Real-time Control Systems: A Tutorial

A. Gambier

Automation Laboratory, B6 23-29, EG. Bauteil C
University of Mannheim, 68131 Mannheim, Germany
gambier@ti.uni-mannheim.de

Abstract

The literature about real-time systems presents digital control or computer controlled systems as one of its most important practical application field. However, it is very difficult to find in these textbooks real-time control aspects. It seems to be more natural that these applications should be treated as part of digital control courses. In spite of that, control system literature rarely includes extensively the real-time subject and it does normally not pay attention to real-time aspects beyond algorithms and choice of sampling times. The aim of this paper is to highlight important issues about real-time systems that should be taken into account at the moment to implement digital control.

1 Introduction

The implementation of digital control systems and real-time systems belong together and they should be connected more or less later in the control engineering curricula. However, it is difficult to find this connection in the standard textbooks, where the real-time implementation is almost always ignored. For example, a very good introduction into computer controlled systems can be found in [1], but no orientation to the real-time software is given there. Mechanisation of control algorithms are given e.g. in [9]. In [8], hardware and software for digital control systems are described shortly. On the other hand, in [3] the real-time system design is treated from the optic of control engineering without to consider implementation aspects.

In general, real-time issues are gradually becoming "transparent" to the control engineering student. This transparency has been considerably increased in the last years with the advent of software tools like Matlab/Simulink ([24]) with its RTW (Real Time Workshop), the RTWT (Real Time Windows Target) and products from other companies like WinCon from Quanser ([25]) and ECP Executive from ECP Systems ([26]). They certainly do the implementation of real-time experiments easier and save much time, but on the other hand they put more distance regarding to the real-life problems, which can emerge during the real-time implementation of control systems. Hence, control concepts become today easier to be exemplified, but control engineering students can lose the real dimension about designing real-time control systems, particularly when they have to deal with time-critical applications.

This paper attempts to give an introduction to the implementation of real-time control systems, where characteristics of real-time systems and digital control issues are taking together into account.

The outline of the paper is as follow. Digital control aspects are introduced in Section 2. Definitions and characteristics of real-time systems are described in Section 3. Here, some common mistakes and misconceptions by developing real-time control software are also discussed. Section 4 treats the implementation of real-time controllers and Section 5 summarizes some important specifications for the implementation of real-time control systems as well as some well-known commercial products. Finally, conclusions are drawn in Section 6.

2 Computer controlled systems

The introduction of digital computers in the control loop has allowed developing more flexible control systems including higher-level functions and advanced algorithms. Furthermore, most current complex control systems could not be implemented without the application of digital hardware. However, the simple sequence sensing–control–actuation for the classical feedback control becomes more complex as well. Nowadays, this sequence can be supplemented as follow: sensing–data acquisition–control law calculation–actuation–data base update. Figure 1 shows an overview of such control systems.

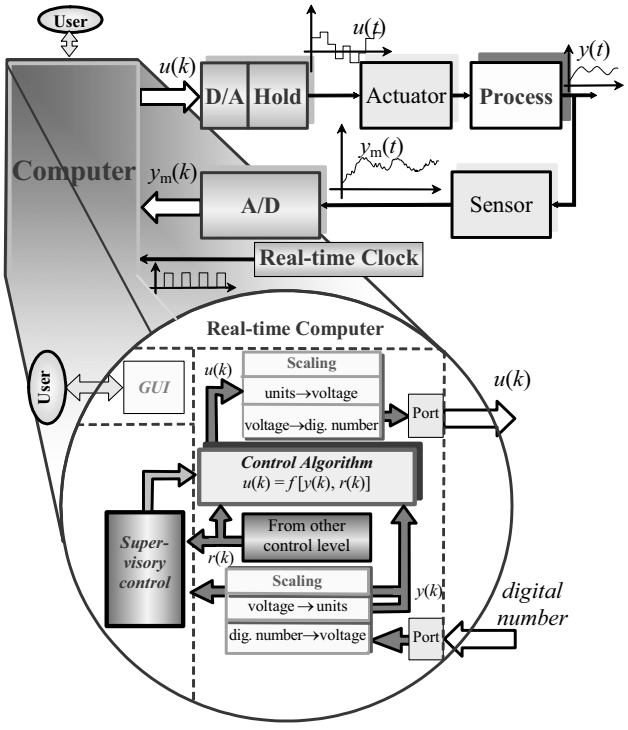

Figure 1. Overview of a computer controlled system

Thus, the control system now contains not only wired components but also algorithms, which must be programmed, *i.e.* software is now included in the control loop. This leads to new aspects to take into account by designing control systems:

1. Errors due to A/D and D/A conversion as well as due to limited length word calculations. This subject is well treated in the literature (see for example [9]).
2. Software developing is prone to errors. Thus, a new concept has to be introduced to consider this aspect, the *verification*, *i.e.* a mechanism to test if the software is doing exactly what it is expected. Here it is necessary to remark that in general a high percent of errors in digital control systems are caused by programming mistakes. Hence, digital control projects need not only control engineers but also engineers with skills in software engineering and computer programming.
3. Standard textbooks on digital control systems normally assume that sampling is uniform, periodic and synchronous. This leads to a case of "*zero-time-execution*" for the control law. However, that is not realistic since the control algorithm also consumes some time producing a control or feedback delay (control or feedback latency), *i.e.* a delay between a sampling instant and the instant at which a control-signal value is applied to the actuator.

