Interfacing Operating Systems components with embedded Java applications

Mateus Krepsky Ludwich and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration – LISHA
Federal University of Santa Catarina – UFSC
Email: {mateus, guto}@lisha.ufsc.br

***Abstract*—In this paper we show a way to interfacing operating systems components with embedded Java applications. This interfacing is achieved using the foreign function interface of a Java Virtual Machine that does the binding between the Java methods and C functions at compile time. The operating system used provides just the necessary support for the application execution, avoiding unnecessary components and consequently, waste of resources. Bringing these components to a Java API gives to applications a more fine grain control of hardware resources, typically needed by embedded systems.**

I. Introduction

Embedded systems applications run near to the hardware in the sense that they use hardware devices such as sensors and actuators to interact with the environment, transmitters and receivers for communication, and timers for real time operations. By definition the operating system (OS) is the responsible for creating hardware abstractions and for providing to the application a more high level way to interact with hardware. A way to perform this abstraction is creating a component to abstract each hardware device. Following this reasoning, a Java platform focused on embedded systems should provides these components in its Application Programming Interface (API).

Foreign function interface (FFI) is mechanism that allows programs written in a certain programming language use constructions of programs written in another. The Java programming language refers FFI as *native interface*. FFI is used in Java to access C and C++ programs. The Java Virtual Machine itself uses native interface to access operating system services such as Input/Output (I/O) in order to implement the Java I/O methods. At the *Java Standard Edition* platform (JSE), the native interface specified is the *Java Native Interface* JNI [1].

In this paper we show how to interface operating systems components with Java applications in the context of Java for embedded systems. To accomplish this interfacing was used the foreign function interface of KESO - a Java Virtual Machine that focus on embedded systems, and EPOS - a operating system that is driven by its applications.

The next sections of this paper are organized in this way, section II presents the EPOS operating system showing a component that will be interfaced with the Java platform, section III presents the KESO Java Virtual Machine and how its foreign function interface can be used to create this interface, and the discussions are made in section IV.

II. Operating System Components

EPOS is a multi-platform operating system that is created according with the needs of the application that should run on it. The methodology *Application-driven Embedded System Design* (ADESD), used to develop the EPOS system, shows how to accomplish this task [2]. ADESD proposes the development of component families that are identified from a domain engineering process. Once identified a component family the component's commonalities will lead to an *inflated interface* that contains all the methods of that family. There is no necessity to implement each component of a family one by one, because these can be generated at compile time by applying aspects that defines an scenario into abstractions that are scenario-independent [2]. In order to abstract specificities from different hardware devices EPOS uses *hardware mediators* that provide a platform independent way for access a hardware device. Using static meta-programming techniques and functions inlining the mediators are dissolved during compile time among the abstractions that use it.

One example of families of components in EPOS are the families for Time management, showed in figure 1. The *Clock* abstraction is responsible for keeping track of the current time, and is only available on systems that feature a real-time clock device, which is in turn abstracted by a member of the *RTC* family of mediators.

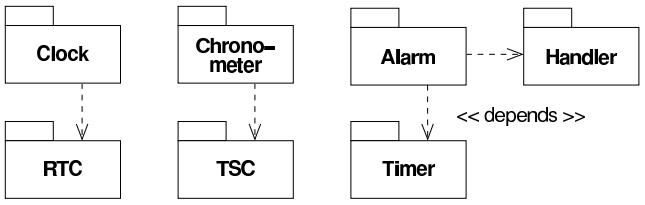

Fig. 1. EPOS timing components

The *Alarm* abstraction can be used to generate timed events, and also to put a thread to *sleep* for a certain time. For this purpose, an application instantiates a handler and registers it with an Alarm specifying a time period and the number of times the handler object is to be invoked. EPOS allows application processes to handle events at user-level through the *Handler* family of abstractions depicted in Figure 2. The *Handler_Function* member assigns an ordinary function supplied by the application to handle an event. The *Handler_Thread*

member assigns a thread to handle an interrupt. Such a thread must have been previously created by the application in the suspended state. It is then resumed at every occurrence of the corresponding event. Finally, the *Handler_Semaphore* assigns a semaphore, previously created by the application and initialized with zero, to an event. The OS invokes operation *v* on this semaphore at every event occurrence, while the handling thread invokes operation *p* to wait for an event.

