in form of vectors with 16-bits integers.

This interpretation was not perfect, and the signals could interfere with each other. For example, the puck on close range could be mistaken for a goal far away. This generated some problems, especially when having the puck and searching for goal. This was solved through design, by obscuring the puck to the sensors when it was close enough to be controlled.

The sensor data was used mainly to compute the steering, but also for some qualitative tasks. If the puck was not seen, for example, the robot was put in a defensive mode, see section 6.2. Different threshold values for the sensor signals were also used to decide when to shoot, to keep out of defensive goal area, and to affect the locomotion in different aspects. Apart from one tactile proximity (contact) sensor, only IR sensors were used as described in section 5.2.

5 Navigation

5.1 Locomotion and Steering

The wheels were placed directly on the motor axis. One could maybe have considered gearing down, to increase torque by reducing speed, but we chose not to, for a more simple construction. The use of differential steering, which means applying different speed on the two driving wheels to control direction, also seemed natural to us.

5.2 Steering Control System

Soon after the positioning and orientation of the IR sensors, as described in the planning section, a steering system was constructed. The first aim was to enable following the puck, but basically the same system was later used to find the goal. As long as the middle (forward) sensor gave some readings, a continuous encoding from sensor data to motor action was applied. In all other cases we ordered a full turn to the direction of the strongest rear sensor reading. Full turn means maximum speed on both wheels in different directions.

The steering used only the three forward sensors. It consisted in comparing the difference between right and left sensors with the sum of these three sensors. We wanted the robot to move forward while turning, therefore we always let one wheel use maximum speed, and reduced the other to get the desired speed difference, as implied by (1).

$$speed\ reduction = max\ speed \cdot \frac{abs(left\ sensor - right\ sensor)}{sum\ of\ all\ front\ sensors} \qquad (1)$$

That is, we used the very simplest form of control, proportional error corrections.

To help the puck search further we also chose the forward sensors to be placed close to where we wanted to catch the puck.

5.3 Manipulation

Because of the shape of the robot, the manipulation became an inherent part of the navigation, that is, the only way to manipulate the puck was by moving or rotating the robot.

Having caught the puck we used the same control algorithm as in the puck-hunting case to chase the goal, except for one small difference. When there was no (or very low) readings in the middle sensor we always made a full left turn, which means max forward speed on right motor and max backward speed on left motor. That is because our robot design made it much easier to keep the puck under control during a left turn than during a right turn.

This means, having the puck, if the goal was somewhere behind us, the motors was ordered to make a full turn left. To actually make such a turn without loosing puck control was impossible. In fact, this was precisely what we did to shoot. What happened was that after a very short turn our puck sensor lost puck contact. This made the robot go into the `dont_have_puck` mode, and it started speeding after the puck, which here means going straight forward. The puck was now recaptured, and the cycle started over, every cycle making the robot turn a little bit. The stick was fixed on the right front side of the robot, directed to the front right as seen in figure 1.

This proved a rather efficient method to make a self-adjusting maximally tight turn, and the fact that the motors were set to full turn under these circumstances also gave the robot good abilities to break free from border.

5.3.1 Scoring Sequence

If the robot had the puck, it went for the opponent's goal. However, to reach an optimal shooting position it did not head straight for the goal. Rather, if far away it aimed to the left of the goal, if closer to the right of the goal, as shown in figure 2. This enabled the robot to use its shooting abilities to the maximum. Having the puck on the right, it preferred shooting diagonally to the left. This behavour was reached by using (1), but manipulating the appropriate sensor data by bit-shifting it in a feasible fashion. See C, `have_puck()`.

When the sum of the three front sensors readings of the goal IR-signal reached a certain threshold value, Hink considered itself to be within shoot range. It then went into the shoot state. This meant first a very short turn to the right, and then full turn to the left until the goal showed up behind the robot. After this it headed home, while signaling goal. At first we thought of doing this home-going by calling the defensive mode see 6.2, which speeds for our own goal, but instead we chose to simply let the motors run forward for a short while. This not to risk that the sensors mistake the opponent's goal for ours, it being so close.

6 Table Hockey Strategy

There were two possible play situations to prepare for, playing with or without an opponent. Of these the one without opponent was very much easier to try out, and therefore was much more thoroughly investigated.

6.1 Playing Alone

Alone in the rink we were finally able to score from almost any starting puck position. The most difficult locations were in the corners close to the

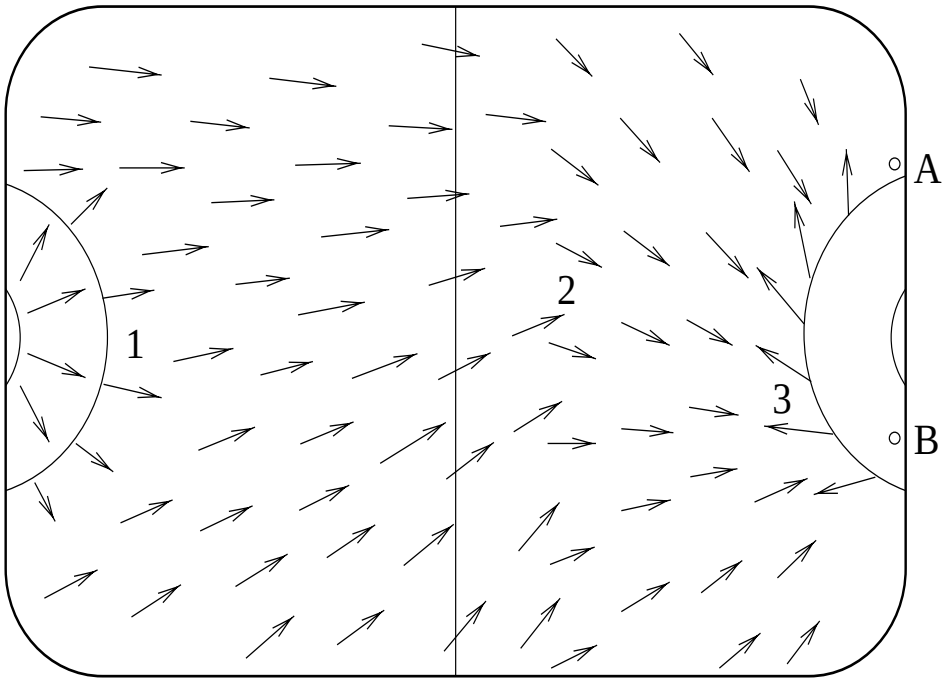


Figure 2: *Approximate potential field describing how the robot moves when in control of the puck. The robot's own goal is to the left and the opponent's goal is to the right. At its own goal (1), it avoids the goal zone. Between (1) and (2), it starts driving towards a point left of the opponent's goal (A). When close enough to the goal (2), it changes direction and aims to a point right of the goal (B). When the robot considers itself to be within shoot range (3) it shoots by turning suddenly to the left.*

goal areas. The robot could also easily get stuck against the border in some difficult situations, this however was solved by a time-out routine, which did different and suitable things depending on which side of the field it was stuck.

