Design and Implementation of an Object Oriented and Real-Time Microkernel for Embedded Systems Using Design Patterns

Kasim Sinan YILDIRIM Aylin KANTARCI

Computer Engineering Department, Ege University, 35100, Bornova, İzmir, Turkey
sinan.yildirim@mail.ege.edu.tr
aylin.kantarci@ege.edu.tr

Abstract. Many embedded applications need a real-time and embedded operating system for an easy operation environment. In contrast to general purpose operating systems, real time embedded operating systems must be configurable and restructurible in accordance with application requirements. In this study, the properties and requirements of real-time embedded systems have been examined from the viewpoint of software engineering concepts. Within this context, eGe Gomulu Isletim Sistemi (eGIS – eGe Embedded Operating System), which is a portable, real-time, embedded, object oriented and configurable operating system, has been designed and implemented. In this paper, the developed system has been introduced and the design patterns used in the implementation of the embedded operating system eGIS have been explained and discussed.

1 Introduction

Software development for embedded systems has many different requirements than that for traditional operating environments. For example, embedded systems offer limited amount of resources to the applications. In addition, most of the software developed for embedded systems have real-time restrictions. Therefore, general purpose operating systems fail in fulfilling the demands of embedded applications. What is needed is a real-time operating system that can be customized, reconfigured and even restructured easily in accordance with the requirements of specific applications and application domains [1].

It is difficult to extend and configure many current real-time embedded operating systems (REOS). Addition of new functionality makes such operating systems more complex and harder to maintain. In addition, highly coupled modules and interfaces make it hard to modify and understand the system. A well defined, loosely coupled, extensible and flexible architecture is necessary for modern REOS. Reusability of system components, portability, scalability and reconfigurability are the key concepts in the design of modern operating systems [2]. In order to fulfill these requirements, modern software development methods and concepts must also be employed in the design of modern REOS.

Design patterns are one of the modern software engineering tools that can be used in the construction of modern REOS. Design patterns refer to descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context and they offer powerful and generalized designs that can be reused on various software systems [3]. Applying design patterns to software

systems increases software quality by improving distributability, reusability, portability, extensibility and maintainability of the system being developed [4].

Many of the design patterns depend on the concept of **interface inheritance** rather than **class inheritance**. Class inheritance defines an object's implementation in terms of another object's implementation. On the other hand, interface inheritance describes when an object can be used in the place of another. An object's interface does not include any implementation details and different implementations of the same interface by different objects make them interchangeable with each other. Consequently, clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect [3].

Another important point to emphasize is that, design patterns favor **object composition** over **class inheritance**. In class inheritance; subclasses reuse the code of parent classes and they are dependent on the implementations of their parent classes. Any modification on parent classes affects subclasses. In contrast, object composition is a *black box reuse*. The added functionality is gained by composing another object. The compositor is unaware of any implementation details of the composed object; it only knows their interface. Internal changes on the composed object have no effect on the compositor. In addition, an object of the same type can be replaced by another without any change on the compositor. Furthermore, systems that are designed using class inheritance generally grow into huge class hierarchies that are unmanageable and hard to change. On the other hand, object composition results in a clean object oriented structure [3].

Design patterns can be categorized according to the design problems they solve. ***Creational*** design patterns make a system independent of how its objects are created, composed and represented [3]. For example, *Singleton* and *Abstract Factory* creational design patterns abstract the instantiation process of concrete classes. ***Structural*** design patterns are concerned with how classes and objects are composed to form larger structures and they mostly depend on object composition. *Bridge* and *Facade* are the examples of design patterns in this category. *Bridge* decouples abstraction from its implementation. *Facade* defines a higher-level interface. ***Behavioral*** design patterns are concerned with algorithms and assignment of responsibilities among objects. For example, *Strategy* behavioral design pattern abstracts algorithms and makes them interchangeable [3].

Using design patterns in the design of REOS makes it easier to structure the operating system in accordance with the application needs. For example, applying design patterns to the operating system design results in that, a subsystem that fulfills application requirements best can be substituted with the existing one easily. Similarly; portability, a key concept in embedded operating systems, can be provided by using structural design patterns. Additionally, resource management can be performed in a flexible manner by switching between various algorithms that have been implemented with using behavioral design patterns [5].

Many embedded operating systems that have been designed by using modular languages such as C are hard to extend. For example, Ucos-II [6] has a monolithic structure and has not been designed by using modern software approaches. Ucos-II implements priority based preemptive scheduling. In order to change this scheduling algorithm, many parts of the operating system must be recoded. Therefore, structuring Ucos-II according to application requirements is difficult [5].