   If the controller design is based on a model and the delay is constant and known, it could be helpful to use a tool for the description of the inter-sample behaviour (*e.g.* modified z-transform) in order to obtain a discrete-time model more approximated to the real case.
4. The computational time of control algorithms can change from one sampling instant to other (*e.g.* hybrid controller with controller switching mechanism, event based controllers, adaptive controllers with on-line parameter update, etc.). This variation in the delay is called *control jitter* (according to the IEEE, jitter is "the time-related abrupt, spurious variation in the duration of any specified related interval"). Moreover, value calculation for the control signal is usually carried out using multitasking (Subsection 3.2) defining a set of control tasks with respective priorities. Thus, a task can be pre-empted by higher priority tasks. In general, it can be said that the control system is also affected by several kind of jitter depending on context: sampling jitter, control latency jitter, input jitter, output jitter, etc.

Finally, real-time issues are often ignored in the implementation of digital control systems. This is in part a consequence of erroneous definitions and false interpretations. Popular misconceptions from the control engineering community about real-time systems are for example:

- *The computer was connected to the plant by mean of A/D and D/A converters in order to obtain the real-time system.* Analog plants should be connected to the computer through A/D and D/A converters. This link with the "*real world*" does not lead to a real-time system. On the other hand, it is possible to find real-time systems in complete digital contexts.
- *Our plant is so slow that real time is actually no problem.* A slow control system, which does not need a fast computer, can require critical time constraints. It is also possible that a control system does no need any hard real-time requirements but it is not necessarily a consequence of the slow plant.
- *It is not meaningful to talk about guarantying real-time performance.* It is true that occasionally time constraints can be relaxed without introducing additional problems in the control loop. This particularly applies to nice designed laboratory experiments. However, this actually depends on the application, and the time criticality should be proved for each individual case. On the other hand, real-time performance cannot be 100% guaranteed while hardware and software failures cannot be avoided at all.
- *We do not care about real time in our digital control system and even though it works*. This statement is similar to the previous one. The problem here is that you are not able to know when your system can fail.
- *Real-time programming is assembly coding, priority interrupt programming and device driver writing*. It is true that some code is still writing in assembler. However, high programming languages like C, Ada 95, Modula 2 and Real-time Java are normally used to develop real-time software. Device driver programming is necessary for real-time as well as non-real-time systems but they should be provided by the operating system or by the device manufacturer. Interrupt programming should be in principle avoided as much as possible. This point is treated in Subsection 3.5.

In the next Sections, an overview about real-time systems and control systems will be given in order to clarify real-time programming and its most important application.

3 Real-time systems: a short introduction

Real-time computing is a vast field and therefore, a complete discussion about that is outside the scope of this paper. Therefore, only the most relevant aspects will be treated here.

3.1 Definitions and general aspects

It is possible to find in the literature several definitions for real-time systems. Here, a definition that does not contradict the definition given in the IEEE POSIX Standard (Portable Operation System Interface for Computer Environments) will be assumed

> *A real-time system is one in which the correctness of a result not only depends on the logical correctness of the calculation but also upon the time at which the result is made available.*

This definition emphasizes the notion that time is one of the most important entities of the system, and there are *timing constraints* associated with systems tasks. Such tasks have normally to control or react to events that take places in the outside world, which are happening in "real time". Thus, a real-time task must be able to keep up with external events, with which it is concerned.

It should be noted here that *real-time computing* is not equivalent to *fast computing*. Fast computing aims at getting the results as quickly as possible, while real-time computing aims at getting the results at a prescribed point of time within defined time tolerances. Thus, a *deadline* (for this point of time) can be associated with the task that has to satisfy this timing constraint specifying either its start or completion time.

If the task has to meet the deadline, because otherwise it will cause fatal errors or undesirable consequences, the task is called *hard real-time task*. On the contrary, if the meeting of the deadline is desirable but not mandatory, the task is said to be a *soft real-time task*. By extension, one speaks about hard/soft time-constraints as well as hard/soft deadlines.

3.2 Real-time operating systems (RTOS)

In order to implement multitasking real-time systems, two approaches can be used: The first one consists in programming by using concurrent real-time languages and the second one is to use a sequential language and a *real-time operating system* ([4]). There has been a long debate about advantages and drawback of both approaches, which will not be treated here. However, a very important point is that real-time systems and real-time operating systems are not equivalent concepts: A RTOS provides facilities, like multitasking (*i.e*. concurrency or potential *parallelism*), scheduling, intertask communication mechanism, etc., for implementing real-time systems.

Old operating systems are characterised by the fact that each task is a simple program running in its own memory space. In the last years, there has been a tendency to provide facilities for creating several tasks within the same program to have faster task switch, unrestricted access to shared memory and to simplify the communication and synchronization. Such tasks are commonly called *threads*. The most important disadvantage of using threads consists in that the memory is not protected between threads of the same program. Figure 2 illustrates the difference between multitasking and multithread systems.

