Programming paradigm of a Microcontroller with Hardware Scheduler Engine and Independent Pipeline Registers – A software approach

Lucian ANDRIES[1,2], Vasile Gheorghita GAITAN[1,2] , Elena-Eugenia MOISUC[1,2]
[1]Faculty of Electrical Engineering and Computer Science, Ştefan cel Mare University of Suceava
[2]Integrated Center for Research, Development and Innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for Fabrication and Control (MANSiD), Ştefan cel Mare University of Suceava
Suceava, Romania
andries.lucian2002ro@gmail.com, gaitan@eed.usv.ro

***Abstract*— In computer science, for embedded field only two types of microcontrollers exists, that can be used to develop a working system. You can use a single core or a multicore which is much faster but will not be the equivalent of a single core multiple with the numbers of cores, because a small part from the power will be used for inter process communications. Our approach is a little bit different because we use a single core CPU that have a number of finite task that act like different CPU's. In this new architecture there is no need for inter process communication because the processor is a single core and the hardware tasks use the same resources as others. The peripherals of this architecture will improve interrupt latencies and task switching times, what makes this microcontroller the best choice when it comes in interrupt response time.**

Keywords—real time system, static hardware scheduler, microcontroller.

I. INTRODUCTION

Usually, embedded systems contains a real time operating system that can perform very complex tasks for small or big systems. An embedded system can be found almost anywhere on this planet, from the watch at your wrist, the parking barrier that will allow you to enter into the parking space till to the spaceship and satellites from the sky.

For each application areas different microcontrollers and operating systems are used because of different requirements of the operating environment.

In this article we are going to focus on a small niche of the real time embedded field, the field of microcontrollers that need a fast response time and a small jitter for external interrupts.

Usually for this kind of applications the programming language used is C, because provide low level access to memory and have minimal run tine support ([1]). The C programming language is considered a low level programming language that was designed to encourage cross platform programming.

The C++ language is also a general purpose programming language that provide facilities for low-level memory manipulation, but it cannot be used in the embedded field because the gpp compiler ([2] C++ compiler), will generate a lot more machine code than the gcc (C compiler) compiler for C. The gpp compiler will generate much more machine code because of the object orientated nature:

- Code will be generated for constructor and destructor. The code of destructor will never be called because an embedded system is supposed to work nonstop.
- Code will be generated for run time polymorphism
- Code will be generated for protective layer of a class: private, protected.

The C language compared with C++ seems to be really small, but the simplicity makes this programming language so powerful.

We chose the C language, to create the environment for a microcontroller that is a single core, but its hardware threads act like separate processors, because the architecture being developed, is going to be used for fast task switch and small interrupt latencies.

In the following chapter, the way that a gcc compiler can be used to generate code for this kind of architecture will be presented. The programming paradigm for this architecture requires, from the programmer, very good knowledge about hardware, programming language and compilers.

This article describe the programming paradigms of processor architecture detailed in [3] and highlights the consequences that appear when a processor is using hardware tasks and a compiler that is not build for this kind of architecture.

This paper is organized as follows. The programming of nMPRA and NHSE architecture is presented in Chapter II which consists of initialization of the SCPUi followed by examples how to use the hardware task and programming paradigms. The final conclusions are drawn in Chapter III.

978-1-4799-8481-7/15/$31.00 ©2015 IEEE

II. Programming Paradigm of nMPRA and nHSE Architecture

In article [3] is presented a processor architecture that offer hardware support for real time operating system (nHSE) and hardware synchronization between tasks (nMPRA):

- nMPRA (multiple pipeline registers architecture for n tasks): offer support for hardware synchronization between tasks and peripherals.
- nHSE (hardware scheduler engine for n tasks): offer support for static and dynamic hardware scheduler for n tasks.

Writing code in assembler or in a high level programming language the compiler will generate machine code that will be executed sequentially. The used resources are only the 32 registers that are used for local variables, function call and normal computation.

The new architecture offers some important advantages:

- Fast response time for a task switch. The stack and registers are not used for the context switch.
- No task can alter the state of another task.
- The event's that may appear will be served faster.

The first two advantages changed the programming paradigm of a single core microcontroller because the hardware tasks (Semi processor i - SCPUi) are acting like independent control process unit (CPU) despite the fact that it's not the case.

