A Digital PID Controller Using RTAI

R. S. Guerra, H. J. Gonçalves Júnior, W. F. Lages
Federal University of Rio Grande do Sul,
Electrical Engineering Department
Av. Osvaldo Aranha 103, Porto Alegre RS, CEP 90035-190, Brazil
rsguerra@eletro.ufrgs.br, hermes@eletro.ufrgs.br , w.fetter@ieee.org

Abstract

This work presents an implementation of a digital PID (Proportional + Integral + Differential) controller in a PC machine running RTAI. PID controllers are well known and have a wide range of applications in the control of analog processes such as servomotors. The well implemented PID controllers are usually hard real time systems. The PID controller described in this work uses the PC parallel port as an analog I/O interface. Just two bits are used as 'analog' interfaces through a technique called Pulse Width Modulation (PWM). This technique allows an analog interface to be built without the use of conventional A/D and D/A converters. Real time tasks were provided by threads under LXRT hard real time mode. The implementation of PWM under RTAI using the timing functions provided by RTAI are presented and the convenience of using these functions for actual real time control is discussed. In particular, the semantics of `rt_sleep()` is analyzed in detail and it is shown that the current semantics can lead to a confusing interpretation of what a periodic task is and what a task period is. Furthermore the current semantics of `rt_sleep()` is incompatible with the semantics of `rt_busy_sleep()`, which can lead to even more confusion. A new semantics for `rt_sleep()` is suggested.

1 Introduction

1.1 The PID Controller

A *feedback controller* is designed to generate an output that causes some corrective effort to be applied to a process so as to drive a measurable process variable, $y(t)$, towards a desired value, known as the *set-point* or *reference*, here noted $r(t)$. The concept is based, as shown in figure 1, on the re-input of the system own output according to certain laws (hence the name 'feedback'). It is desired for the system output to follow the reference $r(t)$. All feedback controllers determine their output by observing the difference, called *error*, here noted $e(t)$, between the set-point and the actual process variable measurement. This theory is valid for a wide class of systems, which include, but is not restricted to, linear systems [1].

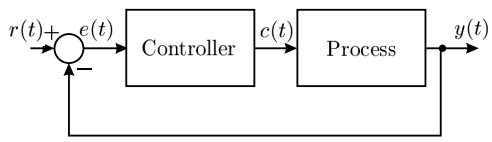

FIGURE 1: *Feedback Control System*

For the class of the linear systems, a very widely used controller is the PID (Proportional Integral Differential). The PID looks at (a) the current value of the error, (b) the integral of the error over a recent time interval, and (c) the current derivative of the error signal to determine not only how much of a correction to apply, but for how long. Each of those three quantities are multiplied by a 'tuning constant' and added together [1]. Thus the PID output, $c(t)$, is a weighted sum as shown in figure 2:

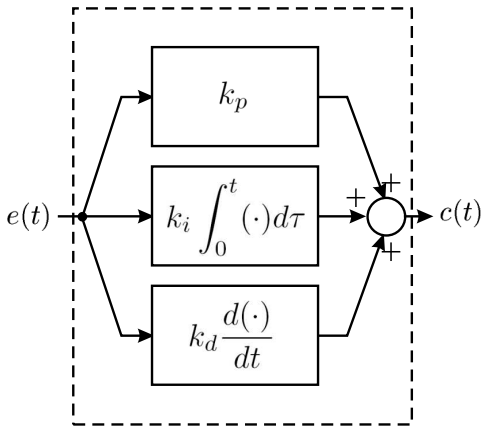

FIGURE 2: *PID Controller*

Depending on the application one may want a faster convergence speed or a lower overshoot. By adjusting the weighting constants, k_p, k_i and k_d, the PID is set to give the most desired performance [1].

1.2 The Digital PID Controller

As explained so far we considered time continuous domain and analog variables. Today, digital controllers are being used in many large and small-scale control systems, replacing the analog controllers [5]. It is now a common practice to implement PID controllers in its digital version, which means that they operate in discrete time domain and deal with analog signals quantized in a limited number of levels [1]. The trend toward digital rather than analog control is mainly due to the availability of low-cost digital computers [5]. A digital version of the PID controller is shown in figure 3:

