EMBEDDED SOFTWARE DEVELOPMENT WITHIN A PRODUCT DEFINITION - BENEFITS AND FRAMEWORK

M. Jenko

Laboratory for Electrical Engineering and Digital Systems
University of Ljubljana, Slovenia
marjan.jenko@fs.uni-lj.si

Keywords: concurrent modeling and design, cross-platform software development, embedded software reuse, software components, virtual prototyping

Abstract*: This paper contributes to paradigms on embedded software design in a market niche where a) application complexity is relatively low (compared to modern high-end mobile phones, for example) and b) the product can not bear the expense of a superfluous microcomputer with a built-in operating system but c) embedded software does run the product. Most modern consumer appliances adhere to these conditions.*

The presented paradigm enforces the development of a functional model of a new product. The model is built in an early phase of product design. It is specifically structured for the purpose of a) being used and refined in discussions on the functionality of the product and b) having most of its internals reused in the production of embedded software that will run the actual product. The new paradigm has built-in mechanisms that ensure quality, shorten design time, give way to cross-platform software production and support team work in potentially remote locations.

The presented paradigm is demonstrated in a case study – the design of an industrial kitchen appliance.

1. INTRODUCTION

Objective of the 5th seminar on Engineering Design in Integrated Product Development (EDIProD'2006) is strengthening design methods aimed at both quality of the product and of the production process. This paper contributes to the quality of the product and to the quality of the product *development* process.

The paper establishes a relation between two phases of a new product development that were historically understood as non correlated activities. These phases are product definition and design of embedded software that governs the new product.

The issues in embedded software development are steadily entering the picture of modern product development since most present products, from fighter jets, space shuttles down to cars, intelligent housing and even further down to piezo-toothbrush or heated skiing shoes get lots of their functionality from the built-in software code.

Competitive engineering is a carefully balanced mix of different fields – mechanical engineering, physics, electronics and embedded software. This paper presents usage of a software development process as a component of a product definition phase. The new approach adds a working model to the product definition phase, and it boosts efficiency and quality of "software manufacturing". This is where this paper touches the quality of production process, the second objective of the 5th EDIProD seminar.

Numerous analyses report that about eighty percent of embedded software projects do not finish on schedule, and about forty percent of such projects failed for different reasons in the previous decade (Stewart, 2004). A breakthrough in technology had to be engineered to get to new generations of feature-rich but reliable products. Like for workstations, an Operating System (OS) had to be developed for embedded applications that can tolerate associated costs.

There is still the unanswered question of how to improve quality and yield (percent of successful designs) in the design of embedded software when a dedicated OS does not exist and the new product can not bear the expenses and time associated with development of a dedicated OS by a specialized company. This paper gives insight into this problem and offers systematics for better practice in the design process.

2. SURVEY OF PRACTICES FOR EMBEDDED SOFTWARE DESIGN

Different companies have different internal practices for embedded software production. Such practices depend on complexity of the product that the company produces or develops for another producer, and on the types of development culture that evolve in companies over the years.

2.1. Small system practice

The design of the Egg Cooker in Figure 1, a new apparatus on the industrial kitchen appliances market (Jenko, 2005) that we released some months ago, represents a typical example of a small embedded system. The apparatus makes pasteurized soft-boiled eggs, which is a novelty on the market. Boiling eggs soft implies a low process temperature while pasteurization implies a high process temperature. There is a very narrow temperature window where both processes coexist.

Source code consists of about 7000 ANSI C program lines. The whole design project, from a vaguely stated specification to the start of production, took us 1.5 years. This is theoretically quite a long time span for a small system design, but constraints on precise temperature regulation (+-0.10 C for twenty liters of water and up to thirty submerged eggs) made the project non-trivial.

The essentials of small system practice, based on a representative experience, Figure 1, and on previous work, are:

a) There is not much parallel work in the design process even though the paradigms of design address and favor parallel work in concurrent engineering. Activities unfold sequentially, and short delays in the development chain just get longer down the development path.

