Proceedings of the
44th IEEE Conference on Decision and Control, and
the European Control Conference 2005
Seville, Spain, December 12-15, 2005

Turning a toy into a didactic industrial regulator

Alberto Leva

Abstract— **This manuscript presents a didactic application based on the LEGO® MINDSTORMS^{TM} RCX^{TM} architecture[1], and the BrickOS operating system. The RCX is turned into a stand-alone regulator, that can accommodate for one or more PIDs in various configurations, and whose operator interface relies entirely on the LCD display and the buttons of the RCX. The so obtained regulator is suitable for several modulating control experiences, that anyone can make at home using only standard LEGO parts. In addition, extending its functionalities is a very interesting and didactically useful exercise, as the problems to tackle are quite similar to those encountered when dealing with real industrial devices.**

I. Introduction and motivation

The way from the specification of a regulator (e.g., a block diagram) to its implementation is long, and heavily dependent on the particular hardware/firmware/software architecture employed. One of the hardest challenges is perhaps when a (reasonably simple) control structure has to be built on a very low-end device, with limited computational resources and barely essential operator interface capabilities. Needless to say, in such a situation also the debugging possibilities are scarce, and no "control-oriented" development tool is available. In one word, the job has to be done with a hardware normally composed of a microcontroller, a small text display, and a few keys, while the development system is typically an assembler or C compiler (often without floating point arithmetic, though in this work this additional - and significant - difficulty will not be addressed).

All the problems above are seldom treated deeply enough in control education, particularly from the practical point of view [1], [2], [3]. As a consequence too many industrial implementations are poorly structured, cumbersome and difficult to manage and maintain [4], and above all the average attitude of the (control) engineer toward the development of small applications on simple architectures (as is the case, for example, in many embedded systems) is often less systematic than it should be.

This manuscript briefly shows how the LEGO MINDSTORMS RCX can be turned into a regulator with three analogue inputs and three analogue outputs, capable of running one or more PIDs (and potentially regulators of different structure) in various configurations, and with an operator interface that relies entirely on the RCX keys and LCD display, and is quite similar to those of the typical stand-alone industrial regulators. The purpose of the presented work twofold. The first is to describe a very simple experimental setup, giving web references allowing the reader to find all the information needed for building regulators on an RCX and setting up some experiences; the idea is that anyone possessing an RCX (a toy that many students have at home) has a small control laboratory. Notice, by the way, that the great majority of control experiments using LEGO (and particularly MINDSTORMS) elements concentrate on logic control, or robotics: a few examples of the huge literature on the matter are [5], [6], [7], [8]. However, only a few experiences (see e.g. [9], [10]) deal with modulating control. More in particular, to the best of the author's knowledge, no didactic experience exists in which a control program 'similar to a real-world one' as long as the programming tools, the code organisation and the operator interface are concerned, has been implemented on the RCX architecture.

The second purpose of the work starts from the idea that, from an architectural point of view, the RCX is quite a good representative of the hardware frequently found when implementing simple controllers [11]: the purpose is then to show a possible way to educate students to implement control software, even and particularly on low-end architectures, based on a systematic and control-theoretically sound approach, avoiding most of the problems that typically arise when too many *ad hoc*, and not methodologically grounded, solutions are taken. This involves interacting in depth with the control code and modifying it; a particular organisation of the code itself is proposed, justified and encouraged to allow easy (cross)debugging—another important, and often overlooked, lesson to learn.

II. The employed tools

The presented work employs the following hardware and software tools.

- the LEGO MINDSTORMS RCX [12], that in our context can be thought of as a microcomputer based on the Hitachi H8/3292 microcontroller with three analog inputs, three analog outputs, an LCD display, four keys, and an infrared communication port [13];
- BrickOS (formerly legOS), an embedded operating system for the RCX released as open source within the terms of the Mozilla license, featuring priority-based multitasking, POSIX semaphores, dynamic memory management, and IR networking, and providing an API for the development of user programs in the C/C++ language [14];
- the gcc cross-compiler for the Hitachi H8/3292.