Many object oriented embedded operating systems are based on class inheritance and they include a huge class hierarchy. For example, eCOS [7] embedded operating system has been implemented using C++ object oriented language and can be configured in accordance with the application requirements. However, eCOS does not have a well defined class hierarchy and its architecture is not based on design patterns. CHORUS [8] and PURE [9] depend on a huge class hierarchy which makes their architectures hard to maintain, modify and extend. On the other hand, EPOS [10] has a configurable structure in which implementations and interfaces have been separated; but its architecture, like eCOS, is not based on design patterns.

In this study, in order to eliminate the weaknesses of existing REOS, a real time, object oriented and embedded operating system *eGe Gomulu Isletim Sistemi*, eGIS, has been designed and implemented. eGIS is a configurable, customizable and portable operating system owing to its architecture based on modern software engineering concepts, especially design patterns. This paper is organized as follows: The architecture of the developed system is introduced in Section 2. Most important design patterns used in eGIS system are discussed in Section 3. Implementation and experimental results are presented in Section 4. Finally, Section 5 is the conclusion.

2 eGIS System Architecture

The most important feature of eGIS is that it has an extensible, customizable and portable object-oriented architecture and that it has been designed to operate on embedded systems with real-time restrictions. eGIS has a **microkernel architecture** [11] which is based on design patterns and is portable in the sense that it operates on different hardware and software platforms. eGIS is composed of three subsystems that fulfill process management, concurrency management and interrupt management services, respectively.

The layered architecture of eGIS is shown in Fig. 1. eGIS includes a *C++ System Interface*, *Subsystems, Infrastructure* and *Platform* layers. The main interfaces zand abstractions are defined in the *Infrastructure* layer and implemented in the *Subsystems* layer. For example, the main process abstraction and related interfaces have been defined in the *Infrastructure* layer; whereas their implementations according to the application requirements have been implemented in the *Subsystems* layer . The *Platform* layer implements the platform specific interfaces and abstractions and provides platform independency. None of the implementations in the *Subsystems* and abstractions in *Infrastructure* layer are dependent on any specific platform. Finally,

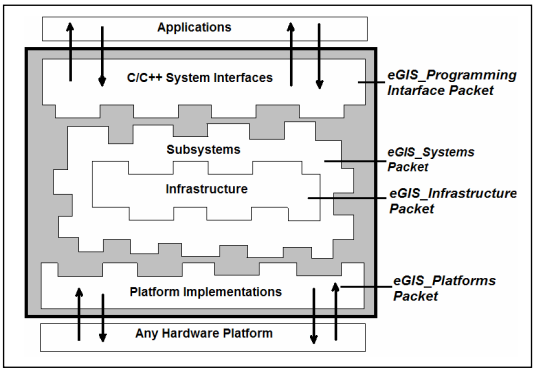

Fig. 1. The layered architecture of eGIS

the *C++ System Interface* provides a C++ API for embedded applications. The developed system interface enables the embedded applications to be isolated from the below layers. Hence, modification of the embedded application is avoided in case of any change in the underlying layers of eGIS.

The internal structure of eGIS microkernel and its interaction with external system components are shown in Fig. 2. Interrupt, process and concurrency management abstractions have been implemented in *Ani*, *Zigana* and *Efes* subsystems, respectively. *Zigana* is the process management subsystem and implements priority based preemptive process scheduling algorithm. *Zigana* provides services for creating, starting, stopping, blocking, unblocking and terminating processes. *Ani* is the interrupt management subsystem and it implements a simple and efficient interrupt management algorithm. *Ani* provides services for registering / unregistering interrupt service managers and basic interrupt handling mechanisms. *Efes* is the subsystem that manages concurrency objects by providing services for creation of mutex and semaphores, locking and unlocking mutexes, signaling and acquiring semaphores.