Figure 1. *Egg cooker, a small system design project*

Planning for mechanics, hardware and software design are parallel activities. Later on, mechanics and electronics hardware design do proceed in parallel, but the majority of software development needs a stable hardware platform as a precondition. Where project constraints are difficult to achieve (compared to the level of the developers' expertise), problems start showing in the integration phase, which is already too late. Such a scenario must be anticipated in advance so the project can progress as fast as possible to the next optimized iteration instead of being cancelled.

b) The project definition phase must not evolve as an open-ended process. Customers in a branch of small embedded systems are not familiar with the particulars of the profession, namely that most late functional changes imply a redesign (another hardware/software loop) rather than a small fix.

c) It is due to ubiquitous spread of PCs that cross platform development[1] is now practically the only way of development. In the early days of embedded programming, a dedicated autonomous Microcontroller Development System (MDS) had to be purchased for work with a particular microprocessor. That has changed – now a modern MDS usually performs as a PC extension card and just as another installable set of applications.

d) It would not be a productive course of action to master assembler language for a particular microcontroller and use it for programming of say, time or footprint-critical functions. Such an approach can easily lead to serious delays and associated expenses. ANSI C is a modern choice of language for embedded programming, since it is designed from the start to be flexible (including both powerful and bitwise operations), since compilers from ANSI C to practically any assembly language exist, and since one can learn ANSI C once and then use it for all subsequent occasions.

2.2. Practice in complex system design

The design of a wireless router for internet traffic, Figure 2, for instance, can represent a typical example of a complex embedded system.

Embedded software (hereinafter referred to simply as “software”) for complex applications is produced in collaboration with different teams and individuals in remote locations. That adds to the complexity of project organization and management but makes the whole project feasible.

From an economic point of view, a new product must get to market in the shortest possible time. This

[1] The software is developed on one platform and used on another.

Figure 2. *Wireless internet router, a complex system design project*

implies purchasing of subcomponents that already exist on the market, development of new components and integration of new and pre-existing components into a new whole, i.e., a new product. This approach may be least desirable from the point of view of the designer as "artist", but most effective for the marketing department in terms of fastest time to market. Such a product is intended to be fully functional in most of its behavioral aspects. As sales progress, the design department can elaborate on optimized versions. Internals of such versions are then structured by the nature of the application, rather than the availability of existing subcomponents. New versions refine current functionality and add new functionality; reliability gets built into the product and production costs get minimized in the optimization process.

Specialized technical journals (Kaven, 2005; Wu 2004) report on development of the product in Figure 2.. We learn that it was incrementally developed through eleven production versions (!). First one included an off-the-shelf PCMCIA card that performed the wireless communication and three off-the-shelf microcontrollers that performed routing of inter- and intranet packets. In eleventh, optimized version only two custom microcontrollers do all the work. Radio frequency (RF) wireless and routing functionality are seamlessly integrated in the optimized version. It is marketing department’s decision that the product should keep the same appearance through all versions.

The incremental approach to hardware design as outlined in this example implies a more incremental approach to software design. The manufacturer’s official web pages show as many as twenty-three sequential releases of the embedded software (Linksys, 2005) for this product.

At first glance, such an intense versioning scheme implies non-optimized usage of engineering resources. Besides, it implies sloppy design work to the uneducated observer. These seem to be logical claims need a short elaboration.

Hardware versioning decreased the street price of this product by thirty percent (from 130 USD to 90 USD). A short calculation confirms that it is worth spending some engineer-years of effort for such optimization if production volume amounts to some ten thousand units. As soon as optimizations involve customization of integrated circuits, then the months start passing by. Where that is the case, an alternative solution (if available) is used in the mean time.

Software versioning resolved some 77 functionality issues and added some 35 incremental contributions to the basic functionality (Linksys a, 2005). These numbers look significant and this, superficially at least, suggests cutting corners in the first designs.

It helps being somewhat familiar with complexity of internet routing and wireless communication to give respect to the designers who created and maintain this product. To illustrate, a current platform for the state of an art wireless router runs at a 200 MHz clock speed, with 32 MB of built-in RAM and 8MB of flash memory. These are the parameters of a personal computer built some seven years ago! Since the application of internet routing and of wireless communication is far from trivial (in the current technical sense) the designers decided to build the aggregate as a combination of a custom application and a standard OS. They chose the Linux OS as a software platform. They built most of the needed functionality in the form of a custom application that runs in the OS environment. Complete source code consists of about 4800 C files with about 30 functional lines on average for the OS and about 1800 C files with about 20 functional lines on average, that all amounts to about 180000 lines of code in C language.

Judging from the tangible properties of such a product, price (about one hundred USD), functionality (routing of wirelessly transmitted internet packets) and an appearance of a mass-produced device, it would be difficult to estimate the amount of work and the level of state-of-the-art technology that went into such an artifact. It is the aspect of mass production, where huge quantities of produced items compensate for enormous efforts and associated costs of development, that gives rise to such modern products of high complexity.

