Embedded Systems and Real-Time Programming

Niklaus Wirth

wirth@inf.ethz.ch

Abstract. Although computers have been employed for decades to control machinery and entire laboratories, the term *embedded system* has received renewed attention and is about to establish itself as a discipline of its own. We therefore try to identify its characteristics and peculiarities. The most outstanding single item is the role of time. The addition of timing conditions to all other specifications of a system causes *real-time programming* to appear as a particularly difficult and challenging subject. We ask how the rapidly growing demands can be met, and emphasize that reliability must be guaranteed in view of the potentially disastrous consequences of failures upon the controlled equipment and its environment.

1. Introduction

For decades we have been told that writing a program is easy, and everybody could do it. That may be so. But certainly getting a program to "run" is already somewhat harder. Providing a complete specification of requirements, and then writing a program that meets these specifications is very much harder, not to speak of analytically verifying its correctness. Designing a program that one is willing to publish in good conscience (for others to study, not just "open source") is even an order of magnitude harder. And writing a program that someone else is willing to adopt and use is almost impossibly hard.

I have used this scale of difficulty for over a decade, and one can readily extend it in many directions. For example, programming concurrent processes presents its own class of new difficulties, particularly if it involves several real processors instead of concurrency simulated by threads. What about real-time programming for embedded systems? Is it also a class of its own? Before we talk about the technical term of *embedded system*, we need to identify the characteristics that distinguish it from conventional programming and perhaps influence the challenges and difficulties encountered in design.

2. Challenges and Difficulties

As the term suggests, an embedded system is part of a larger whole that typically consists of many components, not just computer modules, but also sensors and

T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pp. 486-492, 2001.

actuators. This implies that many activities occur concurrently, and are to be controlled by the embedded computer system part. This results in a variety of challenges and problems that have no counterpart in conventional programs, neither in commercial data processing nor in scientific computing. Without claim for completeness, let me list a few:

The various ongoing activities imply multiple, *concurrent computing processes*, and with them the problem of their synchronization for harmonious cooperation in their exchange of data, be it via messages or a common store.

The activities in the system run at a given speed, and thereby impose constraints on the delays with which a computing process must generate reactions and responses. These *real-time constraints* are requirements unknown in conventional programs, where the only limitation of delay is typically the impatience of a human user. It is indeed one of the great achievements of digital computing that we can abstract from time. The only condition that matters is that the programmed operations are strictly performed in the prescribed sequence, one after the other.

In mechanical systems, *economy* is more important than it is usually considered by computer programmers. Economy of computing power and of memory are essential, although in a declining degree. The ever growing speed of microprocessors and capacity of memory chips have the negative consequence that careful and economizing design becomes less respected.

An embedded computer system receives its inputs from a variety of *sensors* of voltage, temperature, pressure, light, acceleration, rotation (gyros), radiation, chemical gases. It delivers its outputs to *actuators*, lights, signals, current sources, motors, steppers, relays, valves. The challenge for the programmer is to manage various devices outside the digital world through interfaces, that often are subject to stringent real-time constraints, and at the same time inadequately specified.

Recently, a criterion has entered considerations that has hardly played a role in the past: Electrical *power consumption*. It has been brought into the limelight by battery-powered computers; and many embedded applications belong into this category. To an increasing degree, it is the ratio of computing power over power consumption that is decisive in the selection of a processor. Low power may render cooling equipment unnecessary, thereby reducing overall consumption even further.

The addition of a flexible computer to control mechanical equipment makes it possible to operate this equipment without human intervention. The resulting systems are called *intelligent robots*. This opens new possibilities to operate them in places where humans would not dare to enter, in *hostile environments*. This application calls for special properties of hardware components, such as radiation resistance, but also imposes additional requirements on programs, such as a fail-safe strategy.

Finally, this brings us to the topic of *reliability*, which plays a much heavier role than in pure computing applications. A failure no longer results in a wrong number or a black screen, but in a crash, the loss of very expensive equipment, or even of lives. Programming becomes a serious activity, where personal responsibility cannot be abdicated. Reliability must be a permanent property and have overriding importance.

The consequences of a mistake may affect adversely many people, as the following story will show. It happened in Basel on June 15. Basel is a large railway station connecting systems of Switzerland, Germany and France:

Traffic in Basel's main railway station collapsed totally on Friday between 9:30 and 11:15. 50 trains and thousands of passengers were afflicted. The reason for this standstill was the failure of a computer system in the main switch controlling station.

