A Systematic Approach to Software Peripherals for Embedded Systems

Ass. Prof. D. Lioupis
Computer Technology Institute, 61 Riga Feraiou St, Patras, Greece
+3061960359
lioupis@cti.gr

A. Papagiannis
Dept. of Computer Engineering & Informatics, Univ. of Patras, Greece
+3061997750
papagian@ceid.upatras.gr

D. Psihogiou
Dept. of Computer Engineering & Informatics, Univ. of Patras, Greece
+3061997750
psixogiu@ceid.upatras.gr

ABSTRACT

The continued growth of microprocessors' performance and the need for better CPU utilization, has led to the introduction of the software peripherals' approach: By this term we refer to software modules that can successfully emulate peripherals that, until now, were traditionally implemented in hardware. Software implementations offer great flexibility in product design and in functional upgrades, while they have high contribution in the cost/performance ratio optimization. We focus on embedded applications, where the cost and the short time to market are the leading issues. In this paper, we study the hardware and software requirements for developing a generic microprocessor with support for software peripherals. Additionally, we present three software peripherals, a Universal Asynchronous Receiver Transmitter, a keypad controller and a dot matrix LCD controller, and we analyze their impact in CPU occupation. Finally, we explore the impact of using a software UART on system power dissipation.

Keywords

Software peripherals, embedded processors, reconfigurable architectures

1. INTRODUCTION

Embedded microprocessors are used in a wide range of applications, from automotive control systems to Palmtops and communication devices. These different markets have a common point: The need for low cost microprocessors, with high level of integration and performance. The growth of the embedded applications' market has brought an increasing migration from application specific logic to application specific code running on embedded processors [9]. The main reasons for this transition from hardware to software are lower cost, flexibility and reduced time to market that software solutions can provide.

The current state in embedded processor market includes a number of different core CPU architectures implemented by several vendors [2]. Examples of such architectures are Motorola's 68000, Intel's i960, Sun's Sparc, Hitachi's SuperH, etc. There are two strategies for integrating peripherals on such microprocessors. According to the first strategy, a basic core and additional logic for a custom device are integrated onto the same die. The second approach uses a standard microprocessor together with a companion chip that serves applications' specific needs [1]. These strategies, and especially the first, are leading to chips that are produced in relatively small volumes due to the fact that they serve only a small range of embedded applications (usually one). Small production volumes are translated in higher final cost. Additionally, developers' choice becomes quite difficult, when they have to choose a microprocessor that covers exactly their needs from the provided variety. This mean more time for searching and learning which entails higher cost of the final product, and longer time to market.

A solution to the problem stated above, is to produce more generic microprocessor chips, which can be software configurable, implementing several peripherals, allowing the resultant generic microprocessor to be tailored to many application areas. In this paper, we study a possible structure for such a microprocessor, which will provide the appropriate flexibility and will be able to constitute a common platform for the application designers.

The remainder of the paper is organized as follows: in the next section we present the current state of the art in the domains of hardware to software migration and reconfigurable architectures. In section 3 we give a brief description of the microprocessor's schema and in section 4 we present the expected benefits of this approach. The utilization of the CPU is the key issue in such a design, thus in section 5 we present the performance analysis concerning a system with three software peripherals. In section 6 we study the effect that a software peripheral may have in power consumption, presenting the comparison between a software and a hardware implemented UART. Conclusions are presented in section 7.

2. RELATED WORK

A large embedded processor manufacturer such as Advanced RISC Machines (ARM) claims that many modern 32-bit RISC processors can be used to implement many functions in software, including signal processing [3]. Recently, software modems have appeared in the market trying to replace modems traditionally implemented in hardware. Similarly, Motorola Semiconductors has developed the SM56 PCI software modem [14] and they aim to establish a 'software communication' market. With this term, they mean that in the future everything except the physical interface will be implemented in software including control, error correction, compression/decompression and modulation. These efforts are focused on high performance desktop processors. In our approach we deal exclusively with embedded processors taking into account their inherent limitations.

Ubicom Inc., (formerly Scenix) [15], has introduced the concept of 'Virtual Peripheral^{TM}', a method of using a portion of the processor's power to perform peripheral functions in software. Their 8-bit RISC-based microprocessor is the platform for running the virtual peripheral modules. Combining advanced architectural features the device is able, in spite of the small data bus, to implement hard real-time functions as software modules to replace traditional hardware functions.