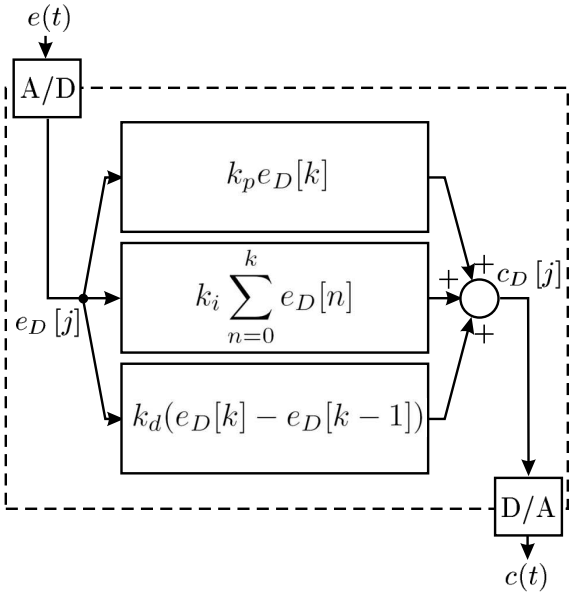

FIGURE 3: *Digital PID Controller*

In its digital version, the integral becomes a sum and the differential a difference. The continuous time signal $e(t)$ is sampled in fixed time intervals equals a determined *sample period*, here called T_c (in figure 3 $T_c = 1$). An A/D (analog to digital) converter interfaces the input and a D/A (digital to analog) converter interfaces the output. This sampled and digitalized input, called $e_D[j]$, exists only in time instants $t = kT_c$ for all $k \geq 0 \in \mathbf{Z}$. It is assumed that these digital values are processed instantly and the result is posted immediately, which obviously is not true. Even if it is possible to deliver the results faster from time to time it is most desirable to maintain a fixed and rigid sample period [1].

Then it is desirable for the controller (a) to have the sample period T_c as small as possible and (b) to have as many levels of quantization as possible. A lower bound for the sample period is the computing time of a whole cycle of the digital PID (which includes the A/D and D/A conversion). In most practical situations noise filtering may imply another lower bound for the sampling period. The number of levels of quantization of the input and output analog variables will depend on the resolution of the A/D and D/A converters respectively. Converters of high resolution are expensive. Helpfully the state-of-the-art technology in the field of the microprocessors makes possible to have good computation time with low cost hardware [5].

1.3 The PWM Technique

Analog voltages and currents can be used to control processes directly. As intuitive and simple as analog control may seem, it is not always economically attractive or practical. Analog circuits tend to drift over time and can, therefore, be very difficult to tune. By controlling analog circuits digitally, system costs and power consumption can be drastically reduced. Pulse Width Modulation (PWM) is a powerful technique for controlling analog circuits with digital signals [4].

PWM is a way of encoding digitally analog signal levels. The duty cycle of a square wave is modulated to encode a specific analog signal level, as shown in figure 4. The PWM signal is still digital because, at any time instant, it is either *on* or *off*. The relation between the *on*-time and the *off*-time varies accordingly to the analog level to be represented. The analog level is obtained through a series of *on* and *off* pulses. Given a sufficient bandwidth, any analog value may be encoded with PWM [3].

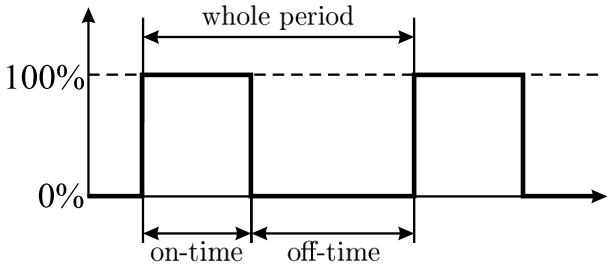

FIGURE 4: *An example of PWM wave*

The cycle period must be short if compared to the process response time to a change in the switch state [4].

The mean value of the PWM wave corresponds to the analog value it represents, proportionally to the

wave duty cycle. A process with slow time response (if compared with the PWM period) may naturally decode the PWM signal into the analog signal it represents dispensing a 'PWM decoder'. In fact, PWM decoding is actually done through a low-pass filtering, either naturally when the process has this effect, or maybe forced low-pass filtering if the process does not have this effect.