All trains were blocked; they could neither enter nor leave the station. The failure occurred in a new computer system that was put in operation only recently. It controls all the switches and signals in the entire railway station.

Four days later, the computers failed for the second time paralyzing traffic completely:

The cause for this disaster remains unclear. Further such break downs therefore cannot be excluded.

Two weeks later, reports said the error had been found. The most illuminating comment was that the two computers had failed to understand each other, also causing the backup computer to clonk out. Reliable sources later said that it was still unclear whether the mistake had actually been identified.

3. Ways to Overcome the Complexity of Real-Time Programming

Indeed, the design of embedded systems appears to be of an overwhelming difficulty. One must wonder how so many successful systems had been designed in the past, considering the often flaky techniques and tools at the engineers' disposal. Their achievements are truly remarkable. Large embedded systems control gigawatt power stations, huge water dams, atomic reactors, they guide high-speed trains, aircraft and missiles. "Large" here means megabytes of code and millions of lines of program text. One wonders how it was at all possible to build systems of such a staggering complexity, when it seems utterly impossible to guarantee their reliability and one is content with a reasonably low probability of failure. This state of affairs is well-known in other fields of engineering, where, however, this rest of uncertainty originates in mechanical and physical properties of materials. In software, dealing with abstract artifacts that do not wear out, we should strive to do better.

To master complexity is evidently the name of the game. Therefore, a large effort has recently gone into studying the foundations of programming and into establishing a theory. With a solid theory about the subject, so the theory goes, we will be able to develop programs within a framework and with tools that control the process to the effect that mistakes will become, if not eliminated, then at least rare. An axiomatic theory and a discipline of programming have been created, automated tools, themselves of a staggering complexity have been built, and text books have been written about program verification and programming with inherent correctness proofs. These efforts deserve praise, but they have hardly begun to take the added problems of real-time programming into account. If one considers the very limited degree in which these theories are being practiced, severe doubts appear as to their effectiveness in the world of computing at large. Surely ambitions and expectations have been scaled down.

How do most people learn to program? By learning rules of a language and then writing. This is in contrast to learning how to compose prose. There we first read and read again before testing our own writing talent. Our compositions are scrutinized by the teacher, correcting our spelling mistakes and gradually improving our style by making suggestions. In programming, our compositions are rarely scrutinized by a teacher; our teacher is the computer telling us whether or not "it runs". And sometimes it runs without being correct, and sometimes it fails in spite of appearing correct. Inspecting programs is an unappealing business, and the more intricate and

inscrutable a writing is, the less is it in danger of being inspected. As a teacher I have seen some orderly and much disorderly code. But every time I looked at a larger "piece of code" (notice the term *code*, implying some secrecy), I discovered that much of its intricacy did not stem from the problem to be solved, but from poor mastery of the art of programming. In other words, much of the complexity is home-made. Every time it was possible to achieve the same goal in a simpler, often much simpler way, by factors of 2 to 10. This is staggering!

The truth remains that for good programming talent is required. Complicated theories and overly complex tools are not enough. In fact, sometimes they even foster bad style and add complexity. They seduce the programmer into the belief that he can easily abdicate responsibility to the tools, and the tools will guarantee proper form and style. This is no crusade against sound theoretical foundations and honest tools, but the best, in my experience, is the rule to *combat complexity like the devil*.

4. An Example of Reducing Complexity

Let me tell you about one of my recent encounters with that ubiquitous devil. I am interested since my youth in flying objects. Six years ago, I heard about a project with the aim of constructing a model helicopter for autonomous flight, controlled by an on-board computer [1]. As a layman, I was surprised that this was not standard practice in every helicopter. But actually this is not so, and in fact controlling a model helicopter is harder than a big craft, because the latter's big rotor has a much larger moment of inertia, leaving more time for corrections. Nevertheless, occasionally helicopters crash due to pilot errors in difficult situations. Hence, a computer-based stabilization system might be handy.

Our model, with a rotor diameter of 180 cm, was equipped with two commercial computer boards, each with an Intel 486 processor and an 8 Mbytes store. The software was based on a commercial real-time operating system, chosen because of the many concurrent processes to be managed. The resulting machinery with computer boards, inertial guidance system with gyros, servos, compass, telemetry and, last but not least, batteries weighed some 20 kg, and consumed some 20W. It did not take me long to believe that a simpler system could be designed, the software being only part of the simplification.

