AD&R
project report

Michael Park, Stefan Esterer, Elias Pschernig

Abstract

In this report, we detail the hardware and software of our Lego^{TM} Mindstorms^{TM} robot, as developed for the embedded systems course.

Contents

1 Motivation

The original motivation was to improve the Butler James project [1]. The robot followed a black line, picked up an obstacle and dropped it close to a light source.

The intention of AD&R was now to improve the path finding program to distinguish between left and right turn and between branches without stopping, or just trying which possible paths are available.

During the development another important issue was to improve the software robustness against external interference like irregularities of the path which means the robot should perform course correction automatically.

2 Hardware Implementation

2.1 Sensor Array Concepts

Our conclusion was that at least three sensors are required to fulfil the requirements. Two static light sensors can distinguish between left and right turn but not branches, and the robot has to recognize when it drifts away, and when to turn where.

The first concept was to increase the resolution by using three static light sensors. The disadvantage of this concept, except the obviously static sensors which can only scan their given area, is the gap between the sensors. If the program recognizes a pattern in the information given by the sensors, it is merely speculation when the pattern matches a left or right turn or a branch. Unfortunately only two light sensors were available, which forced us to drop this concept anyway.

Because of this circumstance we developed our second idea, to combine a static light sensor with a movable light sensor mounted on a torque sensor. This concept allows wide area scanning and more flexibility when finding directions.

The disadvantage or flaw was the spinning speed of the movable sensor. It has to spin fast enough without losing sight of something and without harming the construction itself.

The first picture in figure 1 shows the two concepts. The top left three circles are the original first concept and the top right three circles are an improved version of the first concept to avoid misinterpretation by the software

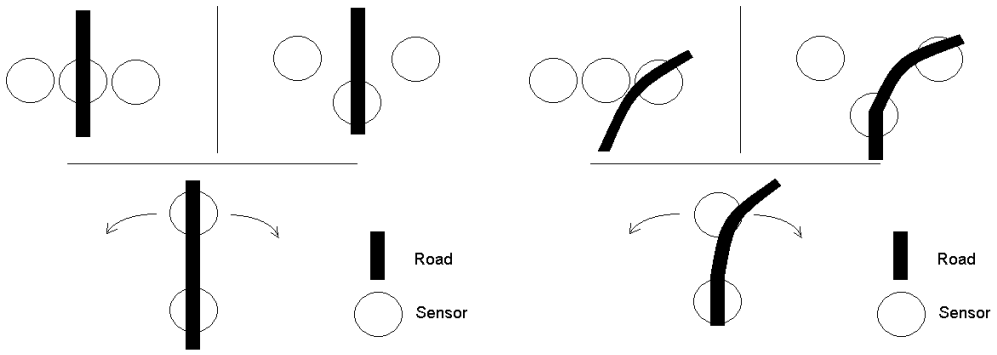

Figure 1: Sensor concepts

in some situations like when all three sensors are parallel to the path and for better scanning, shown in the second picture.

Another issue is the technical limitation. The torque sensor distinguishes only between 16 directions which denotes an arc of 22,5 degrees per direction, which turned out to be a very large arc. The problem occurred at first when the movable sensor had to ignore the straight main path, otherwise the sensor would assign the main path either to the left or right side and as a result find a non existing turn direction.

But the sensor must not ignore the path direction when the robot moves into a turn and still has to turn itself. So we improved the head length and rotation speed, but still there are some situation when the head ignores the correct turn at the wrong time when the robot is moving off road.

Other problems concerning the light sensors are the environmental light and the width of the path. If the environmental light is too bright or too dark the sensor might not be able to find the path or if the width of the path is much smaller than the scanning area of a light sensor.

As a result of this hardware complexity the underground for the road must be white an the road itself black and at least about 1cm width.

2.2 Hardware Implementation Issues

The robot is a crawler-mounted vehicle. During turns one side of the drive is moving forward while the other is moving backward, which enhances the turning speed and reduces the turning arc.

The static main light sensor is mounted in front of the robot, on top of

the torque sensor and the engine for moving the head with the second light sensor left and right.

The robot is capable of path finding on a black line and distinguish between left and right turns and branches without stopping or just trying, as long as the information about the possible turning directions is stored before the main sensor leaves the black line. In this case the robot uses the last information about turning to find its way back on the road.

3 Software Implementation

3.1 Basics

For developing the code of our Lego robot, we wanted to get full control over the RCX, but still not write our own new OS for it. An ideal solution seemed to be BrickOS. A small operating system for the RCX, which allows to read all the sensors and control the motors, and also things like timing and even multi-threading. The programming language used with it is C, and there also exists a C++ interface which we used.