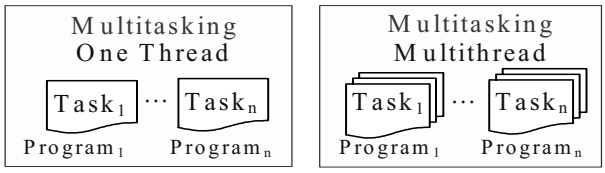

Figure 2. Multitasking and multithreading concepts

Together with parallelism, *determinism* is another important property of RTOS. A RTOS is *predictable* if the time necessary to acknowledge a request of an external event is know in advance. The end point of this predictability scale is called determinism, in sense that this time is exactly known in advance. This concept should not be confused with *responsiveness*, which is the time (after the acknowledgement) elapsed till the request is attended. Determinism and responsiveness make up the response time to external events. This is also called *system latency*.

Modern RTOS include in general the following features: fast switch context, small size, preemptive scheduling based on priorities, multitasking and multithreading, intertask communication and synchronisation mechanisms (semaphores, signals, events, shared memory, etc.), real-time timers, etc. However, RTOS are similar to standard operating systems from a structural point of view, since functional components as interrupt handler, task manager, memory manager, I/O subsystem and intertask communication are proper of both kind of operating systems.

3.3 Real-time scheduling

The distinctive part of a RTOS is the task manager. It is composed by the *Dispatcher* and the *Scheduler*. The Dispatcher carries out the *context switch*, *i.e.* the parameter saving for the outgoing task and the parameter loading for the incoming task, and the CPU handing over to the task that is becoming active.

The Scheduler has the function of selecting the task, which will obtain the processor as next. This choice is given by means of algorithms and this is the point where RTOS and non-RTOS are mostly distinguished. Real-time systems need special algorithms to schedule a set of tasks. This is a very active area of research in computer science and many algorithms have been proposed. In this paper, only the most important uniprocessor scheduling algorithms for real-time requirements will be presented. Fig. 3 presents an overview about some well-known scheduling algorithms (for details see *e.g*. [17], [12]).

Scheduling algorithms can be grouped in two classes: static and dynamic algorithms. A *static scheduling* requires that the complete information about the scheduling problem (number of tasks, deadlines, priorities, periods, etc.) is known *a priori*. Thus, the scheduling problem is solved before the schedule is executed. Such scheduler is also called *clairvoyant*. If at run time the feasibility can be determined and changes in the configuration may be carried out, then the *scheduling is said to be dynamic*.

Static schedules must always be planed *off-line*. Dynamic schedules can be planed either off-line if the complete scheduling problem is known *a priori* but with an on-line implementation, *i.e*. the configuration is changed at run time, or *on-line* if the future is unknown or ignored. Advantage of off-line scheduling is its determinism and the disadvantage its inflexibility. On the contrary, an on-line scheduling is very flexible but poor in determinism. Moreover, an on-line scheduling does not perform well if the system is overloaded. However, on-line scheduling is clearly the only option in a system whole future workload is unpredictable.

The guarantee that all deadlines are met can be taken as measure of the effectiveness of a real-time scheduling algorithm. If any deadline is not met, the system is said to be *overloaded*. Liu and Layland ([13]) showed that the *total processor utilization* for a set of n tasks given by

$$U = \sum_{i=1}^{n} \frac{C_i}{\min(D_i, T_i)} \qquad (1)$$

can be used as schedulability test. C is the execution time, D the deadline and T the task period. If the task is aperiodic or the deadline is smaller than the period, then the deadline is used in the equation. Figure 3 presents a classification for the most well-known dynamic scheduling algorithms for uniprocessor systems. In the following, the

most popular algorithms for scheduling tasks with real-time requirements will be shortly presented.

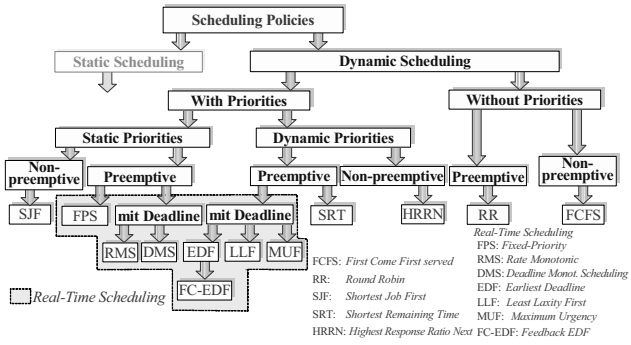

Figure 3. Classification of uniprocessor scheduling algorithms

Fixed-Priority Scheduling (***FPS***). In this approach, each task has a fixed static priority which is computed pre-run time. The runnable tasks are executed in the order determined by their priorities. If all tasks are periodic, a simple priority assignment can be done according to the statement: *the shorter the period, the higher the priority*. This approach is known as Rate Monotonic Scheduling (RMS) and it was proposed in [13]. The schedulability analysis for this algorithm presumes that all tasks are pre-emptive, periodic with deadlines equal to the period and independent (*i.e.* no task precedence between tasks exists). In this case the total utilization has an upper bound given by

$$U \le n(2^{1/n} - 1) \qquad (2)$$

This bound converges to 0.693 for $n \to \infty$, to 0.88 when the periods are uniform and to 1,00 only when the periods are harmonics of the smallest period. Under the conditions given above, it can be showed that the algorithm is optimal among fixed priority policies (*i.e.* given a set of tasks, RMS always produces a feasible schedule for this set, if any other algorithm can do that). This approach is easy to be implemented and if there are schedulability problems, the first task to fail is the task with the longest period, *i.e.* if the system becomes overloaded, deadlines are missed predictably. The most important drawbacks are its low utilization (under 70%), the fixed priorities, which can lead to starvation and deadlocks, and the fact that all deadlines should be equal to the periods.