The first decision was to build an entirely new system on a single board. As processor I chose a Strong ARM (DEC 1035), a RISC architecture delivering the computing power of (at least) the two Intel 486s with a power consumption of little more than 1W. The next decision was to eliminate the RT-OS, as it seemed possible to do essentially without concurrent processes in the form of threads. The third decision was to program the entire software in Oberon [2], which is very suitable for "programming close to the hardware". This implied, however, to first build an Oberon compiler for the Strong ARM. This turned out to be easy (1 man month) because much could be taken over from existing compilers [3], and provided the invaluable advantage to add SA-specific inline procedures for handling processor initialization and device interfaces. In order to slightly simplify the task of compiler construction, a few features of Oberon were omitted, and a few new ones were added (see Appendix).

The task of the helicopter controller is essentially to sample sensors (accelerometers and gyros), compute new values for power, pitch, roll, and yaw, and output the computed values to the respective servos. This can be done in a single loop which is triggered in equal time intervals, in this case every 20ms. Because for every new group of outputs 4 preceding groups of inputs are required as arguments, the inputs are buffered. Input and output interfaces were implemented by two PLDs, converting the PWM signals to binary (8-bit) values and vice-versa. They could be considerably simplified by converting the (up to) 8 signals sequentially. As a consequence, the interrupt time slice was chosen as 2ms, and a counter is incremented to identify the signals being input and output.

As a result, the time critical operations are kept in a single interrupt handler, triggered by a clock generated in one of the PLDs. This routine is, and must be, very short. It reads a single byte from the input PLD and deposits it in the input buffer, and it picks a single byte from the output buffer and feeds it to the output PLD. The "main program" is a single loop (after initialization), at one point waiting for the next change of the counter. In this loop, four value sets are taken from the input buffer, and a new output set is computed. All that had to be done concerning real-time constraint was to measure the time needed for this computation. Fortunately it turned out to be less than half of the 20ms time slice.

5. Some Conclusions

Certainly, this example may be called academic, because it is simple, or rather turned out to be simple. I am fully aware that different criteria apply in other cases, and that "real-world" systems "out there" are much more intricate. Yet, this example does show that drastic simplifications are possible – even in simple cases! With the addition of further input sources such as a compass and a GPS, for example, affairs become more complicated, because those inputs cannot simply be sampled, but arrive at unpredictable intervals. Hence, the temptation to introduce threads (and a system handling them) arises, and even may turn out to be justified.

However, one must be aware of the consequences. Task switching becomes hidden, and it becomes impossible to guarantee real-time constraints. It may all be very well, as long as processing power is available in excess, i.e. if the processor idles most of the time. But "throwing in resources in abundance" is not sound engineering practice, even if it appears to be cheap.

If we are to design reliable real-time systems, we must demand precise specifications, and then be able to guarantee that the computing time required between specified events is shorter than their interval of occurrence. This implies the availability of tools (compilers) that provide data about the time required to execute certain sequences of statements. Such tools are hardly available to my knowledge. Instead, "break-points" are inserted in the program, and a tool is used to measure the time consumed. But this is like program testing vs. analytic verification: The result applies only to the specific test case, but not in general The practical idea is that the test case is sufficiently "representative", and a safety factor is thrown in, in case of doubt.

The dubiousness of this practice is compounded by two circumstances:

If interrupts are involved, the interrupt handler steals time from the interrupted process (thread). If neither place nor duration of the interrupt are known, the effect on time bounds of the interrupted process may be considerable, and the worst case condition taken for analysis may be so much bigger than the "normal' case that the temptation to ignore the (very rare) worst case may be large.

Modern processors commonly use on-chip cache memories and pipelining, resulting in rather unpredictable performance variations. The time for an instruction sequence cannot be taken as the sum of the time of the individual instructions. Yet, the real-time program must reliably handle the worst case, which may be 10 or more times slower than the average case. Temptations to ignore that ugly worst case loom large.

Not even sophisticated analytical tools help to overcome these intrinsic difficulties with machinery displaying a stochastic behavior. As a consequence, we recommend the following two rules of thumb for the real-time programmer:

Refrain from using interrupts. If interrupts cannot be excluded, make sure that the time consumed by interrupt handlers is orders of magnitude shorter than any specified real-time condition. Handlers must be free of loops (with an unknown number of repetitions).