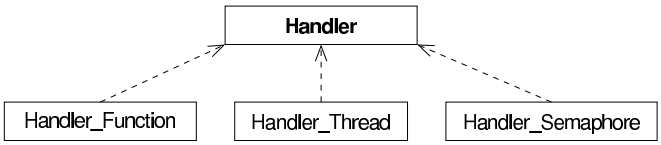

Fig. 2. EPOS event handlers

Chronometer is the EPOS abstraction/component that is used as the case study of this work. It is also a member of Time management families. The *Chronometer* component is used to perform time measurements, it is started/stopped with the *start*/*stop* methods, and the time is obtained by the *read* method. It is also possible to know in which frequency the *Chronometer* operates (*frequency* method), get directly the ticks counted (*ticks* method), and restart it (*reset* method). The *Chronometer* uses the *TSC* (*time stamp counter*) hardware mediator to obtain the current time stamp value. The a *TSC_Common* interface specifies the *time_stamp* method, that must be implemented by each *TSC* realization. There is a *TSC* for each computer architecture (e.g. *IA32_TSC* for IA32 architecture). The figure 3 shows a class diagram presenting *Chronometer* and *TSC*.

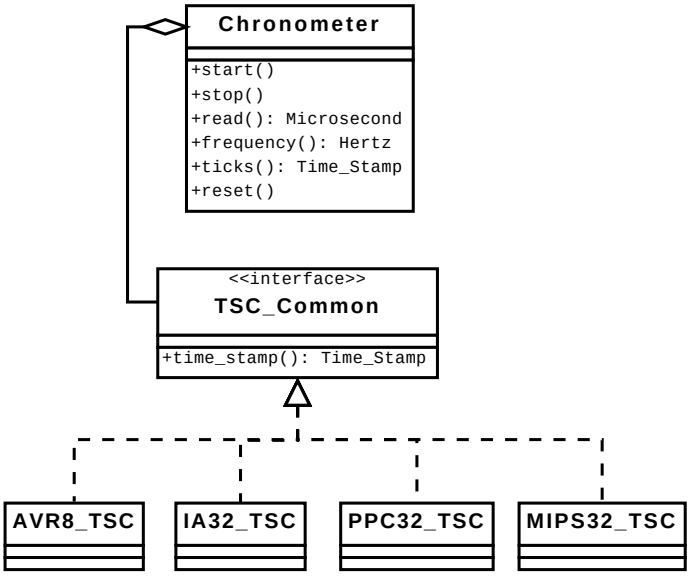

Fig. 3. Chronometer and TSC

III. Components interfacing

KESO is a Java Virtual Machine (JVM) that focus on embedded devices and networks of micro controllers [3]. KESO JVM uses a ahead-of-time compilation strategy, translating the application from java bytecode to C, and generating C code that corresponds to some parts of the JVM needed by the application, such as the garbage collector. Translating the application java bytecode to C and then compiling the C program into native code eliminates the overhead caused by java bytecode interpretation at runtime. Also this strategy enables the generation of the JVM parts that the application really needs.

KESO relies on OSEK/VDX operating system. OSEK/VDX is an organization that specifies a set of standards for automotive systems, including a operating system standard [4]. Recently was developed a way to interface KESO with EPOS [5]. This is accomplished by a layer that does EPOS behave like a OSEK operating system. Using this layer KESO interacts with EPOS as interacts with any other OSEK operating system.

Because KESO relies on OSEK/VDX operating system it exports to Java applications the concepts of this operating system, through a OSEK/VDX API. These concepts includes the OSEK/VDX scheduling and synchronization policies and mechanisms. Besides that, KESO also provides a foreign function interface (FFI) like the Java Native Interface (JNI) to interfacing with C and C++ code. This work uses the KESO FFI to interface EPOS components to the Java applications. This approach is showed in figure 4. The “FFI C gen code” at figure 4, is the code responsible for the interface with EPOS components. “KESO C gen code” corresponds to the Java application bytecode translated to C and the generated parts of KESO JVM.