2 Methodology

The authors have implemented a Digital PID Controller in a standard PC machine running Linux-RTAI. It was used a Linux-2.4.18 kernel patched with the RTAI-24.1.9 . The I/O was done by PWM through two pines of the parallel port: one for input and the other for output, dispensing the analog-to-digital converter which is the most expensive component in a data-acquisition system [5]. The PWM coding/decoding has been dealt by software routines implemented in Linux threads with LXRT RTAI hard real time constraints. The main routine, which is responsible for the actual PID calculation, has also been implemented in a thread with hard real time constraints. The communication between threads was made through global variables shared with mutual exclusion, guaranteed by semaphores. The next sub-sections explain each part in detail.

2.1 The PID Task

The PID task is the core of the application, where the controller calculation is actually made. The PID controller receives as input the error, given by

$$e(t) = r(t) - y(t) \tag{1}$$

and computes the control variable $c(t)$ which is its output. The PID controller has three terms: (a) the proportional term P corresponding to proportional control, (b) the integral term I giving a control action that is proportional to the time integral of the error, and (c) the derivative term D, proportional to the time derivative of the error. The control signal $c(t)$ is calculated as follows [1]:

$$c(t) = \underbrace{k_p e(t)}_{P} + \underbrace{k_i \int_0^t e(\tau)d\tau}_{I} + \underbrace{k_d \frac{de(t)}{dt}}_{D} \tag{2}$$

It is common to define $k_p = K$, $k_i = K/T_i$ and $k_d = KT_d$ [1]. Then the following well known 'text book' version of the equation arises:

$$c(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{3}$$

It is necessary to *discretise* the controller, that is, to approximate the integral and derivative terms to forms suitable for computation by a computer. From a purely numerical point of view we can use:

$$\frac{de(t)}{dt} \approx \frac{e(t) - e(t - T_c)}{T_c} \tag{4}$$

$$\int_0^t e(\tau)d\tau \approx T_c \sum_{n=0}^{k} e(nT_c) \tag{5}$$

In the above equations, T_c is the sampling period and in equation 5 it is assumed that $k = \lfloor t/T_c \rfloor$. Notice that equations 4 and 5 give us unbiased estimators as $T_c \to 0$. By this way the equation 3 may be approximated by:

$$c(kT_c) = K\left(e(kT_c) + \frac{T_c}{T_i}\sum_{n=0}^{k} e(nT_c) + T_d\frac{e(kT_c) - e((k-1)T_c)}{T_c}\right) \tag{6}$$

The discrete PID equation comes from equation 6, by referencing the iterations instead of the time, that is, substituting $f(kT_c)$ by $f[k]$. Then, the discrete PID equation is:

$$c[k] = K\left(e[k] + \frac{T_c}{T_i}\sum_{n=0}^{k} e[n] + T_d\frac{e[k] - e[k-1]}{T_c}\right) \tag{7}$$

This particular form of the PID algorithm is known as the *positional* PID controller.

The figure 5 shows the sequence of operations for the PID task [1].

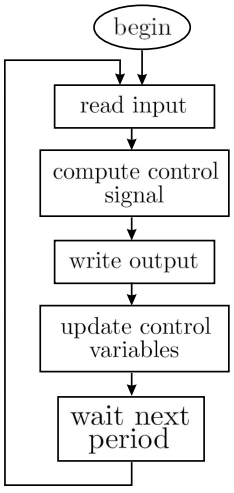

FIGURE 5: *PID Task Diagram*

2.2 The PWM Task

The PWM task is responsible for two things:

1. for encoding the PID digital output, which is stored in a variable, into an external PWM wave signal measurable through a parallel port pin, and;
2. for decoding an external PWM wave signal, input through another parallel port pin, into a digital signal, which will be stored in another variable, to be the PID feedback.

This can be done by dividing a whole PWM cycle into several fixed time intervals, each one hard real time constrained, as illustrated in the figure 6. In this case the PWM instant signal value is to be considered either *on* or *off* within an interval, allowing changes to be made or acknowledged only between these intervals. By doing so, the represented analog signal is quantized in as many levels as many divisions were made.

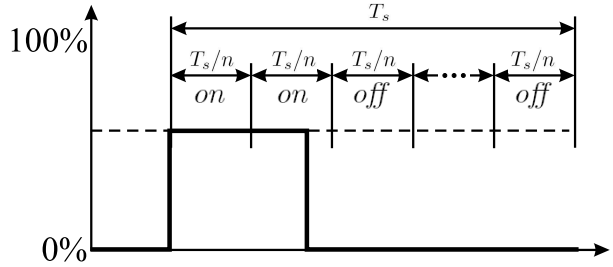

FIGURE 6: *PWM cycle divided in n fixed time intervals*

Let's call the PWM period T_s. If the PID has a lower update frequency than the PWM output wave, which means $T_c > T_s$, then several PWM pulses will be generated before the PID completes a cycle, which is generally the case. On the other hand, being $T_c > T_s$, in order to read the PWM input wave, detecting just one of these pulse widthes within a PID period would be sufficient. This is shown in figure 7.