2.3. Most important decisions when shaping practice for a design of intermediate complexity

The two examples above illustrate design practices for moderate and highly complex products whose functionality stems from the operation of embedded software. The level of complexity is the most influential factor when shaping practice for a particular design. The most relevant decisions are taken in the following two areas:

Project management.

The level of application complexity influences decisions on project management to the greatest extent. Complexity implies long-term design efforts with associated expenses. Time wise, long-term designs are not favored in industry. To make the design process shorter, more engineers need to collaborate on the project. Issues of human collaboration and structuring the project for parallel development add further complications. It is a matter of overall project optimization to decide which measures to take to come to a successful design within some realistically estimated time span. Before the onset of the project, decisions are to be made about the amount of resources for research into feasibility issues, outsourcing to specialized companies, load balancing among existing specialists, temporary hires of focused specialists, project milestones and frequency of design reviews. Thorough and frequent design reviews (Trader, 2004) are critical in keeping non-trivial projects on track.

The actual design flow is to be worked out for each specific project. Large projects require teamwork and managers, and the project organization needs to enforce collaboration of different specialists for design and testing. Small projects can be most efficiently executed by a small group of specialists, even by one (Jenko, 2005).

Design with an OS or without it?

Complex applications cry out for the inclusion of an OS and for an associated powerful hardware platform. Less complex applications are designed faster, more simply and more cost-effectively without a built-in OS. It is a matter of optimization to decide whether to develop and code the application in its entirety or to decide on a platform where the OS yields elementary functionality and the application builds from there further on. The decision for a synergetic functionality of an OS and a lean application has advantages and disadvantages. The advantage is that the OS takes care of the basics. It schedules and runs processes, performs input-output (IO) functions and provides a rich set of ready-to-use functions on a high level, a so-called “application programming interface (API)”. As a general principle, usage of a platform with a built-in OS leads to shortening design time. Application portability gets simpler since the OS API is the application interface to hardware and not the actual hardware registers.

One disadvantage of OS usage is associated cost. Royalties for OS usage are to be paid for each item in production. When that presents a significant issue, a public domain OS can be used, such as a version of Linux, as in the example above. Such an approach is in favor of costs but calls for a higher level of expertise on the part of the developers. Usage of public software, or "as is" software, needs skillful developers with long-term conceptual and practical experience. Such developers do not exist in large numbers and they do not come cheap.

When designing with an OS, developers essentially need to master the microcontroller and the OS. In the other case, only mastering the microcontroller is involved.

A decision to design with an OS also implies usage of a more powerful platform.

2.4. Current evolution of design practices and design processes

The current level of maturity in structuring design practice is relatively high. As such, it implies space for mostly incremental progress only. Current practice suffices to successfully bring any feasible project to a successful end *in a given period of time while incurring a certain cost*. Incremental progress, i.e., evolution, takes place in different directions:

a) Practices like Extreme Programming (XP) (Grenning 2004) that systematically enforce teamwork, diffusion of knowledge from experts to less experienced but ambitious engineers, remote collaboration, critical design reviews, an open work space, coding standards, customers as design team members and others.

b) build-up and enforcement of practices for software portability and reuse (Stolper, 2004). Portability implies running non-modified code on different platforms. Software reuse implies usage of non-modified code in different applications.

c) Making a simpler coding process. Attempts are made to exchange some coding for visual programming and programming by patterns (Douglass, 2004), as is the case with modern tools for programming in workstation environments and networks.

d) better visualization and comprehension of behavioral functionality that needs to materialize. Universal modeling language (UML) visual tools and compilers from UML to C (and then further on to binaries) are appearing on a market (Skrtic, 2004).

e) better comprehension and better solutions for problems from the field of control systems theory. Continuing new releases of a tandem Matlab Simulink present a state-of-the-art tool for visualization, comprehension and achievement of solutions in control systems problems. The most tangible results, the work which a Matlab Simulink set-up can produce, are ready to use C coded functions that encapsulate just-developed and simulation-verified control algorithms.

