Designing an Embedded Hard Real-Time System: A Case Study

Matjaž Colnarič[1], C. T. Cheung[2] and Wolfgang A. Halang[3]

[1] University of Maribor, Slovenia, colnaric@uni-mb.si
[2] Hong Kong Polytechnic University, csronnie@comp.polyu.edu.hk
[3] FernUniversität Hagen, Germany, wolfgang.halang@fernuni-hagen.de

Abstract. In this paper a description of a consistent design of an embedded hard real-time control system is given. To provide for the overall predictability of tasks' temporal behaviour, which is the ultimate requirement in such systems, all influencing factors are taken into account in a holistic manner: system and hardware architecture, operating system issues, programming language and application design methodology. Based on the resulting guidelines, a consistent prototype was implemented.

1 Introduction

Instead of computer speed, which cannot guarantee that specified timing requirements will be met, almost a decade ago a different ultimate objective in designing consistent systems for embedded hard real-time applications was generally accepted: predictability of temporal behaviour. While, for the systems usually employed in process control, testing of conformance to functional specifications is well established, temporal circumstances, being an equally important design aspect, are seldom verified consistently. Almost never it is proven at design time that such a system will meet its temporal requirements in every situation that it may encounter.

Although the domain of real-time systems substantially gained interest in recent years, the results of fundamental research are not broadly used in hard real-time applications, yet. One of the reasons for this situation is that most studies consider selected topics only, and assume that other system constituents behave predictably. This assumption makes it very hard for the application designers to set up consistent systems from different components. For that reason we initiated a project systematically addressing all crucial layers to provide practically usable design guidelines and techniques for hard real-time control systems. A consistent prototype implementation of an embedded hard real-time system was designed and will be briefly sketched in this paper. For those who are interested in the details, thorough reference to our detailed publications on specific topics resulting from this project is given, where also comments on, and references to, related work of other research groups can be found.

2 Concept of the Experimental Hardware Platform

Using static scheduling algorithms, severe problems regarding the non-determinism of temporal task execution behaviour could be avoided. However, following the very nature of real-time applications, it is necessary to provide for dynamic task scheduling. The algorithms which guarantee that, once successfully scheduled, all tasks will meet their deadlines, are referred to as feasible. In the literature, several such algorithms have been reported. For our purpose, the earliest-deadline-first scheduling algorithm, which was shown to be feasible for scheduling tasks on single processor systems, was chosen and implemented in the kernel of an operating system [3].

For process control applications, where process interfaces are usually physically hard-wired to sensors and actuators establishing the contact to the environment, it is natural to implement either single processor systems or dedicated asymmetrical multiprocessors acting and being programmed as separate units. Thus, the earliest-deadline-first scheduling policy can be employed without causing any restrictions.

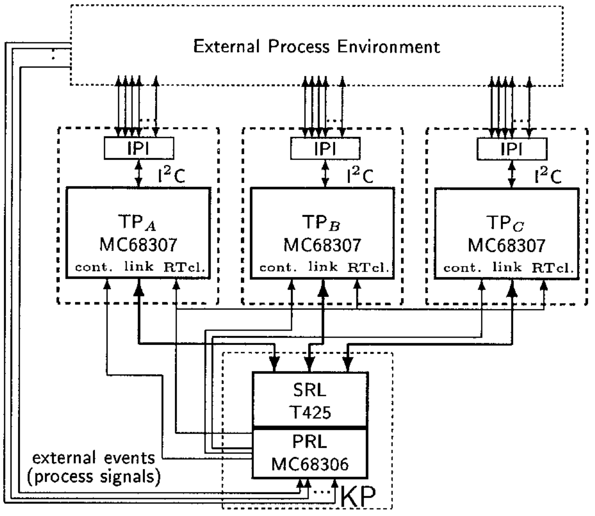
External Process Environment
IPI
I²C
TP_A MC68307 cont. link RTcl.
TP_B MC68307 cont. link RTcl.
TP_C MC68307 cont. link RTcl.
SRL T425
PRL MC68306
KP
external events (process signals)

Fig. 1. Scheme of an experimental hardware platform

In classical computer architectures the operating systems are running on the same processor(s) as the application software. In response to any occurring event, the context is switched, system services are performed, and schedules are determined. Although it is rather likely that the same process will be resumed, quite a bit of computer capacity is wasted by superfluous overhead. This suggests to employ a second, parallel processor, to carry out the operating system services. Such an asymmetrical architecture turns out to be advantageous, since, by dedicating a special-purpose, multi-layer processor to the real-time operating system kernel, the user task processor(s) are relieved from any administrative overhead. The kernel processor and the task processors are connected point-to-point by serial links, thus avoiding collisions on common communication media as a possible source of non-determinism. This concept was in detail elaborated in [7] and [2]. The implementation is shown in Figure 1.

The **kernel processor** (KP) is responsible for all operating system services. It maintains the real-time clock, and observes and handles all events related to it, to the external signals and to the accesses to the common variables and synchronisers, each of these conditions invoking assigned tasks into the ready state. It performs earliest-deadline-first scheduling on them and offers any other necessary system services.

External processes are controlled by tasks running in the **task processors** (TP) without being interrupted by the operating system functions. A running task is only pre-empted if, after re-scheduling caused by newly invoked tasks as a consequence of an event, it is absolutely necessary to execute one of the arriving tasks immediately in order to allow for all tasks to meet their deadlines. Each task processor supports a peripheral serial I^2C bus. To these buses intelligent peripheral interfaces are connected, enhancing fault tolerance by adding certain intelligence to them enabling reasonable reactions in exceptional situations, and increasing system performance by providing higher level I/O services.

