DEVELOPING EMBEDDED APPLICATIONS FOR A COMPONENT-BASED OPERATING SYSTEM

Nikolaos S. Papaspyrou, Ioannis P. Panagopoulos, Stergios Stergiou, George Papakonstantinou
Department of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens, Greece
Tel: +30 1 772 2486; fax: +30 1 772 2519
e-mail: nickie@softlab.ntua.gr, {ioannis, stergiou, papakon}@cslab.ece.ntua.gr

ABSTRACT

Continuing advances in system software and hardware components have allowed the building of embedded systems for an increasing variety of applications. However, from the programmer's point of view, building such systems efficiently presents a number of challenges. The customization of an operating system to support an embedded application has been proposed as a way to reduce the sofware cost. Recent research moves in the direction of designing an operating system in such a way that it can be tailored to the needs of specific applications with small programming effort. In this paper we exemplify this approach using a custom-made operating system called EMPIX, decomposed into very small components. Given an embedded application, the components that are necessary to support the application are automatically combined. This process is controlled by a meta-interface language, which allows the programmer to inform EMPIX of the application's requirements and to direct the components of EMPIX to obtain better performance.

1 INTRODUCTION

The wide deployment of computers has brought forth a huge interest in embedded systems lately. Embedded systems can be found in almost any facet of our lives, e.g. in house appliances, cars, various electrical devices and tools [1, 2]. Programming such systems with minimal cost is bound to be one of the most crucial problems in this area in the years to follow. The selection of an appropriate *Operating System* (OS) plays an important role in the minimization of the software cost. A general purpose OS may not be efficient for a specific application; on the other hand, an OS designed so that it can be automatically tailored to satisfy the requirements of a specific application is a better solution. Many approaches have been proposed towards this direction [3, 4, 5]. In this paper, the approach that we follow is based on:

1. The decomposition of the OS into very small components.
2. The automatic combination of only those components of the OS that are necessary for a specific application. Thus, the application is encapsulated into the OS that is generated from the necessary components.
3. The development of a meta-interface language, which allows the application programmer to inform the OS of the application's requirements and to direct its components to behave in such a way as to obtain better performance. In this way, the process of creating a minimal application dependent OS based on EMPIX can be automated.

Although there are commercial ventures doing component-based OSs, there is a long way to go until we reach a level of certitude that the selected components shall meet the requirements of an application and work satisfactorily and efficiently [6, 7].

We have exemplified the above approach using a custom-made OS called EMPIX [8], which has been developed for the family of Intel x86 and Pentium processors. EMPIX is a clone of XINU [9] but differs from this in many aspects as discussed in Section 3. Based on a BIOS virtual machine, EMPIX has limited operational capabilities and is appropriate for small-scale soft real-time applications. On the other hand, it can easily be ported to different computer architectures. Like XINU, it supports semaphores, messages and dynamic memory management. EMPIX has been used mainly for educational purposes.

The decomposition into small components has been done using separate object files for almost each function and for the critical variables of the OS. The automatic combination of these components is accomplished using: (1) a special meta-interface language, allowing the user to direct the generation of the tailor-made OS, and (2) the linker, which automatically selects the necessary object files. The first step is necessary when selection of code must be done inside some modules, e.g. inside the table where the interrupt vectors are stored.

In the re-development of the OS, we apply a top-down methodology, i.e. the size of the components gradually becomes smaller during the development phase. At this level we believe that we have not yet reached the least possible size for the components. In the near future, the proposed system will be used in real-life medical applications requiring multi-tasking operations such as data acquisition, display and automatic diagnosis of the ECG test, at rest or during exercise [10, 11, 12, 13, 14, 15, 16].

2 MOTIVATION

EMPIX [8] is an OS which has been used for many years in the National Technical University of Athens for educational purposes in the "Operating Systems" course. Most of the programming exercises based on EMPIX involved the implementation of new device drivers, the addition of new shell commands and system calls or the modification of existing ones. Based on this experience, we have concluded that most of these changes to the sources of EMPIX can be described in a formal way, since the procedure for their implementation was mechanical, to some extent.