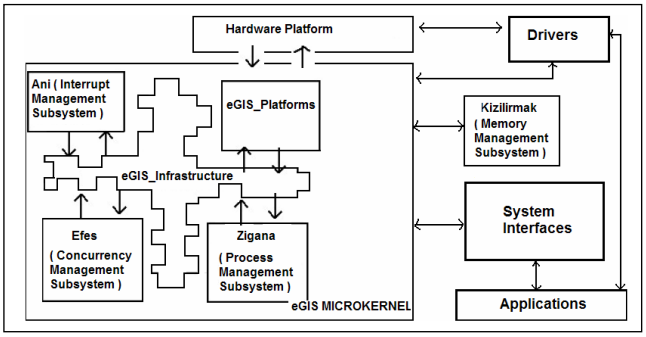

Fig. 2. The internal structure of eGIS Microkernel and its interaction with external components

As seen in Fig. 2, the subsystems are completely isolated from each other and the underlying hardware platform. Consequently, in the development phase, these subsystems have all been tested independently on a PC platform. Using a microkernel architecture that is based on interface inheritance and object composition improves the extensibility and scalability of eGIS. Addition of a new feature to the operating system is no more than registering a new subsystem to the microkernel. New subsystems, that share the same interface with different implementations, can be registered to the microkernel easily and can be interchanged with existing subsystems. For example, *Zigana* process management subsystem, which implements a priority based preemptive scheduling algorithm, can be replaced by a different subsystem that implements a simple round-robin based process management policy. Furthermore; in eGIS system, interfaces and their implementations have been separated and a clean object oriented architecture has been obtained.

As shown in Fig.2, eGIS communicates with ***Embedded Applications, Drivers*** and a ***Memory Management Subsystem. Embedded Applications*** are unaware of any internal details about the system structure, subsystems and even underlying hardware. They only need to know the programming API that resides in the system interfaces layer. ***Drivers*** have not been implemented in the context of eGIS, they have been implemented independently and they use the services provided by eGIS microkernel. The ***Memory Management Subsystem***, *Kizilirmak*, has been implemented independently outside of the microkernel. *Kizilirmak* implements sequential fit

memory management algorithm. A memory manager subsystem that is more efficient and suitable for real-time systems may be replaced with *Kizilirmak*.

As a result of the layered microkernel architecture of eGIS; portability, extensibility, maintainability and testability of the general system are highly improved. This results in a real application-oriented operating system [12]. It should be noted that the main structure of eGIS is based on the design patterns. Design patterns have been applied to the points that may be changed in accordance with the application requirements. The design of eGIS favors object composition and interface inheritance over class inheritance which results in a clean and reusable object-oriented architecture. The following section explains how design patterns have been used in eGIS and discusses the gained advantages.

3 Integration of Design Patterns to eGIS Operating System

Design patterns used in the eGIS operating system deal with the internal structures of the subsystems and they can be classified into three groups: creational, structural and behavioral design patterns.

Singleton [3] and ***Abstract Factory*** [3] creational design patterns abstract object creation and make eGIS independent of the underlying representations and implementations of the objects. Creation of objects using their concrete class names implicitly decreases system configurability. To prevent this situation, creational dependencies on concrete classes must be minimized in the system.

Singleton design pattern guarantees that the classes that need to have a single instance, have only one unique instance within the system and the creation of this single instance is isolated from the clients . An example class which is based on the Singleton design pattern is the *eGIS_Microkernel* class. Subsystems register the *eGIS_Microkernel* object in a centralized way. Consequently, this single microkernel instance must be accessed in a controlled and well-defined way and clients must be decoupled from the creation of this object.

Abstract Factory creational design pattern is used to create objects in eGIS system without declaring their concrete classes, making them independent from the concrete types. An example usage of this pattern in eGIS system is shown in Fig. 3. Platform specific process context data and necessary process context switching codes are implemented in classes that extends *plt_eGIS_ProcessContext* interface. Classes that implement *eGIS_Process* interface are only aware of *plt_eGIS_ProcessContext* interface. Underlying hardware platform dependencies are embedded in platform specific classes. However, platform specific classes must be created somewhere in the system. If the creation of these classes is embedded in classes that implement the *eGIS_Process* interface, there occurs a

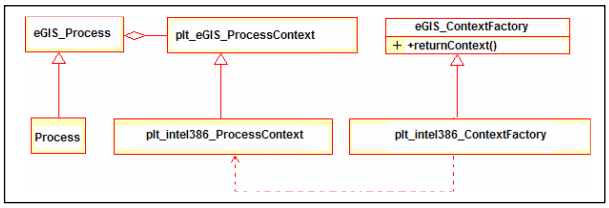

Fig. 3. The usage of Abstract Factory in eGIS

dependency on platform specific implementations. To avoid this interobject dependency, platform specific classes have been created in the context of classes that implement *eGIS_ContextFactory* interface. Consequently, eGIS system is completely independent of the creation of the concrete classes. Client objects within eGIS system are unaware of the concrete platform objects, they only know their interfaces.