3 Concept of a Real-Time Programming Tool

To program applications for the above hardware platform a tool is being constructed [8], into which also a specific exception handling mechanism [4] is integrated. Its ultimate objective is to produce the best possible program code for embedded hard real-time applications, and realistic upper bound estimations of the code's execution time. In the tool two parts are closely interleaved: a compiler for an adapted standard real-time programming language, and a program execution time analyser. The latter is providing the necessary information for a schedulability analyser currently being beyond the scope of our research.

Being aware of the unpopularity of defining new languages and writing "own" compilers, we had several arguments for doing so. One is that commercially available and widely used high-level programming languages and compilers still do not meet all the needs of hard real-time application programming. Another argument in favour was the co-design of the particular experimental hardware architecture, the corresponding operating system, and the application development tool rendering it impossible to use any of the existing compilers. Finally, experience with run-time analysis of high-level source code programs demonstrated that access to the internal structures of a compiler is required in order to gain reasonably realistic results.

The language introduced was called miniPEARL [8]. It is a simplified version of PEARL [6], a standard language for programming real-time applications, which, however, may produce temporally unpredictable code for several reasons. To eliminate these problems, PEARL's syntax was modified. Further, to support efficient mapping onto typical target architectures certain features were removed.

Finally, it was enhanced by some constructs specific to real-time systems, as proposed by Halang and Stoyenko [7].

The main differences between PEARL and miniPEARL are:

- no GOTOs (LOOP and REPEAT instead);
- pointers and recursion renounced;
- number of loop iterations strictly bounded;
- signals not directly supported;
- temporally bounded statement execution, including synchronisation mechanisms and process I/O operations;
- possibility to explicitly assert execution time of software blocks;
- PEARL's DATIONs not implemented;
- improved task activation scheme and scheduler support.

To allow for schedulability analysis, precise execution times of application tasks must be known in advance. In our tool, two methods for the estimation of program run-times are supported:

1. *Analysis of executable code.* In this method, an automatic analyser is used to estimate execution times. Source code is transformed into an intermediate form (modified syntax tree) prior to the generation of executable code. Each element of this form is associated with a macro block that is used for two purposes. The first one is to generate code, and the second one to obtain its execution time. Since the execution time of a block can be data-dependent, as much information as possible about operands should be passed to it. An operand can be a register, a constant, a local or a global variable. When a macro is expanded, the sum of times needed for accessing these operands is added to the basic execution time of the macro.

2. *Direct measurement of executable code.* This method can be used when a more precise value for an execution time than estimated is desired. To achieve this, object code is executed on the target system and the execution time is recorded. Direct implementation of this method has some obvious disadvantages. First, the complete target system must be implemented; thus, co-design of the hardware and the software of an application is not possible. Further, through recording, only average execution times can be obtained. For a usable analysis, however, worst-case execution times are needed, and it is usually difficult to create a test scenario leading to a worst-case situation. Finally, the impact of the delays caused by the input/output devices is dependent on the run-time circumstances.

By our approach, these disadvantages are eliminated. Only a task processor or its equivalent must be implemented. The longest path through a task is determined by the compiler, and "pilot code" is generated running only along that path. From a set of alternative constructs (IF and CASE statements, for example), the longest one is statically routed. All time-guarded commands, system calls and input/output variable accesses are replaced by corresponding delays. This pilot code is then executed on the hardware platform.

To practically and adequately support the design of embedded real-time applications, it was considered to include new, and more explicitly emphasise the existing, features in miniPEARL to enhance its suitability also for purposes of hardware and software system specification [5]. Instead of using strictly formal specifications, systems can be described in a simple and straightforward manner with a terminology which is close to application programmers and their way of thinking. Such descriptions are mixtures of clauses in syntactically correct formal notation and natural language inserts. Employing appropriate artificial intelligence methods [1], specifications are then gradually refined until programs in the real-time programming language miniPEARL are obtained.

4 Conclusion

To provide for application layer predictability of an embedded real-time system, the main objectives pursued in its design as presented here were determinism and predictability of each of its layers. The alternatives to design such a system to an as large as possible extent of consistency with the guidelines set, using commercial off-the-shelf components and other easily available means, were verified and validated. Applications designed this way fulfill the requirements of hard real-time systems, viz., timeliness, simultaneity, predictability, and dependability, better than conventionally constructed ones.

References

1. C. T. Cheung. Transforming Mixed Formal and Natural Language Specifications into High Level Language Code. Internal Report, Department of Computing, Hong Kong Polytechnic University, 1997.
2. Matjaž Colnarič and Wolfgang A. Halang. Architectural support for predictability in hard real-time systems. *Control Engineering Practice*, 1(1):51–59, February 1993. ISSN 0967–0661.
3. Matjaž Colnarič, Wolfgang A. Halang, and Ronald M. Tol. Hardware supported hard real-time operating system kernel. *Microprocessors and Microsystems*, 18(10):579–591, December 1994. ISSN 0141-9331.
4. Matjaž Colnarič, Domen Verber, and Wolfgang A. Halang. Supporting high integrity and behavioural predictability of hard real-time systems. *Informatica, Special Issue on Parallel and Distributed Real-Time Systems*, 19(1):59–69, February 1995. ISSN 0350-5596.
5. Matjaž Colnarič, Domen Verber, and Wolfgang A. Halang. A Real-Time Programming Language as a Means to Express Specifications. *Control Engineering Practice*, 5(7), July 1997.
6. *DIN 66 253: Programming Language PEARL, Part 1: Basic PEARL*. Beuth Verlag, Berlin, 1981.
7. Wolfgang. A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, Boston–Dordrecht–London, 1991.
8. Domen Verber, Matjaž Colnarič, and Wolfgang A. Halang. Programming and time analysis of hard real-time applications. *Control Engineering Practice*, 4(10):1427–1434, October 1996. ISSN 0967-0661.