EFFICIENT DEVELOPMENT OF HIGH QUALITY SOFTWARE FOR EMBEDDED SYSTEMS

Stanislav Gruden

Iskraemeco d.d., Kranj, Slovenia

INVITED PAPER
MIDEM 2003 CONFERENCE
01.10.2003 - 03.10.2003, Grad Ptuj

Abstract: *New electronics products are being developed with a constantly growing pace today. The development must meet very tough criteria: short time-to-market, continuous use of currently the best available technology in order to reach high performance requirements, etc. More and more we see that the cost for this is a decreasing quality of the products, especially the low cost consumer electronics. The problems are most often due to the insufficiently tested software of the embedded systems used. On the other hand there is no need to make the software optimized for performance anymore, usually the more efficient way of optimizing the overall cost and resources is just to use more powerful hardware.*

In order to increase the software quality in these hard development conditions some measures have to be taken into consideration: rigorous testing is one of them, but a lot can also be achieved by using of high level programming languages wherever applicable, making code as portable and reusable as possible, using tested other party software whenever accessible, etc.

Some techniques that can be used to make software more portable and reusable are presented. A common characteristic of these techniques is they use some of the system resources like memory or CPU time in exchange for structural organization that makes the code much easier to maintain, distribute between many developers and test. The technique of compiling and testing the code on strong personal computers or workstations before using it on a real system is described. This technique takes up some additional development resources at the beginning but saves them lately because it makes developing and testing a new code much easier, makes it portable and it is possible to have a large part of application completed even before the actual hardware is obtained, etc.

Učinkovit razvoj programske opreme za vgrajene sisteme

Izvleček: Živimo v svetu, kjer elektronske naprave razvijajo z vedno hitrejšim tempom. Ta razvoj se odvija v težkih pogojih: vstop na tržišče mora biti hiter, hkrati je potrebno slediti napredku na področju tehnologije z namenom ves čas delovati v optimalnem področju. Vedno bolj je očitno, da to gre na račun kvalitete izdelkov, posebno to velja za nizkocenovne širokopotrošniške naprave. Najpogosteje problemi nastanejo zaradi nezadostno preverjene programske opreme vgrajenih sistemov. Po drugi strani se izkaže, da ni več potrebe po pretirani optimizaciji programske kode, ampak je za boljšo izkoriščenost razvojnih virov in manjše skupne stroške boljše vzeti zmogljivejšo strojno opremo.

Za povečanje kakovosti programske opreme v teh zaostrenih pogojih so potrebni določeni ukrepi. Natačno testiranje je najpomembnejše. Mnogo se da doseči z uporabo višjih programskih jezikov, kjer je to mogoče, ter z izdelavo čimbolj prenosne programske kode in uporabo preverjene programske opreme drugih proizvajalcev.

Prispevek prikazuje nekaj načinov, kako programsko kodo narediti dobro prenosljivo in primerno za ponovno uporabo. Splošna značilnost takih metod je, da boljša strukturiranost programa gre nekoliko na škodo porabljenih virov - pomnilnika, procesorskega časa. Končni rezultat je lažje vzdrževanje kode, preprostejši prenos med razvijalci in učinkovitejše testiranje. Opisali smo postopek, kako na močnih osebnih računalnikih ali delovnih postajah prevesti in preveriti programsko kodo brez uporabe končne strojne opreme. Postopek zahteva dodatne razvojne vire na začetku, ki pa jih več kot prihranimo kasneje, saj sta razvoj in testiranje nove kode mnogo preprostejša, tako dobljena koda je sama po sebi dobro prenosljiva, večino kode lahko dokončamo tudi v primeru, da končne strojne opreme še nismo dobili, itd.

1. Introduction

Many articles and books have been written on the software crisis that has been continuously happening since the first commercial computer applications /1/. They address the huge problem of software quality, late time-to-market, etc.