There were three possible situations alone at rink. If the robot did not see or control the puck, it went into the defensive mode, see section 6.2.

If it saw the puck, it went straight for the puck, avoiding nothing but coming too close to its own goal, to stay out of the goal area.

6.2 Playing with Opponent

To compensate for the difficulties introduced by an opponent on the field, we introduced a defensive mode . In the final implementation of the robot, we enter this mode only if Hink can not see the puck, without regard to the position of the other robot (see flowchart in figure 3). The reason for

not seeing the puck is, however, in most cases that it is obscured by the opponent.

In defensive mode the robot returns home, to its own goal. If it is already nearby home, it turns the motors off, unless it is also facing our own goal, in which case it first makes a turn. All this also give a good starting position for scoring, for most positions of the puck. This motivate the defensive mode to be entered regardless of if an opponent is present on the field or not.

6.3 Improvements Not Used

There was also a fully developed, but never tested method for determining whether or not the puck was controlled by the opponent. It consisted in computing a measure of correlation between the directions and distances to the opponent, the puck and our own goal. This would let us know when the puck, the opponent and the defensive goal were approximately in line at the same time as the puck and the opponent were close to each other. This we would interpret as an enemy robot with the puck on its way to our goal, with ourselves on the wrong side, see appendix F. When this situation was discovered we would enter the defensive mode.

Because of this use of the defensive mode, we also planned to change this to include the opponent's position in its computations. For instance we discussed how to overtake our opponent in an efficient way, without bumping into it.

Of the ambitions mentioned above, the only thing that survived was the defensive mode, which headed straight home without regard of the opponent, and was entered only after the trivial check if we could not see the puck. Thus almost all our tactics against an opponent was not completed, mainly due to three things:

- Late delivery of the new IR PIC processor with ability to detect the opponent signal. This implied not only that experiments with our algorithms that included the opponent signal were impossible for a long time, but also that all time after the arrival of the new PIC, the last days before the competition, had to be spent on getting the robot work as good with this new hardware, as with the old one. These adjustments were very troublesome, for instance we suddenly had problems with signals from different sources interfering with each other, something that we had not experienced with the old IR PIC processor. This meant among other things that the puck signal made it difficult to decide where the goal was when the puck was close. We also had to adjust all our threshold values, something we would have liked to have done earlier before all groups was working simultaneously, all demanding time with the puck and on the field.

- Lack of two goal experimenting possibilities. Since two goals with two power sources were not readily available and of comparable strengths until the last day before the competition, a full field test discriminating between defensive and offensive goal was never possible to perform.
- Lack of opponent signal generator not attached to any robot. This made it difficult to simulate an opponent and thus try out, for example, our correlation algorithm.

7 Integration of Robot Behaviors

The programming of the robot was made to implement a reactive behavior. Therefore the behavior was almost completely determined by the robots current desires, and these were derived from the sensor readings. In most cases the robot did not use any memory, and continuously went through the same main loop, taking different actions depending on the sensor readings, see figure 3.

8 Problems and Debugging

Hardly surprising, building a real thing combining hard- and software, we ran in to numerous problems. Many were difficult to foresee and took much time to find and correct, but most are of no further interest. Some, however, can be good to remember.

Initially we suffered much from unexplainable stops in program execution. This was also difficult to track because using serial port debugging such as `printf` seemed to slow the process down enough to prevent this problems to occur. By using only LEDs (light emitting diodes) to debug we managed to decide that the problem was an inability to divide by even powers of two! This probably was due to some strange function of the C compilator. We solved this by not dividing by any such number.

We also experienced some trouble when using many subsequent subroutine calls; some things just did not work when coded this way, but worked perfectly when executed in an other manner. Maybe this was due to a limited stack depth.

When strange problems arised it could also always be the accumulators that needed re-charging, these problems could take virtually any form.

9 Implementation

9.1 Software

The programming environment used was MPLAB and the C compiler was licensed to KTH from HI-TECH Software. To load programs to the PIC-