As explained in the following, care was taken to isolate RCX-specific code into convenient modules, so that the

A. Leva is with Dipartimento di Elettronica e Informazione, Politecnico di Milano, Via Ponzio 34/5, 20133 Milano, Italy leva@elet.polimi.it

[1]LEGO, MINDSTORMS and RCX are registered trademarks of The LEGO Group.

0-7803-9568-9/05/$20.00 ©2005 IEEE

experience made with the presented system can be effectively employed when implementing control code for different architectures.

III. The RCX-based regulator

The control software in the RCX is entirely written in C, and is organised in three modules:

1) the *control module*, that includes control, I/O and timing functions,
2) the *event handler* that deals with user input (the RCX keys), and
3) the *display module*, that manages the RCX LCD.

Each module is in turn organised in two parts:

1) a *core* part, that includes only computations, or standard C instructions, and
2) a *RCX* part, that isolates all the RCX-specific code (e.g., calls to BrickOS primitives that deal with the RCX inputs and outputs).

This structure allows to write, compile and test the complex and application-specific part of the control code (i.e., the core part) on a different architecture than the RCX, typically a PC. The RCX part of the code is much less tied to specific applications, and once stabilised it is seldom touched. Organising the code as suggested greatly eases the development and debugging of applications, and should be encouraged when developing embedded systems. It should be made clear to the students that the possible availability of an emulator for the target architecture is not necessarily of help: when control code is ported on a real-time target, there is no such thing as single-stepping.

The PID regulator implemented is in the 2 degree of freedom (2-d.o.f.) standardised ISA form [15], i.e., in the frequency domain, the control law

$$\begin{aligned} CS(s) &= K\Big((bSP(s)-PV(s))+\tfrac{1}{sT_i}(SP(s)-PV(s)) \\ &\quad +\tfrac{sT_d}{1+sT_d/N}(cSP(s)-PV(s))\Big) \end{aligned} \tag{1}$$

In (1) the symbols $SP(s)$, $PV(s)$ and $CS(s)$ denote, respectively, the Laplace transforms of the set point, the controlled variable and the control signal, K the PID gain, T_i and T_d the integral and the derivative time, N the ratio between T_d and the time constant of a second pole required for the controller properness, and b and c the set point weights in the proportional and derivative actions. The control law (1) is then discretised with the backward difference method, written in incremental form, and completed with antiwindup, bumpless transfer between automatic and manual mode (the control signal being subject in the latter case to manual increment or decrement), and the so-called "increment/decrement locks", i.e., two boolean inputs (NoInc and NoDec) that prevent CS from increasing and decreasing, respectively. These features make the algorithm suitable for use in the most important control structures.

The control software allows at present for several PID regulators, possibly connected together to form structures like cascade or multivariable controls, and is based on the following data types and structures.

```
// DATA TYPES
typedef float         REAL;
typedef unsigned char BOOL;
typedef struct {   // PID parameters
  REAL K;          // Gain
  REAL Ti;         // Integral time
  REAL Td;         // Derivative time
  REAL N;          // Term 1+sTd/N
  REAL b;          // SP weight in P
  REAL c;          // SP weight in D
  REAL Ts;         // Sampling time
  REAL CSmax;      // Upper CS limit
  REAL CSmin;      // Lower CS limit
  REAL deltaCSMAN; // Manual CS inc/dec
  } PIDPARAMS;
typedef struct {   // PID context
  REAL SP;         // SP
  REAL SPold;      // Previous SP
  REAL PV;         // PV
  REAL PVold;      // Previous PV
  REAL CS;         // CS
  REAL CSold;      // Previous CS
  REAL Dold;       // Previous D
  BOOL MAN;        // TRUE in manual
  BOOL MANinc;     // TRUE in CS man inc
  BOOL MANdec;     // TRUE in CS man dec
  BOOL HIsat;      // TRUE when CS=CSmax
  BOOL LOsat;      // TRUE When CS=CSmin
  BOOL NoInc;      // Prevent CS inc
  BOOL NoDec;      // Prevent CS dec
  BOOL ForceMAN;   // Force PID in man
  BOOL IAmRunning; // TRUE when PID runs
  } PIDLOOPDATA;
```