Be skeptical about sophisticated processors with caches and pipelines. Test your programs with caches turned off. (I am afraid this implies that the caches may just as well be left out anyway). Digital signal processors (DSP) typically do not feature caches, and are therefore to be preferred.

What renders the seemingly hopeless situation less desperate is the fact that in most cases the real-time constraints are weak; they are specified in seconds (or hundreds of milliseconds), while the processor executes millions of instructions per second, implying that real-time constraint can almost be ignored. However, in the case where several interrupt sources must be handled, it might be wise to employ an individual processor for each of them. After all, silicon is now cheap, and system reliability is dear. We should remember that interrupts were originally introduced to spare the use of separate, expensive processors for events that rarely occur and then take a negligible amount of time for a very simple, straight-forward action.

We might note that in the case of embedded systems the requirements concerned with time are in addition to all other correctness specifications. Could it therefore be possible to separate the two categories? Separation of concerns has always been a useful design principles. Hence, the appearance of a formalism (language) allowing real-time constraints to be considered and checked separately, perhaps by separate software tools, would mark a desirable step ahead.

Here we are concerned with systems containing computers embedded as crucial parts of a larger whole. The consequence is that the designer of the computer section must understand the whole, or at least be able to rely on complete and precise specifications of the interfaces between the rest and the computer. Often these interfaces are themselves unnecessarily complex (particularly if they comply with established standards!), giving rise to misunderstandings and mistakes. A second corollary of embededness is that failures of the whole may be due to other than computer malfunction, requiring much wider horizons in the search for errors. The designer of embedded systems should be a mechanical, electrical, and software engineer all in one. In our helicopter example, mechanical oscillations and resonances, loosening contacts, over sensitivity of sensors to vibrations, electrical

noise of the combustion engine, or mechanical fatigue of the exhaust muffler caused failures in spite of correctly functioning software, where crashes could be avoided only by an immediate fall back to the remote pilot, acting as a human reset button. In many applications no such easy solution exists. The Pathfinder vehicle landing on Mars comes to mind.

The major difference between “regular” and “embedded” systems programming, however, lies not so much in the compounded difficulties of the latter, but rather in the more stringent reliability expectations. If a system crashes, it may be fatal. It is as if every mistake may force you to restart construction from scratch, to step back to square zero. This increased awareness of consequences of malfunction and of our responsibility should make us doubly anxious to look for simple (not simplicistic!) solutions, and at the same time triply skeptical towards black boxes that we do not fully understand but nevertheless have to assume responsibility. I also warn against an exaggerated reliance on sophisticated development tools. Although they may be helpful, they cannot be a substitute for detailed understanding. I have witnessed cases where the mastery of the toolbox took a greater effort than the solution of the problem at hand, thereby diverting attention from the essence.

In our programming plight, we rightly expect to receive support from programming languages and their compilers. But even at run-time we are used to rely on checks detecting mistakes such as indices out of array bounds, arithmetic overflow, and other rare cases. When relying on such implicit checks, we are in a state of sin, because they detect mistakes that should have been avoided a priori by proper design. In the case of embedded systems, this becomes manifest by our uncertainty of the actions to be taken in the case of failure. For instance, in the case of the model helicopter, what should be done in the case of an array bound violation, what in the case of an arithmetic overflow? There is no reset button!

In summary, I do not only plead for avoiding complexity in program design, but also in the tools used. The more difficult the problem at hand, the more we must strive for reasonably simple solutions. They must never exceed our mental abilities to understand them fully. And the simpler the theories and tools employed, the more perspicuous are the engineer’s designs, and the more realistically can he take responsibility for a design. Past achievements of clever engineers have been wonderful; think only of the automated flights of satellites around the earth, to the moon and past planets. But the fight against mistakes and the overestimation of our infallibility will last for ever. It must not keep us from trying hard to do better.

References

1. J. Chapuis, C. Eck, M. Kottmann, M. Sanvido, O. Tanner, "Control of Helicopters," in K. J. Åström, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, R. Sanz (Eds): Control of Complex Systems, Springer Verlag.
2. N. Wirth. The Programming Language Oberon. *Software – Practice and Experience, 18,* 7, (July 1988), 661-670.
3. N. Wirth. *Compiler Construction*. Addison-Wesley, 1996, ISBN 0-201-40353-6.