In order to get out of the last problem, the Deadline Monotonic Scheduling (DMS) was proposed in [11]. They generalized the RMS allowing deadlines less than periods, where the fixed priority of a task is inversely proportional to its deadline.

Deadline-based Dynamic Scheduling. There are some dynamic scheduling algorithms which are based on assigning priorities according to their deadline. The simplest algorithm in this class is the *Earliest Deadline First* algorithm (EDF), where the task with the earliest (shortest) deadline has the highest priority. Thus, the resulting priorities are naturally dynamic. This algorithm can be used for both dynamic and static scheduling. However, absolute deadline are normally computed at run time and hence the algorithm is presented as dynamic. This algorithm was also proposed in [13]. They also showed that if all task are periodic and preemptive, then the algorithm is optimal and its utilization is $U \le 1$. A disadvantage of this algorithm is that the execution time is not taken into account in the priority assignment.

Another algorithm that is also optimal for scheduling preemtive tasks on one-processor system is the *Maximum Laxity First* (MLF) algorithm ([6], also called Least Slack-time First, (LSF) or Least Laxity First (LLF) algorithm). At any time, the laxity (or slack) of a task with deadline is equal to

$$Laxity = deadline - remaining \text{ execution time} . \qquad (3)$$

The MLF algorithm assigns priorities based on their laxities: the *smaller the laxity, the higher the priority*. This algorithm requires the knowledge of the current execution time, and then laxity is essentially a measure of the flexibility available for scheduling a task. Thus, MLF takes into consideration the execution time of a task, and this is it advantage in front of EDF. On the other hand, the execution time is normally not known until the task complete and therefore estimation is necessary. Because this estimation is used to schedule the set of task, the resulting schedule can be incorrect. This is its most serious disadvantage.

The MLF algorithm also has a schedulable bound of 100% for all task sets. A problem with EDF as well as MLF is that there is no way to predict which tasks will fail in transient overboard situations. This has led to another algorithm called Maximum Urgency First (MUF) algorithm ([19]), where an explicit description of urgency is assigned to each task. This urgency is defined as a combination of two fixed priorities, and a dynamic priority, which is inversely proportional to the task laxity. One of the fixed priorities, called *task criticality* has precedence over the dynamic priority. The other fixed priority, called *user priority*, has lower precedence than the dynamic priority. The criticality helps on-line algorithms to distinguish more important from less important tasks. Finally, it is necessary to remark that all mentioned dynamic algorithms do not remain optimal if pre-emption is not allowed or the system has multiple processors.

If sporadic or aperiodic tasks must be scheduled, two algorithms can be used: the Deferrable Server Algorithm (DSA) and the Sporadic Server Algorithm (SSA). However, only SSA conforms to the RMS schedulability analysis.

3.4 Intertask Communication and Synchronisation

The intertask communication can be carried out as for non-RTOS by using mailbox, pipes and shared memory.

Synchronization is very important in real-time systems for two reasons: (*i*) tasks may experience unpredictable delays due to blocking on shared resources to which they require exclusive access (*e.g.* A/D and D/A converters), and (*ii*) some tasks should be executed depending on results of other tasks.

It can be shown that the addition of mutexes in real-time programs makes the general scheduling problem a non-predictable one ([14]). To solve this problem with an EDF algorithm *kernelized monitor protocol* can be used and *priority ceiling protocol* if the scheduling algorithm is RMS.

3.5 Common programming mistakes

By programming real-time system, many typical mistakes are often founded ([18]). Some of them are:

- *Large or many if-then-else and/or case statements*. These statements introduce in the code many different paths with varying length so that the code will also have different execution time. This becomes more significantly when the path is very long. Variable functions, state machines, lookup tables should be used instead the mentioned statements.
- *Delays implemented as empty/dummy loops*. This leads to inaccurately delays depending on the hardware. RTOS timing mechanisms should be used here for implementing exactly time delays.
- *Indiscriminate use of Interrupts*. Interrupts affect seriously the real-time predictability: They cannot be scheduled, and they have always very high priorities. Programs based on interrupt are very difficult to debug and to analyse. Moreover, they operate in kernel context.

  There is a very popular myth, which says that interrupts save CPU time and they guarantee the execution start of a task. This can be true in small and simple microprocessor based systems. However, it is not the case for complex real-time system, where non-preemptive periodic tasks can provide similar latency with better predictability and CPU utilization.