The reasoning in this article is based on real life situations, which differ greatly from the theoretical situations. Theoretically the less expensive development would consist of a thorough problem analysis and after that, a complete design of software structure. The actual coding would not start until these two phases are finished and confirmed by the customers and design team. Unfortunately, this almost never is a case. The system analysis would usually take too much effort if it is to be complete and all the facts about the system are taken into account. This would also require the customer to really study the analysis and to make all necessary comments on time. Many customers are just not prepared to do this and leave the important decisions to the developers. The most frequent reason for omitting the analysis steps are the deadlines. We just simply live in a world where time is money and the 'theoretically less expensive development' would actually cost more because of the lost opportunities on the market /4/.

There exist possibilities, despite the real world requirements, to build up a better structured, more documented, portable and less error-prone code. Early and frequent integration /2/ does so with so-called front-loading - the problems should be detected as soon as possible. Designers must meet earlier and resolve the conflicts earlier. Following some guidelines ensures this to happen virtually automatically while doing the 'preferred' work - the coding.

Thorough testing must be performed through the whole development process. Some possibilities of how to set up and use testing features are presented.

The presented ideas are directly applicable when using standard programming languages like C, C++. They may not always be useful if higher-level design tools, which automate code generation, are used since these tools may already force the way the program is designed.

2. Relation between development resources, time to market, reliability and product price

The basic requirements for any development are:

- The final product must have a quality that is expected by the customers /2/. This may differ largely from one application to another. Mass consumer products like cheap children toys need not to be very reliable, sometimes it is even expected that they will be in use for some days and then abandoned; on the other hand the professional equipment is supposed to work without any problems for a long period of time. The highest level of reliability must be assured when human lives depend on the proper operation of the system.
- The production costs of the product tend to be minimized although the effort of doing this depends on application type. Very large quantity products are hardware minimized to the highest possible limit, since every cent is important. The software and hardware development costs for such products are low and represent a negligible part of the final price. On the other side high quality products need a lot of intensive development. Being sold in much smaller quantities means that every device incorporates in its price a significant portion of the development costs. These costs may be much higher than the hardware costs and in this case hardware cost minimization is not so important.
- Development resources are always limited. The most important of these resources are developers. Additional money, development tools and equipment can be obtained one way or another i· case of time pressure, but human resources are not trivial to add. This fact was known long ago, as early as in year 1975 /1/. A lot of time is needed to find people that are able to do the work; their training and getting to know the project cause another delay. This is true even with experienced developers. In case of tough deadlines one usually does not have any other choice but to count only on the available developers. The exceptions are possible in a case a very distinctive module can be separated from the entire product, which has a simple application programming interface (API), easily describable black box functionality, and is at the same time so complex that it takes a lot of work to implement. The functionality and requirements for such a module can be easily documented and presented to a new, skilled developer.
- Time to market is a big issue /3, 4/. In the world where the market competition is the main driving force, it is absolutely necessary to get to the market as soon as possible. At the beginning the demand for a new product is at the strongest, there is less or no competition and the prices are high. When the competition comes to the market the prices may fall to production cost, development may not be covered anymore and the profit disappears. The applications also have short lives. In some cases a fast development also decreases development costs /3/.

The goal is to make a good compromise between these issues of which each one generally contradicts the other. Lower product price means weaker hardware, which needs more development resources to get the work done, otherwise the reliability will be worse or development time will be unacceptable high. As a very simple approximation it could be stated that a weighted sum of these parameters is a constant value.

The use of better tools and equipment can help reducing this value but may not always be affordable. On the other hand, a better organization of the development process can greatly improve the final quality of the product - the number and the severeness of operation errors (bugs), which are an inevitable part of the products, especially complex ones.

For quality devices it usually turns out that it is better to use more advanced platform than the minimal one (more memory, higher speed). Namely, despite good planning and previous analysis the amount of system resources on the platform needed is usually underestimated. With stronger platform the programming is easier because the focus can be put to the problem solving and not to the resource optimization. The code can be written in a cleaner way, with higher programming languages. Such a platform is easier to maintain and upgrade later. Electronic elements nowadays are developed very quickly and one moment sophisticated and expensive platform very quickly gets a successor, which is even more powerful. The price of the older platform then decreases and when the device is in production it is not so expensive anymore.