To specify a control structure, it is first necessary to allocate the following global entities (the example reported refers for brevity to a single PID).

```
// GLOBAL VARIABLES
// For each PID, define and allocate
//   2 PIDPARAMS structures (PIDa,PIDb)
//   2 pointers to them (PIDedit,PIDrun)
//       PIDrun points to the current
//       parameters, PIDedit to those
//       that the user can modify
//   1 PIDLOOPDATA structure for the
//     loop context
#define      NUMBER_OF_PIDs 1
PIDPARAMS    PIDa[NUMBER_OF_PIDs],
             PIDb[NUMBER_OF_PIDs],
            *PIDedit[NUMBER_OF_PIDs],
            *PIDrun[NUMBER_OF_PIDs];
PIDLOOPDATA  PIDloop[NUMBER_OF_PIDs];
//index of initially current PID
```

```
int          currentPID = 0;
//Initial tracking status
BOOL         trackingCS = FALSE;
// Initial mode
BOOL         manual     = FALSE;
```

The C language implementation of the PID follows.

```
// 2-d.o.f. ISA PID ALGORITHM
void PID(PIDPARAMS* R, PIDLOOPDATA* L) {
  REAL deltaSP,deltaPV,deltaP,deltaI;
  REAL D,deltaD,deltaCS;
  deltaSP = L->SP-L->SPold; // SP var.
  deltaPV = L->PV-L->PVold; // PV var.
  if (!L->MAN && !L->ForceMAN) { // AUTO
     deltaP  = R->K
       *(R->b*deltaSP-deltaPV);  // P
     deltaI  = R->K*R->Ts/R->Ti
       *(L->SP-L->PV);           // I
     D = (R->Td*L->Dold+R->K*R->N*R->Td
       *(R->c*deltaSP-deltaPV))
       /(R->Td+R->N*R->Ts);
     deltaD  = D-L->Dold;        // D
     deltaCS = deltaP+deltaI+deltaD;
     if ((deltaCS>0 && L->NoInc)
       || (deltaCS<0 && L->NoDec))
          deltaCS=0;             // lock
     } else {                    // MAN
     deltaCS = 0;
     if(L->MANinc && !L->MANdec) {
        deltaCS =  R->deltaCSMAN;
        L->MANinc = FALSE; }
     if(L->MANdec && !L->MANinc) {
        deltaCS = -R->deltaCSMAN;
        L->MANdec = FALSE; }
     D = 0; }
     L->CS = L->CSold+deltaCS;
  // Antiwindup
  if(L->CS>R->CSmax) L->CS=R->CSmax;
  if(L->CS<R->CSmin) L->CS=R->CSmin;
  // Signal saturations
  L->HIsat = (L->CS==R->CSmax);
  L->LOsat = (L->CS==R->CSmin); }
  // State update
  L->CSold = L->CS; L->SPold = L->SP;
  L->PVold = L->PV; L->Dold  = D; }
```

The control code is organised in threads, to ensure that real time requirements are met. The following code excerpt is relative to the control thread of the single PID application.

```
// CONTROL THREAD (example with 1 PID)
void executePID() {
// This thread manages the execution of
// a single PID regulator
  time_t time_cycle = 0; // Set up
  double CSraw = 0;
  motor_a_dir(fwd);
  motor_a_speed(0);
  while (1) {
    // Time stamp: start of PID exec
    time_cycle = sys_time;
    // Disable context change (Edit/Run)
    PIDloop[0].IAmRunning = TRUE;
    // Read input
    PIDloop[0].PV = (REAL)(100-LIGHT_1);
    // PID algorithm exec
    PID(PIDrun[0],&PIDloop[0]);
    // Write output
    if (PIDloop[0].SP == 0
      && PIDloop[0].MAN == FALSE)
        CSraw = 0;
    else CSraw = 255*(PIDloop[0].CS
      -PIDrun[0]->CSmin)
      /(PIDrun[0]->CSmax
      -PIDrun[0]->CSmin);
    motor_a_speed((int)CSraw);
    // Enable context  change (Edit/Run)
    PIDloop[0].IAmRunning = FALSE;
    //Check if sampling time is kept
    wait_event(&checkSampleTime,
       time_cycle); }
```

Other threads, that normally do not need modifying on the part of the user who creates a new control application, manage the keys, the display, and so forth. Thread synchronization is obtained through the BrickOS semaphores, which is standard practice in POSIX-compliant environments. The only additional precaution is that the system prevents changing the parameter set of a particular regulator (i.e., swapping its PIDrun and PIDedit pointers) amidst the execution of a control step involving that regulator, for obvious reasons.