DSP Processors [4] with the appropriate software routines can replace hardware modules of a design (e.g. modems). This category of processors has a special architecture, which help them to execute software related with signal processing quite fast. DSPs can handle in real time tasks that demand high processing power. This kind of processors have been used in several application areas, like cellular telephony, audio and video products etc. This novel market demands has led the manufactures of microprocessors to include DSP functionality into their chips which is achieved by adding fast multipliers, Multiply and Accumulate (MAC) units or adding separate DSP cores into the same chip. Interesting architectures that fall under this category are SH-DSP from Hitachi[11], Picollo from Advanced RISC Machines[12], and Tricore from Siemens[13].

FPGA manufacturers follow a different approach to reconfigurability, based on the idea of reconfigurable hardware. Software, written in some Hardware Description Language, can program the FPGAs to operate as an ASIC. CPU cores are provided as software modules by a lot of companies like Altera Corporation and Xilinx Inc. Additionally, any peripheral can be added to the FPGA as a separate soft macro. New studies in reconfigurable architectures try to integrate FPGA, RISC core, and SRAM within the same die. Garp architecture [5] sets a trend to incorporate RISC cores with FPGA arrays. Towards this direction, Triscend Corporation produces the A7 device family [16] that combines a 32-bit ARM™ processor core with programmable logic and many other system functions onto a single chip. This solution gives great flexibility and could achieve significant speedups compared to traditional General Purpose Processors. However, adding reconfigurable hardware to implement peripherals introduces greater cost and requires more silicon area. In addition, such devices need to be programmed individually before they are used in the field. This limits the scope of such microprocessors to small volumes and thus higher cost.

In our approach, we propose a careful combination of both hardware and software methods to develop peripherals. Our primary concern remains the hardware to software migration. Nevertheless, there are functions that cannot be implemented efficiently with software. Additionally, the purely software approach can be proven inadequate, when we have to deal with demanding peripherals. In this work we present a case where a minor addition in hardware can have beneficial effect in system performance.

3. SYSTEM ARCHITECTURE

In this section we present the hardware and software requirements for developing a generic microprocessor with support for software peripherals. High performance and fast interrupt response are two important requirements, as software peripherals are individual tasks that need to be executed always on time, and microprocessor must be capable to satisfy this demand. Another important issue is the definition of the set of minimal hardware, which is essential for the efficient system operation. The system must also provide a fast and simple way for upgrading peripherals through a well-specified programmable interface, and must be capable to achieve optimal synchronization of all tasks running concurrently, with a robust scheduling algorithm. We suggest that in an embedded system, software peripherals should not occupy more than the 20% of the CPU time. Setting this limitation, we ensure that software peripherals will never introduce great overhead to the system, leaving up to 80% of CPU time for the main application. In our analysis, which is presented in section 5, we will show that this threshold is adequate to emulate our peripherals.

3.1 CPU Architecture Requirements

The functional blocks required by a microprocessor to make it an ideal platform for 'software peripherals', have been defined as hardware functions in the CPU. Figure 1 shows the resultant CPU architecture. As mentioned above, the embedded microprocessor must include:

High performance core: High performance is a critical issue for the described microprocessor. Common techniques to achieve high performance are high clock frequencies, RISC core and pipelining. A high performance CPU can meet our goal of being able to emulate several peripherals, without disrupting the main application.
Fast Interrupt Response: Implementing peripherals in software, increase the software contexts in an embedded system. We use banked registers to reduce context switch time and achieve fast interrupt response.

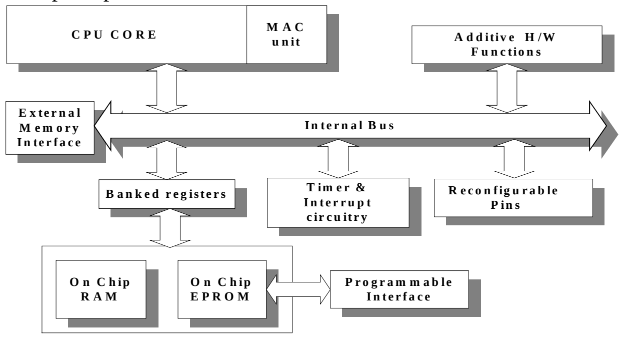

Figure 1. Block diagram of the microprocessor

Set of hardware functions: As it was previously mentioned, there are functions that cannot be implemented efficiently with software. In this work, we use basic hardware functions like timers and interrupt handlers, to implement software peripherals. In other embedded areas like multimedia applications, additional functions in hardware, as for example digital to analog converters, are necessary to retain processor's performance at high level. Furthermore, additions in CPU core, like Multiply and

Accumulate Units, should also be included in requirements to support performance demanding DSP applications.

