Exploring Embedded Software Concepts using the Hellfire platform in an Undergraduate Course

Alexandra Aguiar and Fabiano Hessel
alexandra.aguiar@pucrs.br, fabiano.hessel@pucrs.br
Pontificia Universidade Catolica do Rio Grande do Sul – PUCRS
Porto Alegre, Brazil

***Abstract*—Information Technology-related undergraduate programs, such as Computer Science and Computer Engineering, offer courses that cover a broad spectrum of embedded systems' topics. An embedded system is composed of hardware and embedded software. The software that composes these systems has become noticeable and its presence increases the number of features offered by the embedded systems. However, it is not always clear to undergraduate students the division and the cooperation that must exist between hardware and software in such systems. In this paper we report on our experience teaching students about the structure of embedded systems and how relevant software is in this context. More specifically, we have used a tool named Hellfire Framework to decrease this knowledge gap. Hellfire is a platform for developing embedded applications with real-time constraints. The framework consists of a set of tools that suggests a design flow to guide the development of a complete hardware/software solution. In our course, we provide a simplified version of one of the framework components, the HellfireOS. This component is used for analysis and modification of the system source code. By using the HellfireOS, students can better understand the impact of code changes on the overall system operation. We have been using the Hellfire framework for over two years. Over these years we have noticed that students have increased their comprehension of the hardware-software interaction in embedded systems.**

I. Introduction

Embedded systems have been used throughout the years in a very wide range of applications impacting the way people live their lives. These systems, which are adopted in entertainment devices, medical equipment, automotive features have some key characteristics. The most prominent characteristic is the goal of the system, since many embedded systems are developed to perform a well-established and known task. However, this scenario has changed. Although there are still embedded systems with restricted features, converging entertainment devices have motivated a new set of development possibilities.

Nevertheless, besides seeing diverging types of embedded systems, there is another characteristic that can help us to categorize them: the presence or absence of real-time constraints. In this case, when real-time constraints are present, the entire development design flow must be redesigned.

Therefore, many universities address both embedded and real-time systems' topics in their undergraduate programs aiming to bring the student closer to the real market. Although it is important to better develop students' programming skills of embedded systems, a general concern among teachers remains: how to teach students real life matters without using real systems?

In this context, we developed a lite version of the Hellfire Framework aiming to decrease that gap. Hellfire Framework is an academic project developed at PUCRS (Pontifcia Universidade Catlica do Rio Grande do Sul), a Brazilian university. The framework provides its own design flow. The student who is developing an embedded system can add applications (in C language), run it over the Hellfire OS (which is a real-time operating system), and simulate the developed system on an instruction-set simulator (ISS) or prototype it in an FPGA-like board.

Thus, Hellfire System can be used following two different approaches. First, the class can be based in exploring the user application aspects. Therefore, no further acknowledge of the internals of the operating system is needed and only the C application stays in focus. It is possible to explore, for instance, the impact of a given system call in the performance of the application. Still, the user can see in a very user-friendly web-based interface, the energy consumption results of the application and even its real-time behavior.

Second, Hellfire Lite Version allows a much more intimate relationship with the OSs source code, since it allows the student to download the entire OS and see how a RTOS works in real life. In spite of not providing the total number of features available in the full version, Hellfire Lite is a more suitable option for real-time and embedded systems designers whereas the full version helps application designers.

To demonstrate the use of our approach, we have applied a questionnaire both in the beginning and in the end of each term aiming to capture the students' expectations before and the perceptions after using the platform. Thus, we identified that in those terms where the tool was not used, it was harder for the students to understand the real behavior of embedded real-time systems. However, since we adopted Hellfire framework (both lite and full version), which has happened for the last two years, we could perceive an increase of the learning quality experience and on the content relevance. We also perceived a stronger confidence of the students regarding designing and developing an embedded system.

The remainder of the paper is organized as it follows. Section II presents some details regarding the platform followed by Section III where we present the methodology used in the classes. Section IV discusses some results we achieved whilst

978-1-4673-0840-3/12/$31.00 ©2012 IEEE

Section V concludes the paper.

II. Hellfire System

The Hellfire System (HFS) is an academic project born at the Embedded Systems Group (GSE) of the Pontifical Catholic University (PUCRS), in Brazil. The project provides its own design flow that comprehends different abstraction levels, from C application development to FPGA prototyping. This design flow is supported by several tools and modules that compose the Hellfire Framework (HFFW).

