Physics Competitions Vol. 15 No 1 & 2 2013

Programming Low End Robots

[1]B. Tiefengraber, [1]C. Jung, and [1–3]M. Stifter
[1]HTBLuVA Wr.Neustadt, Department of Computer Science,
Wiener Neustadt, Dr. Eckener-Gasse 2/Austria, tiefengraber.bruno@gmail.com
[2]Danube University Krems, Center for Integrated Sensor Systems,
Wiener Neustadt, Viktor Kaplan Straße 2/Austria, michael.stifter@donau-uni.ac.at
[3]Vienna University of Technology, Institute of Sensor and Actuator Systems,
Vienna, Gusshausstraße 27–29/Austria

Abstract

This paper discusses the problem of programming low end robots like an Arduino platform from the perspective of two students at the age of 18 [1]. B. Tiefengraber and C. Jung are currently students of the Federal Secondary College of Information Technology located in Wiener Neustadt and members of an amazing team. This publication is aimed at supporting those secondary school students who would like to enter a robot tournament. Hopefully, they will benefit from the experience of the authors who have successfully participated in the Global Conference on Educational Robotics 2013 in Norman/Oklahoma [2]. Optimization of low end robots is important due to their limited resources in respect of the programming capabilities. Other constraints of low end robots are usually the limited sensor sensitivities, resulting in a sophisticated data processing.

Introduction

Platforms like Arduino are rapidly claiming the commercial aspect of Robotics. But most of the commonly used platforms sacrifice high performance in order to maintain accessibility. These low performance robots are available to a broad audience, but programming complex algorithms on these platforms is a greater challenge than that for high end robots or PC's. Nevertheless, the low performance of these robots does not limit its application, as long as the implemented code is efficiently written by the programmer [3].

The proposed projects on the HackADay – website provide an insight into how comprehensive the applications of the Arduino platform can be [4]. Young engineers deal e.g. with trivial things like a mailbox robot notifying you if there is a new mail to more advanced implementations like an Arduino-controlled QuadroCopter [5, 6].

A server-client technology is often applied to centralize the logic platform behind the robot. Robots simply act as a client, transmitting sensor values and executing commands from the server. The bottle neck of this approach is the limited capability regarding the response time in this command structure. Programs that run directly on the autonomous robot react faster to any problem that occurs during the operation of the robot.

This publication will provide a guideline on programming low end robots and on the way how to optimize the program code for these demanding platforms.

Methods in Programming on Low Cost Robots

The most important method that needs to be established when working on low end Robots is using the Preprocessor [7].

The Preprocessor helps writing a source code by defining constants that get replaced by literals before being compiled. It makes the code more readable, ensuring that you cannot edit the values during runtime and taking up less memory during runtime when using standard variables.

Knowledge of your Firmware

When programming on desktops you really appreciate high level application programming interfaces (API's). They won't require excessive testing and generally help you programming. There are often a lot of API-layers.

Robot API's do exist, but most of the time they are hardly optimized and therefore not suited for low performance robots. There is a way out – most of the low end robots have an Open Source firmware, enabling an open access of the source. Downloading, reading and modifying the firmware can improve the performance a lot. It also helps you understand how the firmware operates.

Robots start to support more and more programming languages. Until recent years you were limited to C and maybe some C dialects. Nowadays you get a wide variety of languages ranging from C++ to Java or even Python. With the support of wide spread desktop languages like Java a huge step to improve the applicability is done. In the case of low performance robots you must carefully select the considered programming language.

Java, for example, is really widely spread, which makes the chance very high that the young programmer is already familiar with it. But you have to consider the additional performance the Java virtual machine (JVM) needs [8]. Additionally, Java doesn't run without the JVM. The JVM is another abstraction layer which is not controllable by the programmer. Hence, robot programming requires a thorough knowledge of the

programming techniques and the programming language chosen for the software development.

Code Efficiency

Writing an efficient code is strongly recommended on low end robots. Correctly implemented software even at low end hardware can run the most complex algorithms. Control algorithms require quite a lot of experience regarding optimization, but there are also a few smart tricks you can use for better performance in general.

```
while(analog_read(0)>10){   //check distance of front sensor
      int distance = analog_read(0); //save  distance into a variable
      … //do something with it}
```

This is an example of a widely applied code snippet. The program reads the distance of the front sensor from sensor-port 0. If the distance is larger than 10, it loops over the operation. In the loop the distance is read-out continuously and saved into a variable. Then the distance determines the next steps of the robots actions.

At first glance this code looks quite simple and you wouldn't even think about optimizing it, but there are two major flaws in this bit of code.