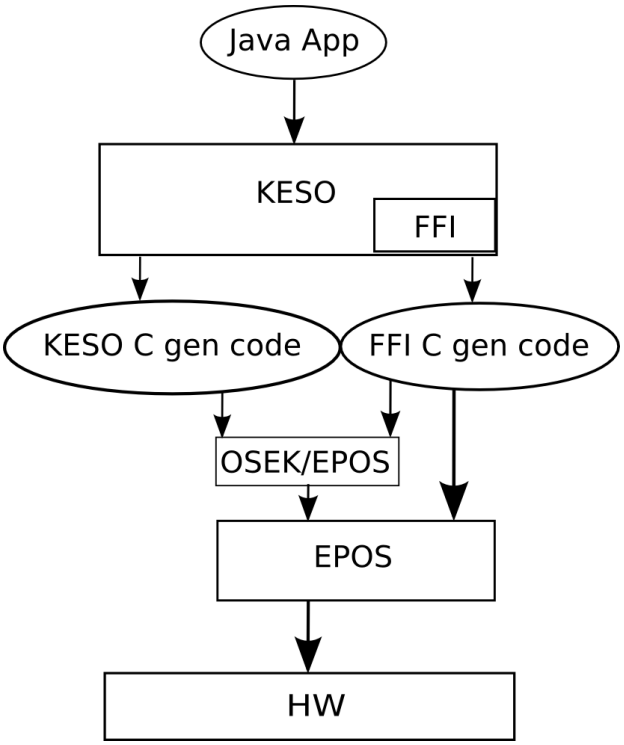

Fig. 4. Proposed approach

The FFI of KESO also use a static approach, i.e. the binding between the Java methods and the correspondent C or C++ methods are done at compile time. In order to do this binding is necessary extends the class *Wavelet* of KESO, and overwrite some of its methods. In the constructor of a class that extends *Wavelet* (a class called *WaveletImplementation*, for example) we specify the *join point*, i.e. which class will be affected by the "weaving". Overwriting the *ignoreMethodBody* to return *true* we tell KESO to do not try generate code for methods of the "join point class". Overwriting the *affectMethod* method is possible to determine the C code that KESO should generate for each method of the "join point class". In this work our *WaveletImplementation* is the *ChronometerWavelet* class. The generated code specified in *affectMethod* is basically a call to the *Chronometer* class at EPOS. The figure 5 shows the *Wavelet* and *ChronometerWavelet* classes, and the overwritten methods.