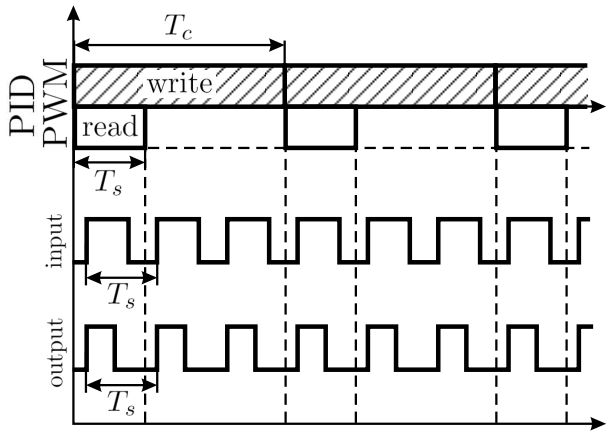

FIGURE 7: *General case: $T_c > T_s$*

2.2.1 Special case: $T_c = T_s$

Let's suppose the PWM cycle period as being the same as the PID cycle period, that is $T_c = T_s$. Then every PWM output pulse needs to be generated, and, in this case, each PWM input pulse needs to be read because there will be only one PWM pulse *per* PID cycle (See figure 7 for $T_c = T_s$). Breaking down this T_s period into n fixed time intervals, as shown in figure 6, means that a whole PWM would need to be generated and read piece by piece in a periodic cycle with period T_s/n. A way of implementing this, is shown in the block diagram of figure 8. Note that, since there will always be a writing *and* a reading, this can be done in the same loop.

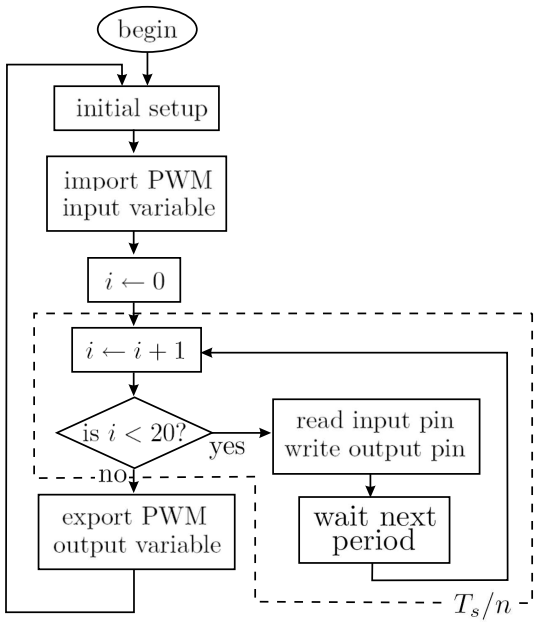

FIGURE 8: *Block diagram of the PWM task for $T_c = T_s$*

2.2.2 General case: $T_c > T_s$

For the general case, where $T_c > T_s$, it is useful to treat the reading procedure and the writing procedure separately.

For the PWM input wave reading, one never knows exactly where the change from *on* to *off* will occur. Due to this fact, the reading must be done in the polling fashion of the block diagram of figure 8.

In general, the PWM I/O wave period will be several times fewer than the PID cycle period [4]. In this case the computational burning of this polling process may be attenuated because it is necessary to read just one of the several PWM input wave cycles,

to be feeded to the PID within his period. During this T_s period, the input pin will be checked cyclically, as many times as many levels one wants for the numerical precision, but only once in T_c, freeing resources periodically during $T_c - T_s$ seconds. It is summarized as follows:

1. create a periodic task with period T_c;
2. read the state of the parallel input pin, which will correspond to $1/n$ of the total PWM input value;
3. 'sleep' a T_s/n period;
4. do item 2 and 3 n times;
5. wait for the next task period, T_c.