Nowadays, embedded applications gain day by day a greater part of the market. In our laboratory we have built embedded systems using EMPIX as the underlying OS. The reason why we prefer EMPIX is that it is a very light-weight, easy-to-learn OS and we are familiar with its source code. However, the task of customizing the system according to the needs of a given application is a tedious task. In order to automate this process, we have designed the meta-interface language described in this paper.

Despite an initial overhead, the proposed approach presents many advantages. The fast and automatic generation of application-dependent OS quickly outweighs our effort for defining the meta-

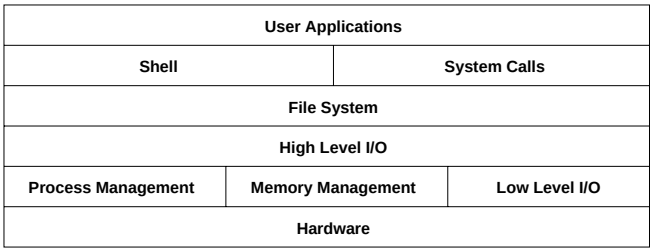

Figure 1: The architecture of EMPIX.

interface language, implementing the meta-interface processor and redesigning EMPIX according to the new requirements. The resulting OS are free of useless system calls, unnecessary interrupt handlers, device drivers and daemon processes that slow down the overall performance and utilize memory that might be valuable for small scale embedded applications.

3 THE EMPIX OPERATING SYSTEM

EMPIX [8] is a multitasking and multiuser OS, initially implemented for educational purposes. Although it is a small OS, it is powerful enough to form the basis for the development of small scale embedded applications. Its architecture is depicted in Figure 1.

As shown in Figure 1, EMPIX has a layered structure. It has been designed in such a way that each level uses only services provided by lower levels. The lowest level is responsible for memory and process management and the low-level part of the I/O drivers. In the second level, there is the high-level part of the I/O drivers. The file system resides on top of the I/O drivers. The higher layer consists of the shell and the primitive calls mechanism. User applications are located on top of all OS layers. Applications use the facilities of EMPIX through system calls. Alternatively, EMPIX can be used through the shell.

EMPIX combines ideas and source code from other OS, such as XINU [9], MINIX [17], MS-DOS and UNIX. More specifically EMPIX has used: (1) the philosophy of UNIX, (2) structures and routines from XINU, (3) part of the source code of diskette and screen drivers from MINIX, and (4) the MS-DOS filing system.

Although EMPIX is based on all the previous systems, there are some aspects with respect to which EMPIX compares favourably. It is more powerful that XINU (multiuser, process hierarchy, loading of static processes from the disk, file compatibility with MS-DOS, dynamic memory allocation, more advanced message passing mechanism). It is smaller and simpler than MINIX, easier to use and with more facilities in process management. Furthermore, it supports multitasking and multiple users, in contrast to MS-DOS.

EMPIX is a standalone OS. It does not need MS-DOS to boot or run. In the current phase of development, there are no tools available for the development of applications in the EMPIX environment. However, EMPIX is file compatible with MS-DOS and thus developers can use off-line all tools existing in the MS-DOS environment (e.g. text editors, C compilers, x86 assemblers, linkers, libraries) to create EMPIX executable binary files, provided that they link it with EMPIX's standard library. Then, the binary file can be moved to the EMPIX environment and executed.