3. DESIGN AND CODING OF EMBEDDED SOFTWARE WITHIN A PROJECT DEFINITION

Traditionally, the process of embedded software design for new products unfolds as follows: Phases of product definition, software design and coding, and product testing make the process. A written functionality specification ends the product definition phase. Then, hardware and software designers are supposed to work in parallel. Usually, there is a gap between software design and software coding since the electronics and mechanical parts need to be produced before the onset of coding. Ideally, there should be no iterations involved in the design and test phases. As the coding progresses, the hardware is simultaneously tested and, if the design is not too ambitious for the engineers at hand, the efforts gradually converge to a working prototype.

The objective is to determine the final behavioral properties of a new product at the end of the product definition phase, and to have working prototypes at the end of the product design phase.

The presented approach starts with designing and coding a fully functional model of a new product in the early stage of the product definition phase. The model evolves concurrently with product definition. The model serves at this early stage of the product life cycle as an effective discussion tool. Various potential features are easily demonstrated. Then, decisions on feature implementation are made and tested for acceptance in a functional model. A fully functional model is available at the end of the product definition phase. It is important that all parties involved in the new product definition have this executable model at hand to explore its functionality and to propose changes. The functional model defines the future product at the end of the product definition phase and at the onset of the product design phase. Internals of the fully functional model are built in such a structured way that they are reused as part of the actual embedded software that governs the actual product. The rest of the embedded software is built by well-established practice as the hardware becomes available. Integration of mechanics, electronics and software, and holistic testing of the whole, completes the software development effort.

We propose an internal structure of the functional model that permits reuse of most of the model's internals further on in the actual product development. Software reuse helps in adjusting the time of embedded software development to times that are needed to develop mechanical and electronic sub-systems. The essential points of the presented approach are:

- Conceptualization of concurrent development of a behavioral simulator and embedded software.
- Establishment of a framework for its realization.
- Design of a behavioral simulator, and of embedded software within the proposed framework.
- Implementation, i.e., coding.
- Integration of the embedded system.

4. CONCEPTUALIZATION

It is difficult for anybody to just sit down and write a detailed specification of a new product behavior in such a way that it would not be changed by someone at some moment within the design process. To address these boundary conditions of new product design and to get productivity of project collaborators to the next level, we suggest a new paradigm of embedded software design.

The requirements are: thorough product definition in the form of a behavioral simulator, short development time, effective budget usage, software reusability, platform independence of most of the software and concurrent development of software, electronics and mechanics.

In a project for a new product release we first introduce a visual behavioral simulator of a new product, which is used for discussions. Second, we structure the simulator internals into areas of simple functionality i.e., into components with simple and well-defined interfaces. Structuring is such that most of these components gets reused in the embedded software that controls the final product.

A component, Figure 3, is defined as a part of an application that is developed and tested independently and integrated into the application later through simple communication.

Mandatory component parts are:

- Input ports. There is no component data input except through component input ports.
- Output ports. There is no component data output except through component output ports.
- Configuration ports. Component configuration can be achieved through configuration ports only. Components can be configured during run-time or when compiled.
- Internal functions.

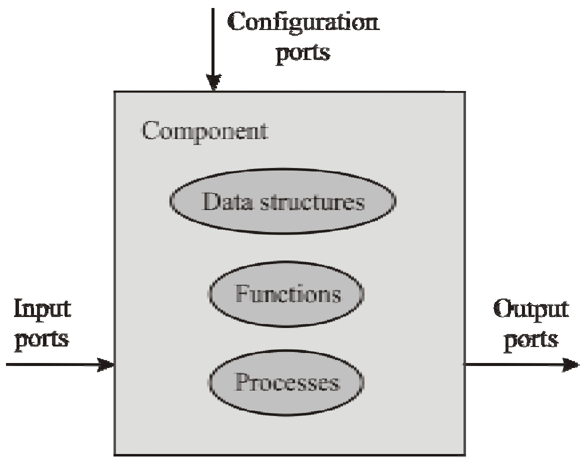


Figure 3. *Typical software component*

- Internal data structures.
- Processes. One process usually suffices, but more processes per component are not to be excluded from the component framework.

Based on functionality, we classify software components into a class of application-dependent components and a class of hardware-dependent components.

Software development is to be performed on a generic platform, i.e., on a well loaded workstation. The reason is that mechanics, electronics and software must be defined concurrently, and they must be designed concurrently to achieve a short development time. A target platform does not exist during most of the software development time. It is also a matter of convenience to work on an efficient and powerful generic cost-effective platform (compared to development system for target platform).