  Periodic polling threads should be used if it is possible. Interrupt services routines should be programmed in such a form that its only function is to signal an aperiodic server.
- *Configuration information fixed using #define or similar statements*. Programmers frequently use #define statements in their code to specify register addresses, limits for arrays, and configuration constants. Although this practice is common, it is undesirable because it prevents on-the-fly software patches for emergency situations, and it increases the difficulty of reusing the software in other applications. Changes in the configuration require that the entire application has to be recompiled.
- *Implementation based on a big single loop.* One big loop leads to the fact that the complete software executes to the same rate. In order to assign different and proper rates concurrent techniques for pre-emptive RTOS should be used.
- *Use of message passing as primary intertask communication mechanism.* Message passing reduces real-time schedulability bound and it produces significant overhead leading to many aperiodic servers instead of periodic tasks. Moreover, deadlocks can appear in closed loops systems. Shared memory and proper synchronization mechanisms to prevent deadlocks and priority inversion should be used.
- *To think that problems can be fixed magically.* Programming errors, which become seldom visible in the debugging phase, can appear exactly at the time when the application is running and it is not possible to correct the mistake. It is necessary to find all mistake causes before the software is released.
- *Do not analyse memory during the design.* The amount of memory in most real-time systems is limited. Frequently, programmers have no idea about how much memory a certain program or data structure uses. Moreover, they are normally wrong by an order of magnitude. A memory analysis is quite simple with most of today's development environments.
- *Design without execution-time measurement.* It is very common to assume that the program is short enough and the available time is sufficient. However, measuring of execution time should be part of the standard testing in order to avoid surprises. Hence, the system should be designed so that the code is measurable all time.

4 Computer implementation of control systems

Building a real-time control system requires two stages in general: controller design and digital implementation. At controller design stage, normally a control performance index is defined (because optimal control is normally preferred to specify control performance requirements) and a controller is designed which optimise this index while maintaining stability and rejecting disturbances. SISO-controllers in an imput/output approach, *e.g*. PID (Proportional Integral Derivative), GMV (Generalized Minimum Variance) GPC (Generalized Predictive Controller), pole placement, etc., can be represented by the general equation

$$P(q^{-1})\,u(k) = T(q^{-1})\,r(k) - Q(q^{-1})\,y(k) \qquad (1)$$

where P, T, and Q are polynomials, u is the control signal, r the reference signal and y the plant output. (Notice that for the PID controller $T(q^{-1}) = Q(q^{-1})$). State Space controller with observer are given in general by

$$\mathbf{u}(k) = \mathbf{K}_r\,\mathbf{r}(k) - \mathbf{K}_x\,\hat{\mathbf{x}}(k) \qquad (2)$$

$$\hat{\mathbf{x}}(k+1) = [\mathbf{A} - \mathbf{K}_o\mathbf{C}]\,\hat{\mathbf{x}}(k) + [\mathbf{B} - \mathbf{K}_o\mathbf{D}]\,\mathbf{u}(k) + \mathbf{K}_o\mathbf{y}(k) \qquad (3)$$

where $\mathbf{K}_o$ is the observer gain. The controller is executed cyclically according to the sampling time, whose value is assumed to be correctly chosen, *i.e.* satisfying not only the condition given by the Shannon's sampling theorem but also achieving the desired performance (the control design is normally based on a time-discrete model which also depend on the sampling time). In order to satisfy the "zero-execution time" requirement, it is desired that the new value for the control signal be delivered as soon as possible.

At implementation stage, multiple control tasks should be scheduled to run on microprocessors or microcontrollers. All tasks should be scheduled with limited available computing resources. The chosen sampling time should take into account the limited computation time provided by the hardware. Thus, the computation time delay (control latency) τ is always in conflict with the sampling time T_o. Depending on the magnitude of τ relative to T_o this conflict can be classified into either a delay ($0 < \tau < T_o$) or loss ($T_o \geq \tau$) problem. Since the control latency is usually affected by control jitter, delay and loss can occur alternately in the same system at different times. The loss of the control signal $u(k)$ is equivalent, to the case when the controller computer fails to update its output during any one sampling interval and $u(k\text{-}1)$ is applied again. Because this could occur randomly at any time, the failure to deliver a control

signal can be treated as a correlated random disturbance $\Delta u(k)$ at the input of the plant.

The interaction between control performance and task scheduling has been investigated in [16]. Research results led to the conclusion that separated design produces suboptimal performance. Hence, the digital control system design has to be revisited in order to introduce considerations about real-time computing. In [10] and [16], the task attribute assignment with respect to control performance was focused on task period selection for a single task model of the controller like the example shown in Figure 4 (Matlab syntaxes is used for simplicity).