4 THE META-INTERFACE LANGUAGE

In the proposed system, the process of developing embedded applications is depicted in Figure 2. The code of the embedded application consists of three parts:

- The *meta-interface*, written using a special meta-interface language. It contains: (i) information about the requirements of the application; (ii) directions to improve the performance of the standard EMPIX components that will be used; and (iii) definitions of additional components that will be encapsulated in the OS. The meta-interface can appear in the application code in several different places, each of which must be surrounded by the special brackets `@{` and `}@`, to be separated from the rest of the application.
- The *OS extension*, containing the implementation of the additional components defined in the meta-interface, if there are any. This part can be written in C, assembly language, or a combination of both. Since it interracts directly with the OS kernel, some knowledge of the EMPIX internals is assumed.
- The *application code*, implementing the embedded application using the standard EMPIX components and the additional components defined in the meta-interface. This part is typically written in C.

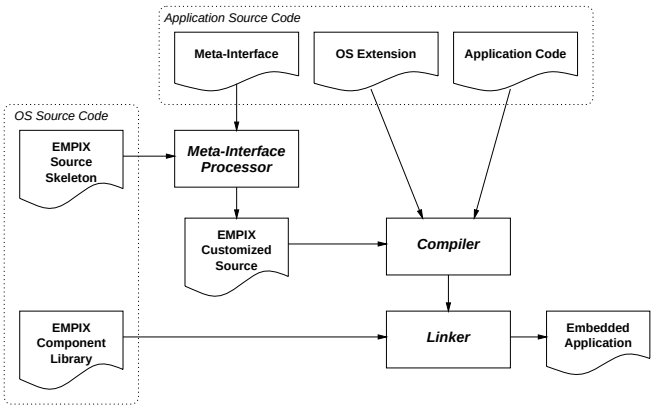

Figure 2: Development process for embedded applications.

⟨*interface*⟩ ::= ⟨*directive*⟩ "**;**" [⟨*interface*⟩]
⟨*directive*⟩ ::= "**define**" ⟨*class*⟩ ⟨*id*⟩ ["**(**" ⟨*formals*⟩ "**)**"]
"**with**" (⟨*def*⟩)*
["**global**" ⟨*globals*⟩] "**end**"
| "**use**" ⟨*class*⟩ ⟨*id*⟩ ["**(**" ⟨*actuals*⟩ "**)**"]
| "**if**" ⟨*cond*⟩ "**then**" ⟨*interface*⟩ [⟨*else*⟩] "**end**"
⟨*class*⟩ ::= "**command**" | "**device**" | "**interrupt**"
| "**module**" | "**shell**" | "**syscall**"
⟨*def*⟩ ::= ⟨*id*⟩ "**=**" ⟨*constant-expression*⟩ "**;**"
⟨*formals*⟩ ::= ⟨*id*⟩ ("**,**" ⟨*formals*⟩)*
⟨*globals*⟩ ::= (⟨*external-decl*⟩)*
⟨*actuals*⟩ ::= ⟨*constant-expr*⟩ ("**,**" ⟨*constant-expr*⟩)*
⟨*cond*⟩ ::= "**using**" ⟨*class*⟩ ⟨*id*⟩ | "**true**" | "**false**"
| "**(**" ⟨*cond*⟩ "**)**" | ⟨*cond*⟩ "**and**" ⟨*cond*⟩
| ⟨*cond*⟩ "**or**" ⟨*cond*⟩ | "**not**" ⟨*cond*⟩
⟨*else*⟩ ::= "**else**" ⟨*interface*⟩

Figure 3: The grammar of the meta-interface language.

The meta-interface is extracted and processed by a special tool of our system, which tailors the EMPIX source code to satisfy the requirements of the application. The customized EMPIX code is compiled together with the rest of the application source code and linked with the necessary components from the standard library of EMPIX.

The grammar of the meta-interface language is presented in Figure 3. It supports a fixed set of *meta-classes* which abstract the functionality of various OS components. A *meta-object* of some meta-class provides a specific implementation of this functionality, and several meta-objects may belong to the same class thus imple-

menting different functionalities. The most important part of the language is the directives **define** and **use**. The former defines a new meta-object of a given meta-class, which is encapsulated in the OS and can be used by the embedded application. The latter informs that the embedded application will use a named meta-object, which can either be one of the standard EMPIX meta-objects or one of the additional ones defined using **define**. Meta-objects may have a number of formal *parameters* in their definition. These parameters take their actual values when the meta-objects are used.