Implementation details for device drivers[2] have to be thought out while developing software on a generic platform. However, in this phase of software design the device drivers exist only on an abstract level since the hardware does not yet exist. Here, we borrow a principle of abstraction[3] from object-oriented languages (Stroustrup, 2000). Abstraction allows actual device drivers to be substituted for a functional equivalent that is feasible to produce in a behavioral model of the actual product. A class of hardware-dependent components exists for this purpose.

Adhering to this approach results in the following benefits:

- The help of a behavioral simulator is available through most of product definition phase.
- Most of embedded software is platform-independent. Porting functional components from the behavioral simulator to embedded software that controls the product is a matter of using different compilers. Switching from one microcontroller to another later in the product life cycle mostly implies usage of a new compiler.
- The organized methodology of designing and coding the behavioral model at an abstract level and then only having to re-code the hardware details is very efficient from a code reliability point of view.
- Last but not least, embedded software is reusable. When applied to a new target system (a new microcontroller), most of software is to be recompiled only.

5. FRAMEWORK

The framework consists of conventions for building platform-independent software components on a generic platform[4].

What these software components have in common with Object Oriented Programming (OOP) is that the functionality is enclosed (encapsulated) in a shell, i.e., in a component or object with well defined interfaces.

Regarding encapsulation, this is the most important part of the OOP paradigm, and it is built into tools for OOP. Encapsulation in component software is achieved by means of self-discipline only. Compilers for OOP in embedded system programming are more an exception than a rule. Component software is coded in strict ANSI C, which is a common ground among different compilers for different platforms.

Programming of software components does not rely on inheritance and polymorphism, which are essential concepts of OOP.

The presented framework proposes a generic platform usage not only for design and coding, but also for building a fully functional model for product definition and for code verification.

Besides, and this is new, we propose such an internal structure of the functional model that gives reuse of most of simulator’s internals further on in the actual product development.

This approach was so beneficial for successful design in our projects that it ought to become a common practice in many designs after it gains full recognition.

5.1. Design

A typical appliance needs to handle the following functionalities:

- apparatus appearance, i.e., user interface (buttons and displays),
- control of the process that the appliance performs,
- process variables monitoring,
- governing the actuators, and
- self-monitoring for alarm conditions.

[2] Device drivers directly address hardware registers that define particulars of the hardware functionality.

[3] Abstraction: we can implement a component in several ways. It is important that a user doesn't need to know how we do it. As long as we keep the interface unchanged, a user is not affected by the component internals.

[4] A platform is the specification of an execution environment for a set of models. Examples of platforms include operating systems like Linux, Solaris and Windows, the Java platform, specific real-time platforms and hardware platforms without an OS, where application software is the only software in the system.

These functionalities are encapsulated in different components in a case study – the design of the industrial kitchen fryer in Figure 4. Application-dependent components (ADC) always reflect the nature of the actual application. It is about frying food in the application in Figure 4. Were it about a more complex application, there would be other components and more components where interrelations and interfaces were of higher complexity. The hardware-dependent components (HDC) in Figure 4 have nearly similar interfaces and functionality in any application. They are designed for simple interfaces that represent different microcontroller ports and sub-systems on an abstract level. The interfaces are simple by design, to ease the exchange of model HDCs for the HDCs of a target system.

5.2. Coding

Macros are defined for the purpose of working in the same integrated design environment (IDE) while programming the model and the target application.