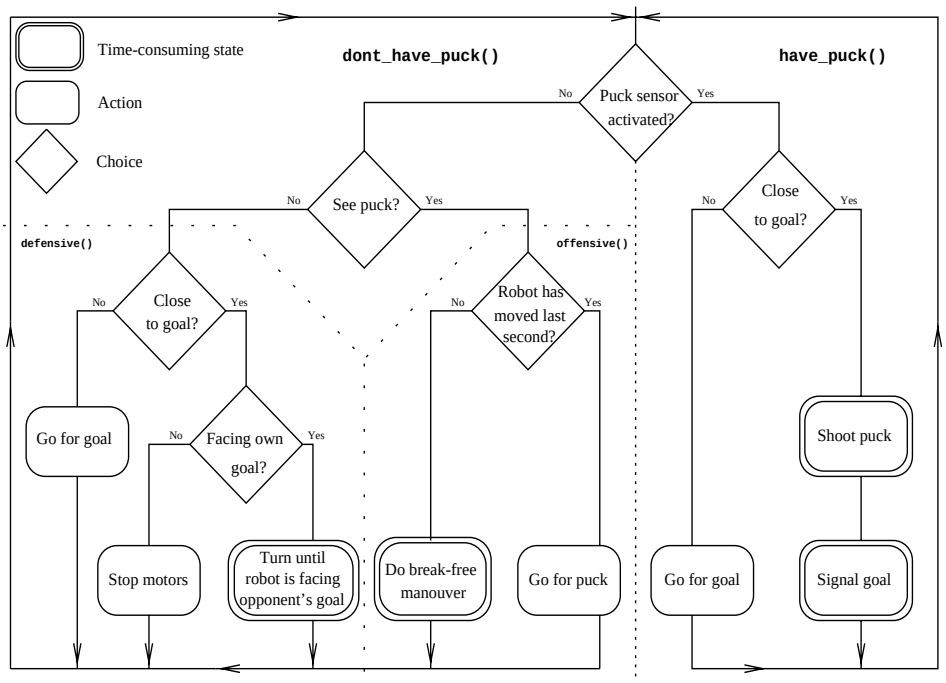


Figure 3: *Flowchart of robot behavior. As long as it does not enter any time-consuming state, each iteration, going top to bottom, lasts approximately 1/128 second. Starting at the top it enters either* `have_puck()` *or* `dont_have_puck()` *depending if the puck sensor is pushed in or not. In* `dont_have_puck()` *it enters* `offensive()` *if it sees any puck to chase, otherwise it goes to* `defensive()`.

processor a bootloader program in the PIC and an RS-232 serial cable were used.

9.2 Hardware

The material used to build the robot was two 7.2 volts rechargeable accumulators, two 12 volts DC motors Michro motors HL149.12.10, one motherboard, five IR sensors of type Texas Instruments TSL261, one micro-switch, two wheels, a lot of lego and some tape. The motherboard contained one preprogrammed PIC16F876 micro controller dedicated to filter the signals from the five IR sensors, and the PIC16F877 main micro controller, which stored the program for the robot behavior.

We also used a bucket, to protect the robot and give it a round and smooth surface. In Swedish the name for bucket is 'Hink', hence the name of the robot.

9.2.1 Connections

The ports B and D, located at J19 and J20 on the motherboard (see appendix A) was used as debugger tools. Connections were set up between the main PIC and L293NE for pwm control of the motors. RC1 was the pwm pulse and RA3 the direction for motor one. RC2 was the pwm pulse and RA4 the direction for motor two. These pins were connected to J8. Our tactile sensor was connected to RC0. The serial communication with the host PC was connected to RC6, RC7, Vdd and Vss. The five IR sensors were connected to J3-J7. The IR emitter was connected to J22.

10 Results

Hink was proven to be the quickest hockey player in the competition, while having the puck under control. After Hink had caught the puck, it always chose the closest way to the goal. At the goal zone, Hink always tried to shoot, without any hesitation. The shooting worked with different efficiency different days. When Hink had its best days, it never missed a shot, and then it could shoot so hard that the puck was bouncing back out from the goal. Days when the shooting did not work that well, the scouring frequency went down to about 50 %.

In the competition, Hink was the only robot without any defeats. Hink actually won every match it played. In the group play Hink won over Agro Tank by 6–0, Ad Hoc by 2–0, and Jas by 2–1. The statistics of shots on goal Hink won by even larger numbers. In the final against Jas, the result was 1–1 after full time. This led to sudden death. At this time Hink's accumulators were almost completely discharged, and it had not enough power to score goal when it was shooting. After a short break in the sudden death, when both robots had to re-charge, the match continued. With fully charged batteries Hink scored the winning goal on its second shot.

11 Conclusions

We all enjoyed very much participating in this course and building our robot Hink. It gave many opportunities to experience typical problems in robot building. We did not have to develop everything by ourselves, but had much help from the course teacher with low level function, and this was good. It enabled us to put more time on higher level problems, such as behavior implementation and steering algorithms.

We got very fond of reactive behaviors, and learnt to appreciate the small numbers of computation this could lead to. On the other hand we could sense that our type of solutions would become inadequate in a more complex environment, where the coordination between behaviors is more

difficult. We always had very simple choices to make.

All in all the construction of Hink and the competition turned out much as we had hoped, and we are very pleased with our robot.

A Motherboard circuit diagram

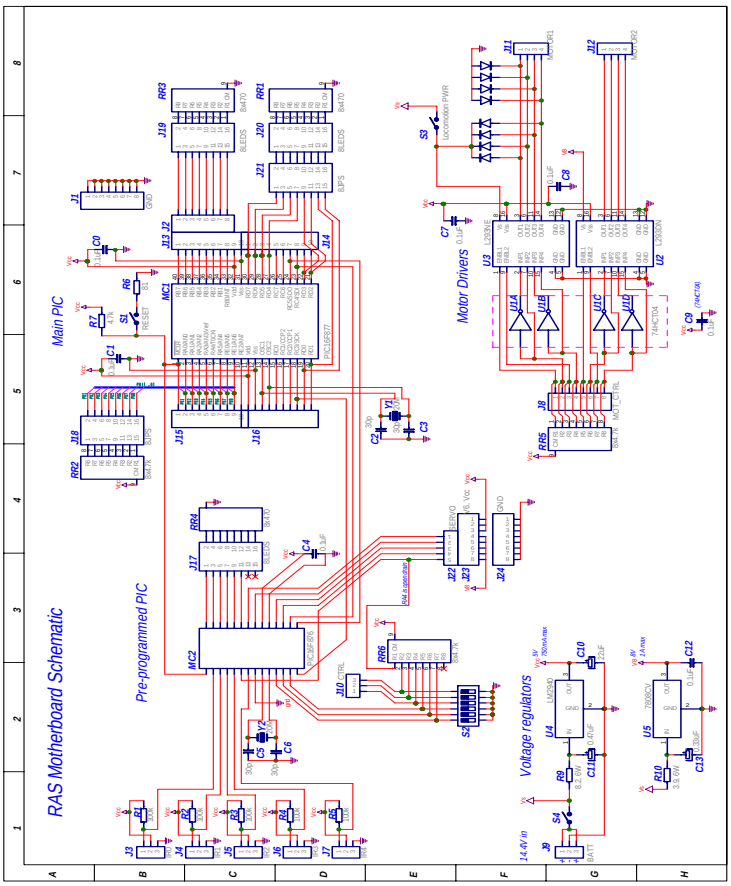


Figure 4: *Circuit diagram of motherboard without extra connections.* `http://www.nada.kth.se/kurser/kth/2D1426/motherboard.jpg`