4.1 Predefined Meta-classes and Meta-objects

EMPIX supports six predefined meta-classes representing different aspects of the OS. The application programmers cannot define new meta-classes; however, they may define new meta-objects that implement any of these meta-classes, in any way that suits their needs.

- The meta-class **syscall** represents EMPIX *system calls*, i.e. system-specific functions that provide the basic interface between the OS and the application. System calls are usually installed by other EMPIX components.
- The most generic meta-class is **module**. *Modules* are used primarily to implement OS components and extensions. EMPIX supports many predefined modules. One such module is the **memory**, which provides the basic system calls for dynamic memory allocation and deallocation. The **scheduler** is responsible for the management of processes which share the computer's CPU time. The **semaphores** and **messages** implement two methods for process communication between processes. **fs** implements the upper level of EMPIX's file system, whereas **floppy** and **hdisk** implement the lower level for file systems residing on floppy and hard disks. Most of these modules accept parameters which alter their behaviour and/or performance.
- The meta-class **device** abstracts the common functionality of EMPIX I/O *devices*. A number of predefined objects support the basic I/O devices of the PC. The **console** implements the computer's keyboard and monitor. There are other devices for two serial ports (**tty0**, **tty1**), two floppy disk drives (**floppy0**, **floppy1**) and two hard disk drives (**hd0**, **hd1**). A number of devices is also reserved for open files by the **fs** module. Some of the predefined devices install device-specific system calls.
- The meta-class **interrupt** represents a routine that is used as an *interrupt handler*. The only predefined interrupt handler in the standard library of EMPIX is **clock**, which implements the computer's real-time clock. Many of the predefined modules rely on this handler to perform their functionality.
- The meta-class **shell** abstracts the functionality of the *shell*, i.e. EMPIX's user interface. The predefined shell **sh** implements an extensible UNIX-like command-line user interface.
- The meta-class **command** is used in order to extend **sh** with new *commands*. Some basic predefined commands are supported, e.g. **cmd_exec** and **cmd_exit**, but modules may install additional commands, e.g. the **fs** module installs commands for file management such as **cmd_ls** and **cmd_cd**.

4.2 Some Simple Examples

Let us assume that an embedded application requires the use of semaphores, as implemented by the standard module provided by EMPIX. The meta-interface part of this application could be:

```
@{
   use module semaphores (int, 10);
}@
```

The actual parameters passed to the **semaphores** meta-object specify that there will be 10 semaphores available to the application and that each semaphore will have a value of type `int`.

As a second example we consider a multi-process application requiring message passing for exchanging information of type `double` and access to a hard disk. Its meta-interface could be:

```
@{
   use module scheduler (5, 20);
   use module hdisk;
   use module fs (2, 1024);
   use module messages (double, 20);
}@
```

Four EMPIX modules are needed: the scheduler, the high-level part of the file system, the low-level part of the file system supporting the hard disk and the messages module. The scheduler will support a maximum of 5 processes and the time quantum for each process will be set to 20/50 sec. The file system will support a maximum of 2 files open at a time, with a buffer of 1024 bytes for I/O. Finally, there will be a queue of up to 20 messages for each process and each message will contain information of type `double`.