From a single processor point of view, the designer can develop the application C code and run it over the Hellfire Operating System (HellfireOS), which is a highly configurable real-time modular micro-kernel based OS. From the platform point of view, the designer can add up to 128 processors to the system, configure each one of them and, by using the HellfireOS API, develop parallel embedded applications which are able to exchange data and even able to migrate tasks. Currently, only MIPS-based processors are allowed.

A. HellfireOS

The HellfireOS (HFOS) [1] is a real-time operating system (RTOS) developed in order to ensure maximum flexibility in its configuration and allow a high level platform customization. In order to allow such feature, HFOS was implemented in a modular way, where each module corresponds to some specific functionality.

Figure 1 depicts the kernel modular organization. All hardware-specific functions are defined in the first layer, known as HAL (Hardware Abstraction Layer) and the uKernel lies just above it. The communication, migration, memory management and mutual exclusion drivers, as well the API are placed over the uKernel layer. The user applications belong to the top layer.

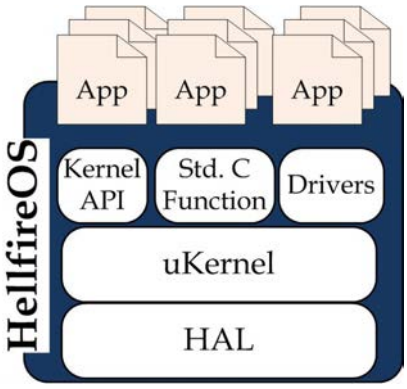

Fig. 1. HellfireOS Structure

Due to its modular implementation, HellfireOS is easily portable to other architectures, requiring only the rewrite of hardware-dependent functions, such as the interrupt service, its initialization routine and the context switch. In order to decrease the kernel final size, allowing the HFOS usage even in architectures with severe memory limitations, parameters such as user tasks' maximum number, stack and heap size and drivers usage, are configurable.

The users' applications are written using basic *C* programming language and the HellfireOS API. In Figure 2 a user task example that prints the CPU usage in the standard output of the system is shown.

```
void cpu_usage_test(void) {

    while (1) {
        printf ("\nMemory Usage %d", OS_GetCpuUsage());
    }
}
```

Fig. 2. Task Example

Another configurable parameter is the activation bit used by the timing register, which determines the system tick size. The tick size corresponds to the minimum periodic timing unit of the system. This unit varies from 0.32ms to 83.88ms. Figure 3 shows the relation between the processor clock frequency rate and the activation bit.

Core Frequency	Activation bit	Tick time (ms)	Ticks/second
25 MHz	15	1.31	763.36
	16	2.62	381.68
	17	5.24	190.84
	18	10.48	95.42
	19	20.97	47.69
	20	41.94	23.84
	21	83.88	11.92
33 MHz	15	0.99	1010.10
	16	1.98	505.05
	17	3.97	251.89
	18	7.94	125.94
	19	15.88	62.97
	20	31.77	31.48
	21	63.55	15.74
50 MHz	15	0.65	1538.46
	16	1.31	763.36
	17	2.62	381.68
	18	5.24	190.84
	19	10.48	95.42
	20	20.97	47.69
	21	41.94	23.84
66 MHz	15	0.49	2040.82
	16	0.99	1010.10
	17	1.98	505.05
	18	3.97	251.89
	19	7.94	125.94
	20	15.88	62.97
	21	31.77	31.48
100 MHz	15	0.32	3125.00
	16	0.65	1538.46
	17	1.31	763.36
	18	2.62	381.68
	19	5.24	190.84
	20	10.48	95.42
	21	20.97	47.69

Fig. 3. Tick Time x Core Frequency

B. Hellfire Framework

The Hellfire Framework (HFFW) [1] allows a complete deployment and test of parallel embedded applications, defining the HW/SW architecture to be employed by the designer. The HFFW is divided in three modules as it follows, which are discussed throughout this section.

- HellfireOS, discussed previously in Section II-A;
- N-MIPS MPSoC Simulator, and;
- architecture builder.

The N-MIPS MPSoC Simulator is an Instruction-Set Simulator (ISS) based on MIPS-like [2] cores. It was also written in *C* and provides simulation of up to 128 processing cores.

A very important feature is the possibility to prototype in FPGA the same resulting image file used during the simulation, which increases the chances of prototyping success after the simulation.

The last highlighted module, named *architecture builder* is used to specify the target architecture and to configure all the HellfireOS parameters.

The HellfireFW design flow is basically divided in three steps: the application design, the project creation and simulation/refinement, explained in the remainder of this section.