B Header file (hink.h)

```
/*
 *   HINK controller program, header file
 *
 *   By Andreas Berndt, Johnny Bigert and Marcus Marcusson
 *   Robotics and autonomous systems VT2000
 *
 */

/* definition of constants */
#define TARGET_PUCK         0
#define TARGET_DEFENSIVE    1
#define TARGET_OFFENSIVE    2
#define TARGET_OPPONENT     3

/* unused bits in the sensor input */
#define PUCK_REMOVE_BITS    5
#define GOAL_REMOVE_BITS    3
#define OPP_REMOVE_BITS     5

/* sensors */
#define SENSOR_RIGHT        0
#define SENSOR_MID          1
#define SENSOR_LEFT         2
#define SENSOR_RIGHT_REAR   3
#define SENSOR_LEFT_REAR    4

/* speed settings */
#define SPEED_MAX           127
#define SPEED_CLOSE         100
#define SPEED_REAL_CLOSE    80

/* thresholds */
#define THRESH_CLOSE            15
#define THRESH_REAL_CLOSE       50
#define THRESH_HOME             15
#define THRESH_TURN             35
#define THRESH_SHOOT            140
#define THRESH_KEEP_LEFT        30
#define THRESH_NOISE            3
#define THRESH_STUCK_FAR        10
#define THRESH_STUCK_IS_NEAR    50
#define THRESH_STUCK_NEAR       20
#define THRESH_STUCK_NOT_NEAR   10
#define THRESH_AVOID_FRONT      130
#define THRESH_AVOID_SIDE       80
#define THRESH_AVOID_NOISE      5
#define THRESH_AVOID_BACK       100



/* declarations of functions, main functions */
void init_hink();
```

```
void dont_have_puck();
void have_puck();
void offensive();
void defensive();

/* helper functions */
void shoot_puck();
void turn(char target);
void signal_goal();
void evasive_manouver();
void stuck();
int is_offensive();
int inside_goal_area();
char get_reduced_speed(char max, char left, char right, int sum);
void avoid_goal_area(char target);
void sensor_data();

/* externals, given in the RAS course */
void motors(char, char);
void init_pwm(void);
```