Reconfigurable Pins: Reconfigurable pins correspond to a common Programmable Peripheral Interface. According to the peripheral set that is loaded to the system, these pins obtain the appropriate functionality.

Sufficient Amount of Memory: The microprocessor should include sufficient internal memory (RAM & EPROM) to satisfy increased system demands due to software peripherals. Through a programmable interface the appropriate peripherals and application code will be loaded or updated. Nevertheless, an external memory interface is necessary for more demanding applications.

A chip designed in such way, allows efficient implementation of software based peripherals and permits its integration in any embedded system. External peripherals can still be employed if necessary.

3.2 Implications on System Software Design

Software peripherals and main application program must execute concurrently. We consider software peripherals as tasks that are waiting for their service. It is possible that complex software peripheral can consist of several simpler ones. We can therefore build a software peripheral based on a hierarchy of simpler functions. These peripherals can be combined in a second level to construct a new peripheral and so on.

Software peripherals introduce extra tasks to the system software design. Scheduling these tasks on the processor, so that all the critical constrains are met, is a difficult problem. A great deal of work has been done on scheduling of embedded systems [6], including those with mixed workloads [7]. We can classify the scheduling policies for real time systems into two categories: *Static or preruntime*, where the scheduling algorithm runs offline and the tasks are well known in advance, and *dynamic or runtime,* where the scheduling is decided online. Each policy suits well in specific cases. In our case, the workload introduced by software peripherals is highly dependent of the target application and so does the scheduling technique. Static scheduling technique can offer a very good optimization when the time that events occur is well known in advance. Round-robin method is probably the simplest solution to our problem. Going a step further, we can use more sophisticated algorithms such as the interval scheduling described in [8]. In the scenario described in section 5, software peripherals are implemented as timer routines having well known occurrences. Thus, static scheduling is applied. On the other hand, when we cannot predict the arrival and the execution time of tasks, dynamic scheduling gives us great flexibility providing on-line scheduling, increasing though the system complexity.

4. System Approach Rationale

The system designed and implemented as above offers the following advantages to the system designer:

- ♦ **Fast Upgrade:** Software peripherals introduce a new fast and simple method of adding peripherals to a system or upgrading the existing ones through programmable interface.
- ♦ **Multiple configurations:** The microprocessor in the described schema has a set of reconfigurable pins. According to the application, the peripheral set is loaded to the processor and the reconfigurable pins obtain the appropriate properties. In this way, multiple configurations of the same chip are possible.
- ♦ **Common development environment:** Application Developers will have one processor for all the different applications that they design. This means less time for learning, great save to expenses of buying different evaluation boards, and shorter time to market for the final product.
- ♦ **Gain in Space:** The microprocessor designers can utilize the saved silicon area to enrich the features of the main CPU core and increase its performance, while at the same time unused functions are eliminated.
- ♦ **CPU Utilization:** CPU power is fully exploited since it is now also used for the execution of peripheral functions and it does not remain inactive for long period of times.
- ♦ **Chip Count Reduction:** The processor will be able to substitute external chips, simplifying the PCB design and reducing the critical time-to-market for the final product.
- ♦ **Low Power Consumption:** The overall power consumption of the application depends on the main processor utilization and the minimal set of hardware functions and not on external chips and circuits. In section 6, we present a case where the software solution is competitive to hardware solution.

Despite the referred advantages, there are also open issues that need to be resolved:

- ♦ **Performance:** Software, of course, cannot replace hardware without trade offs in performance. Emulated peripherals are expected to have lower performance than the hardware ones. Nevertheless the effect of slower peripherals is expected to be minimal as processors become faster.
- ♦ **Synchronization:** In a complex application with several peripherals running in parallel, the synchronization of all tasks is a critical issue. The scheduler's operation and optimization should be carefully studied.