Application Design. The first step of the design flow is the development of an application implemented in *C* language and manually divided into a set of tasks. These tasks are defined as n-uples $(id_i, r_i, WCET_i, D_i, P_i)$ and the parameters stand for identification, release time, worst case execution time, deadline and period of each task respectively.

HellfireFW Project Creation. After the application design, the designer must create a project in HFFW. This step corresponds to the architecture design and configuration. The designer must define the initial hardware architecture, specifying the number of processors and their frequencies and configuring the HellfireOS parameters, as presented in Section II-A. After that, the mapping of the tasks along the processors must be performed. This mapping is done manually and is based on the designer's experience.

As the output of this step, an MPSoC platform is expected, having a given number of processors, a personalized instance of HellfireOS on each processor and a static task mapping.

Simulation and Refinement. When the designer triggers a simulation, the MPSoC ISS Simulator is activated. After the simulation is completed, several reports are presented to the designer both in textual and graphical way. The textual reports are the following:

- standard output of each processor;
- report with all Plasma instructions used, the number of times that each was used and an usage percentage of each group (logical, arithmetic, etc) of instructions;
- an energy consumption summary, based on [3];
- report containing the main characteristics of the system, such as deadlines misses and CPU load, and;
- individual report of the cycle to cycle operation of each processors. All information contained on the system stack is shown in this report.

III. Methodology for Class Support

The framework described in the former section presents the solution with all possible resources. In spite of that, during our earlier years of educational practice, without any kind of practical tool, we observed that students had trouble understanding the basics of an embedded real-time operating system. Therefore, we decided to split the Hellfire project, creating a lite version where only the most basic features were available.

The main goal of creating Hellfire Lite was to distribute the source code freely among the students, therefore no other kind of graphic interface (such as the web-based framework) was required. Initially, we present the states that a task can assume, as shown in Figure 4.

```
/* task status definitions */
#define TASK_READY      0       // task has run, and is ready to run again
#define TASK_RUNNING        1       // task is running (only one task/core can be in this state)
#define TASK_BLOCKED        2       // task is blocked, and stopped executing. can be resumed later.
#define TASK_WAITING        3       // task is waiting on a semaphore, and will be resumed soon
#define TASK_NOT_RUN        4       // task is not ready, since it has never run
```

Fig. 4. Hellfire Lite Task States

Once the students are familiar with the states a given task can assume, we present and discuss the Task Control Block - TCB and each one of its parameters, shown in Figure 5.

```
typedef struct{                 // the task control block
    unsigned char id;               // task id
    char description[TASK_DESCRIPTION_SIZE];    // task description (or name)
    unsigned char status;               // 0 - ready,  1 - running,  2 - blocked, 3 - waiting, 4 - not run
    unsigned int ticks;             // total task ticks executed
    unsigned int last_tick_time;            // last tick duration
    unsigned int memory_usage;          // task memory usage
    context task_context;           // task context
    void (*ptask) (void);           // pointer to task entry point
    void (*stack_ptr) (void);           // pointer to task stack address (bottom)
    unsigned char *stack;           // pointer to task stack buffer
    unsigned int stack_size;            // task stack size
} tcb;
```

Fig. 5. Hellfire Lite Task Control Block

After presenting the TCB structure, we show some important variables to the OS, such as the task set itself, presented in Figure 6.

```
tcb OS_task[OS_MAX_TASKS];                  // global vector of task control blocks
tcb *OS_task_entry;                     // pointer to a task control block
```

Fig. 6. Hellfire Lite Task Set Variable

Following, after explaining the basics of the task set we present another crucial portion of the system, which regards to the way these tasks are scheduled. Since the goal is to provide the students the opportunity of putting in practice concepts only studied theoretically, we provide only a simple circular scheduler, with no timing constraints compliance. This scheduler is presented in Figure 7.

```
static void OS_DispatchTasks(void){
    if (OS_tasks > 0){
        OS_current_task = OS_TaskBestEffortReschedule();            // schedule best effort tasks
        OS_UpdateTCB();
        OS_RestoreExecution(OS_task_entry->task_context,1);         // restore task context
        OS_Panic(PANIC_UNKNOWN);                        // not reachable code
    }else{                                      // hope we never get here..
        OS_Panic(PANIC_NO_TASKS_LEFT);
    }
}

static unsigned char OS_TaskBestEffortReschedule(){
    static unsigned char i=0;

    while(1){
        (i<=OS_max_index)?(i++):(i=0);
        OS_task_entry = &OS_task[i];                    // set pointer to tcb address
        if (OS_task_entry->ptask){                  // is there a best effort task in this slot?
            if ((OS_task_entry->status == TASK_READY) || (OS_task_entry->status == TASK_NOT_RUN)){      // ready to run?
                return i;
            }
        }
    }
}
```