As a last simple example, let us assume that an embedded applications requires access to the computer's parallel port, which is not currently supported by EMPIX. A new **device** meta object must be defined in the meta-interface part of the application.

```
@{
   define device parallel (devno, iv, ov) with
      num   = devno;      getc  = pp_getc;
      minor = 0;          putc  = pp_putc;
      init  = pp_init;    ioctl = pp_ioctl;
      open  = pp_open;    ivec  = iv;
      close = pp_close;   ovec  = ov;
      read  = pp_read;    iint  = pp_iint;
      write = pp_write;   oint  = pp_oint;
      seek  = ioerr;      oblk  = NULL;
   end;
   use device parallel (PPORT, PP_IVEC, PP_OVEC);
}@
```

The first directive defines the new device and the second informs that this device will be used, specifying some hardware-specific constants as parameters. Furthermore, the device's functionality must be implemented. In the OS extension part, the programmer must define the functions `pp_init`, `pp_open`, `pp_close`, etc.

4.3 A Complete Example: Adding Events

As a more involved example, in this section we describe how to implement an event list in EMPIX. A new module called **events** is defined, together with a system call **ev_schedule** for scheduling new events. If the embedded application uses the EMPIX standard shell, a new command called **alarm** is also defined. This command schedules an event that rings a bell after a given number of seconds.

For the implementation of events, we use the computer's real-time clock. Assuming that no more than 10 jobs will be required in the clock's interrupt handler, the meta-interface begins with:

```
@{
   use interrupt clock (10);
```

Next, we define the **events** module, introducing the maximum number of events that can be scheduled simultaneously as a parameter `max`. We also specify the name of the module initialization function: `ev_init`. Moreover, we specify that when the module is used, the global constant `MAX_EVENTS` should be added to the EMPIX programming environment and its value should be equal to the value of the given parameter `max`.

```
   define module events (max) with
      init = ev_init;
   global
      const int MAX_EVENTS = max;
   end;
```

After its definition, the new module can be used. Here we set the maximum number of events to 10.

```
   use module events (10);
```

The next thing to do is to define and use the system call **ev_schedule** for scheduling new events.

```
define syscall ev_schedule with
   bytes = sizeof(unsigned long) +
           sizeof(INTERRUPT (*) ());
   func  = ev_schedule_func;
end;
use syscall ev_schedule;
```

The name of the function implementing the system call is `ev_schedule_func`. It takes two parameters. The first has type `unsigned long` and specifies the time after which the event will occur, in 50ths of a second. The second is a pointer to the function that will be called when the event occurs.

Finally, we need to define and use the new shell command **alarm**, provided that the standard EMPIX shell **sh** is used. The name of the function implementing the new command is `ev_alarm_cmd`.

```
   if using shell sh then
      define command cmd_alarm with
         name = "alarm";
         run  = ev_alarm_cmd;
      end;
      use command cmd_alarm;
   end;
}@
```

This concludes the meta-interface for the event list. What remains to be written is the C code for the event list's implementation. At least the following functions, that are mentioned in the meta-interface, need to be defined:

```
int    ev_init          ();
SYSTEM ev_schedule_func (unsigned long time,
                         INTERRUPT (* f) ());
int    ev_alarm_cmd     (struct lex * head,
                         struct lex * lptr);
```

together with everything else that is needed for the implementation, e.g. a data structure representing the event list, a function that will be added as a job to the real-time clock interrupt handler and a function that will ring the bell when an alarm event occurs.

5 A CASE STUDY: MEDICAL APPLICATIONS

EMPIX has been successfully used for small scale embedded systems, mainly for medical applications. The first attempt was to build an embedded system for the acquisition and storing of long-term electrocardiograms using a 12-lead digital electrocardiograph [14, 12]. The digital output of the electrocardiograph is connected to an RS-232 serial port. The embedded application consists of two processes: one collecting samples and storing them temporarily in a local buffer, and another copying the data from the local buffer to permanent storage. As a hardware platform, we used Jumptec DIMM-PC/386-B on a custom designed board providing two serial ports and an interface for a text LCD display.