5. PERFORMANCE ANALYSIS

In this section we study the performance of software peripherals implemented in our lab. We chose to implement a combination of three software peripherals that are used in a wide range of embedded applications like cellular phones and Personal Digital Assistants. These peripherals are a) a UART, b) a keypad controller and c) a dot matrix LCD controller. The integration of these peripherals in the embedded system is presented in Figure2:

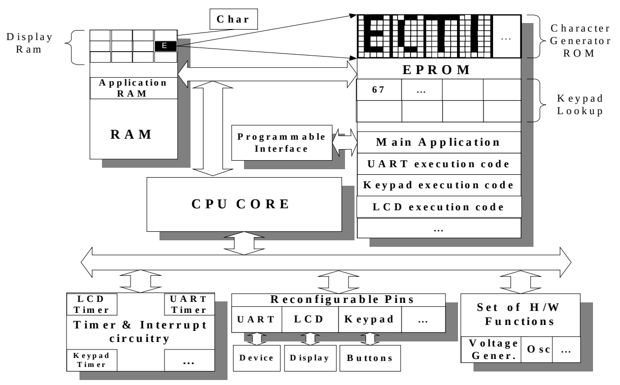

Figure 2. System loaded with three S/W peripherals

We assume that the target processor has the following characteristics: 16 or 32 bit RISC processor with CPI equal to 1 and clock frequencies that vary from 30 to 100 MHz, fast interrupt response, low power features and at least one internal timer for synchronization. These assumptions are not far from reality as there are several processors in the market that correspond to the above characteristics. Examples of such processors are microSPARC-IIep by Sun Microelectronics, the MCF5104 by Motorola, or the 80960HA/HD/HT by Intel etc. All three peripherals mentioned above are analyzed in the next paragraphs.

a) In case of the **UART** implementation, we initially consider the popular 1 Start bit, 8 Data bits, 1 Stop Bit, No parity, which is the simplest and needs the minimal number of instructions. First, we estimate the CPU percentage that the software UART occupies during execution. For the transmission and the receive of one bit the CPUtime is calculated as follows:

CPUtime = (Instruction count x CPI + interrupt response cycles)x(clock frequency)-1 (5.1)

For the CPU occupation, we use the relation:

CPUocc = CPUtime x Baud rate (5.2)

From the implemented UART program, the instruction count is equal to 16. Assuming that CPI is equal to 1, the Clock frequency is in a range from 30 to 100 MHz and the interrupt response is 8 clock cycles, we can conclude that CPU time for one bit lies also in a range from 800nsecs to 240nsecs.

Table 1. CPU Occupation for the Uart

Rate	19200 bit/s	38400 bit/s	57600 bit/s	115.2 Kb/s
MHz				
30	1.536%	3.072%	4.608%	9.216%
50	0.922%	1.843%	2.765%	5.530%
75	0.614%	1.229%	1.843%	3.686%
100	0.461%	0.922%	1.382%	2.765%

As we can see from the above table, the CPU occupation varies from 0.5 % to 10% for all the different frequencies and baud rates. These values indicate that such software peripheral can be incorporated in any application.

In the scenario that we used to calculate the above CPU occupation values, we assumed that one bit is received and transmitted at the same time (Full Duplex). In real applications this scenario rarely happens. The most common case is that some bytes are transmitted and some other bytes are received in separate time intervals (Half Duplex). To serve one of the two functions each time, processor should execute only 10 instructions from our UART code. This observation can reduce the calculated CPU occupation values for about 25% and can make the software solution even more attractive. If we add parity generation and checking functions, the CPU occupation is increased by 11.5% in full duplex mode and 7.6% in half duplex.

b) Next we study the case of a **keypad controller** for an embedded system implemented in software. We simulated the functionality of such a controller, by writing an appropriate software peripheral in assembly language. The keypad controller supports a 4x8 keypad. We used the technique of ‘row scanning’: Each row and each column is connected to the microprocessor. We shift a ‘1’ in each row and we read the output from columns. If a key is pressed a ‘1’ will be scanned in the specific column. The combination row-column reveals the identity of the pressed key through a look-up table. This software peripheral can be fairly implemented as an internal timer routine that is executed every 0.01 sec (scanning frequency). The code length of such a keypad controller is no more than 12 instructions for the service routine and another 32 bytes for the look-up table. An execution of this routine involves about 32 executions of instructions, due to the shifting loop. Then, CPU occupation for the keypad is:

CPUocc = CPUtime x Scanning Frequency (5.3)

where CPU time is calculated from (5.1).