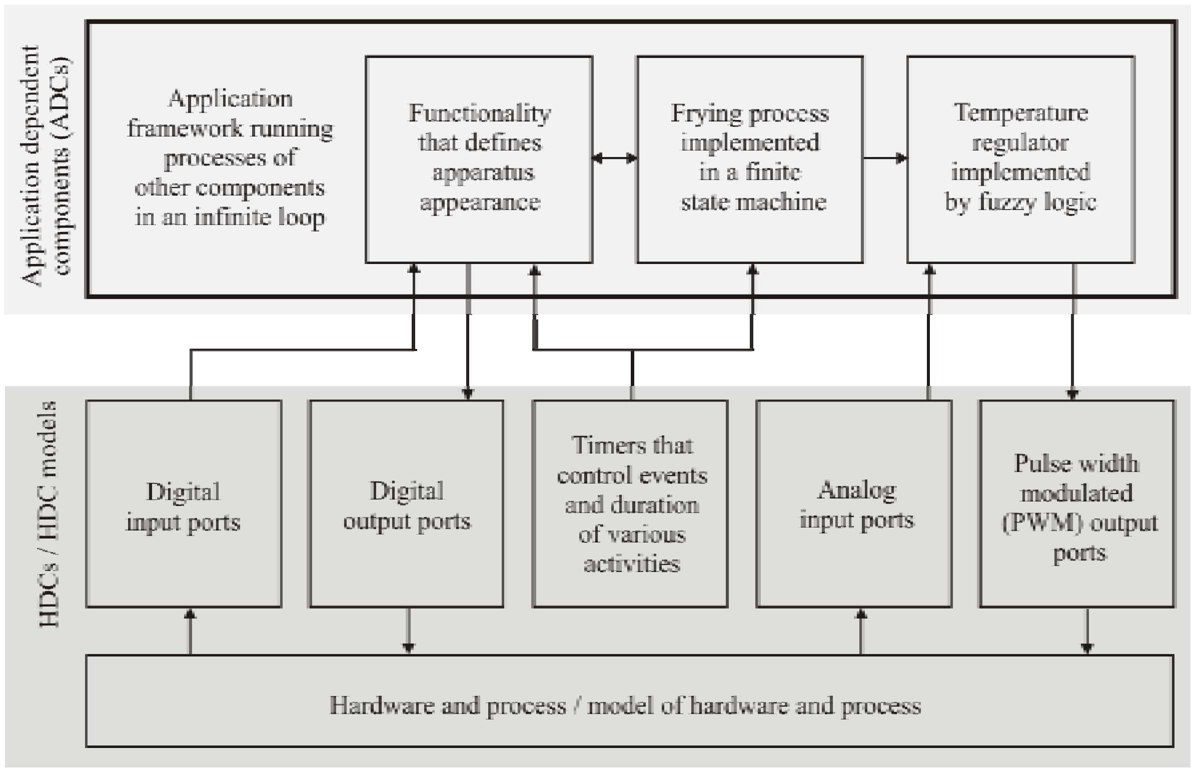
```
#ifdef _MODEL

#define FUNCTION() MFunction() // a model function

#else   /* __TARGET__ */

#define FUNCTION() Function() // target system function
```

Then, macros are called only:

```
FUNCTION();
```

Switching between the target and generic platform project is a matter of *_MODEL* definition only.

Coding of a visual model starts near the onset of the project feasibility study. The model’s internal structure, consisting of interfaces and components, is less than perfectly enforced in the first model version. Weak structuring is not an issue at this stage since the model is mainly used in discussions. As product definition grows to a mature level, the model has lived yet through most of its functional and visual modifications. At this stage, internal


Application dependent components (ADCs)
Application framework running processes of other components in an infinite loop
Functionality that defines apparatus appearance
Frying process implemented in a finite state machine
Temperature regulator implemented by fuzzy logic
HDCs / HDC models
Digital input ports
Digital output ports
Timers that control events and duration of various activities
Analog input ports
Pulse width modulated (PWM) output ports
Hardware and process / model of hardware and process

Legend: ADC: Application dependent component, HDC: Hardware dependent component

▢ Functional model ADCs materialize 80 percent of a functional model code. They are reused (recompiled) for microcontroller ADCs that build 60 percent of a microcontroller code.

▨ HDCs, a model of hardware and process materialize 20 percent of a functional model code. They are replaced by mechanics, process properties and microcontroller HDCs, which build 40 percent of a microcontroller code

Figure 4. *Component structure of the model and of software for the product*

structuring becomes important since the level of future modifications is supposed to remain low.

Coding of hardware specific components starts as soon as the hardware is built. Hardware-dependent components are coded and verified sequentially in small stand-alone projects. This way, a low-end MDS with limited source-level debugging capabilities suffices for development. This is important from the project budget perspective. Practically, different microcontrollers need different MDSs, and the fixed costs of a design firm just add up.

Component-based software design enables thorough verification at each step of the implementation, i.e., at each step of the coding process. Each component is thoroughly tested. Simple test environments are written for this purpose.

Component verification is fast, since components are self-contained, and their interfaces are well defined.

The difference between application and component verification is subtle. The smallest parts of components are code lines. The smallest parts of application are components, i.e. complex code lines, already tested and verified. Such, *per partes* verification, is more reliable than verification of a whole at once. Additionally, it is also simpler.