IV. The documentation

Another important aspect of the didactic activity (so important, in the author's opinion, to deserve a section in this manuscript) is the creation of a suitable documentation for the user of the produced control application.

Contrary to intuition, that task proved to be quite difficult for the students, due basically to two reasons. One is that the user interface of the RCX is very simple, so that performing all the required operation involves many complex key sequences. The second is that the system is configurable, and therefore the documentation has to be easy to extend adapt if a different control configuration (e.g., a cascade control system) is built.

The choice adopted was to structure the documentation along a basically graphical approach, explaining the required user operations visually. Figure 1 reports a small example, namely the explanation of how the RCX keys are used, and how numerical items are entered.

According to the students, writing a clear documentation was almost as difficult as programming the system. They tested their work by having other students, not involved

in the project, using the regulator. The unanimous opinion was that producing the documentation was a very important experience, and made the students aware of several too frequently overlooked problems.

V. The possible didactic experiences

A. *Using the regulator* as is

The RCXD-based regulator can be used for a number of experiences, and the reader is surely able to imagine many possible ones. The LEGO system provides a lot of different sensors and actuators, so that many of those experiences can be made with LEGO parts only: for example, the LEGO lamp and temperature sensor can be used to build a very simple temperature control system. In the following, a speed control application is presented to give an idea of what can be done with the presented regulator. In any case, experience leads to conclude that the most natural didactic objectives to pursue with the regulator used *as is* are the following:

- understand the physical structure of a control system, and learn to recognize its elements in a very simple implementation they can fully inspect;
- get accustomed with the user interface of simple, stand-alone regulators, and learn to perform the typical control operations (startup, automatic/manual switching, and so on);
- learn to apply systematic procedures for PID tuning (e.g., methods based on the identification of simple models 'in the field') using the limited data provided by the user interface of such regulators;

Such experiences can be made easily also at home, based on simple guidelines given in the regulator's documentation. Interested students can take profit of this activity starting from the basic courses of Automatic Control.

B. *Extending the system*

Students can also extend the system, either by constructing different control systems based on the basic blocks already available (e.g., a cascade or a multivariable system), or by building new blocks (e.g., a new type of regulator, a filter, and so on). The first type of activity is not particularly difficult, requires C programming capabilities but little or no knowledge of BrickOS, and leads essentially to the following didactic achievements:

- understanding the organisation of the control software into threads, and consequently how to structure a given application into real-time and non real-time tasks, using the BrickOS priority system accordingly;
- understanding the profound difference between a 'multi-tasking' and a 'real-time' application (two concepts that are too frequently confused and wrongly overlapped by the students), i.e.,
  - that multitasking is useful for real-time applications, but not strictly necessary, and
  - that a multitasking application, if overloaded, works with degraded performance, while an overloaded real-time application simply does not work;
- understanding the user interface primitives made available by BrickOS and by the code developed in the presented project, and using them effectively.

Such an activity can be proposed to undergraduate students, and helps them a lot connecting concepts that they feel as coming from 'computer science' and from 'automation control', which is a very important educational result.

The second type of activity requires good programming and code structuring capabilities, and in some cases quite deep knowledge of BrickOS, and is suitable for graduated students. it is also possible, once an application has been developed on the RCX, to port that application on a 'real-world' microprocessor-based controllers. Some preliminary studies in this direction appear promising from the didactic standpoint.

VI. An application example

In this section, a simple application example is presented. The system under control is composed of two LEGO MINDSTORMS motors, connected with a flexible transmission as depicted in figure 2. One of the motors drives the transmission, while the second acts as mechanical load and tacho generator at the same time. The controlled variable is the load rotational speed, while the control variable is the motor command. A bar can be lowered manually to increase the overall transmission friction; two bushes under the bar allow for easily repeatable experiments. The experimental setup used, together with some similar ones, is described with more details in [9].