The calculated CPU occupation of this peripheral is extremely low: from 0.013% for a 30MHz CPU to 0.004% for a 100MHz CPU. The main reason for these low values is because the interrupt service routine is executed rarely. Adding more features to the keypad controller, increasing its code complexity and length, will have little impact on CPU occupation which remains far below 1%. The disadvantage of the described solution is the large number of pins that should be used (4+8=12 pins). This problem can be moderated if we use encoder/decoder circuits at the columns/rows of the keypad respectively, reducing the number of used pins to 2+3=5.

c) Finally, we study the case of a **dot matrix LCD controller**. Examples of such LCD controllers are HD44780 from Hitachi [17] and MSM6222B from OKI [18]. These controllers support several features like two different character sizes (5x8 or 5x10), on chip display RAM, on chip character generator ROM, small set of user-programmable character patterns, on chip LCD signal drivers, cursor manipulation instructions etc. Our software LCD controller supports 5x8 dot characters, variable length RAM, according to the size of target LCD and variable length EPROM, according to the size of the character set of the application. There is no need of external RAM and EPROM memories, as the LCD controller can use portions of the on chip RAM and EPROM. For example, a 4x16 character LCD demands 64 bytes for display RAM. Assuming that the target application uses a set of 128 characters, our software controller needs another 5x8x128=5120 bits or 640 bytes of EPROM to store all the character patterns.

This software peripheral can be implemented efficiently enough as an internal timer routine. To calculate the exact frequency that this timer interrupt should occur, we take into consideration the LCD refresh rate and the total size of the display in dots:

Timer frequency = Refresh Rate * X * Y (5.4)

where X and Y are the numbers of dots in horizontal and in vertical dimension respectively. It is efficient to define that the refresh rate of the LCD is equal to 60Hz. Thus, for a 2x16 character display, where each character has 8x5 dots, the calculated timer frequency is:

Timer frequency = 60*(2*8)*(16*5) = 76800 Hz

The software LCD controller should shift a bit to the output every 1/Timer frequency seconds. This bit will be shifted in an external LCD driver like MSM5260 [19] from OKI semiconductor, which will be responsible for the interfacing between the software controller and the target display. The software controller should execute the following operations in order to emulate successively the functionality of a hardware implementation: a) read a character from display RAM, b) find the correct character pattern in character generator ROM, c) load the appropriate 5-bit value that corresponds to the current displayed horizontal dot line, d) shift one bit out and e) occasionally, proceed to the next character, load new horizontal dot line, or go to the next character line.
Shifting is the only operation that is always executed at the timer frequency. All the other operations have fewer occurrences than the shifting operation in the same time interval. For example, the read-from-RAM and the corresponding 5bit-load-from-ROM, occur every 5/(timer frequency) seconds, or the dot-line-change occurs every 5*Y/(timer frequency) seconds, where Y is the number of horizontal dots. Although in our case the total number of instructions is about 35, the average number of instructions executed per interrupt is less than 10 (9.3 in our implementation). Consumed CPU occupation due to this software peripheral is:

$$\mathbf{CPUocc = CPUtime \times Timer\ Frequency} \quad (5.5)$$

where CPUtime is calculated from (5.1) assuming that CPI is equal to 1, as in the case of the UART.

Table 2. CPU Occupation for the LCD controller

LCD Size	2x16	2x20	4x16	4x20
Clock freq	(16x80)	(16x100)	(32x80)	(32x100)
30MHz	4.61%	5.76%	9.22%	11.52%
50MHz	2.76%	3.46%	5.53%	6.91%
75MHz	1.84%	2.30%	3.69%	4.61%
100MHz	1.38%	1.73%	2.76%	3.46%
Display RAM	32 bytes	40 bytes	64 bytes	80 bytes

In table 2 we experimented with four different display sizes and processor's clock frequencies and as we see, the results are quite encouraging for the pure software solution. The maximum CPU occupation is less than 12%, which occurs in the worst case of the lowest frequency processor with the maximum LCD size. CPU occupation increases linearly, thus for a double size LCD, we expect the occupation also to be doubled. In case we wish to save CPU resources for a demanding application we can use an external or internal hardware shift register to reduce CPU occupation by a factor of N, where N is the size of the shift register.

