An Open Source Hands-on Course with Real-Time Linux

Herman Bruyninckx, Pieter De Troyer, Klaas Gadeyne
Katholieke Universiteit Leuven, Dept. Mechanical Engineering
Celestijnenlaan 300B, B-3001 Leuven, Belgium
herman.bruyninckx@mech.kuleuven.ac.be

Abstract

This paper describes the documentation that comes with our hands-on course on real-time systems, as we present it to fourth-year mechatronics engineering students. This hands-on course is one part of a cluster of three, the others being more theoretical courses on real-time software, and on processors and interfacing for real-time systems.

The hands-on course is using real-time Linux (RTLinux and RTAI), and its examples focus on mechatronic systems for mechanical engineers, i.e., mostly motion control applications. The course was first given in Dutch, but its documentation and code examples are now reworked and extended for this year's course into an English version, which will be available under the GNU Free Documentation License (FDL) license [3]. The aim is to serve as a starting point for cooperation in the field of educational material for real-time courses for beginning engineering students, because there is a big need to complement the excellent real-time operating system projects with "higher-level" documentation and guidance.

1 Introduction

Both RTLinux [11] and RTAI [6] have become mature real-time operating systems, which are ready for serious industrial use. They both also have succeeded in building up an active community of users and co-developers (e.g., the Comedi library for DAQ-like device drivers [7]). However, offering real-time *operating system primitives* of good quality is not sufficient to reach real-time *application programs* of good quality: one needs documentation and course material of comparable scope and quality. And currently, the lack of both documentation and courses in the open source real-time world is apparent.
The goal of this paper is to stimulate the creation of an open source initiative to fill this gap. As with development of code, also documentation and course projects need an initial design and implementation, that others can then comment on and extend, in order to eventually reach a decent level of contents and scope. The authors believe that their course notes on real-time systems could be such a starting point.
This course is given to fourth-year mechatronics engineering students at the K.U.Leuven, Belgium, and consists of three parts:

- A theoretical course on real-time software.
- A theoretical course on processor architectures.
- A set of hands-on sessions.

For the hands-on sessions, an extensive documentation is currently being developed, and accessible on-line on `http://www.mech.kuleuven.ac.be/~bruyninc/rthowto/`. This course has been given last year, in a shorter, Dutch version, and is currently being extended and translated into English.
The following Sections explain the hands-on course (Section 2), our vision on how to further develop the contents of the accompanying documentation (Section 3), and how we see an open source project being created around it (Section 4).

2 Hands-on

The public of the course are mechatronic engineering students. This means:

- they have only a very basic knowledge about computing (and certainly about operating systems) and about electronic interfacing.
- their field of interest is mainly in mechatronics, such as machine control and robotics.

In many ways, this is a perfect public to try out general documentation and course notes, because almost everything is new to them. (More advanced courses

could be derived from the general one, by a combination of selection and addition, see Section 4).
The hands-on sections have the following purpose:

- To illustrate the two theoretical courses. It has been our experience, however, that students are not always motivated to prepare the hands-on sessions by first going through the theoretical classroom notes; therefore, we decided to make the notes of the hands-on sessions as self-contained as possible. (Another reason for this decision is that the theoretical courses don't publish their material in open source form.)

- To demonstrate and explain the software behind typical mechatronic systems. We've built a couple of low degrees-of-freedom motion control setups, that illustrate the following:

  compiling, linking and loading. Students have almost no experience and understanding of these basic things...

  threads: scheduling, periodic timing, static priorities.

  interrupts: an ISR shapes the pulses for a stepper motor; another ISR detects when a probe comes in contact with the environment; still another ISR is responsible for filling in the initial encoder value during a homing sequence.

  interprocess communication (IPC): the motion controller consists of: (i) the *command interpreter* thread, that receives commands from the user, and distributes the appropriate information to the other threads; (ii) the *trajectory generator* thread, that processes high-level motion commands into setpoints for the servo thread; (iii) the *servo thread*, that takes these setpoints as inputs to its control algorithm (a simple proportional control in the hands-on); and (iv) the *diagnosis* thread, that returns status information about the whole motion control system. The trajectory generator and the servo thread communicate with a FIFO (i.e., the *producer-consumer* pattern).

  device drivers: for a parallel port, incremental encoder, DA and AD conversion, and a force sensor.