5.3. Work with an MDS

Only the hardware-dependent components need to be tested and verified on an MDS. The same is true at the end for the whole application. The approach minimizes work with MDS.

5.4. Integration

At the onset of the integration phase, components are verified to be functional.

The gist of integration is enforcement of a thorough abstraction for interfaces of hardware-dependent components. As both flavors (model and target) of these components start performing identically then the application runs in the target environment in exactly the same way as in the model. This is a criterion of successful integration.

5.5. Verification and testing

Verification is a process that confirms that the embedded system performs in accordance with expectations. A well thought out verification scenario is a basis for verification procedures. It is our experience that it is the best for small and medium complexity of an embedded system, when the designer himself works out the verification scenario.

Testing is about verification of each product on a production line. It is the same software that runs all product instances. As the software gets verified for proper functionality it does need individual testing on each product. Testing needs to confirm that the hardware performs properly. Hardware testing is less complex than software verification.

6. CASE STUDY

Figure 4 shows the internal structure of the case study. It consists of about 8000 lines of ANSI C code encapsulated into three application-dependent and five hardware-dependent components. Figure 5 shows a corresponding functional model of the product. Figure 6 shows the final product. The reader will notice that the user interface of the product and the model are of the same form. They have identical functionality. The same is true for the whole product and for the whole model. The model and its condensed users' manual are available for readers' evaluation (see a link in the Appendices).

7. RESULTS

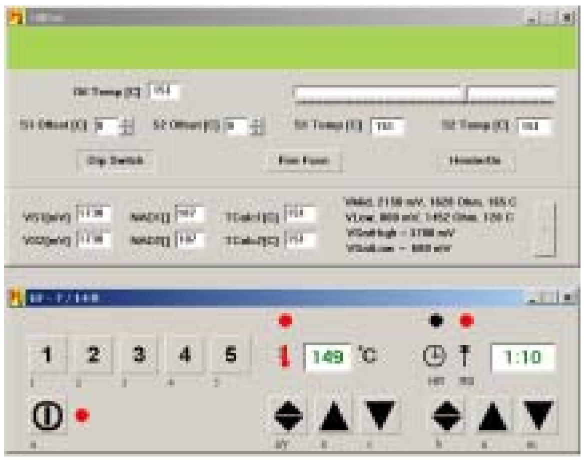


Figure 5. *Visual behavioral simulator of the product*

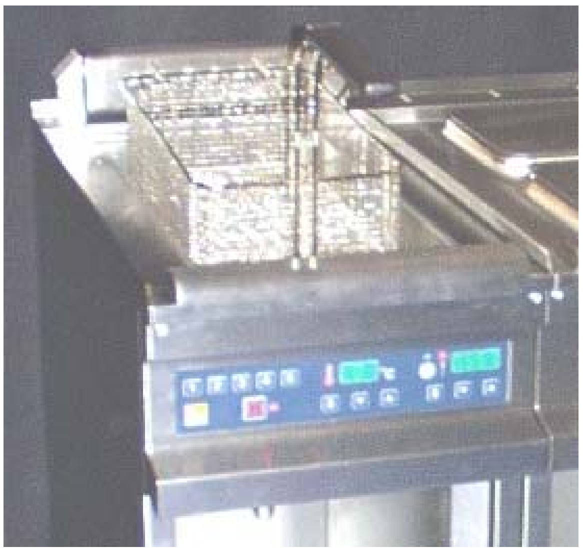


Figure 6. *Finished product*

About 60 percent of the code (application-dependent components) for the final product were designed and written within a week when starting with a solid project definition. The amount of effort spent on the remaining 40 percent of the code depends mostly on a previous experience with a particular microcontroller. Speculating for state-of-the-art familiarity with a particular microcontroller, hardware-dependent components were coded and verified within two weeks. The presented design and coding procedures result in a simple integration of application- and hardware-dependent components into a final embedded system.

8. DISCUSSION

Simulation and modeling are key tools in product development, particularly in control system development. They provide the means for problem understanding and performance evaluation. Simulation is part of the development process at many points. It has been used for a long time in different techniques and problem areas such as finite element analysis for mechanical systems, circuit analysis for electrical systems and discrete event type analysis for optimizations in manufacturing. Behavioral modeling of the entire product adds to already existing simulations.

Two factors lead us to the presented software design by behavioral modeling: one is conceptual and one is sociological.