Structural design patterns in eGIS system are used to combine objects and classes to form larger structures and to compose objects to obtain new functionality. As a result, it is easy to extend and to port eGIS to new architectures. In addition, interobject dependencies between objects have been eliminated. The main structural design patterns in eGIS are ***Bridge*** and ***Facade*** [3]. *Bridge* pattern abstracts platform specific operations and separates their implementations. Fig. 4 shows an example usage of this pattern which is based on its object composition functionality. Objects that implement the *plt_eGIS_ProcessContext* interface include platform specific elements such as registers and switching code between processes. Process objects that implement *eGIS_Process* interface compose objects that implement *plt_eGIS_ProcessContext* and they are unaware of the implementation of the platform specific context. The composed object may have a process context implementation for MIPS architecture as well as Intel i386 architecture. Consequently, the process objects are isolated from their platform specific context and this improves the portability of eGIS. In order to port eGIS to another hardware platform, new implementations of the target platform must be implemented and integrated with the operating system code at compile time. No other modification is necessary.

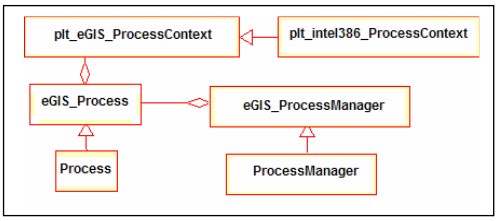

Fig. 4. The usage of Bridge in eGIS

eGIS includes many classes and abstract interfaces. The objects instantiated from these classes have communication and interactions with each other. However, clients that use the services of eGIS system must be unaware of this complex system structure. The system must have a general and simple interface. ***Facade*** [3] structural design pattern introduces a high level and simple interface for clients in eGIS system. Fig. 5 shows how *Facade* is used to provide a general programming interface for embedded applications. As a high level interface for application programs, *eGIS_SystemInterface* hides the internal structure and interactions from clients. Clients that use system API only interact with *eGIS_SystemInterface*. Consequently, the clients are independent of the subsystems and a clean and simple system interface is provided. In addition, clients communicate with fewer objects.

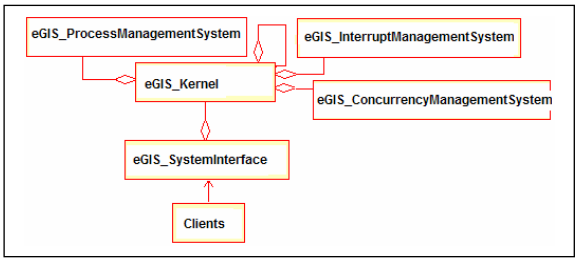

Fig. 5. The usag of Facade in eGIS system

Behavioral design patterns used in eGIS system prevents dependencies on algorithms and defines object responsibilities. ***Strategy*** [3] is the main behavioral design pattern used in eGIS operating system. Different system management algorithms can be adapted to eGIS system easily with the use of *Strategy* pattern. This results in a great flexibility which allows for restructuring the operating system in order to change the management algorithms without affecting the rest of the system. A suitable implementation of the algorithm interface that fulfills application requirements best can easily be linked with operating system code and can be made ready to run. Fig. 6 shows the extensibility gained with the use of Strategy pattern in eGIS system. Every process scheduler in eGIS must implement the *eGIS_ProcessManager* interface. The clients that use the services of the process scheduler are only aware of the abstract process scheduler interface; they are not coupled to the algorithmic implementations in process managers. Different process management algorithms can be integrated to the eGIS system by only replacing the existing classes with their new implementations without affecting the rest of the system. This results in the independence of eGIS from the process scheduling algorithm. In addition, separating implementations and interfaces makes classes that have algorithmic implementations more reusable.

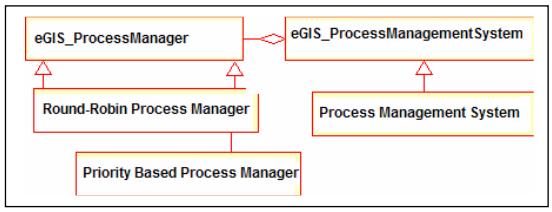

Fig. 6. The usage of Strategy in eGIS

To summarize, patterns used in eGIS make entire architecture more extensible and flexible as well as more changeable. As a result, eGIS can easily be structured in accordance with the application requirements. In addition, eGIS can easily be ported to different execution platforms easily. The overhead of using design patterns will be less important on current powerful embedded processors. Also, the improvement in hardware and memory speeds and decrease in the cost of hardware will make eGIS much more advantageous than the traditional embedded software systems.