3 Documentation

The documentation that comes with the abovementioned hands-on course consists of **two major parts**. We don't give full details of the contents, because these things are quite well known in the real-time Linux community, and the full course notes are readily accessible on-line, [2].

- Part 1. Introduction to the *concepts* of a real-time operating system.

- Part 2. Discussion about how to *design* a real-time application.

3.1 Part 1: Concepts

The contents of the first Part are:

Real-time and embedded operating systems. A brief overview of the whole first Part, i.e., the responsibilities of an operating system (general-purpose and real-time); trade-offs that designers of an OS have to make; the importance of the concept of "time" in an RTOS; a discussion on the appropriateness of standard Linux for real-time and embedded applications; operating system standards.

Task management and scheduling. With the following sections: POSIX thread management; scheduling; priority-based scheduling; the Linux scheduler; real-time scheduling for Linux.

Interrupts, with emphasis on the need for, and the difference between, an Interrupt Service Routine and its Deferred Service Routine(s).

Interprocess communication (IPC), subdivided into *synchronization* (i.e., events and locks) and *data exchange* (i.e., FIFOs, messages, buffers, etc.).

Memory management.

Real-time device drivers, with a discussion on *Comedi*, and the *rt-net* and *rtcom* projects.

Linux variants for real-time and embedded, with discussions on RTLinux, RTAI, miniRTL, AtomicRTAI, uCLinux and Etlinux.

Non-Linux RTOSs, with discussion on eCos, RTEMS, CubeOS, FIASCO and DROPS, Real-time micro-kernel, and KISS Realtime Kernel.

Compilation, tracing and debugging, with, for example, gdb and the Linux Trace Toolkit.

As can already been seen from this contents description, the documentation tries to cover the whole range of open source software for real-time programming (and embedded programming, although this part of the documentation is much less mature still), and doesn't want to concentrate on RTLinux or RTAI alone. The authors think there is a definite need for this "larger-scale" view on the open source evolutions in the real-time domain.

3.2 Part 2: Design

The second Part wants to guide the students in their task to *design* the software for a real-time project they have to work on. Our experience is that design is by far the most difficult part of the course, because: (i) there exists very little literature on what to do when designing software for a real-time application, (ii) the standard theoretical course on real-time software doesn't spend much time on design because it has its hands more than full on explaining the RTOS primitives, and (iii) design is inherently difficult, especially for non-experienced users. So, we go into a bit more detail in presenting the contents of this Part 2:

Decoupling structure and functionality. This chapter speaks about the basic issues in design: components, modules, architectures, coupling, etc. Students should understand the importance of separating the *structure* of an application from its *functionality*. The functionality is the set of all algorithms needed to perform the purpose of the application; the structure is the way in which the algorithms are distributed over tasks (threads), and how these tasks are synchronized. Most (mature) application domains have a relatively fixed structure of cooperating tasks that has grown over the years by experimenting in the domain, while the functionality of some of the tasks tends to change more quickly (adding features, trying out alternative functionality, etc.) Examples of such mature domains are: telecom, motion control, networking, data bases, ...

A lot of the real-time developments are done in small, isolated groups, where often the majority of developers are not computer scientists but specialists in the application domain; and new students in the field are concentrating more on understanding the real-time primitives than on learning to design software applications. This often leads to "spaghetti code" or abuse of the available real-time and IPC primitives.

Software patterns. A Software Pattern [4, 8] is a proven, non-obvious, and constructive solution to a common problem in a well-defined context, taking into account interactions and trade-offs ("forces") that each tend to drive the solution into different directions. A Pattern describes the interaction between a group of components, hence it is a higher-level abstraction than classes or objects. It's also not an implementation, but a textual description of a solution and its context. Patterns are very important while designing the *structure* of an application, because finding the relevant pattern(s) solves most of the structuring problem already.

The patterns discussed in the course are: the distributed components patterns (described already very well by Douglas Schmidt in ACE [9] and TAO [10]; and the patterns for low-level motion control.