By adding the worst cases for all three peripherals, the maximum occupancy on a 30 MHz CPU is 20.75%. In all other cases, the sum of occupation lies between 1.85% to 13.83% far bellow the requirement of 20%. This result proves that further addition of software peripherals is possible, allowing at the same time enough CPU power for the execution of the main application.

6. POWER CONSUMPTION

Power consumption is an important issue affected by software implementation of peripherals. To estimate the effect on power consumption, we use the 1 Start bit- 8 Data bits- 1 Stop Bit- No parity Full Duplex UART compared with another similar UART implemented in hardware.

In the software implementation the only component that consumes power is the main processor. The main application and the software UART are running concurrently. When the CPU is idle, we consider that it is in power-down mode. In the second implementation there are two components: the main processor as well as the external UART chip. While the CPU is idle, we consider that it is in power-down mode as previously, but for greater time intervals than in software implementation, as the external UART deals with data receiving and transmitting. The system current drain for the first case is:

$$\mathbf{I_{SW} = I_{CPUactive} * y + I_{CPUsleep} * (1\text{-}y)} \quad (6.1)$$

where y is the time portion the software UART occupies the CPU, $I_{CPUactive}$ is the current when the CPU is active and similarly $I_{CPUsleep}$ is the current when the CPU is in power down mode. In the second case the relation becomes as follows:

$$\mathbf{I_{HW} = I_{CPUactive} * (y/8n) + I_{CPUsleep} * (1\text{-}y/8n) + I_{uart}} \quad (6.2)$$

where n is the size of UART FIFO buffer, I_{uart} is the current that the external UART consumes while $I_{CPUactive}$, $I_{CPUsleep}$ and y are same as above. The main application time portion has not been taken into account because it is the same for both occasions and it does not affect the final result.

The factors that affect the power consumption are baud rate, size of FIFO buffer, number of instructions per interrupt, interrupt response time, clock per instruction (CPI), operating frequency, CPU active current, CPU sleep current and external UART current. We present two different cases. In both, the instructions per interrupt is 16, the interrupt response takes 8 cycles, the CPI is equal to 1, the FIFO buffer in the external UART is 16 bytes and the Iuart is 15mA. In the first case we use a 30 MHz CPU with $I_{CPUactive}$ = 300mA and $I_{CPUsleep}$ = 10mA. In the second case we use a 100 MHz CPU with $I_{CPUactive}$ = 900mA and $I_{CPUsleep}$ = 20mA.

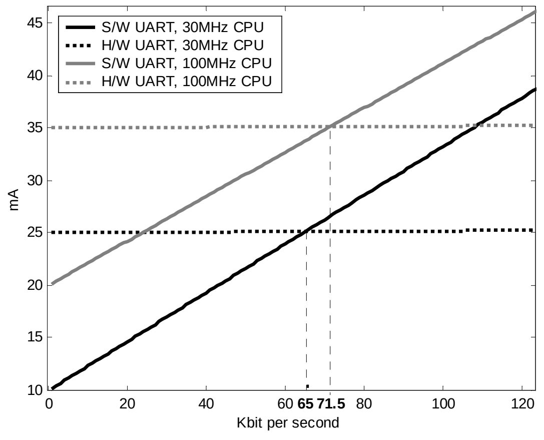

Figure 3. Power consumption curves

As we can see in figure 3, the software implementation consumes less than the hardware implementation up to the point the baud rate reaches close to 70 Kbits/sec. To obtain these results we used a simple power estimation model. We also made conventions about the way the microprocessor operates. For example, we assumed that there is no time or consumption penalty during a transition from sleep mode to operational mode and vice-versa. More detailed power estimation models are described in [9], [10].