Apart from storing the electrocardiogram, the software can also extract and store the heart rate signal, using a QRS detection algorithm [12] based on the length or energy transform. The development of this application provided us with motivation to design the meta-interface language described in this paper. In the near future, we plan to develop several embedded medical applications. The first one will be a system that acquires and stores an electrocardiogram and calculates in real-time the Athens QRS score [10, 11]. This is a much more complex application and we expect that the use of the meta-interface language will facilitate the programmers' task.

6 CONCLUSION

Embedded systems are currently "at the forefront of use and justification of computer products" [2] and the need for effortless and costless development of efficient and reliable embedded applications has become one of the first priorities in today's computer industry. In this paper we propose a powerful meta-interface for programming embedded applications, based on customized versions of the EMPIX operating system that can be automatically configured. The resulting applications use only the necessary components of EMPIX; they are light-weight and efficient.

Future research will be directed towards possibilities of further customization. At its current form, the meta-interface language does not allow the application programmers to define meta-classes of EMPIX components; there is a fixed number of predefined meta-classes and the programmers can only define instances of meta-objects. Using a full object-oriented language for the definition of EMPIX components, allowing new meta-class definitions and inheritance, would only be a natural evolution of the current meta-interface language. This would introduce an additional *meta*-level to our system and would allow for a more detailed customization.

References

[1] W. Wolf, *Computer as Components: Principles of Embedded Computing Systems Design*. Morgan Kaufmann, 2000.

[2] D. Milojicic, "Embedded systems", *IEEE Concurrency*, vol. 8, 2000.

[3] "RT-Linux as an embedded operating system". `http://www.rtlinux.org/`.

[4] "Windows CE 3.0". `http://www.microsoft.com/`.

[5] "RedHat eCos." `http://www.redhat.com/embedded/technologies/ecos/`.

[6] G. Kiczales, "Foil for the workshop on open implementation". `http://wwwparc.xerox.com/csl/groups/sda/projects/oi/workshop-94/foil/`, 1994.

[7] S. Shumilov, "Flexible metalevel architectures: A review". `http://www.informatik.uni-bonn.de/~shumilov/research/mop/mop.html`.

[8] G. Papakonstantinou, C. Delarokas, and P. Tsanakas, *The EMPIX Operating System*. Symmetria, 1990. In Greek.

[9] D. Comer, *Operating System Design: The XINU Approach*, vol. 1, PC Edition. Prentice Hall, 1988.

[10] A. P. Michaelides *et al.*, "Signifance of ST segment depression in exercise-induced supraventricular extrasystoles", *American Heart Journal*, vol. 117, pp. 1034–1041, 1989.

[11] A.P. Michaelides *et al.*, "Exercise-induced QRS prolongation in patients with coronary artery disease: A marker of myocardial ischemia", *American Heart Journal*, vol. 126, pp. 1320–1325, 1993.

[12] A. Alexandridi *et al.*, "An integrated system for the diagnosis of cardiac pathology through the analysis of heartbeat interval variability", in *Advances in Physics, Electronics and Signal Processing Applications*, pp. 24–29, World Scientific, 2000.

[13] A. P. Michaelides, *et al.*, "Improved detection of coronary artery disease by exercise electrocardiography with the use of right precordial leads", *New England Journal of Medicine*, vol. 340, pp. 340–345, 1999.

[14] F. Gritzali, G. Fragakis, and G. Papakonstantinou, "Detection of P and T waves in an ECG", *Computers and Biomedical Research*, vol. 22, pp. 83–91, 1989.

[15] A. Thanos, G. Economakos, G. Papakonstantinou, P. Tsanakas, and L. Nikolaidis, "An open system for ECG telemedicine and telecare", in *Proc. Computers in Cardiology*, 1998.

[16] A. Koulouris, G. Papakonstantinou, and P. Tsanakas, "A decentralized multichannel length transformation for real-time ECG monitoring", *Computers and Biomedical Research*, vol. 33, pp. 227–244, 2000.

[17] A. Tanenbaum and A. Woodhull, *Operating Systems, Design and Implementation*. Prentice Hall, 1999.