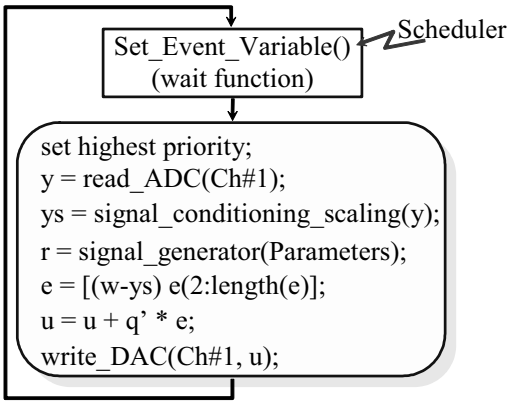

Figure 4. Controller implemented with one real-time periodic task

Here, in order for delivering the control signal $u(k)$ as soon as possible one-step-ahead predictive controller can be used in the form

$$u(k+1) = f[r(k+1), r(k), \cdots, r(k-n), u(k), \cdots, u(k-m), \hat{y}(k+1), y(k), \cdots, y(k-n)] \quad (4)$$

where $r(k+1)$ is assumed to be known and $\hat{y}(k+1)$ is calculated by a simple linear predictor given by

$$\frac{\hat{y}(k+1)-y(k)}{(k+1)-k} = \frac{y(k)-y(k-1)}{k-(k-1)} \quad (5)$$

$$\hat{y}(k+1) = 2y(k) - y(k-1) \quad (6)$$

Hence, the control task started first delivering $u(k)$ (which was calculated at time k-1), after this the remaining operations are carried out, *i.e.* read $y(k)$ from A/D converter, calculating $\hat{y}(k+1)$ and then $u(k+1)$. The disadvantage of this approach is that the control signal is calculated based on a prediction. This prediction can be however improved by using a model of the plant but in this case, more execution time is necessary.

A second approach was proposed by [5]. This is based on the implementation of two periodic real-time tasks. The first one calculates the control signal directly after reading and conditioning $y(k)$ and the second one updates the states after the control signal value was delivered. For the I/O representation, eq. (1) can be implemented by using a realization in the form of a ladder structure given by

$$\left[\begin{array}{c} \mathbf{x}_r(k+1) \\ \hline \mathbf{x}_y(k+1) \end{array}\right] = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{A} \end{array}\right] \left[\begin{array}{c} \mathbf{x}_r(k) \\ \hline \mathbf{x}_y(k) \end{array}\right] + \left[\begin{array}{c|c} \mathbf{B}_r & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{B}_y \end{array}\right] \left[\begin{array}{c} r(k) \\ \hline y(k) \end{array}\right] \quad (7)$$

$$u(k) = \left[\begin{array}{c|c} \mathbf{C}_r & -\mathbf{C}_y \end{array}\right] \left[\begin{array}{c} \mathbf{x}_r(k) \\ \hline \mathbf{x}_y(k) \end{array}\right] + \left[\begin{array}{c|c} d_r & \mathbf{0} \\ \hline \mathbf{0} & -d_y \end{array}\right] \left[\begin{array}{c} r(k) \\ \hline y(k) \end{array}\right] \quad (8)$$

where the matrices $\mathbf{A}$, $\mathbf{B}_y$, $\mathbf{B}_r$, $\mathbf{C}_y$, $\mathbf{C}_r$, d_y and d_u are obtained from some realization of

$$u_r(z) = \frac{T(z^{-1})}{P(z^{-1})} r(z) \text{ and } u_y(z) = \frac{Q(z^{-1})}{P(z^{-1})} y(z)$$

and contains the controller parameters. Figure 5 illustrates a possible implementation of this idea.

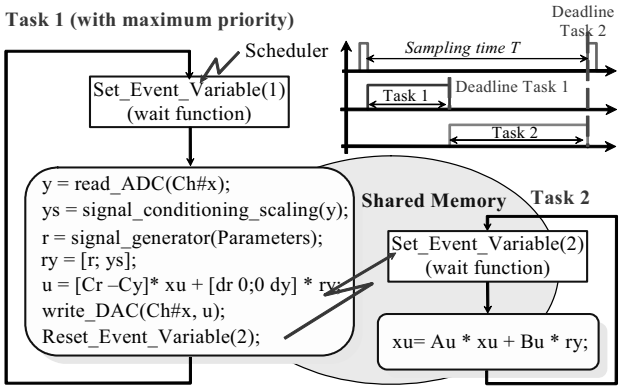

Figure 5. I/O Controller implemented with two real-time tasks

Space-state controllers naturally fit this task model (Figure 6). All these approaches are based on optimal control design. In [15], the control performance is specified by rise time, maximum overshoot, settling time and steady state error. The task scheduling was carried out by a heuristic approach.

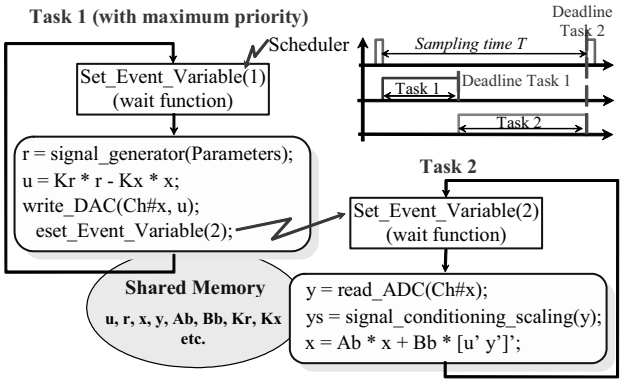

Figure 6. Real-time implementation of a state-space controller

A supervisor can be implemented as an independent task. A possible scheme is shown in Figure 7.