3. Design considerations

It is a good idea to take a lot of time at the beginning of the project to split the system into smaller logically separated units. The most important part is to define good boundaries between them. These are the so-called APIs (application programming interface). They should be as simple as possible, but must,implement all the needed functionality. The policy of how to use the APIs must be as simple as possible to avoid confusion and misunderstanding between developers. The most important API is the one that delimits the platform independent from the platform dependent code as shown in Figure 1. The simpler it is, the easier the porting to a new system will be if this is needed in the future. Additional well-defined APIs are used to delimit modules that will be assigned to different developers.

A good directory scheme of source code must also be designed, which fits well to the structure of developers' modules. In ideal case each module can be assigned a special directory, which will be maintained by one developer only. This is rarely a case and to avoid confusion and backup problems use of a version tracking system (CVS, Microsoft SourceSafe) is needed.

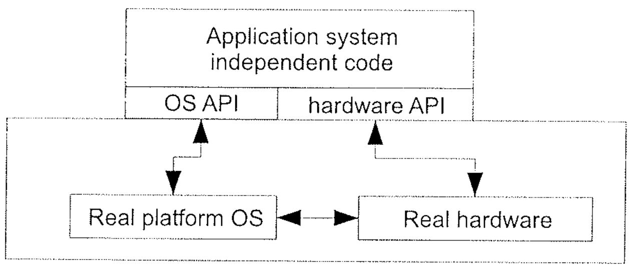

Figure 1: API used as a boundary between the platform independent and the platform dependent code.

A common programming 'technique' is use of 'copy and paste'. While this gives good enough results very quickly, it can be a source of hard to understand errors later. The reason is that when some part of such a code must be corrected, one usually forgets to make a correction on all the places where this code was also used. It is better to use functions performing the task (with parameters if the task is not exactly the same everywhere), macros, which will be expanded, or templates. Macros are deprecated because they produce name spacing problems and are hard to debug (breakpoints can usually not be set inside macros).

The higher the level of the programming language, the easier the programming is. The reason is that on the low level, the programmer must also focus on the correct use of code. The result is the distraction from the main problem, which the developers try to solve.

Humans can easily understand complicated data structures, especially when hierarchically organized, on the other hand it is very hard to trace all but the simplest program flows. Object oriented programming languages are specifically designed to make programming easier by focusing the programmers attention on data structures instead of algorithms (a case in classical programming).

Integrated debugging environments (IDE) with powerful graphical user interface are ideal for making an application very fast (rapid prototyping), but are less suitable when it comes to maintaining the code, reusing the code, automating the process of compilation, source saving, etc. Typical such programs make a lot of auto-generated code, which usually resides in predefined directories. Their configurations are typically in binary form, so they are less manageable then text based configurations. The only access possible is by mouse (sometimes it is very frustrating if a lot of clicking is needed to make a lot of identical changes, which could have been done with a simple 'find and replace'). Classical tools such as make, command line compilers and powerful script shells offer much greater flexibility but take a lot of time to learn how to use and setup. This time is very often saved later. They are very portable, flexible and easily automatically generated.

Data types used are very often a big problem when it comes to code portability. Apart from big/little endian compatibility (not so hard to manage properly) the most problems are caused by the difference of storage length and precision for simple data types (in C programming language), especially integer and sometimes also floating point data types. The same code that may work perfectly on one platform would fail - during runtime usually - on the other. There exist recommendations about this issue but care must always be taken to prevent problems. Using only specially defined (typedef) types helps a lot (using types directly from headers of different operating systems often introduces more confusion than it solves), but the type promotion rules in C always make problems.

The recommendations about the coding style and other rules may differ very much from person to person and cannot be generalized. Even related project teams do not always agree (/5/ and /6/). It is a good idea to read a lot of them so every valuable piece of information is taken into account. Very often some largely accepted rules just are not so efficient as generally thought, for example, extensive use of comments usually just puts additional load on developers, with no real benefit.