7. CONCLUSIONS AND FUTURE WORK

We have presented a systematic approach to peripherals for embedded systems, implemented in software. We tried to exploit the extra performance that modern processors offer, replacing traditionally hardware peripherals, with equivalent software ones. The basic idea that led us to this direction of 'software migration' was to produce flexible embedded systems without any 'glue logic'. We constructed in software three popular peripherals, an UART, a keypad controller and a dot matrix LCD controller. We investigated their efficiency and the load that they introduce to the main processor. In the case of the UART we also studied its behavior from the scope of power consumption, comparing it with that of an external hardware UART. We conclude that we can have an equivalent system using software peripherals, at an acceptable performance. In particular:

- ♦ Software peripherals can provide a feasible alternative, offering great flexibility and simplifying the microprocessor design as well as the design of the final embedded system.
- ♦ They can dramatically reduce the final cost of an embedded application and retain the overall performance in a satisfactory level, giving an excellent cost/performance ratio.
- ♦ Software peripherals can follow the rapid microprocessor advances. As the microprocessors get faster the performance of software peripherals will also increase.

All the three peripherals that we studied had little impact on CPU performance, which decreases linearly as the clock frequency of the processor is increased. We should also point out that when a software peripheral overcomes the desired threshold of CPU occupation, small hardware additions, like the addition of a shift register in the LCD controller case, might have catalytic impact in the system performance.

The future directions of this work will be the thorough definition of a minimal set of hardware peripherals that are used by a wide range of embedded applications and cannot be implemented in software. Additionally more complicated software peripherals will be implemented and studied. Finally, we will also turn into the domain of embedded scheduling and study the feasibility of systems with a substantial number of software peripherals and mixed workloads.

8. ACKNOWLEDGEMENTS

This work was supported by the Caratheodory Programme of the University of Patras.

9. REFERENCES

[1] Manfred Schlett "Trends in Embedded-Microprocessor Design", IEEE Computer Aug. 1998 pp 44-49

[2] J.Turley, "Evaluating Embedded Processors", Micro Design Resources, Sebastopole, Calif., 1997.

[3] Dave Walsh, "Reducing System Cost with Software Modems", IEEE Micro August/July 1997.

[4] Jennifer Eyre, Jeff Bier, "DSP processors hit the mainstream", IEEE Computer, Aug. 1998 pp 51-59

[5] Hauser, J. R., Wawrzyneck J., "Garp: A MIPS Processor with a Reconfigurable Coprocessor" Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.

[6] Felice Balarin, Luciano Lavagno, Praveen Murthy, and Alberto Sangiovanni-Vincentellii "Scheduling for Embedded Real-Time Systems" IEEE Design & Test of Computers, Vol. 15, No. 1, January/March 1998.

[7] Mark K. Gardner, Jane W.S. Liu, "Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload", Proc. of 11th Euromicro Conf. on Real-Time Systems, June 1999 York, England.

[8] Pai Chou, Gaetano Borriello, "Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems", 32nd ACM/IEEE Design Automation Conference, 1995.

[9] V. Tiwari, S. Malik, A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization", IEEE Trans. on VLSI Systems, Dec. 1994.

[10] Jeffry T. Russell, Margarida F. Jacome, "Software Power Estimation & Optimization for High Performance 32bit Embedded Processors" Proceedings of ICCD '98, 5-7 October 1998 in Austin, Texas.

[11] SH-DSP Microprocessor Overview. Hitachi Semiconductors Inc, 1998 Doc. Number PMH1DTB001D1.

[12] ARM Signal Processing Architecture Reference Manual. ARM Ltd, 1997. Doc. Number: ARM IP0025B-07.

[13] TriCore Architecture Overview. Siemens Microelectronics, Inc., 1997. Order Number: M32T008.

[14] Motorola Corporation software modem Home Page, http://www.mot.com/softmodem

[15] Ubicom (formerly Scenix) Corporation Home Page, http://www.ubicom.com

[16] Triscend Corporation, A7 CSoC family: http://www.triscend.com/products/indexA7.html

[17] Hitachi, HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver, ADE-207-272(Z) '99.9, Rev. 0.0.

[18] OKI Semiconductor, Dot Matrix LCD Controller with 16-dot Common Driver and 40-dot Segment Driver MSM6222B-xx. Version November '97.

[19] OKI Semiconductor, 80-dot Common/Segment Driver MSM5260. Version November '97.