The MIPS32 Release1 architecture was used and tested using the gcc toolchain built from the following tools and the C programming language:

- binutils-2.24.
- gcc-4.9.1.
- gmp-6.0.0a.
- mpc-1.0.2.
- mpfr-3.1.2.
- newlib-2.1.0.

The new paradigm will be detailed in the following lines.

A. SCPUi initialization

Each SCPUi must initialize its own registers and coprocessor 2 (COP2) module correctly for a proper operation because these hardware resources are not shared (Figure 1).

After RESET only SCPU0 is active and must be used to initialize the others SCPUi in the following steps:

- SCPU0 will set the recurrence of each SCPUi (function ***initThread***, line 23 from below code example) with the lowest recurrence starting with the least SCPUi priority because the SCPU0 (line 26 from below code example) must be the last task that is called before initialization of the Scheduler ([3]). Only SCPU0 has the rights for configuring the Scheduler.
- SCPU0 configure the Scheduler to use a non-preemptive algorithm.
- SCPU0 configure the pointer of each SCPUi task (line 44, 45 from below code example) from the Scheduler peripheral with the address of a function that will initialize the registers and local registers of nHSE, nHSE_lr. Each init function for SCPUi, after the initialization has finished will inform the Scheduler that the task has finished successfully and will enter to an endless loop. If not, the task will run forever because the Scheduler will consider that the task is still running.

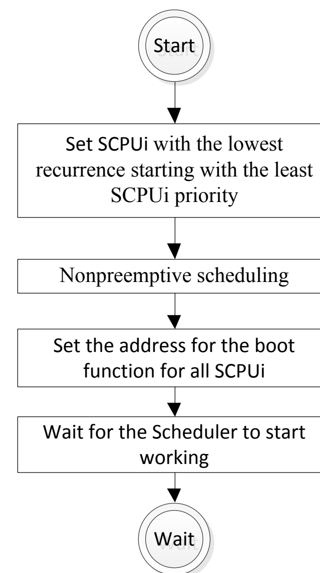

Figure 1 - Boot process for SCPUi

- The first task that will start to execute code will be the task with less priority, in our case task1, followed by the task with higher priority

Code example for SCPUi initialization of two tasks:

```
1.    #define SCHEDULER_ADDR  0xE0000000
2.    #define REGISTER_ASSIGMENT(val) {\
3.        __reg = (unsigned long)(val);
4.        asm("addiu   $8,$8, 0\n\t");\
5.    }
6.    //force variable _reg to be placed in register $8
7.    volatile register unsigned long __reg asm ("$8");
8.
9.    inline void taskFinishExecution(void){
10.       REGISTER_ASSIGMENT(0); asm ("mtc2 $8, $31");
11.   }
12.
13.   void initThread(void);
14.   void Task0Init();
15.   void Task1Init();
16.
17.   void main(void){
18.     initThread();
19.
20.     while(1);
21.   }
22.
```

```
23.   void initThread(void){
24.     //SCPU1, SCPU0 recurrence
25.     *((volatile uint32_t*)SCHEDULER_ADDR+3) = 300;//300 machine cycles for
      SCPU1
26.     *((volatile uint32_t*)SCHEDULER_ADDR+2) = 10; //10 machine cycles for SCPU0
      in order to start immediately
27.
28.      /*Scheduler preemptive, no prescaler for SCPUi recurrence
29.      Bits: 0-3: SCH_TMR1 prescaller (SCPU0 clock prescaller)
30.            4-7: SCH_TMR2 prescaller (SCPU1 clock prescaller)
31.            8-11: SCH_TMR3 prescaller (SCPU2 clock prescaller)
32.            12-15: SCH_TMR4 prescaller (SCPU3 clock prescaller)
33.            16-19: SCH_TMR5 prescaller (SCPU4 clock prescaller)
34.            20-23: RoundRobin timeStamp prescaller
35.            27: enable: preemtiv, disable: nepriemtiv
36.            28: enable SCPU1
37.            29: enable SCPU2
38.            30: enable SCPU3
39.            31: enable SCPU4
40.       */
41.      *((volatile uint32_t*)SCHEDULER_ADDR+0x10) = 0xF8111111;
42.
43.      /*initialize the address of each task*/
44.      *((volatile uint32_t*)SCHEDULER_ADDR+0x10) = &Task0Init;
45.      *((volatile uint32_t*)SCHEDULER_ADDR+0x11) = &Task1Init;
46.
47.      /*Scheduler nonpreemptive, no prescaler for SCPUi recurrence*/
48.      *((volatile uint32_t*)SCHEDULER_ADDR+0x10) = 0xF0111111;
49.      /*the Scheduler will start running*/
50.      /*enable the Scheduler*/
51.      *((volatile uint32_t*)SCHEDULER_ADDR+0x0D) = 1;
52.
53.      taskFinishExecution();
54.   }
55.   void Task0Init() {
56.      asm( "jal initTask0_NHSE_CP2 \n\t"//jump to function
57.           "nop\n\t"
58.           "jal SCHEDULER_Init\n\t"//jump to function
59.           "nop\n\t");
60.       taskFinishExecution();
61.   }
62.   void Task1Init(){
63.      asm("jal initTask1_NHSE_CP2 \n\t" //jump to function
64.          "nop\n\t");
65.      /*reset the recurrence of the task*/
66.      *((volatile uint32_t*)SCHEDULER_ADDR+3) = 0xFFFF;
67.       taskFinishExecution();
68.   }
```