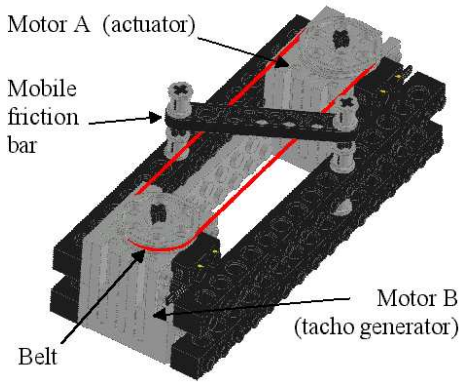

Fig. 2. The speed control system.

The system is entirely composed of LEGO elements except for a passive RC filter with gain equal to 0.5, necessary to clean up the tacho output from noise, and to equalize the typical tacho output voltage produced with a 1:1 transmission to the RCX input range. The filter is composed of one resistor and one capacitor, and is easily constructed on a breadboard, or even cutting a LEGO cable and connecting the resistor and the capacitor to the so obtained cable terminals, as indicated in figure 3. The transfer function from the tacho voltage V_T to the RCX input voltage V_{RCX} derived from the scheme of figure 3 is

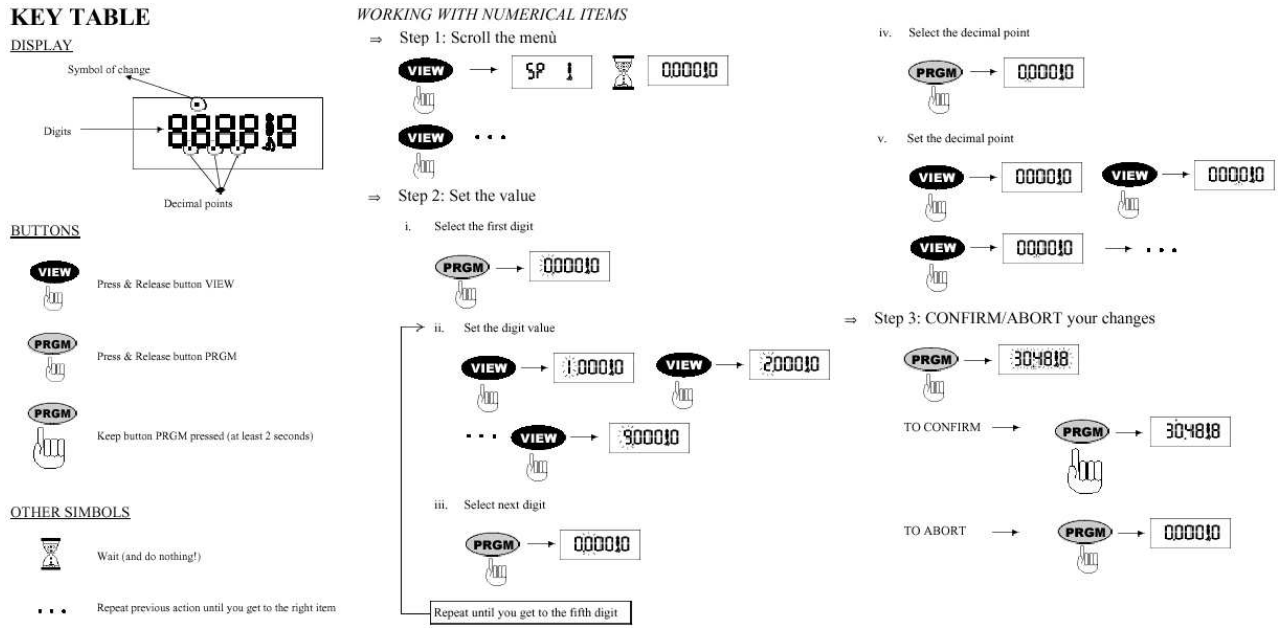

Fig. 1. *Excerpt* 1 from the user manual: key table for the RCX, and how to manage numerical items.

$$\frac{V_T(s)}{V_{RCX}(s)} = \frac{1/2}{1+sRC/2}, \tag{2}$$

and therefore suitable values for R and C are 15 kΩ and 8.2 μF, respectively, leading to a time constant of approximately 60 ms, suitable for the speed control experiments of this work.