Multiple checking of the data validity takes additional effort to implement but may help reveal some errors. On both sides of an API a different range of data may be valid. For example, a key pressing is detected by the hardware and processed by the driver, which always implements some sort of glitch removal. At this stage typically only very quick (some microseconds) changes are considered as a glitch. This may or may not be suitable for the application, it depends on what is the behavior required by the customer. It therefore makes sense that the application implements its own data check algorithm (longer glitch detection in this case). Making so ensures the driver for the keyboard can remain the same regardless of the application requirements and is therefore useful also for future applications. However, the excessive use of double-checking leads to bad efficiency.

It is considered a bad programming technique to:

- Use global variables; instead, all data should be put locally on stack. This is slower and needs more memory but is much less error-prone. Even worse is reusing global variables for more than one purpose in order to save memory.
- Write code in an optimized way; it is better to concentrate on the clarity/readability of the code. That is especially true today when good compilers make much better optimization than any programmer could, and larger or slower final executable is not a problem either.
- Not to use operating systems, multi-threading, etc., except for extremely simple applications.

To make the code compile on many platforms two basic techniques are possible:

- Using the same modules with compilation switches.
- Using a common platform independent modules and separate modules for each platform that implements the dependent code.

The first approach is better only if the number of differences is very small (for example, using sockets under Unix or Microsoft Windows). Usually the second approach results in a more clear and easy to understand code. If the first approach is used, it is better to code the differences as macros or templates in a separate header in order to make the main code more readable.

The software documentation is often a problem. It is sometimes updated only few times during the development, for example, when explicitly required by the customer. It is not kept up-to-date when the changes in the software happen. The reason is almost always the time pressure. When there are not enough resources available to make the complete documentation, the priorities must be set up. It is usually sufficient to have a coarse description of the system architecture and algorithms used, which does not change much during the development. The details in the code should be commented in a way that the author, or any other normally skilled programmer, would understand them at any time in the future. Excessive use of comments is not a solution, doing so can make the code harder to understand, it is also very likely that the comments do not follow the code changes, which renders them misleading and wrong.

The APIs, on the other hand, deserve extremely detailed documentation, which consists not only of software interfaces (function prototypes, macros, enumerations, etc.), but also of detailed description of what conditions have to be fulfilled for the code on both sides of the interface in order to function properly (for example, thread safety policy, speed and processing capabilities limits). These conditions must be kept in mind for all the developers: those that implement the functionality should know what they must implement and what they need not to (but are allowed to if they can); those that use the functionality should be aware of API usage limitations and use only the documented features. It is very risky to count on the knowledge about the other parts of the system and to use non-documented features because they may change at any time. Any small change in APIs must be documented immediately.

4. Device simulation

Almost no development is possible without first making the product prototype. The same also applies for the embedded system software development. A powerful way to do this is simulating the behavior of the device and its environment.

Very often there exist simulators for target CPU and peripherals, which allow testing of native executables. Using one of them makes the debugging much easier. But the first level of simulation can be done in an environment used more generally by the developers.

The basic idea is to use a good compiler and debugger on a strong personal computer or workstation, to produce the first working prototype of the new application as a simulation. The application code (the platform independent code) is shared between this simulation and the real device that will be developed later in the process with a new platform as shown in Figure 2.