4.1 Some common mistakes in the implementation of real-time control systems

In the laboratory, it can be frequently observed that control engineering students commit some of following mistakes:

- *Overlook the anti-aliasing filter*. Anti-aliasing filter is necessary to bind the highest signal frequency in order to determine correctly the sampling time. The filter can be dispensed with if the highest frequency of the signal is known and the sampler can be set accordingly.
- *Implement the anti-aliasing filter in software (as digital filter) after the sampling*. This is a typical error committed by students. Because the filter is used to avoid

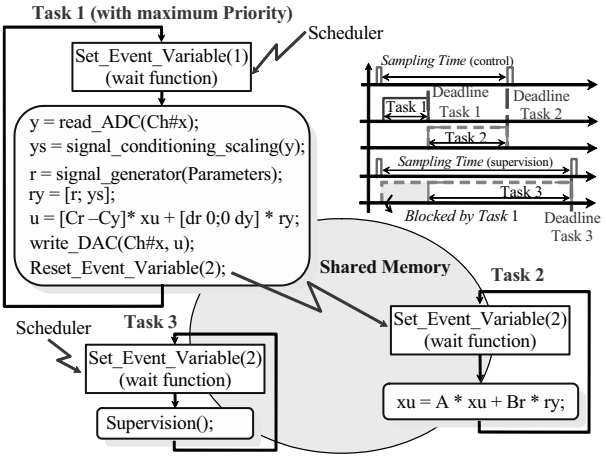

Figure 7. Real-time implementation of a state-space controller

aliasing at the sampling stage, the filter must be situated before the sampling and it has to be analogue.

- *Overlook the signal scaling*. Controllers are normally designed by using a model that is parameterized according to physical or normalized units. Thus, sampled signal have also to be converted to the corresponding units, before they can be used for the control signal calculation.
- *Unnecessary implementation of continuous-time controllers*. Students frequently design a continuous-time controller in Simulink and then they transfer the algorithm to real-time target system using the Real Time Workshop. The consequence is that the controller is implemented using a fixed-step solver for differential equations (normally fourth-order Runge-Kutta) with a sampling time equal to the integration step. Thus, the real-time task suffers an unnecessary overload. If it is possible, discrete-time controllers should be implemented.

  Here it is important to remark that sometimes discrete-time control systems perform poorer than continuous-time ones and increasing the sampling rate does not always lead to a performance improvement. Moreover, some problems are caused by the properties of sampling zeros of pulse transfer functions at high sampling rates. In these cases, continuous-time control should be applied and the above advice could be incorrect. This clarifies the sense of the expression "*if it is possible*".

5 Real-time platform

Nowadays it is very difficult to choose a software/hardware configuration for real-time experiments because there are many manufacturers that offer a variety of well designed systems. Thus, it is necessary to be careful at the moment to define the specifications for such systems.

Today it is very common to use two computers in a host/target configuration to implement real-time control systems. The host is a computer without real-time requirements, in which the develop environment, data visualization and control panel in the form of a Graphic User Interface (GUI) reside. The real-time system run on the target, which can be a second computer or an embedded systems based on a board with a DSP (Digital Signal Processor), a PowerPc or a Pentium family processor.

This separation is not necessary for small systems, since hard real-time PC operating systems such QNX ([7]), LynxOS and RT-Linux have solved the problem of deterministic response of real-time tasks, which coexist together with non-real-time tasks on the same computer. However, if the project has some spread then the host/target architecture brings more flexibility, order and computational power. An additional advantage is that the real-time system will continue working still in the case that the host crashes, increasing the reliability of the system. Requirements for a real-time system could be:

- *Preemptive Multitasking for hard real-time requirements*. Multitasking and multithreading are necessary if *e.g.* there are many independent control loops. It is also necessary to implement supervisory control as well as adaptive control.

  Laboratory plants do not usually need hard real-time response to obtain an acceptable performance for a simple control loop. However, this property becomes essential if the project includes research in the area of hybrid dependable systems or the algorithms should be tested for time-critical applications.
- *POSIX compliance*. Posix (*Portable Operating System Interface*) is an IEEE standard for operating systems. Norms 1003.1b, 1003.1d, 1003.1j specify requirements and compatibilities for real-time systems. Posix compliant software is easy to be ported.
- *Support for real-time scheduling*. Several algorithms are available for this task, *e.g.* RMS, EDF, MLF and MUF.
- Small latency such that sampling times in the area of 1 *ms* should be possible for several control loops.
- *Integration with Labview*. Labview is a graphical programming environment from National Instrument Inc. that combines development with a powerful programming language allowing 2-D and 3-D data presentation and visualization.
- The use of Matlab/Simulink/RTW should not be the exclusive tool to implement real-time software but an additional facility. Therefore, a real-time operating system with a develop environment to write software is also needed.