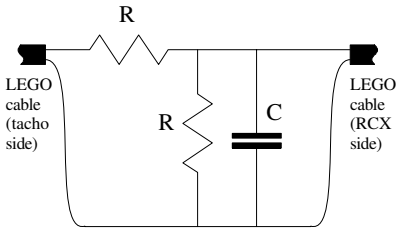

Fig. 3. The tacho generator cable and filter.

A better, and slightly more complex, tacho interfacing circuitry is shown in figure 4, and is recommended wherever it is possible to do some (quite basic) breadboard assembly. Neglecting for simplicity the role of the OP-AMP buffer, the diodes and the adapting resistor R4, the transfer function of this circuit, with the same notation of (2), turns out to be

$$\frac{V_T(s)}{V_{RCX}(s)} = \frac{\frac{R_2}{R_1+R_2}}{1+sRC\left(\frac{R_1R_2}{R_1+R_2}+R_3\right)}. \tag{3}$$

As a consequence, the values of figure 4 lead, as required, to a static gain of 0.5 and a time constant of 68 ms.

To acquire the analog speed value on the RCX, the input used (input 1 in the presented application) is configured as a light sensor, since that configuration is, among hose provided by BrickOS, the nearest to the acquisition of a constant voltage value, and the measurement is obtained with the simple instruction

```
PIDloop[0].PV = (REAL)(100-LIGHT_1);
```

For further details, the reader can refer to the RCX and the BrickOS documentation.

Figure 5 shows a closed-loop response of the system to a speed set point step. To record the required signals, a PC is used with a National Instruments 6024E analog interface card, and a simple application written in the LabVIEW programming language. The PC is not necessary for the controller operation, it is just used to obtain data for visualization and analysis, so the idea of 'allowing for control experiences at home' is preserved. It would be possible to have the RCX record the data and then send them to a PC through the IR port, but one should take into account the limitations of the RCX memory. In any case, implementing such an extension would be very easy.

VII. Future work

The system is open to a wealth of activities. Some examples may be

- designing and implementing additional control experiments, involving other types of (LEGO) sensors, and other control structures;
- creating a PC-based application, e.g. in the java language to improve portability, capable of interacting with the RCX-based regulator via the IR port, to provide the system with a more powerful and user-friendly interface, data logging capabilities, and so forth;
- employing the RCX IR port to create a network of RCX-based regulators and possibly PCs, adhering to some FieldBus standard.

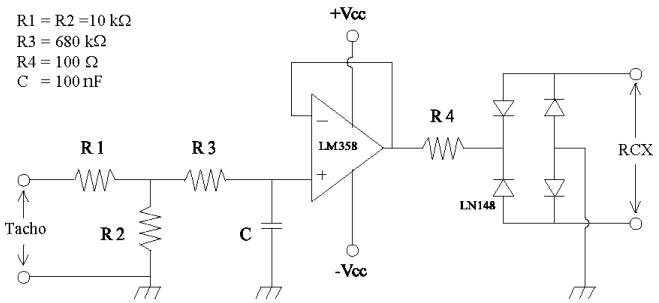

Fig. 4. Improved interfacing of the tacho generator.

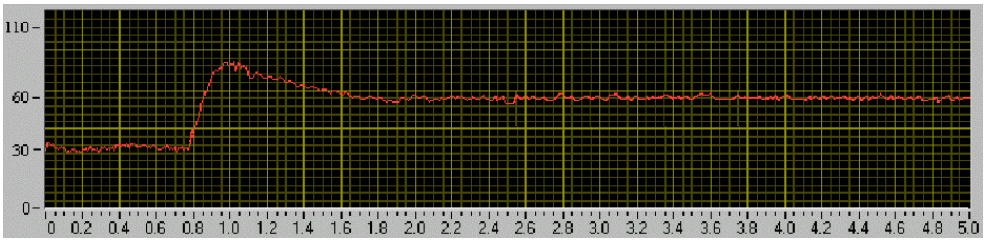

Fig. 5. Closed-loop set point step response of the speed control system.