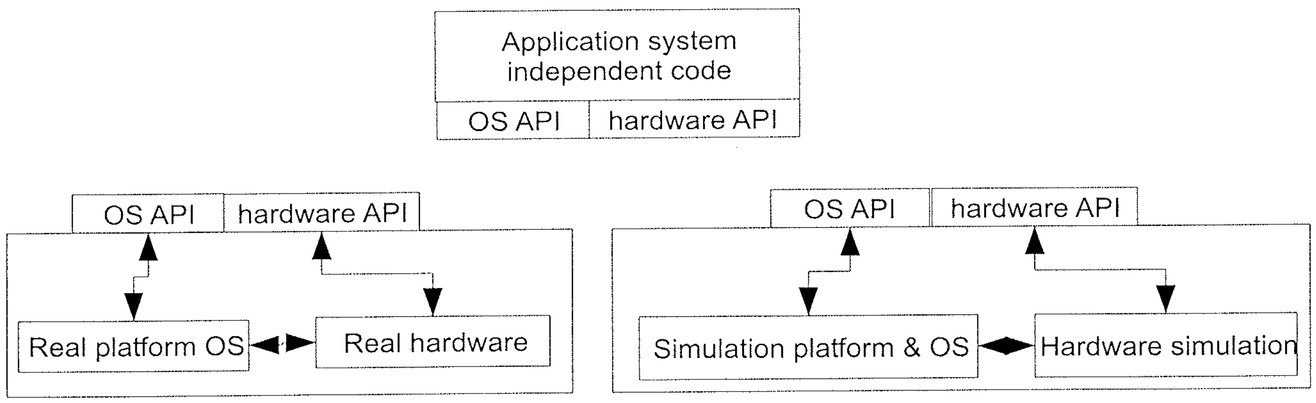

Figure 2: Replacement of the platform dependent code; the real platform and the simulated platform are interchangeable.

It must be noted that according to the general recommendations the prototype code should not be used in the final product, instead the code should be rewritten from scratch. Due to usual time pressures this is only seldom feasible.

This technique has one strong drawback - it is necessary to make an emulation of every part of the platform (operating system - OS, hardware, etc.) that will be present in the final product and set up one additional parallel project environment, which takes some time and efforts at the start of the project development. However, the simulation system usually turns out to be very simple; for example, in the real world sensing the state of a simple switch requires using an I/O port and writing the driver to handle this information, on the simulation that may be a simple button that is incorporated in a matter of seconds. LCD driver in a real world is complicated, but drawing bitmaps in any window environment is much easier.

Virtually all the other outcomes of this technique have a positive influence on the development, mainly because of the so-called front-loading effect /3/.

The main application development may be started long before the actual platform is available, tool packages set up and all the necessary drivers for the product are completed.

API between the system dependent code and the application, as well as any other APIs, can be evaluated and optimized. The API definition is more likely to be correct and complete if tested on many platforms. Any weak points or exceptions show up sooner and can be documented more reliably. This operation also forces the project manager and the programmers to think about how the code should be organized and split between the platform dependent and the platform independent part. The code organized this way is much easier to split between the developers and to port to a new hardware or OS platform if needed, since only the dependent part of the code has to be rewritten. This may happen sooner as predicted, because newer, better and more suitable platform elements emerge on the market very rapidly.

Debugging on the real platform is usually hard to do or is poorly supported. The code has to be loaded to the target, run and then connected with the host application. The ease of the debugging process depends on the debugging tool maturity, communication channel bandwidth that is established between the target and the host, etc. Available target simulators are platform oriented, which means a new simulator is usually needed if a new processor is chosen. In the case the producer remains the same, the existing simulator may still be useful, but changing to another producer usually means setting up an entirely different environment, which may not even have the same needed functionality.

Debugging the simulation on the personal computer or workstation is easy because standard, powerful and well-known tools are used. It is much faster, simpler and allows operations not possible on the real platform. Practically only the platform dependent code must be debugged directly on the target.

Device simulation can be connected to environment simulator, a program, which simulates the system where the device will be operating, to check if the behavior of the application is correct.

The simulation platform dependent code can be reused, usually with slight changes and enhancements, for the projects in the future.

Compiling the platform independent code on more than one hardware or OS platform and with different compilers can reveal warnings and errors that may only be discovered later during run-time, when they are much harder to track down. This way the code truly becomes independent.

It is very common that the development is started with insufficient analysis of the problem and customer needs, and continued with too weak emphasis on consistent software design basement. Instead, the coding starts very soon, usually because of tight schedule. This is hard to avoid. The development for parallel platforms forces this coding to be much more consistent and is thus less likely that corrections in the future will be needed.

