Project report

REAL TIME SYSTEMS - PROJECT

Merta Michal, Rábek Martin, Valúšek Matej, Vlasák Michal

May 11, 2014

Task

Purpose of our robot is to hold a ball - and if somebody takes it away, find it and then move back to starting position.

Workflow

At the start, robot is holding the ball in its claws. It measures the intensity of brightness, using its light sensor and if that brightness increases significantly, we assume, that the ball is missing and robot starts to search for it. He moves forward by a specified amount and then starts to look around - 90 degrees to left and 90 degrees to right. If he doesn't find the ball, he moves forward again and whole procedure is repeated. If the robot doesn't find the ball till the third iteration, he gives up and returns back to its starting position.

If the robots see something, what is closer than given threshold, during his phase of looking around, he stops in that direction, opens his claws and moves this way by a portion of distance to the object he saw. Then he starts to look around again, but in smaller angle - just to be sure, that he is still moving towards the object. When the object is near, the robot close the claws and starts to measure the light brightness. Once it get sufficiently near to the brightness at the beginning (when he held the ball), the robot assumes, that the ball was found, and he returns back to its starting position.

Movement

Part of the project was to design API (Application Programming Interface) for movement subroutines, e.g. move on for desired centimeters and rotate by given angle. Resulting subroutines are named according to their function - `go_forward(int cm)`, `go_back(int cm)`, `rotate_left(int angle)`, `rotate_right(int angle)`.

With this approach, there is high efficiency of reduced lines of code because only thing you need is to call one of these subroutines with an attribute. Distances were debugged and tested and full control over them is with predefined macro functions `CM2DEG(x)` and `AN2DEG(x)` - so you can simply adjust movement API for specific type of surface. In fact, because movement is independent and robot backtracks each move, there is no need for distances or angles to be highly precisely defined. In earlier version, we also used `AN2DEG` macro, but synchronized rotating of robot wasn't working as it should, so we had to rewrite it and use another method instead. So while moving forward and backward is programmed utilizing the synchronized rotation of motors, rotating is programmed using timed rotations, when the motor is started and after a certain amount of time it's turned off. This approach allows us to stop the rotation of robot from another task. That's something not possible with synchronized motor movement, because that method is "thread-safe" so it is unpreemtable and no other thread can acces motors at this time. Only possibility would be to use a method for setting the remaining angle of motor rotations, but this method is available only in newer firmware (which we didn't have).

Movement API suffers from several inaccuracies which could not be resolved. First is caused by third wheel which brings some latency to movement operations because some time is needed to rotate it into movement direction. Secondly, it may appear that robot does not move straight forward but it slightly changes angle of move - this is caused by motors and weight of robot. These two inaccuracies highly influences the destination position backtracking operation.

Backtracking

Each move performed by robot is stored into stack. This stack contains complete history of all performed actions. Actions are stored during performation of subroutines `turn_left`, `turn_right` and `go_forward`. Subroutine `push_action` stores action on stack.

Each movement in stack is represented as two integers - first one represents operation code and second one is argument (angle or distance). When operation is about to be inserted on stack, there is always performed merge with operation on the top of stack - if these operations are of the same kind. For example actions `go_forward(10)` and `go_forward(20)` are in result stored as `go_forward(30)`. This reduces number of steps performed during backtracking.

When the ball is found (or not found during third iteration of search task), robot starts to backtrack. Robot reads all actions from the stack and performs them in the opposite manner in order to reach the starting position. Main problem with this aproach is parallelism during ball searching - task search can stop the motors, but by that time, the action is already stored on the stack. Attempt to solve this problem is based on subtraction of the amount of time taken by action up to the point of interruption. However, this process doesn't work correctly, because of problems with precision of motors, estimations in navigation in space, rounding errors and inaccuracy of time measuring.

Concurrency

Concurrency is used, when robot rotates and search the space around him, and simultaneously checks, whether he "see" something - and if it is the case, robot stops. Programming this in one task would significantly slower the operation of rotate because it is time demanding to compute if robot sees something. Thereby, there exist two parallel tasks - `search()` and `rotate()`.

Since the tasks are parallel and are simultaneously accessing one variable (which defines whether the ball was found) it can happen that even though ball is found in search task, the rotating tasks continues to turn around. This resulted in mistakes such as either the robot missed the ball completely because the search angle was lowered or he hit the ball and misplaced it. Therefore after finding the ball we added a slight rotation to the left side (robot searches from left to right) so that the miscalculations can be minimized.

Changes from abstract

Robot is not placed in the garage. Because then, he would find the garage, instead of the ball. So the garage can be drawn on the floor/table - you can imagine one, for convenience. Robot does not walk in rectangles around the garage. Instead of it, he walks toward and search his surroundings in a given radius. It is better for demonstration purposes and we believe, that extension to walk-in-rectangle would be easy, but would serve no purpose at this time.

Complications

Ultrasonic sensor Sometimes it happened that robot did overlook the ball during movement. This resulted in catching the ball into one side of opened claws and robot started rotating desperately. There were also situations, when even though the ball was not placed in robot's surrounding, ultrasonic sensor saw something and robot was trying to get it.

Wheels Two motor wheels together with third light wheel bring some inaccuracies to the movement subroutines as described earlier.

Display Most of the time it was impossible to see anything on the display. Its best times were, when the display was both poor and blinking. Thus, robot display must have been controlled via Bricx CC IDE.

IDE and drivers We have been using Bricx CC and official drivers from included Lego Mindstorm CDs. Initially, it took some time to make drivers to work on Windows 7/8 64bit operating system. Some members of our team had problems of seeing the connected robot with Bricx, via USB. This was caused by improper and incorrect official Lego Mindstorm drivers. Fortunately, this was fixed with drivers accessible on homepage of Bricx CC project.

Obsolete firmware It does not support some methods for movement and tasks.

Teamwork

Working in team has been smooth, with no problems. Team was able to meet together whenever needed, decide what to do, assign tasks to individual members and solve many emerging issues. Every individual contributed to project in these areas:

Merta Michal Programmer, Designer, Safety inspector

Rábek Martin Programmer, Movement specialist

Valúšek Matej Designer, Tester, Debugger

Vlasák Michal Programmer, Designer