Where:

- ***SCHEDULER_ADDR*** (line 1 from above code example) is the base address of the Scheduler peripheral that is located on the slow bus.
- ***initTask0_NHSE_CP2*** (line 56 from above code example) and ***initTask1_NHSE_CP2*** (line 63 from above code example) are the functions that are called to initialize the COP2 for task 0 and task 1. Each task will have a different function for initialization because the nHSE_lr can behave differently for each task.
- ***SCHEDULER_Init*** (line 58 from above code example) is the function that is called to initialize the scheduler with the correct recurrence of all available tasks.
- **SCH_Task0Finished()** (line 60 from above code example) and ***SCH_Task1Finished()*** (line 67 from above code example) function will inform the Scheduler that the tasks has finished successfully and will enter deep sleep produce by COP2.

B. Using the hardware tasks

In order to use the power of the hardware tasks, the task recurrence (the recurrence, which each task will be called) and start address of each task function (Figure 2), must be configure correctly.

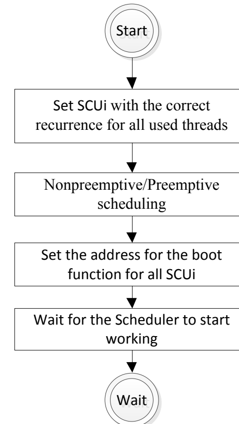

Figure 2 – Steps for Scheduler configuration

Code example for initialization of two tasks:

```
1.   static void Task0Function(void){}
2.   static void Task1Function(void){}
3.
4.   void main(void)
5.   {
6.      //SCPU0, SCPU1 recurrence
7.      *((volatile uint32_t*)SCHEDULER_ADDR+2) = 1000;//1000 machine cycles for
     SCPU0
8.      *((volatile uint32_t*)SCHEDULER_ADDR+3) = 2000;//2000 machine cycles for
     SCPU1
9.
10.     /*Round robin timer that is used to supervise the active time   of the
11.       current task.*/
12.     *((volatile uint32_t*)SCHEDULER_ADDR+1) = 1000;//RBTM timer(machine cycles)
13.
14.      /*Scheduler preemptive, no prescaler for SCPUi recurrence
15.      Bits: 0-3: SCH_TMR1 prescaller (SCPU0 clock prescaller)
16.            4-7: SCH_TMR2 prescaller (SCPU1 clock prescaller)
17.            8-11: SCH_TMR3 prescaller (SCPU2 clock prescaller)
18.            12-15: SCH_TMR4 prescaller (SCPU3 clock prescaller)
19.            16-19: SCH_TMR5 prescaller (SCPU4 clock prescaller)
20.            20-23: RoundRobin timeStamp prescaller
21.            27: enable: preemtiv, disable: nepriemtiv
22.            28: enable SCPU1
23.            29: enable SCPU2
24.            30: enable SCPU3
25.            31: enable SCPU4
26.      */
27.     *((volatile uint32_t*)SCHEDULER_ADDR+0x10) = 0xF8111111;
```

```
28.
29.     /*initialize the start address of each task*/
30.     *((volatile uint32_t*)SCHEDULER_ADDR+0x10) = &Task0Function;
31.     *((volatile uint32_t*)SCHEDULER_ADDR+0x11) = &Task1Function;
32.     /*the Scheduler will start running*/
33.     *((volatile uint32_t*)SCHEDULER_ADDR+0x0D) = 1;
34.
35.     while(1);
36. }
```