For the PWM output wave writing, it is always known *a priori* where the change from *on* to *off* will take place. With this information one may avoid the computational burning of the cyclic loop, which, in the case of the writing, would be done not only once within a PID period, but for every PWM pulse generation. The alternative solution for writing may be as follows:

1. create a periodic task, with period T_s;
2. in the beginning of the loop, turn *on* the PWM output pin of the parallel port;
3. make this task 'sleep' for a period equivalent to the desired *on* time of the PWM output wave form;
4. immediately after the sleep, switch the *on* to *off* and wait until the end of the T_s period;
5. go to item 2.

The advantage is to free the CPU while the task 'sleeps'. By this way it is possible to have a better quantization without the computational burning of the polling approach.

3 Analyzing 'sleeping' RTAI Functions

As shown in the PWM reading and writing procedures, sometimes it is desirable to make a task 'sleep'. To do this, there are three different RTAI functions [2]: (a) `rt_busy_sleep()`, (b) `rt_sleep()` and (c) `rt_sleep_until()`. The first one 'sleeps' without freeing resources, while the last two do it freeing resources. The functions `rt_sleep()` and `rt_sleep_until()` differ only in that for the first one, it is given the time period the task will sleep, while for the second one, it is given the absolute time until the task will need sleep, but internally the implemented code is almost the same.

When called, the function `rt_busy_sleep()` sums the current time plus the period it is desired to sleep, storing the result. Then a `while` keeps comparing this stored time to the current time. When the current time passes the stored time the function proceeds. The implemented code, file `rtai_sched.c`, is as follows:

```
void rt_busy_sleep(int ns) {
  RTIME end_time;

  TRACE_RTAI_TASK(TRACE_RTAI_EV_TASK_BUSY_SLEEP,
                                          ns,0,0);

  end_time = rdtsc() +
           llimd(ns, tuned.cpu_freq, 1000000000);

  while (rdtsc() < end_time);
}
```

No change is made on the task period, but, as the name of this function suggests, the CPU remains busy while 'sleeping', which is not desirable for the PWM writing and reading procedures.

The `rt_sleep()` and `rt_sleep_until()` functions free the CPU while the task 'sleeps'. These functions do so by using the same structure which `rt_task_wait_period()` uses, that is: it reprograms the task resume time, here called T_r. The function `rt_sleep()` changes the resume time to be the sum of the current time plus the desired sleep time:

$$T_r[k] = T_{now} + T_{sleep} \tag{8}$$

The code implemented in the `rtai_sched.c` file is as follows:

```
void rt_sleep(RTIME delay) {
  unsigned long flags;

  TRACE_RTAI_TASK(TRACE_RTAI_EV_TASK_SLEEP,
                                    0, delay, 0);

  hard_save_flags_and_cli(flags);
  if ((rt_current->resume_time =
   (oneshot_timer ? rdtsc():rt_times.tick_time)+
                           delay) > rt_time_h) {
      rt_current->state |= DELAYED;
      rem_ready_current();
      enq_timed_task(rt_current);
      rt_schedule();
  }
  hard_restore_flags(flags);
}
```

It works well, but there is a big inconvenience in doing this: it does change the task resume time which is used for the task period calculation. The function `rt_task_wait_period()` is in charge of maintaining the periodic task being called periodically. This is done by reprogramming the task resume time to be its previous resume time plus the task period:

$$T_r[k+1] = T_r[k] + T_p \tag{9}$$

The code of this function is also in the `rtai_sched.c` file, as follows:

```
void rt_task_wait_period(void) {
  unsigned long flags;

  TRACE_RTAI_TASK(TRACE_RTAI_EV_TASK_WAIT_PERIOD,
                                        0, 0, 0);

  hard_save_flags_and_cli(flags);
  if (rt_current->resync_frame) {
      rt_current->resync_frame = 0;
      rt_current->resume_time = rt_get_time();
  } else if ((rt_current->resume_time +=
              rt_current->period) > rt_time_h) {
      rt_current->state |= DELAYED;
      rem_ready_current();
      enq_timed_task(rt_current);
      rt_schedule();
  }
  hard_restore_flags(flags);
}
```

If no other function changes the resume time $T_r[k]$, for other purpose it does work well. But if `rt_sleep()` has been called within the task period, then the last resume time $T_r[k]$ has been replaced by the 'sleep' resume time, and as a consequence *this* value is summed with the task period T_p to calculate the next resume time in the `rt_task_wait_period()` function. Implemented this way, `rt_sleep()` does change the task period, which is not desirable and is incompatible with the semantics of the `rt_busy_sleep()`.