4 Implementation and Experimental Results

eGIS has been implemented in C++ using Gcc C++ compiler [13], Gdb debugger [14] and Bochs i386 PC emulator [15]. So far we have ported eGIS to i386 platform and a port to Mips32 architecture is in progress. At the time being, eGIS microkernel consists of 3 subsystems with more than 60 classes in total. An important point to be noted is that eGIS needs a small memory of 32K for text and 4K for bss segments.

The main entry point of eGIS is shown in Fig. 7. The subsystems that implement process management and interrupt management services are registered to the microkernel and objects that implement platform specific requests are also associated with the related subsystems. As it is seen, the flexibility of pattern-oriented microkernel architecture facilitates reconfiguration of the system in accordance with application specific requirements and restrictions. This configuration is performed

through the selection of subsystems at system configuration time without affecting other parts of the system. Porting eGIS to a new architecture is no more than registering related objects that have platform specific implementations to the subsystems.

```
/**
 * eGIS System Entry Point
 */
int main()
{
    ProcessManager processManager;
    InterruptTable interruptTable;
    ProcessManagementSystem processManagementSystem;
    InterruptManagementSystem interruptManagementSystem;
    plt_intel386_ContextFactory contextFactory;
    plt_intel386_Interrupt plt_Interrupt;

    /* Register Zigana process management subsystem */
    processManager.setPlatform(&contextFactory);
    processManagementSystem.setProcessManager(&processManager);
    eGIS_KERNEL->setProcessManagementSubsystem(&processManagementSystem);

    /* Register Ani interrupt management subsystem */
    interruptManagementSystem.setInterruptTable(&interruptTable);
    interruptManagementSystem.setPlatform(&plt_Interrupt);
    eGIS_KERNEL->setInterruptManagementSubsystem(&interruptManagementSystem);

    interruptManagementSystem.initialize();
    processManagementSystem.initialize();

    systemInitialize();
    eGIS_KERNEL->start();

    while(1);

    return 0;
}
```

Fig. 7. eGIS system entry point

Fig. 8 shows a general thread interface for embedded applications in eGIS. Threads in eGIS system are represented with *eGIS_Process* interface internally. On the other hand, a more general and clean thread interface *e_Thread* hides internal operating system data from applications. Every thread in eGIS must implement e_Thread interface and override pure virtual *executionMethod* method. With such system programming interfaces, embedded applications only know these services and internal complexity of eGIS is hidden. Applications are isolated from any internal modification on the structure of eGIS, They do not know which subsystems have been registered, even the hardware platform that they are running on.

```
class e_Thread : public e_Object
{
    public:

        e_Thread(Priority_t priority);
        virtual ~e_Thread();

        /* the general entry point for all threads */
        static void *GeneralEntryPoint(void *parameter);

        void start();
        uint8_t changePriority(Priority_t priority);

        virtual void executionMethod() = 0;

    protected:

        ThreadId_t id;
        eGIS_ThreadProperty threadProperty;
};
```

Fig. 8. eGIS application thread interface

The overhead introduced by using design patterns in some basic services of eGIS has been presented in Table 1. These results have been collected on Bochs emulator. For example, accessing the single instance of eGIS_Microkernel class takes only 12 clock cycles. Except for the parts of the system that have hard real time restrictions and in which efficiency is quite important; this pattern can be used safely. Process scheduling is another system block that *Bridge* pattern has been used. Deciding which thread to run next and switching to this thread takes approximately 168 clock cycles. In the context switching operation, the request is handled through the *Bridge* layer and 29 clock cycles has been spent to delegate context switch request to the relevant platform context objects. If we consider the separation of implementations and interfaces, achieved portability and clean object structure with the usage of *Bridge*; 29

clock cycles will be quite reasonable. *Facade* layer is the entry point of all the requests of embedded applications. A thread creation request of an application first passes through the *Facade* layer and then it is delegated to the related subsystem. Total thread creation is performed in approximately 1146 clock cycles. The request is prepared in *Facade* layer and then delegated to the process management subsystem in 234 clock cycles. It should be noted that all the operations in *Facade* layer are not specific to this design pattern. For example, some operations such as passing the thread creation parameters to the eGIS microkernel would have to be performed even if the Facade pattern were not used. Due to the inefficiencies of Bochs emulator, a fine grain overhead measurement has not been possible. Therefore, we can say that overhead of Facade is bounded with 234 clock cycles.