The developers must resist the temptation to quicken the coding process by putting to the same module the parts of the code that are different in nature. This largely extends the complexity of such module and makes it much harder to understand, document and especially maintain. Generally, unless 'copy and paste' is used, porting the code to many platforms quickly discloses such coding style and forces the programmers to organize their code properly.

5. Environment simulation

Any electronic device typically operates in three basic stages:

- Capturing of the input data from the environment.
- Processing of the data.
- Making actions, returning the information to the environment.

Usually, the most complex part of the device software is the data processing. On the other side very often only small amount of the data is captured from the environment. That makes the simulation of such an environment extremely simple and easy to implement.

Example:

Electricity power metering device can capture the following input data:

- Voltage and current samples.
- Control inputs.

- Receiving characters over a serial communication.
- Detection of pressed keys on the console.

The information that is transferred to the output is:

- Data for the LCD display.
- Sending characters over a serial communication.
- Control outputs.

In this case input and output interface are very simple, compared to what this power meter device has to process:

- Calibrating the input data and calculating many different values: powers (active, reactive, apparent), voltages and currents (effective, maximum, minimum, harmonic analysis), frequency, etc.
- Tariffing and other processing of the data.
- Responding to input requests over communication and keys: parsing input data, output data formatting.
- Real time clock, security, checking data validity, etc.

For this system it is very simple to make the simulation of the environment. It consists of a simulating of the input signal samples, key presses and the communication channel, and on the output, of the LCD simulation and communication channel responses.

It turns out that such simplicity is very often a case. There may be some situations when simulating of the environment is hard to implement. For example, simulating the sensor data to the walking robot would be a complex task because of the strong feedback of robot's movements to the input sensors; this feedback is not trivial to simulate. But even in such cases the early simulation saves development resources /3/. First simulation may be low-fidelity and enhanced later, if needed. It can be made programmable and may support automated test procedures. This is useful during development, to test the new functionality, as well as for the device maintenance, when the device program is being changed because of fixing errors and sometimes adding new features. Namely, changing the program of the device may cause side effects that are not anticipated, and a complete system has to be tested as a consequence to ensure quality.

Besides testing of the device simulation, the environment simulator can be made to support the testing of the device on the real platform as well. The same test procedure is used; only the communication channel between the application and the environment simulation must be replaced. Pure simulation may run in the same executable as the environment simulation, communicating directly through the function calls. Real platform may connect to the environment simulator via one of available hardware communications (serial, Ethernet).

The general principle of how to incorporate the environment simulation to the system, is shown in Figure 3. The remote channel support represents the used communication means for connection to the environment simulation. In the Figure 3 the two hardware APIs implement the same functionality, which is identical to the general hardware API in the Figures 1 and 2. This way, the expanded application can directly replace the original application.

The switch can be realized as an object which all of the used information paths use to make the registration to. These information paths do not necessarily support all of the functionality, so they register itself each one with its own set of capabilities. Such implementation enables:

- Redirection of the output data from the application to all the data paths that support that data.
- Gathering input data from the data paths that can provide that type of data and deciding which input data path is in control at any given moment.