First Flaw: Lots of Garbage

Every iteration cycle of the loop the integer distance will be allocated. This results in a lot of allocated memory space without any pointers [9]. Some languages like Java use a garbage collector that interrupts your program temporarily and clears allocations to variables. Garbage collectors interrupt the program at scheduled times and are often the reason of randomly appearing mysterious error messages in Java based robot programming.

Other languages like C++ don't have a garbage collector in the background. This results in a lot of memory space which is allocated and therefore unfeasible for program operations. The programmer could now de-allocate the allocations but that is cumbersome.

The central problem in robotics is that you want the most current values you can get. So if the robot is driving straight ahead to a wall and you want to stop it in front of it minimizing the distance between the wall, every additionally millisecond of controlling the robot results in a lot of iterations through this loop.

With the following code snippet you can fix this problem:

```
int distance;
while(analog_read(0)>10){  //check distance of front sensor
      distance = analog_read(0); //save  distance into a variable
      … //do something with it}
```

Now the integer distance is allocated once again and every iteration cycle is overwritten.

Second Flaw: Multiple Reads

While the programrs always want to have the most current values, the procedure of reading twice in two consecutive lines is not very helpful. Sometimes it even happens that one of the read-out data is a noisy value. It depends a lot on the firmware and how the data can be discriminated.

There are events where the firmware buffers values for all sensors at all times and updates them every, e.g., 100 milliseconds. This is often a problem since the value which the program receives could be up to 100 milliseconds old.

Very often sensor values fluctuate a lot, which makes accurate calculations nearly impossible. When a firmware does not store the values, this might result in large differences between the two values in the sample code. An efficient way to reduce fluctuation of sensor values are filtering techniques described in section 5/sensor values.

To eliminate the differences between consecutive values you can apply the following code:

```
int distance = analog_read(0);
while(distance >10){  //check distance of front sensor
      … //do something with it
      Distance = analog_read(0); //save  distance into a variable}
```

By initializing the variable distance with a sensor value and updating it at the end of the loop the code doesn't need to be read in the while-condition. Now the code should be running a lot more smoothly.

Sometimes, a loop is used as a method to wait for a certain event when programming robots. In that case the following code snippet is more convenient:

```
driveForward();  //start motors
while(analog_read(0)>10);  //checks distance of front sensor
stop();  //stops the motors
```

After the motors are started, the loop will be repeated until the sensor value is lower or equals 10, but this program will discharge the robots battery continuously.

During execution this loop will use up all the performance of the robot because there is no pause in the loop. This causes problems when operating on robots with just a single core and multiple threads.

```
driveForward();  //start motors
while(analog_read(0)>10){ Sleep(10); } //checks distance of front sensor
stop();  //stops the motors
```

With this code snippet the robot will wait 10 milliseconds after every check. When optimizing an algorithm, you normally start with an inefficient version. The optimization process is limited by the running time or the allocated memory. In robotics you also have to consider applicability of the battery and its lifespan. An optimized algorithm often has much more code lines than the raw code. But the code length should not be a criterion. Paul Masurel wrote on his blog about intersecting linked lists in Python [2]. In the optimized code version he went from a five line code to 50 lines.

Sensor Values

Sensor values tend to fluctuate around a given value. These fluctuations have various reasons and should be handled properly by the program of the robot. For that purpose programs often use smoothing algorithms. These filter techniques can be done by hardware or software actions. In this publication this problem is solved by a code.

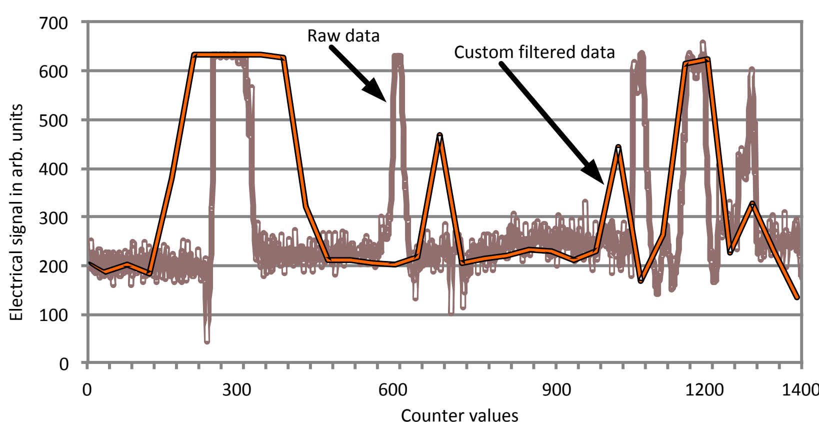

Figure 3 shows the difference a smoothing algorithm can make. The raw data come from an infrared distance-sensor. From the overall 1400 values this particular smoothing algorithm leaves all values that are out of tolerance of the previous ones. This results in an abstract view of the data without losing too much of the reasonable values.