Frameworks. A framework [5] is a set of computer code files that *implement* a reusable software solution for a particular problem domain. In that sense a framework is much broader ("programming in the large") than a software pattern ("programming in the small"). A framework typically contains several patterns, but a pattern doesn't contain frameworks; a framework contains code, a pattern doesn't. Software frameworks are constructed in such a way that similar applications within the same domain can benefit from the same structures and abstractions, but may need re-implementation of some parts of the framework.

The above-mentioned ACE [9] and TAO [10] open source projects are frameworks, based on a lot of Software Patterns. The authors are working on a similar framework for robot control, in the context of the Orocos project (*Open source Robot Control Software*, [1]).

Hints. This Chapter contains a list of *do's and dont's*, i.e., things to be aware of, or good examples of coding, in the context of real-time programming.

There doesn't exist a Part 3 yet, but we envisage it to consist of detailed code examples from different application domains. This should become an archive of high-quality modules, where real-time programmers come and shop for (or contribute new re-usable) components.

4 Collaboration project

Not all of the above-mentioned chapters and topics have already been worked out in full detail. That's the reason why the authors want to start an open source project around documentation and course material. The second reason is to improve the quality and the scope of what is already there.
We realize that the goals of a course or of documenation could be quite different for courses begin taught by different people in their own particular context: one could, for example, need a course for quite experienced C programmers, with interests in telecom applications. This flexibility of shaping the course according to different goals should be one of the design issues to be taken into account. Therefore, while at this moment there is only a need for examples in mechatronics, the parts of the course that treats general real-time concepts is "application-agnostic," in the sense that the concrete examples are separated from the theory, and are only combined at "*compile time*."
We think that the process of writing and maintaining documentation and course notes should use the same tools that have proven to work in the creation of the software they accompany. One could imagine using the well-known *configure; make; make install* cycle to produce a set of course notes with the appropriate selection of general theoretical discussions, with examples of code and design in the relevant application domain. Or to have a configuration tool as used for the building of a Linux kernel.
This means that the documentation should be written in more or less independent modules, separating theory from example code, and with the option to choose the examples from a pre-defined (but extensible) set of application domains (telecom, motion control, process control, data acquisition, ...). We need a meta-formalism for specifying constraints and dependencies between the different modules, such that the user is supported in constructing course notes that are consistent.
Currently, there is no such infrastructure implemented yet for the documentation project, and there is no clear design approach for it either. The authors sincerely welcome suggestions in this area.

5 Conclusions

This paper presented on-going work on the creation of an open source documentation project for real-time programming. The documentation is software project-agnostic (i.e., not targeted to, for example, only RTAI), and consists of three major parts: general theory on real-time software; support for the design of real-time applications; and an archive of re-usable components.
Only the first two parts have been (partially) implemented yet, so it is time to start up an international collaboration to make the initiative grow into a succesful open source project.

Acknowledgements

Herman Bruyninckx is Postdoctoral Fellow of the Fund for Scientific Research—Flanders (F.W.O.) in Belgium. This work was partly sponsored by the Belgian Programme on Interuniversity Attraction Poles (IUAP), and Concerted Research Action GOA/99/04.

References

[1] H. Bruyninckx. Open source robot control software. `http://www.orocos.org/`.

[2] H. Bruyninckx. Real-time and embedded howto. `http://www.mech.kuleuven.ac.be/~bruyninc/rthowto/`.

[3] Free Software Foundation. GNU Free Documentation Licence. `http://www.fsf.org/copyleft/fdl.html`.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[5] R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[6] P. Mantegazza. RTAI: the Real-Time Applications Interface. `http://server.aero.polimi.it/projects/rtai/`.

[7] D. Schleef. Comedi: Linux control and measurement device interface. `http://stm.lbl.gov/comedi/`.

[8] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture. Patterns for concurrent and networked objects*. Wiley, 2001.

[9] D. C. Schmidt. ACE, The Adaptive Communication Environment. `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[10] D. C. Schmidt. TAO, The Ace Orb. `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[11] V. Yodaiken and M. Barabanov. Real-time linux. `http://www.rtlinux.org`.