Table 1. Experimental results on Bochs emulator for i386 architecture

Pattern	***Operation***	***Total clock cycles***	***Cycles spent in the pattern layer***
Singleton	Accessing eGIS_Microkernel class instance	12	12
Bridge	Scheduling next process to run and switching to it	168	29
Facade	Thread creation	1146	< 234

The performance results in Table 1 indicate that the overhead introduced with design patterns is tolerable for many applications. As the current trend in increase in the speed of embedded microprocessors continues, the overhead will be less important.

5 Conclusion

In this study, a new real-time and embedded operating system, eGIS, has been implemented by using design patterns. eGIS is a flexible application oriented operating system in the sense that it can be used in various systems with independent and distinct requirements. During the implementation of eGIS, many problems of the current embedded real-time operating systems have been eliminated by using an architecture based on design patterns. Employment of design patterns resulted in the extensibility, interoperability, platform independency and algorithmic independency of eGIS. The problems solved by the design pattern oriented architecture of eGIS can be summarized as in the following:

- Creation of objects using concrete class names implicitly has been avoided with the usage of *Singleton* and *Abstract Factory* creational design patterns.
- *Bridge* structural design pattern provides portability and eliminates inter-object dependency.
- *Facade* structural design pattern hides internal structure of eGIS from embedded applications and provides a clean and a simple programming interface.
- *Strategy* behavioral design pattern decouples eGIS from management algorithms, allowing for interchangeability.

As a future work, we plan to add new subsystems such as a Ram file system, a TCP/IP stack and an embedded graphical user interface engine to eGIS. Secondly, a

more suitable and efficient memory management subsystem will be replaced with *Kizilirmak* and new process and interrupt management algorithms will be implemented. Thirdly, the design of current subsystems and internal structure of eGIS will be improved by applying *refactoring*, which refers to a modern software engineering technique that eliminates unnecessary inheritance and composition relationships between classes. Moreover, we will run eGIS in real operating platforms with real embedded applications.

References

1. Friedrich L. F., Stankovic J., Humprey M., Marley M., Haskins J.: A Survey Of Configurable, Component-Bassed Operating Systems For Embedded Applications, IEEE Micro, vol 21, Issue 3, pp. 54-68, (2001)
2. Gien M., Next Generation Operating System Architecture, Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, (1991)
3. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, 395p, (1994)
4. Douglass B. P.: Realt-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison-Wesley 332p, (2002).
5. Yildirim K. S.: Design of an Object Oriented and Real-Time Microkernel for Embedded Systems Using Design Patterns, Master Thesis, Dept. of Computer Engineering, Ege University, (2005).
6. Labrosse J. J.: MicroC/OS-II: The Real-Time Kernel, 2nd Ed., CMP Books, 604p, (2003).
7. The eCOS operating system homepage. Available: http://ecos.sourceware.org/
8. Rozier M., Abrossimov V., Armand F., Boule I., Gien M. , Guillemot M., Herrrnann F ., Kaiser C., Langlois S., Leonard P., Neuhauser W.: Chorus Distributed Operating System, Computing Systems, 1(4):305-370, (1988).
9. Beuche D., Guerrouat A., Papajewski H., Schroder-Preikschat W., Spincysk O., Spincysk U.: The PURE Family of Object-OrientedOperating Systems for Deeply Embedded Systems, In Proceedings of 2nd IEEE International Symposium on Object-Oriented Realt-Time Distributed Computing (ISORC'99), (1999).
10. Fröhlich A. A., Schröder-Preikschat W.: EPOS: An Object-Oriented Operating System, In Proceedings of the Workshop on Object-Oriented Technology, Lecture Notes In Computer Science Volume 1743, (1999).
11. Liedtke J.: On micro-kernel construction, In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, (1995).
12. Frohlich A. A.: Application-Oriented Operating Systems, Number 17 in GMD Research Series, GMD - Forschungszentrum Informationstechnik, Sankt Augustin, (2001).
13. The GNU Compiler Collection homepage. Available: http://gcc.gnu.org/
14. The GNU Project Debugger homepage. Available: http://www.gnu.org/software/gdb/
15. Bochs IA32 PC Emulator homepage. Available: http://bochs.sourceforge.net/