The main problem of this current implementation is that the control variable T_r actually plays two very distinct roles:

1. It acts as a way of controlling the scheduler for the next task shot, and;
2. It acts as a reminder of the time when the last period shot happened.

It is desirable for a 'sleep' function to make the task wait for a given delay T_{sleep} without changing its period T_p, that is: the 'sleep' time must also be accounted for the task period calculation. In order to preserve the task period, it must not be under the influence of other functions changes in the task resume time T_r. There are many ways of solving this problem. The authors point three:

1. The `rt_task_wait_period()` function could find the next task resume time $T_r[k]$ by calculating $T_r[k] = T_0 + k \times T_p$, where T_0 is the absolute time of the first task shot;
2. The `rt_task_wait_period()` function could keep track of the time of the periodic shots, as it actually does, but in another auxiliary variable, say T_{aux}, instead of doing this directly on the resume time T_r. The time of the next shot would then be calculated as the sum of T_{aux}, which stores the last period shot, with a task period T_p (equation 10). The result would then be copied to T_r (equation 11);

$$T_{aux}[k+1] = T_{aux}[k] + T_p \tag{10}$$

$$T_r[k+1] = T_{aux}[k+1] \tag{11}$$

3. The functions which change the task resume time (*i.e.* `rt_sleep()`, `rt_sleep_until()`), could store T_r before changing it, and then restore this previous value before going on.

The first solution is elegant and robust and has also the advantage that it does not imply in changing the code of those other functions which change the task resume time T_r. To make it possible one needs to store the time of the task first shot, T_0. The next task resume time may be found without explicitly knowing k. The next task resume time would be

$$T_r[k] = \{\lfloor (T_{now} - T_0)/T_s \rfloor + 1\} \times T_s \tag{12}$$

where '$\lfloor \cdot \rfloor$' denotes the floor function. Obviously, the equation 12 does not necessarily correspond to the optimal code implementation for its calculation.

The second solution also does not change the code of the other functions which may change the task resume time T_r, and has a simpler mathematical implementation (a sum), but must keep updating a control variable in the memory. It intends to maintain the same idea of the current implementation, but treating the resume time variable only as means of setting the time of the next shot, making the period calculation with another auxiliary variable T_{aux}. Once the period has been calculated and stored in T_{aux}, as shown in equation 10, it can be copied to T_r. By this way both variables, T_{aux} and T_r, would be equal, except when another function, say `rt_sleep()`, needs to resume the task earlier (before completing a period). In this case this another function may change T_r. This does not change the task period because the period is summed with the time stored in the auxiliary variable, which may only be changed for the purpose of the next period computation.

4 Conclusion

As shown, by using RTAI, it is possible to build a PWM based PID controller directly in a PC machine running a standard Linux distribution without the need for external hardware, except maybe for an analog to PWM converter to the input in some cases. This is a very flexible and low cost controller and has application in most of the usual control environments (*i.e.* servomotors, temperature control). Depending on the sample period associated and on the PWM resolution, the PC CPU may still be shared with other non-realtime applications such as graphic environments, text editors, etc.

Problems with the semantics of the 'sleeping' commands of RTAI were pointed and some possible solutions were presented.

By changing the control algorithm, the structure of this application may be expanded to implement other feedback controllers (*i.e.* optimal control, adaptive control). It is also possible to run MIMO (multiple-inputs-multiple-outputs) control systems, or several independent SISO (single-input-single-output) systems, by making use of more parallel port pins.

References

[1] ÅSTRÖM, K. and HÄGGLUND, T. , 1995, *PID Controllers: Theory, Design, and Tuning*, Instrument Society of America, ISBN 1-55617-516-7.

[2] BIANCHI, E., DOZIO, L. and MANTEGAZZA, P. , 2000, *A Hard Real Time supportfor LINUX*, DIAP-RTAI Documentation.

[3] BIRD, B. M., KING, K. G. and PEDDER, D. A. G. , 1993, *An Introduction to Power Electronics*, John Wiley and Sons Ltd., ISBN 0 471 92616 7; 0 471 92617 5 (pbk)

[4] MOHAN, N., UNDELAND, T. M. and ROBBINS, W. P. , 1995, *Power Electronics*, John Wiley and Sons Ltd., ISBN 0 471 58408 8 (cloth)

[5] OGATA, K. , 1995, *Discrete-Time Control Systems*, Prentice-Hall International, Inc., ISBN 0-13-328642-8.