3.2 General Idea

Our first version of the code was implemented directly in C++, using two threads, one for controlling the drive wheels and the main light sensor, and one for the turret and its sensors. Now, it would not be true to tell that we switched away from this to make the code more robust and more real-time, or even allow complete time-safety checking instead, we simply wanted to get some better understanding of the schedule-carrying-code paradigm [3]. Therefore, we remodeled our code to use a virtual machine for execution of E/S-code [2], and we should credit the authors of last year's "Sortbot" [4] project for parts of the idea, which was taken from their source code.

3.3 Virtual machine

The C++ main() function is reduced to a virtual machine code interpreter, which can execute simple E/S-code instructions. And there are some low level C++ functions, to check sensor values, and to program actuators. The control of the robot is moved into the E-code and S-code. In a real project, such

code would be generated from a higher level representation like a Simulink model, or a Giotto program. In our case, there only is a simple compiler (or mere assembler) to translate a textual representation of the VM code into binary representation.

3.4 E/S-code

The available E/S-code instructions are:

- **release**: Releases a new task, that is, it is now ready to run.
- **dispatch**: Transfers control to a ready task. A task is implemented in C++.
- **future**: Specifies time at which to continue the VM code, at a specific location.
- **jump**: Directly change location inside VM code.
- **if**: Conditional change of location. A condition is implemented in C++.
- **idle**: Waits for the specified time.
- **return**: Return from a block of VM code.

The E-Code is used to control the execution of the robot. It decides when to run which parts of the code. More precisely, it only decides when the parts are ready to run. The S-Code then contains the scheduling code to actually run them.

In our simple version, there are no drivers or ports. The C++ conditions are run directly when they are encountered, only the C++ tasks are affected by scheduling. And all timing is not done with actual time limits, but with a logical clock. The speed is controlled by sleeping for the given times, but does not take into account the actual execution time. Nevertheless, we could already observe advantages of the changed code. For one, the overall code structure is clearer. The low level C++ part only has to do simple sensor reading checks, and give commands to the motors. All the higher level control is separated into the VM code. And at the same time, this means there is no more need for threads. With only short conditions and tasks, the VM scheduler makes the code run in a defined and timely manner, with no more need to worry about when one C++ function is preempted and jumps to another one, and having to put locks around critical shared variables.

4 Description of the tasks

The system contains a given set of tasks. Namely:

- check_turn
- do_move
- do_turn
- do_turret

All of this tasks have specific duties.

4.1 do_move

It's duty is to make the robot move forward with a given speed.

4.2 check_turn

This task checks periodically if the main light sensor detects the color white. This indicates that the robot isn't on the path. If this is the case, the robot has to turn.

4.3 do_turret

The turret is constantly moving from left to right and vice versa. The robot checks at given time instances if the light sensor on the turret sees the black color of the path. If this is the case, it saves whether it found it on the left or right side.

4.4 do_turn

This task is only executed if check_turn indicates that the robot is not on it's path. In this case the task has to decide whether to turn left or right. This is done by investigating the data do_turret has given the system. Now the robot is rotating on it's position until the main light sensor finds the path again.

4.5 Tasks interaction

All of this tasks are simple C++ procedures and are called by the virtual machine. The only data they "share" is the information about the direction in which the robot must rotate.

One problem occurs with the black path. The turret moves from left to right and when it sees the black path, it saves the occurrence either in the left or right variable. There were thoughts that we could define a small area where the turret would not save his discovery into the variable, but due to the limitations of the hardware we decided to ignore this effect. Thats not optimal but doesn't do a lot of harm. The reason for this is: The only occasion where the left/right variable is read, is when the main light sensor indicates that the robot is not on the path. In nearly all situations the turret has found if the curve goes left or right.

The turret is moving constantly, no matter what the robot does. This also means the turret task runs if the robot is turning, taking away CPU time and wasting battery power. So we decided to deactivate the task if the robot is rotating. If the main light sensor is sensing the black path, the turret task can be invoked. If it is sensing white color (i.e. the robot is about to rotate) it can't be activated. This solves the problem of a dead-locked situation which causes the robot to turn left a bit, than turning right again, and so on.

5 Further work

More work could be done to allow the robot to recognize crossings, and then at the end of a path, turn around and take another path when there was a crossing. This would enable it to make its way past a dungeon. It would require to store all recognized junctions, and use a simple maze solver on the generated data. But since our main goal was to investigate the real time behavior of the robot, we concentrated on the simpler task for now.

References

[1] Werner Hager, Manuel Maier and Harald Röck: *Butler James project* `http://cs.uni-salzburg.at/~ck/teaching/ESE-Winter-2004/butlerjames/`

[2] T.A. Henzinger, C.M. Kirsch: *The Embedded Machine: Predictable, Portable Real-Time Code.* PLDI 2002

[3] T.A. Henzinger, C.M. Kirsch, S. Matic: *Schedule-Carrying Code* EMSOFT 2003

[4] Robert Löffelberger, Rainer Trummer: *Sortbot project* `http://cs.uni-salzburg.at/~ck/teaching/ESE-Winter-2004/sortbot/`