C Main Functions (`main.c`)

```
/*
 *   HINK controller program, main functions
 *
 *   By Andreas Berndt, Johnny Bigert and Marcus Marcusson
 *   Robotics and autonomous systems VT2000
 *
 */
#include "ras.h"         /* prerequisites given in the RAS course */
#include "hink.h"        /* declarations */
#include "pic.h"         /* pic defines */


/*
 *  main()
 *
 *  Main loop. Checks whether the robot has the puck
 *  and calls the appropriate action. Flashes the
 *  ready beacon on top of the robot.
 *
 */
int main()
{
    char count;

    /* initialize robot */
    init_hink();

    /* infinite loop */
    for(;;)
    {
        /*
         * Wait for new sensor readings, data arrives
         * every 1/128th second.
         *
         */
        sensor_data();

        /* flash light to indicate ready mode */
        if(bittst(count, 1))     /* bit one will determine the speed */
            PORTD = 0xFF;        /* port D is wired to LED on top of robot */
        else
            PORTD = 0;
        count++;

        /* if sensor is not pushed in, we don't have the puck */
        if(PORTC & 0b00000001)
            dont_have_puck();    /* sensor not activated */
        else
            have_puck();         /* sensor pushed in */
    }

    return 0;   /* will never be reached */
```

```
}

/*
 *  dont_have_puck()
 *
 *  Checks if the puck can be seen, then gets offensive.
 *  If the puck is not seen, go home (defensive).
 *  If close to own goal area, take evasive action.
 *
 */
void dont_have_puck()
{
    /* check so that robot is not within forbidden (own) goal zone */
    if(inside_goal_area())       /* check and take appropriate action */
        return;

    /*
     *  Determine whether the robot can see the puck or not.
     *  If so, try to take the puck. If not, go to defensive goal
     *  zone since it is probable that the opponent is obscuring
     *  the puck and is heading for goal.
     *
     */
    if(is_offensive())
    {
        offensive();             /* go get the puck */
        stuck_or_not();          /* break free if stuck against the border */
    }
    else
        defensive();             /* defensive mode, go home */
}


/*
 *  offensive()
 *
 *  When called, the puck is in sight, so speed towards it.
 *  If behind robot, do full turn on the spot.
 *
 */
void
offensive()
{
    /* speed values */
    char speed_max = SPEED_MAX;  /* maximum speed of both wheel */
    char speed_left;             /* speed of left wheel */
    char speed_right;            /* speed of right wheel */
    char speed_reduce;           /* difference between fastest and other wheel */

    /* sensor readings */
    char left;                   /* left front sensor */
    char mid;                    /* middle front sensor */
    char right;                  /* right front sensor */
    char left_rear;              /* left rear sensor */
```

```
char right_rear;            /* right rear sensor*/

int sum;                    /* sum of front sensors */

/*
 *  Read sensor data, but use only a few of the bits
 *  by removing some lsb bits by bit shifting and removing
 *  some msb bits (always zero) by using an 8-bit char.
 *  Fewer bits will ensure that the sum of all sensor data
 *  will fit into an int.
 *
 */
right  = target_map[SENSOR_RIGHT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
mid   = target_map[SENSOR_MID][TARGET_PUCK] >> PUCK_REMOVE_BITS;
left = target_map[SENSOR_LEFT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
left_rear  = target_map[SENSOR_LEFT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;
right_rear = target_map[SENSOR_RIGHT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;

/* form sum of front sensors to measure some distances */
sum = left + mid + right;

if(sum != 0)    /* is the puck in front of robot? */
{
    /* lower speed if close to puck */
    if(sum > THRESH_CLOSE)
        speed_max = SPEED_CLOSE;
    if(mid > THRESH_REAL_CLOSE)
        speed_max = SPEED_REAL_CLOSE;

    /* initiate wheel speeds to maximum speed */
    speed_left = speed_max;
    speed_right = speed_max;

    /* get speed reduction for the slowest wheel*/
    speed_reduce = get_speed_reduce(speed_max, left, right, sum);

    /*
     *  Determine which wheel to slow down and
     *  lower speed. If speed reduction is equal
     *  to max speed, the robot will turn on the spot.
     *
     *  If puck is seen most on the right side, the
     *  turn should be to the right.
     *
     */
    if(left < right)            /* turn right fast (hence the factor 2) */
        speed_right -= 2 * speed_reduce;
    else                        /* turn left */
        speed_left -= 2 * speed_reduce;
}
else /* the puck is not in front of robot */
{
    /* full turn since puck is behind robot, turn left if equal */
    speed_max = 127;
```

```
        speed_right = (left_rear >= right_rear ? 1 : -1) * speed_max;
        speed_left = -speed_right;
    }

    /* set motor speed */
    motors(speed_left, speed_right);
}


/*
 *  defensive()
 *
 *  When called, the puck cannot be seen, so we assume
 *  that the opponent has the puck and we speed home.
 *  When home, we do a full turn so that we
 *  face the opponent's goal.
 *
 */
void defensive()
{
    /* speed values */
    char speed_max = SPEED_MAX;  /* maximum speed of both wheel */
    char speed_left;             /* speed of left wheel */
    char speed_right;            /* speed of right wheel */
    char speed_reduce;           /* difference between fastest and other wheel */

    /* sensor readings */
    char left;                   /* left front sensor */
    char mid;                    /* middle front sensor */
    char right;                  /* right front sensor */
    char left_rear;              /* left rear sensor */
    char right_rear;             /* right rear sensor*/

    int sum;                     /* sum of front sensors */

    /*
     *  Read sensor data, but use only a few of the bits
     *  by removing some lsb bits by bit shifting and removing
     *  some msb bits (always zero) by using an 8-bit char.
     *  Fewer bits will ensure that the sum of all sensor data
     *  will fit into an int.
     *
     */
    right  = target_map[SENSOR_RIGHT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    mid   = target_map[SENSOR_MID][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    left = target_map[SENSOR_LEFT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    left_rear  = target_map[SENSOR_LEFT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    right_rear = target_map[SENSOR_RIGHT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;

    /* sum of front sensors */
    sum = right + mid + left;

    /*
     *  Threshold that ensures that the robot remains still,
```

```
     *  when having reached home, even after the turn.
     *
     */
    if(right + left + left_rear + right_rear > THRESH_HOME)    /* have reached home */
    {
        if(sum > THRESH_TURN)                   /* defensive goal in front of robot */
            turn(TARGET_DEFENSIVE);             /* turn around until goal is behind */
        else
            motors(0, 0);                       /* stand guard */
        return;
    }

    /* the goal was not near enough, so we aim towards it */
    if(mid == 0)    /* the goal is not seen in front of the robot */
    {
        /* do full turn to break free from border */
        speed_left = -127;
        speed_right = 127;
    }
    else
    {
        /* get reduced speed */
        speed_reduce = get_speed_reduce(speed_max, left, right, sum);

        /* adjust speed depending on which side the goal is seen */
        if(left < right)            /* turn right */
            speed_right -= speed_reduce;
        else                        /* turn left */
            speed_left -= speed_reduce;
    }

    /* set the motor speed */
    motors(speed_left, speed_right);
}

/*
 *  have_puck()
 *
 *  When called, the forward touch sensor is activated,
 *  indicating that we are in possession of the puck.
 *  So we speed for goal and when passing a shoot
 *  threshold, we fire!
 *
 */
void
have_puck()
{
    /* speed values */
    char speed_max = SPEED_MAX;  /* maximum speed of both wheel */
    char speed_left;             /* speed of left wheel */
    char speed_right;            /* speed of right wheel */
    char speed_reduce;           /* difference between fastest and other wheel */

    /* sensor readings */
```

```
char left;                  /* left front sensor */
char mid;                   /* middle front sensor */
char right;                 /* right front sensor */
char left_rear;             /* left rear sensor */
char right_rear;            /* right rear sensor*/

int sum;                    /* sum of front sensors */

/*
 *  Read sensor data, but use only a few of the bits
 *  by removing some lsb bits by bit shifting and removing
 *  some msb bits (always zero) by using an 8-bit char.
 *  Fewer bits will ensure that the sum of all sensor data
 *  will fit into an int.
 *
 */
right  = target_map[SENSOR_RIGHT][TARGET_OFFENSIVE] >> GOAL_REMOVE_BITS;
mid   = target_map[SENSOR_MID][TARGET_OFFENSIVE] >> GOAL_REMOVE_BITS;
left = target_map[SENSOR_LEFT][TARGET_OFFENSIVE] >> GOAL_REMOVE_BITS;
left_rear  = target_map[SENSOR_LEFT_REAR][TARGET_OFFENSIVE] >> GOAL_REMOVE_BITS;
right_rear = target_map[SENSOR_RIGHT_REAR][TARGET_OFFENSIVE] >> GOAL_REMOVE_BITS;

/* determine sum of front sensors */
sum = right + mid + left;

/* have reached shooting distance? */
if(sum > THRESH_SHOOT)
{
    shoot_puck();           /* shoot by a full turn around */
    return;
}

/*
 *  To get to a optimal shoot position, we will navigate
 *  so that we aim to the left of the goal when far away,
 *  and to the right of the goal when near.
 *
 */
if(sum < THRESH_KEEP_LEFT)  /* we are far away... */
    left <<= 1;             /* ...so keep left */
else                        /* we are getting close... */
    right <<= 1;            /* ...so keep right */

/* sum must be calculated again since left/right will have changed */
sum = right + mid + left;

/* determine the speed reduction for the slower wheel */
speed_reduce = get_speed_reduce(speed_max, left, right, sum);

/* initiate both weels to full speed */
speed_left = speed_max;
speed_right = speed_max;

/*
```

```
     *  If the goal cannot be seen or it is behind robot
     *  do a full turn so that we can break free of the border.
     *
     */
    if(mid < THRESH_NOISE || left + right < left_rear + right_rear)
    {
        speed_left = -127;
        speed_right = 127;
    }
    /* reduce on the side where the puck signal is most prominent */
    else if(left < right)        /* turn right */
        speed_right -= speed_reduce;
    else                         /* turn left */
        speed_left -= speed_reduce;

    /* and finally, set the calculated speed */
    motors(speed_left, speed_right);
}
```