Fig. 7. Hellfire Lite Built-in Scheduler

A. Planned activities - embedded systems

We developed a flow aiming to employ both Hellfire Framework and Hellfire Lite version during the classes. This *Teaching Plan* contemplates the main steps we consider mandatory in the use of these tools in an undergraduate course. Firstly, we highlight mandatory steps for embedded systems classes followed by our recommendations during the adoption of our methodology for real-time systems' classes.

Initially, we need to present both full and lite version and state the difference between them. After this initial contact, we explore several embedded applications that are distributed with Hellfire full version. This initial set of tests contemplates only the use of such applications and it also aims to familiarize the student with the tool and its provided feedback.

After only using and understanding the entire flow of the system, the students are asked to develop different real-time applications to be executed in the full version platform. Results must be analyzed and discussed.

B. Planned activities - real-time systems

The next step contemplates the first of a series of modifications that must be performed aiming to improve the students' knowledge about real-time scheduling. Therefore, students are motivated to develop theoretical concepts about real-time scheduling, such as the use of periodic tasks. When periodic tasks are added, the students must develop the most used scheduling policies: Rate Monotonic - RM [4], [5], [6] and Earliest Deadline First - EDF, [7], [8]. Still, aperiodic tasks can be exploited and their compliance based on polling server is also a viable option.

Finally, regarding real-time tasks the teacher can explore the use of dependent tasks and how they should be treated using RM and EDF scheduling policies and demand students to provide support to this feature.

IV. Discussed results

In this section we briefly discuss the results we observed throughout the adoption of Hellfire Full Framework and the Hellfire Lite Version. Results were taken during the last two years in classes of two different undergraduate courses: computer science, regarding only embedded systems' classes and computer engineering, contemplating real-time systems' classes.

As educators, we could observe that when no practical tool was available to the students, the overall interest level and learning quality was damaged. The main reason for that is that the pedagogic structure of the class was not motivating enough for the students, specially in our specific case where these courses already happen to be taken when the student is too close of graduating (that is, they already developed interest for other specific fields of both computer and engineering courses).

Thus, it took us two semesters to adopt the tools and better adequate them to the existing teaching plan. We could observe the higher interest for the field (in both cases) and how students were more motivated to even search for embedded real-time matters extra-class and bring them to discussion. We still asked them to answer a questionnaire about the use of the tools (full and lite version) and the increase of learning quality experience could be easily perceived.

V. Conclusion

Embedded systems are a solid reality and the inclusion of their basic concepts in both computer science and computer engineering courses has become a common approach. During years of practice, we observed the difficulty level of teaching such an advanced topic for undergraduate students and decided to use a well-known and structured platform during our classes. Therefore, we used the full version of Hellfire Framework especially for embedded systems lessons, aiming the use and development of different embedded applications and the analysis of their results with the framework's facilities. Later, we decided to extend the use of Hellfire in our undergraduate courses and provided a lite version of the framework with less resources but easier to understand. The Hellfire Lite is more intended to include real-time scheduling concepts and, therefore, is more indicated for real-time courses. We could observe that higher confidence of students since they had contact with real and practical embedded systems during their undergraduate courses.

Acknowledgment

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9.

References

[1] A. Aguiar, S. Filho, F. Magalhaes, T. Casagrande, and F. Hessel, "Hellfire: A design framework for critical embedded systems' applications," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, 2010, pp. 730 –737.

[2] O. Cores, "Plasma most mips i(tm) opcodes," http://www.opencores.org.uk/projects.cgi/web/mips/, Accessed, September 2009, 2007.

[3] S. J. Filho, A. Aguiar, C. A. M. Marcon, and F. P. Hessel, "High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping," in *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 27–33.

[4] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[5] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behaviour," *IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.

[6] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, 1982.

[7] W. H. Hesselink and R. M. Tol, "Formal feasibility conditions for earliest deadline first scheduling," Tech. Rep., 1994.

[8] M. Andrews, "Probabilistic end-to-end delay bounds for earliest deadline first scheduling," in *In Proceedings of the IEEE INFOCOM 2000*, 2000.