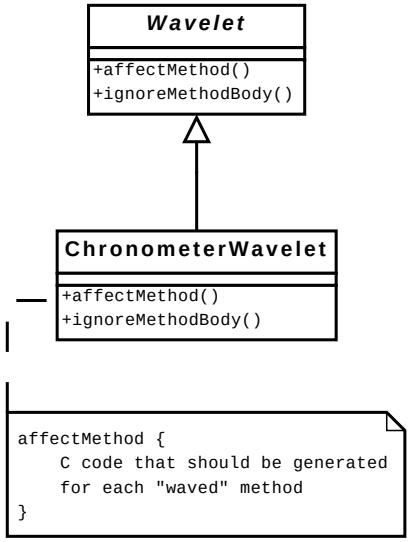

Fig. 5. Using KESO FFI

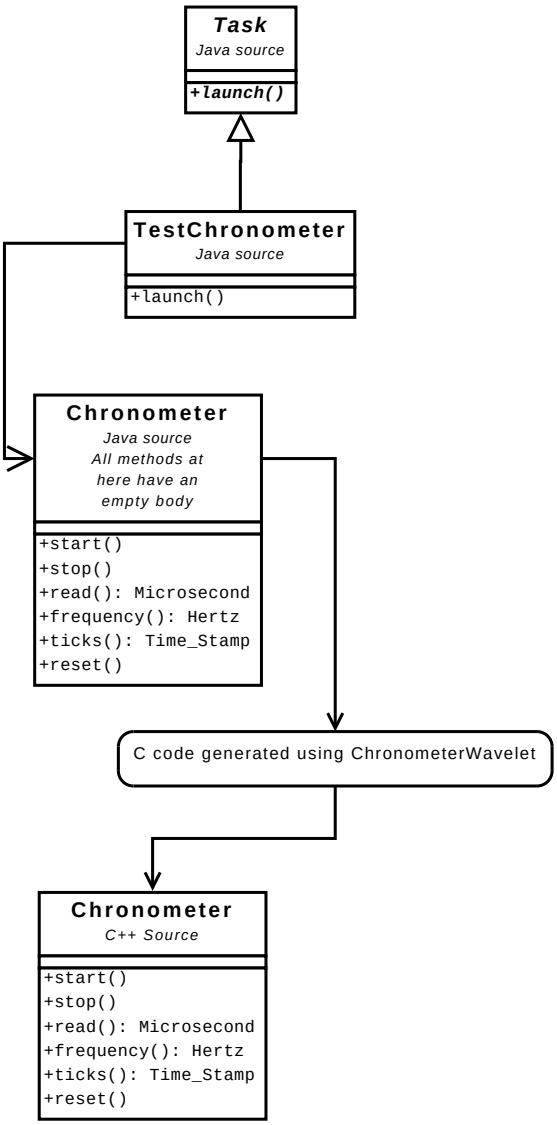

Fig. 6. Overall application

In OSEK and consequently in KESO all tasks are allocated statically, at design time. To create a task in a Java application in KESO is necessary extend the *Task* class and implement the method *launch*, the task entry point at KESO. The class that represents the application in this work (*ChronometerTest*) is the one that extends the *Task* class and implements the *launch* method. This class and the main application classes, including some KESO classes and the binding between Java and C/C++ are showed at figure 6.

IV. Discussion

In this paper we show a way to interfacing operating systems components with Java applications for embedded systems. This is archived using the foreign function interface of KESO JVM. KESO JVM compiles the bytecode of a Java application to C code and generates the parts of JVM needed by applications. The KESO FFI also use this static approach generating the specified C code. Then the C code generated by KESO itself and by KESO FFI are compiled together to native code.

The EPOS *Chronometer* component was used as case study. At EPOS operating system, only the components needed by the application are used. The same strategy used for *Chronometer* can be used for others EPOS components. Bring these components to the Java platform API gives to the applications a more fine grain control of hardware resources, typically needed by embedded systems. At the same time this is accomplished with portability, using the EPOS components that are itself platform independently.

References

[1] S. Liang, *The Java Native Interface - Programmer's Guide and Specification*. Addison-Wesley, 1999.

[2] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[3] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat, “An OSEK/VDX-based Multi-JVM for Automotive Appliances,” in *Embedded System Design: Topics, Techniques and Trends*, ser. IFIP International Federation for Information Processing, S. Boston, Ed., Boston, 2007, pp. 85–96.
[4] “Osek vdx portal,” 2008. [Online]. Available: http://www.osekvdx.org/
[5] H. Bauer, “Entwurf eines OSEK Adaption Layers für das Betriebssystem EPOS,” Master’s thesis, Friedrich-Alexander-Universität, Erlangen-Nurnberg, Deutschland, 2008.
[6] “Josek,” 2008. [Online]. Available: http://www4.informatik.uni-erlangen.de/Research/KESO/josek/
[7] G. Gracioli, “ELUS: Projeto e Implementação de um Mecanismo de Reconfiguração Dinâmica de Software para Sistemas Profundamente Embarcados,” Master’s thesis, Federal University of Santa Catarina, Florianópolis, Santa Catarina, Brasil, 2009.
[8] A. A. F. Hugo Marcondes, Arliones Stevert Hoeller Junior Lucas Francisco Wanner, “Operating systems portability: 8 bits and beyond.” Prague, Czech Republic: 11th IEEE International Conference on Emerging Technologies and Factory Automation, 2006, pp. 124–130.
[9] “Keso,” 2008. [Online]. Available: http://www4.informatik.uni-erlangen.de/Research/KESO/
[10] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz, “An OSEK/VDX API for Java,” in *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, ACM, Ed., New York, 2006, pp. 13–17.