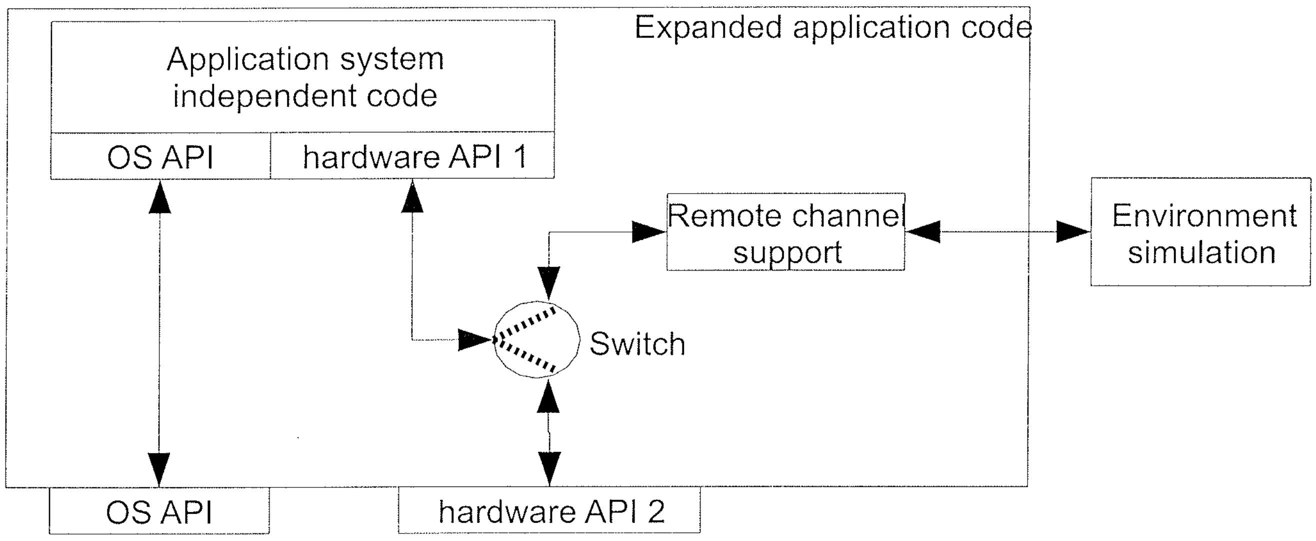

Figure 3: Insertion of a switch to support redirection of the data flow between the platform implementation and the remote environment simulation.

In Figure 3 only two data paths are shown, but this is not a limitation. For example, another path could be added to test a particular piece of hardware through the simulation environment.

Similar to the device simulation, using the environment simulation technique forces the developers to consider the application functionality even more in details. Some new ideas about how the input data could behave may be gathered. The whole system can be very successfully used as a demo that is presented to the customer. The customers very often do not have a good notion of how exactly the final system should behave and this is a good opportunity to compare wishes and reality. Demo can be presented in the earlier stages of the development when the changes are more easily implemented and are thus less expensive.

6. Conclusion

A situation in the software development domain was briefly described, stating that it is still far from reaching the point where high quality software is developed and delivered on time with all the promised functionality.

The simulation method of the application and the environment is particularly useful when developing the software for the embedded systems, especially where the input and output interface to the real world are not too complex, which is usually a case. These simulations can serve as a design aid during development and as a testing tool later.

All of these methods are not necessarily useful for every programmer, but they may be taken into account if they are found to fit into existing concepts of the development group. At least they may serve as an idea that would help produce newer, even better development methods. They are not meant to answer to the hard question how to consistently produce a good and reliably software on time, instead they provide some improvement in the case, which is very usual, when not enough development resources and time is available for the project.

7. References

/1/ F. P. Brooks, *The Mythical Man-month: Essays on Software Engineering*, Addison Wesley, MA, 1995, 0201835959

/2/ W. A. Sheremata, "Finding and Solving Problems in Software New Product Development", *Journal of Product Innovation Management*, 19 (2), 2002, 144-158.

/3/ S. Thomke, T. Fujimoto, "Shortening Product Development Time through 'Front-Loading' Problem-Solving", *CIRJE, Faculty of Economics, University of Tokyo*, CIRJE-F-11, http://ideas.repec.org/p/tky/fseres/98cf11.html.

/4/ P. G. Smith, "From Experience: Reaping Benefit from Speed to Market", *Journal of Product Innovation Management*, 16, 1999, 222-230.

/5/ http://www.purists.org/linux/: Linux Kernel Coding Style.

/6/ http://www.gnu.org/prep/standards.html: GNU Coding Standards

Stanislav Gruden
Iskraemeco d.d.
Savska loka 4, SI-4000 Kranj
E-mail: stanislav.gruden@iskraemeco.si

Prispelo (Arrived): 15.09.2003 Sprejeto (Accepted): 03.10.2003