VIII. Conclusions

A didactic application based on the LEGO MINDSTORMS RCX and the BrickOS operating system has been presented. The RCX is turned into a stand-alone industrial regulator, fully functional and whose computational core and operator interface rely only on the RCX resources. The presented application is relative to a single-loop PID regulator, but the system is entirely open (the code is written in C and totally accessible), so that different and more complex control structures can be accommodated for. Also, some possible future activities were briefly outlined.

The presented work allows anyone possessing an RCX to do modulating control experiments, since all the required tools can be downloaded and used freely, within the open source paradigm. In addition, interacting with the control code permits to experience most of the typical difficulties encountered when developing control software for simple architectures, with limited computational and interface resources. Such an experience greatly increases the implementation capabilities of students, and makes their approach to control software development more compatible with the requirements of the professional domain.

The RCX PID project has its own home page located at the URL *http://www.elet.polimi.it/upload/leva/Projects-/RCXPID2003/*. It is possible to download the project documentation (at present in Italian, an English translation is underway) and the code of the presented example, both in source (.c) and compiled (.lx) form.

IX. Acknowledgements

The author is grateful to the former students Alex Mancastroppa and Paolo Romagnoli, who implemented the system and wrote the documentation for their BSc thesis, working at the Cremona site of the Politecnico di Milano. Many thanks are due also to several other students, who tested the produced didactic application and provided very useful comments.

References

[1] N. Kheir, K. Åström, D. Auslander, K. Cheok, G. Franklin, M. Mastem, and M. Rabins, "Control systems engineering education," *Automatica*, vol. 32, no. 2, pp. 147–166, 1997.

[2] M. Soklic, "Laboratory for real-time and embedded systems," *Computers in Education*, vol. 12, no. 4, pp. 1–11, 2002.

[3] A. Leva, "A hands-on experimental laboratory for undergraduate courses in automatic control," *IEEE Transactions on Education*, vol. 46, no. 2, pp. 263–272, 2003.

[4] ——, "An experimental laboratory on control structures," in *Proc. ACC 2004*, Boston, MA, 2004.

[5] M. Cyr, V. Miragila, T. Nocera, and C. Rogers, "A low-cost, innovative methodology for teaching engineering through experimentation," *Journal of Engineering Education*, vol. 86, no. 2, pp. 167–171, 1997.

[6] B. fagin, "An Ada interface for LEGO MINDSTORMS," *Ada Letters*, vol. 21, no. 2, Sept. 2000.

[7] A. Kumar, "Using robots in an undergraduate artificial intelligence course: an experience report," in *Proc. 31st Annual Frontiers in Education Conference*, Reno, NV, 2001.

[8] J. Schumacher, D. Welch, and D. Raymond, "Teaching introductory programming, problem solving and information technology with robots at West Point," in *Proc. 31st Annual Frontiers in Education Conference*, Reno, NV, 2001.

[9] D. Gasperini, F. Schiavo, W. Spinelli, C. Veber, and A. Leva, "A set of hardware and software tools for control education," in *Proc. IBCE'04*, Grenoble, France, 2004.

[10] P. Gawthrop and E. McGookin, "A LEGO-based control experiment," *IEEE Control Systems Magazine*, vol. 24, no. 5, pp. 43–56, Oct. 2004.

[11] D. Baum, *Dave Baum's efinitive guide to LEGO MINDSTORMS*. New York, NY: Springer-Verlag, 2000.

[12] The LEGO MINDSTORMS official home page. [Online]. Available: http://www.legomindstorms.com/

[13] LEGO MINDSTORMS internals page. [Online]. Available: http://www.crynwr.com/lego-robotics/

[14] The BrickOS operating system home page. [Online]. Available: http://brickos.sourceforge.net/

[15] K. Åström and T. Hägglund, *PID controllers: theory, design and tuning—2nd edition*. Research Triangle Park, NY: Instrument Society of America, 1995.