D Helper Functions (help.c)

```
/*
 *   HINK controller program, helper functions
 *
 *   By Andreas Berndt, Johnny Bigert and Marcus Marcusson
 *   Robotics and autonomous systems VT2000
 *
 */
#include "ras.h"         /* prerequisites given in the RAS course */
#include "hink.h"        /* declarations */
#include "pic.h"         /* pic defines */


/*
 *  shoot_puck()
 *
 *  Shoot the puck by doing a small right turn,
 *  followed by a full turn and then give the goal
 *  signal. While signalling goal, since the robot
 *  is turned toward the own goal, keep going straight
 *  to get an optimal position when the match is on again.
 *
 */
void shoot_puck()
{
    unsigned int count1;
    unsigned int count2;

    /* turn right before shoot */
    motors(127, -127);

    /* delay loops */
    for(count1 = 0; count1 < 0x4F; count1++)
        for(count2 = 0; count2 < 0xFF; count2++);
    motors(0, 0);

    /* this is the actual shot, turn until goal is behind robot */
    turn(TARGET_OFFENSIVE);

    /* engage the motors to get to the center while signalling */
    motors(127, 127);

    /* flash lights (Nightrider style) several times */
    signal_goal();
    signal_goal();

    /* stop speeding since we have reached the center of the court */
    motors(0, 0);
    signal_goal();
    signal_goal();
}

/*
```

```
 *  turn()
 *
 *  This is the routine used both for the
 *  shot and the turn when home. Just apply
 *  full speed on both wheels but in different
 *  directions to do a turn on the spot.
 *
 */
void turn(char target)
{
    char right;
    char right_rear;

    /* full turn to the left during wait */
    motors(-127, 127);

    /* turn, loop until robot is appropriately positioned */
    do
    {
        /* get new sensor data each turn */
        sensor_data();
        right  = target_map[SENSOR_RIGHT][target] >> GOAL_REMOVE_BITS;
        right_rear = target_map[SENSOR_RIGHT_REAR][target] >> GOAL_REMOVE_BITS;
    } while (right >= (right_rear >> 1));  /* now facing away from goal */

    /* stop motors */
    motors(0, 0);
}

/*
 *  signal_goal()
 *
 *  Flashes the LEDs several times in a
 *  Nightrider kind of way. Also flashes
 *  the LED on top of the robot.
 *
 */
void signal_goal()
{
    unsigned char i;
    char j;
    unsigned int k;

    /* initiate LEDs */
    PORTD = 0xFF;
    i = 0b00000001;
    PORTB=i;

    /* flash one way of the Nightrider */
    while(i < 0b10000000)
    {
        for(k = 0; k < 0x2FFF; k++);          /* pause */
        i <<= 1;
        PORTB=i;
```

```
    }

    /* initiate again for the other way */
    PORTD = 0;
    i = 0b10000000;
    PORTB=i;

    /* flash the other way of the Nightrider */
    while(i > 0b00000001)
    {
        for(k = 0; k < 0x2FFF; k++);          /* pause */
        i >>= 1;
        PORTB=i;
    }
}

/*
 *  is_offensive()
 *
 *  Can we see the puck? If so, return 1, otherwise, return 0.
 *
 */
int is_offensive()
{
    unsigned int sum;

    /* sum all puck sensor data */
    sum = target_map[SENSOR_RIGHT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    sum += target_map[SENSOR_MID][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    sum += target_map[SENSOR_LEFT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    sum += target_map[SENSOR_LEFT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    sum += target_map[SENSOR_RIGHT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;

    /* puck in sight? */
    return sum != 0;
}

/*
 *  stuck_or_not()
 *
 *  Uses timer to determine whether the robot has been
 *  in the same position for a long time. If the readings
 *  from the puck varies within a prescribed interval,
 *  we increment a timer counter. When the counter reaches
 *  a certain value, a break-free pattern is invoked.
 *
 */
void
stuck_or_not()
{
    /* static variables to be remembered between calls */
    static int timer = 0;         /* when timer reached thresh, break free */
    static unsigned int old_sum;  /* remember old value as comparison */
```

```
unsigned int sum;             /* sum of all sensors */
unsigned char left;           /* left front sensor value */
unsigned char right;          /* right front sensor value */

int count1;                   /* delay counter when turning */
int count2;                   /* delay counter when turning */

/* get the sensor readings, sum is the sum of all sensors */
right = target_map[SENSOR_RIGHT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
left = target_map[SENSOR_LEFT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
sum = right + left;
sum += target_map[SENSOR_MID][TARGET_PUCK] >> PUCK_REMOVE_BITS;
sum += target_map[SENSOR_LEFT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;
sum += target_map[SENSOR_RIGHT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;

/*
 *  Sensor data is almost the same as before, increment timer.
 *  First, the puck must be near enough. Second, if it's far
 *  away the differance must be within a more narrow interval.
 *  This is to avoid noise.
 *
 */
if(old_sum > THRESH_STUCK_FAR &&             /* too far, don't bother */
   ((old_sum < THRESH_STUCK_IS_NEAR &&       /* far away, use small thresh */
     abs(sum - old_sum) < THRESH_STUCK_NOT_NEAR) ||
    (old_sum > THRESH_STUCK_IS_NEAR &&       /* near, use large thresh */
     abs(sum - old_sum) < THRESH_STUCK_NEAR)))
{
    timer++;       /* increment timer */
}
else               /* sensor data has changed, reset timer */
{
    timer = 0;
    old_sum = sum;
}

/* when n cycles has passed, do break free turn */
if(timer == 0x80)
{
    timer = 0;     /* reset timer */

    /* turn for a short while... */
    if(right > left)              /* puck is on the left side */
        motors(127, -127);        /* turn left */
    else
        motors(-127, 127);        /* turn right */

    /* delay loop when turning */
    for(count1 = 0; count1 < 0xFF; count1++)
        for(count2 = 0; count2 < 0xFF; count2++);

    /* ...and then forward for a while (right-hand border only) */
    if(right > left)
    {
```

```
            motors(127, 127);
            for(count1 = 0; count1 < 0x8F; count1++)
                for(count2 = 0; count2 < 0xFF; count2++);
        }

        /* reset timer and motors */
        motors(0, 0);
        timer_count = 0;
    }
}

/*
 *  inside_goal_area()
 *
 *  Checks whether robot is near own goal area,
 *  and if so, engage motors to escape.
 *  Returns 1 if the motors where used, 0 otherwise.
 *
 */
int inside_goal_area()
{
    /* sensor readings */
    char left;                  /* left front sensor */
    char mid;                   /* middle front sensor */
    char right;                 /* right front sensor */
    char left_rear;             /* left rear sensor */
    char right_rear;            /* right rear sensor*/

    int sum;                    /* sum of front sensors */

    /* get sensor data readings */
    right = target_map[SENSOR_RIGHT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    mid = target_map[SENSOR_MID][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    left = target_map[SENSOR_LEFT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    left_rear = target_map[SENSOR_LEFT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;
    right_rear = target_map[SENSOR_RIGHT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS;

    /*
     *  Check several thresholds to ensure that robot stays
     *  out of own goal area. All sides are checked separately
     *  as is the front
     *
     */
    if(right + mid + left > THRESH_AVOID_FRONT ||
       (2*left + left_rear > THRESH_AVOID_SIDE &&
        left > THRESH_AVOID_NOISE)  ||
       (right + right_rear > THRESH_AVOID_SIDE &&
        right > THRESH_AVOID_NOISE) ||
       right_rear + left_rear > THRESH_AVOID_BACK)
    {
        /* do evasive manouver to escape goal area */
        avoid_goal_area(TARGET_DEFENSIVE);

        /* motors where used, return 1 */
```

```
        return 1;
    }
    else
    {
        /* no action was taken, return 0 */
        return 0;
    }
}

/*
 *  avoid_goal_area()
 *
 *  Uses motors to avoid own goal area when
 *  certain thresholds are passed.
 *
 */
void avoid_goal_area(char target)
{
    /* sensor readings */
    char left;                  /* left front sensor */
    char right;                 /* right front sensor */
    char left_rear;             /* left rear sensor */
    char right_rear;            /* right rear sensor*/

    /* get sensor readings for puck */
    right  = target_map[SENSOR_RIGHT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    left  = target_map[SENSOR_LEFT][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    right_rear  = target_map[SENSOR_RIGHT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;
    left_rear  = target_map[SENSOR_LEFT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS;

    /*
     *  If puck is behind robot, full turn so that it
     *  doesn't get stuck. Otherwise, the robot might not
     *  get out of the corners because of the goal area
     *  and the attraction of the puck
     *
     */
    if(right_rear + left_rear > right + left)
    {
        motors(-127,127);
        return;
    }

    /* read goal sensor data */
    right  = target_map[SENSOR_RIGHT][target] >> GOAL_REMOVE_BITS;
    left  = target_map[SENSOR_LEFT][target] >> GOAL_REMOVE_BITS;
    right_rear  = target_map[SENSOR_RIGHT_REAR][target] >> GOAL_REMOVE_BITS;
    left_rear  = target_map[SENSOR_LEFT_REAR][target] >> GOAL_REMOVE_BITS;

    /* goal is to the right */
    if(right + right_rear > left + left_rear)
    {
        if(right > right_rear)   /* goal is in front of robot */
            motors(-127,127);    /* turn left */
```

```
        else                    /* goal is behind robot */
            motors(127,127);    /* go forward */
    }
    else  /* goal is to the left */
    {
        if(left > left_rear)    /* goal is in front of robot */
            motors(127,-127);   /* turn right */
        else                    /* goal is behind */
            motors(127,127);    /* go straight forward */
    }
}
```