- The conceptual factor is the power of abstraction. One of the most creative ways to approach the distillation of application concepts is through a strategic use of abstraction. The elegance of abstraction is that it distills the application to its root concepts while isolating it from the details of system implementation.

Decoupling things into their separate specific identities builds quality into a design process. Physical interfaces are abstracted to logical interfaces. This makes the exchange of hardware-dependant components from a model to a target platform straightforward.

- The sociological factor is the need for an additional communication domain that helps different parties in the project definition phase come to closure. Non-engineers are not supposed to like programming jargon; they don't have to be enthusiastic about discussing UML models. Instead, they discuss and judge a functional model. Sociology professionals claim that functional models are more geared toward communicating with people than with machines. This way, a functional model serves both worlds, since its internals constitute pure machinery, made by design to be reused in the actual product without modifications to the greatest possible extent.

Hardware modeling in the functional model is very compact since it is based on a visual components library, which is an essential part of any modern visual set of tools for workstation applications. Most of the functionality of the visible final objects is inherited from the visual components library. Inheritance, one of the three foundations of object programming (along with encapsulation and polymorphism) gives us practically for free what could take months when coding model visual objects from scratch. The presented approach to embedded system design would be unrealistic without modern libraries of visual objects – the most essential part of any modern workstation programming tool.

9. CONCLUSIONS

A new concept of concurrent development using a component-based visual behavioral simulator, and of component-based embedded software is presented. This concept contributes to embedded software design and to product definition.

Contributions to embedded software design are:

- Structuring an application into components enforces precise interfaces that are a precondition for ease of debugging, maintenance and upgrading.
- Software is designed in a luxurious graphical workstation environment. Most of it is portable without modifications to the target environment.
- If a need for a different microcontroller arises within a product’s lifetime, software portability is not straightforward, but simplified to the greatest possible extent.
- Project costs are kept as low as possible. Design tools consist only of a capable workstation packed with visual tools and non-ambitious MDSs for different microcontrollers. The classical alternative was an ambitious MDS for each project with a different microcontroller.

Benefits to the product definition are:

- A visual behavioral simulator of a new product becomes available in the early stage of product definition. The visual behavioral simulator is an essential part of developer / management / marketing / customer relationships. Unclear issues are settled on the behavioral model, and not on the iterated product design.
- A fully functional visual behavioral model crisply defines the interface between product definition and product design.
- More than half of the control software is written in the product definition phase while designing and coding the functional model. This yields precious time that is later used in the product design phase for unanticipated activities that are not known in advance.
- Programming progress at the onset of the project is more rapid than the customer might expect. This tends to make the customer

confident about the project’s success and more open to discussion and mutual work on the design spec. Being able to contribute to the specification can later simplify its implementation.

ACKNOWLEDGMENTS

This work was performed within the program of the "Virtual Research Lab for a Knowledge Community in Production (VRL KCiP)" Network of Excellence, Contract Number NMP2-CT-2004-507487, within the EU 6th Framework Program.

APPENDICES

The functional model for the case study and its one-page users’ manual can be downloaded and evaluated from http://www2.arnes.si/~supmjenk.

References

[1] Douglass B, Architecting systems with patterns, Spring embedded systems conference, San Francisco, CA, 2004

[2] Grenning J., Extreme programming and embedded software development, Spring embedded systems conference, San Francisco, CA, 2004

[3] Jenko M., Statistical evaluation of variables in precise process control. Case study: MSP430 based apparatus for pasteurization of soft boiled eggs, accepted for Texas Instruments advanced technical conference, Landshut, December 6-8, 2005

[4] Kaven O, Linksys sweeps all three categories, PC Magazine, issue of June 7th 2005

[5] Linksys, www.linksys.com, wrt54gs, wrt45g, firmware download pages

[6] Skrtic M., State machine graphical design for embedded systems, Technology news, Industry articles about state machines, 2004, www.iar.com

[7] Stewart D. B., The Twenty- Five Most Common Mistakes with Real-Time Software Development, Proc. Embedded Systems Conference, San Francisco, CA, 2004

[8] Stolper S. A., Designing portable software, Spring embedded systems conference, San Francisco, CA, 2004

[9] Stroustrup B., The C++ programming language, special edition, AT&T, 2000, p. 30

[10] Trader M. T., Embedded software inspection overview, Spring embedded systems conference, San Francisco, CA, 2004

[11] Wu X. M., Linksys WRT54GS wireless G broadband router, CNET, issue of March 1st 2004