The above stated requirements are very hart if the acquisition of ready-to-use products is wanted. For example, although EDF, MLF and MUF are well-known scheduling algorithms for real-time requisites, they are hardly to find in commercial real-time operating systems. MLF and MUF are not implemented at all and EDF can be found in JBed ([20]) and RT-Linux ([2]). However, JBed is not Posix compliant, the integration with Matlab/Simulink and LabView is not available and only few CPUs and interface cards are supported. RT-Linux (or RTAI) could be a good choice if software drivers are available for the corresponding acquisition boards and the developers have enough experience installing RT-Linux because this activity is very cumbersome. A limited interface with Matlab/Simulink/RTW and Labview is available.

The first software specification (multitasking/multithreading) excludes products like RTWT ([24]), WinCom ([25]) and

real-time systems based on DSP board, since they are very limited in this aspect. Most WindowsNT-based real-time systems are inadequate for system with hard deadlines and products like InTime (from RadiSys Co.) and Hyperkernel (from Nematron Co.) do not support Matlab/Simulink and Labview. The same is valid for LynxOS ([21]).

6 Conclusions

In this contribution, an introduction to real-time digital control from an educational point of view has been given. Some well-known misconceptions coming from the control system community were clarified and common mistakes in the programming and in the real-time control implementtation have been highlighted. The relevance of the real-time implementation of the control system, particularly in case of time-critical application, can also be taken from the paper.

Finally, the problem to find an adequate commercial real-time operating system was pointed out by summarizing the experience collected in this field.

References

[1] Åström, K. and B. Wittenmark. *Computer Controlled Systems*. Prentice Hall International, 1997.

[2] Barabanov, M., (1997). *A Linux-based Real-Time Operating System.* M.S. Thesis at New Mexico Institute of Mining and Technology.

[3] Bennet, S. *Real-time Computer Control an Introduction*. Prentice Hall International, 1994.

[4] Burns, A. and A. Wellings. *Real-time Systems and Programming Languages*. Addison Wesley, 2001.

[5] Cervin, A. *Improved Scheduling of Control Tasks*, Proc. 11th Euromicro conference on real-time systems, 1999.

[6] Dertouzos, M. L. and A. K. Mok. *Multiprocessor on_line scheduling of hard_real_time tasks*. IEEE Transactions on Software Engineering, Vol. 15, no. 12, 1497-1506, 1989.

[7] Hildebrand, D., (1992). *An architectural overview of QNX*. In USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp. 113-126, Seattle, WA, April 1992. USENIX.

[8] Houpis, C. H. and G. B. Lamot. *Digital Control Systems*. McGraw-Hill, 1992.

[9] Katz, P. *Digital Control using Microprocessors*. Prentice Hall International, 1981.

[10] Kim, B. K. *Task Scheduling with Feedback Latency for Real-Time Control Systems*. Proc. IEEE 5th Int. Conf. on Real Time Computing Systems and Applications, 37-41, 1998.

[11] Leung, J.-T. and J. Whitehead. *On the complexity of fixed priority scheduling of periodic real-time tasks*. Performance Evaluation, 2 (4), 237–250, December 1982.

[12] Liu, J. W. S. *Real-time Systems*. Prentice Hall, 2000.

[13] Liu, C. L., and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment*. Journal of the Association for Computing Machinery, Vol. 20, no. 1, pp. 44-61, 1973.

[14] Mok, A. K. *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment*. PhD thesis. M.I.T., 1983.

[15] Ryu M., S. Hong, M. Saksena. *Streamlining Real-Time Controller Design: From Performance Specifications to End-to-end Timing Constraints*. Proc. IEEE 3rd Real Time Technology and Application Symposium, p. 1-99, 1997.

[16] Seto D., J. P. Lheoczk y, L. Sha, and K. G. Shin. *On Task Schedulability in Real-Time Control Systems*. Proc. 17th Real-Time Systems Symposium, 13-21, 1996.

[17] Stallings, W. *Operating Systems*. Prentice Hall, 2001.

[18] Stewart D. B. *30 Pitfalls for Real-Time Software Developers*. Embedded Systems Programming. Parts 1 and 2, vol. 12, no. 11, 32-41; no. 12, 74-86, 1999.

[19] Stewart D. B. and P. K. Khosla. *Real-Time Scheduling of Dynamically Reconfigurable Systems*. Proc. IEEE International Conference on Systems Engineering, Dayton Ohio, 139-142, 1991.

[20] Tryggvesson J., T. Mattsson and H. Heeb, (1999). *JBED*: *Java for Real-Time Systems*. Dr. Dobb's Journal, 1999.

[21] Wind River Systems, Inc., 1010 Atlantic Avenue, Alameda, CA 94501-1147, USA. VxWorks Programmer's Guide, 1993.

[22] Wittenmark B., J. Nilsson, M. Törngren. *Timing Problems in Real-Time Control Systems*. Proc. of American Control Conference, 1995.

[23] RTLinux. *The Realtime Linux*, www.rtlinux.org.

[24] The MathWorks, 3 Apple Hill Drive, Natick, MA 01760- 2098, www.mathworks.com

[25] Quanser Consulting, 102 George Street, Hamilton, Ontario, Canada L8P 1E2, www.quanser.com

[26] ECP Educational Control Products. 5725 Ostin Avenue Woodland Hills, CA 91367 USA, www.ecpsys.com