E Miscellaneous Functions (misc.c)

```
/*
 *   HINK controller program, miscellaneous functions
 *
 *   By Andreas Berndt, Johnny Bigert and Marcus Marcusson
 *   Robotics and autonomous systems VT2000
 *
 */
#include "ras.h"         /* prerequisites given in the RAS course */
#include "hink.h"        /* declarations */
#include "pic.h"         /* pic defines */


/*
 *  sensor_data()
 *
 *  Wait for sensor data.
 *  Fill the externally given target_map array
 *  with sensor readings from the five sensors.
 *  Each array element is an int. Data arrives
 *  with frequency about 128 Hz.
 *
 */
void sensor_data()
{
    char i,j;

    /*
     *  First wait for new reception of a complete set of data
     *  all targets for all sensors (5x5 int values)
     *
     */
    while(!bittst(rx_spi_flags,FULL_SET)) {}    /* wait until flag gets set */
    bitclr(rx_spi_flags,FULL_SET);              /* and then clear it */

    /*
     *  Now copy received data to another buffer to avoid interference
     *  from interrupt routine.
     *
     */
    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            target_map[i][j]=rx_target_map[i][j];

    /* copy the spi flags, too */
    spi_flags=rx_spi_flags;
}

/*
 *  init_hink()
 *
 *  Initialize the robot (called hink). Set the
 *  appropriate input/output pins for the PIC.
```

```
 *  Initialize motors, turn interupts on, etc.
 *
 */
void init_hink()
{
    /* Set B and D as outputs (for LEDs) */
    TRISB=0;                /* all outputs except B0 */
    TRISD=0;                /* all outputs */

    ADCON1=0b00000110;      /* all porta as digital i/o */
    TRISA=0xFF;             /* all inputs */

    /* 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) */

    /* SMP and CKE bits in SSPSTAT register are left as zeroes for */
    /* compatibility with old PICs */
    SSPEN=1;                /* enable SSP module */

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

    rx_spi_flags=0;         /* no spiflags set */

    /* enable interrrupts */
    PEIE=1;                 /* peripheral interrupts enable */
    ei();                   /* global interrupt enable */

    init_pwm();             /* initiate motors */
}

/*
 *  get_speed_reduce()
 *
 *  Get the reduction speed of the wheel that
 *  have the lowest speed. Return value is a
 *  char between 0 and speed_max.
 *
 */
char get_speed_reduce(char speed_max, char left, char right, int sum)
{
    int tmp;                /* keep temporary values */

    /*
     *  Compute the speed of each wheel by reducing the
     *  speed of the wheel closest to the puck.
     *  The formula used is
     *
     *      reduce = speed_max * diff/sum,
     *
     *   where diff is
```

```
     *
     *      diff = abs(left - right).
     *
     */
    tmp = speed_max * (left - right);   /* difference */

    /*
     *  bug fix to avoid division by power of 2,
     *  since this makes the processor hang!
     *
     */
    if(sum == 0x2 || sum == 0x4 || sum == 0x8 || sum == 0x10 ||
       sum == 0x20 || sum == 0x40 || sum == 0x80 || sum == 0x100)
        sum++;
    tmp /= sum;
    tmp = abs(tmp);         /* tmp is now unsigned speed reduce */

    /*
     *  the speed must not be reduced more than max_speed,
     *  or the char type will underflow
     *
     */
    return min(speed_max, tmp);
}


/*
 *  service_routine()
 *
 *  Handles the interrupts due to communication between
 *  the processor and the co-processor.
 *
 */
void interrupt 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)
                                      this number (0..9) in the sequence */
    char rxdata;                   /* temp storage for received byte */

    /* the SSP interrupt receives and sends SPI data and moves the servos */
    di();
    if(SSPIF){
        SSPIF=0;      /* clear flag */

        /* store received data and... */
        rxdata=SSPBUF;

        /*
         *  ...load next byte to be sent (when *the next* byte is received)
         *  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)
 *
 */
if(currbyte==10)
    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
     *
     */
    SSPBUF=*((char*)servopos+currbyte-1);
}

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

/*
 *  check if we got any 0xFF bytes
 *  (two 0xFF means start of 12 byte sequence)
 *
 */
if(rxdata==0xFF && bittst(rx_spi_flags,GOT_1ST))
{   /* two FFs in sequence */
    /* reset SPI */
    bitset(rx_spi_flags,RECEIVING);
    bitclr(rx_spi_flags,GOT_1ST);
    currsens++;

    if(currsens>4) {
        /* new 5x5 set starts, clear error flags */
        currsens=0;
        bitclr(rx_spi_flags,OVERFLOW);
        bitclr(rx_spi_flags,BAD_DATA);
    }
    /* check if last reception was finished */
    if(currbyte!=10) bitset(rx_spi_flags,BAD_DATA);
    currbyte=0;

}
else
{
    /* we have received one FF */
    if(rxdata==0xFF)
        bitset(rx_spi_flags,GOT_1ST);
    else
        bitclr(rx_spi_flags,GOT_1ST);

    if(bittst(rx_spi_flags,RECEIVING))
    {
        /* store received data in appropriate location */
```

```
                *((char*)(&(rx_target_map[currsens][0]))+currbyte)=rxdata;
                currbyte++;
            }
            if(currbyte==10)
            {
                /* last byte received */
                if(bittst(rx_spi_flags,OVERFLOW))
                    bitset(rx_spi_flags,BAD_DATA);
                bitclr(rx_spi_flags,RECEIVING);    /* reception complete */

                /* was this the last sensor? */
                if(currsens==4) bitset(rx_spi_flags,FULL_SET);
            }
        }

    }
}
```