The averaging from four to ten regular values is sufficient.

```
int analogSmooth(int sensor){
      int total = 0;
      int i;
      for(i = 0; i < 5; i++){
            total += analog_read(sensor);
      }
      return total/5;}
```

Sometimes pauses are added in order to get a better distribution of the values. An extended value range can be helpful when the firmware is caching sensor values. This can cause some confusion from time to time.

```
int analogSmooth(int sensor){
      int total = 0;
      int i;
      for(i = 0; i < 5; i++){
            total += analog_read(sensor);
            sleep(10);
      }
      return total/5;}
```

The next algorithm is a custom-made algorithm which tries to cut out values that have a high fluctuation while having a backup strategy and running out of time. The algorithm should not exceed 50 milliseconds, where the smallest quantity of values in the calculations is 12.

The choice of a certain smoothing algorithm depends on the context in which you use the following: (*see next page*)

```
int SmoothVal(int Sensor){
       return (ana-
log10(Sensor)+analog10(Sensor)+analog10(Sensor)+analog10(Sensor))>>2;
}

int SmootherVals(int Sensor, int maxJump){
       int count = 0; int target = 2; int val1; int val2; int val3; bool error; int
retrys = 0;
       do{
          retrys += 1;
          error = false;
          val1 = SmoothVal(Sensor); //read a smoothed value
          msleep(2);
          val2 = SmoothVal(Sensor); //read a second smoothed value
          error = distance(val1, val2) > maxJump;
          if(!error){
              int tryCnt = 0;
              while(error && tryCnt < 5){
                 tryCnt += 1;
                 val3 = SmoothVal(Sensor);
                 msleep(1);
                 error = distance(val2, val3) > maxJump;
              }
              if(error){
                     return (val1 + val2) >> 1;
              }else{
                   int abs1 = distance(val1, val2);
                   int abs2 = distance(val2, val3);
                   int abs3 = distance(val1, val3);

                   if(abs1 < abs2 && abs1 < abs3)
                           return (val1 + val2) >> 1; // average of val1 and val2
                   else if(abs2 < abs3 && abs2 < abs1)
                           return (val2 + val3) >> 1; // average of val2 and val3
                   else
                           return (val1 + val3) >> 1; // average of val1 and val3
                   }
              }
       }while(error && retrys < 10);
       if(error) val3=val1;
       return (val1 + val2 + val3) / 3;}
```

If values need to be up to date as possible, consider simple algorithms like the moving average. On the other hand, if you want a value that represents a time period rather than a particular moment, some more complex algorithms must be implemented.

Conclusion

This publication has shown how complex optimizing programs for low performance robots can be. Optimizing is necessary to run more complex programs on low end robots. There are many applications from analysis of sensor values to optimizing robot algorithms.

Additionally, in this publication value smoothing approaches are compared and disenssed quantitatively. These approaches differ from those that are commonly applied in robots.

Acknowledgement

We would like to thank our principal, Mrs. Mag. U. Hammel, for the continuous support, such as providing a robotic laboratory. Furthermore we would like to thank all our sponsors.

Comments

Amazing Team participated in the **G**lobal **C**onference on **E**ducational **R**obotics in Norman/Oklahoma during july 6 to July 10 2012 (GCER13) under the supervision of Dr. M. Stifter.

The team members won the World championship in the Alliance Challenge. All team members in alphabetical order: Hovorka Markus, Jung Clemens, Langenau Thomas, Lütge Philipp, Podest Patrick, and Tiefengraber Bruno [10].

References

(1) V. Georgitzikis, O. Akribopoulos, I. Chatzigiannakis; “Controlling Physical Objects via the
   a. Internet using the Arduino Platform over 802.15.4 Networks,“ Latin America Transactions,
   b. IEEE, vol.10, no.3 (2012).
(2) http://www.botball.org/2013/events/international-botball, September 2013.
(3) http://fulmicoton.com/posts/intersecting_link_list/, Paul Masurel Blog, September 2013.
(4) http://hackaday.com/category/arduino-hacks/, September 2013.
(5) http://hackaday.com/2013/03/13/mailbox-notifier-texts-when-the-letter-carrier-arrives/,
   a. September 2013.
(6) http://hackaday.com/2013/03/19/diy-arduino-pro-mini-quadcopter/, September 2013.
(7) D. Compton; “Application of an Olfactory Data-Preprocessing Algorithm to Chemotactic
   a. Robotic Navigation,“ Journal of Young Investigators, (2008, July).
(8) D. Nardi, “Robot programming, C++ vs Java,“ (2010).
(9) T. Dean, “Building Intelligent Robots,“ (2002).
(10) http://www.robo4you.at/, September 2013.