F C code not used (improvements_not_used.c)

```
/*************************************************************
 * C code of not used improvements         *
 *************************************************************/

// This code was to be part of the function is_offensive, which returns 1 if the robot is to chase the pu
tmp = sum;
sum = target_map[SENSOR_RIGHT][TARGET_OPPONENT] >> OPP_REMOVE_BITS;
sum += target_map[SENSOR_MID][TARGET_OPPONENT] >> OPP_REMOVE_BITS;
sum += target_map[SENSOR_LEFT][TARGET_OPPONENT] >> OPP_REMOVE_BITS;
sum += target_map[SENSOR_LEFT_REAR][TARGET_OPPONENT] >> OPP_REMOVE_BITS;
sum += target_map[SENSOR_RIGHT_REAR][TARGET_OPPONENT] >> OPP_REMOVE_BITS;

/* if puck > opponent or puck is near then go after puck (offensive) */
if(tmp > sum || tmp > 70)
return 1;

/* check if opponent is heading towards own goal */
right = 1; /* avoid overflow */
right *= mymin(0b00011111, target_map[SENSOR_RIGHT][TARGET_OPPONENT] >> OPP_REMOVE_BITS);
right *= mymin(0b00011111, target_map[SENSOR_RIGHT][TARGET_PUCK] >> PUCK_REMOVE_BITS);
right *= mymin(0b00011111, target_map[SENSOR_RIGHT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS);

left = 1; /* avoid overflow */
left *= mymin(0b00011111, target_map[SENSOR_LEFT][TARGET_OPPONENT] >> OPP_REMOVE_BITS);
left *= mymin(0b00011111, target_map[SENSOR_LEFT][TARGET_PUCK] >> PUCK_REMOVE_BITS);
left *= mymin(0b00011111, target_map[SENSOR_LEFT][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS);

right_rear = 1; /* avoid overflow */
right_rear *= mymin(0b00011111, target_map[SENSOR_RIGHT_REAR][TARGET_OPPONENT] >> OPP_REMOVE_BITS);
right_rear *= mymin(0b00011111, target_map[SENSOR_RIGHT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS);
right_rear *= mymin(0b00011111, target_map[SENSOR_RIGHT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS);

left_rear = 1; /* avoid overflow */
left_rear *= mymin(0b00011111, target_map[SENSOR_LEFT_REAR][TARGET_OPPONENT] >> OPP_REMOVE_BITS);
left_rear *= mymin(0b00011111, target_map[SENSOR_LEFT_REAR][TARGET_PUCK] >> PUCK_REMOVE_BITS);
left_rear *= mymin(0b00011111, target_map[SENSOR_LEFT_REAR][TARGET_DEFENSIVE] >> GOAL_REMOVE_BITS);

/* own goal, opponent and puck is on the same side of robot */
if(right + left + right_rear + left_rear > 0xFF)
return 0;

return 1;
}
```