In the example above the Scheduler ([3]) module can be access as any other peripheral from the slow bus and the code for void main function is executed by the SCPU0 for the first time. After the initialization of all hardware task is finished the Scheduler will enable the task with the smaller recurrence from the system to start executing code.

C. *Using shared variables between tasks*

Using global variables that are shared between SCPUi, the compiler might consider that the code generation can be optimize (internal implementation of the compiler) to use local variables that can be stored in registers. This case may appear when a variable is initializing globally and read in each task. Only the task used for initialization of this variable will see the value because is the first task to execute code. In this case the hardware task may not working properly because it's expecting a value that is not present. For eg: A global variable that is used to enable the reading of an analog pin. The first task that execute the initialization routine will use the correct value from the variable while the other tasks will not read the analog pin because the value that enable the readings is not present.

The same result will be if a variable is modified in a task and read in other task, because the variable will be optimize by the compiler and used locally.

To avoid this kind of behavior the variable must be forced to be placed in RAM (MIPS32 architecture cannot perform operation in directly in RAM, therefore the values from RAM must be loaded into the local registers) by the compiler or to use the mechanisms provided by the nHSE architecture, the hardware mutexes that are atomic operations.

D. *Interfacing with COPROCESSOR 2 (nHSE_lr)*

The local registers, of nHSE architecture, which are implemented as COP2, are appointed as nHSE_lr and the global registers of nHSE are appointed as nHSE_gr.

A coprocessor can be used to supplement the function of the CPU. The MIPS architecture support 4 coprocessors:

- Coprocessor 0 (COP0): supports exceptions and virtual memory system.
- Coprocessor 1 (COP1): reserved for floating point custom implementation.
- Coprocessor 2 (COP2): available for user defined implementation.
- Coprocessor 3 (COP3): reserved for floating point module in Release 1 implementation of the MIPS64 architecture and on all release 2 implementations of the architecture.

The MIPS32 architecture has provided a means by which can be upgraded to do custom computation. In this case we discuss about the COP2 that has some standard assembler instruction that can be used to interface with:

- mtc2 – move word to coprocessor 2
- mfc2 – move word from coprocessor 2

These two assembler instructions are enough for a proper use of the new module. The gcc compiler is not offering support for the C programming language to work with the COP2 assembler instructions. Therefore we had to mix C with assembler instructions.

1) *Writing to nHSE_lr registers*

When, only one variable is used, inside a function, the register $8 (t0) is used to store the value (Table 1).

Table 1 – Assembler code generated from C

C language	Generated assembler
1. uint32_t localVariable = 0;	1. move t0,a0 2. addiu t0,t0,0

Further we will present a function that can be used to write into the registers from COP2 (Table 2):

Table 2 – Function used to write to nHSE_lr registers

C language	Generated assembler
1. //force variable _reg to be placed in 2. //register $8 3. volatile register 4. unsigned long __reg asm ("$8"); 5. 6. inline void crTR_SET(unsigned long val){ 7. //val will be assigned to reg $8 8. __reg = (unsigned long)(val); 9. asm("addiu $8,$8, 0\n\t"); 10. //load reg $8 value to reg $0(crTR) 11. asm("mtc2 $8, $0"); 12. }	1. 00000e14 <crTR_SET>: 2. e14: 00804021 move t0,a0 3. e18: 25080000 addiu t0,t0,0 4. e1c: 03e00008 jr ra 5. e20: 48880000 mtc2 t0,$0

2) *Reading from nHSE_lr registers*

The same principle applied to writing to nHSE_lr registers will also be applied in this context as can be seen in Table 3:

Table 3 - Function used to read from nHSE_lr registers

C language	Generated assembler
1. //force variable _reg to be placed in 2. //register $8 3. volatile register 4. unsigned long __reg asm ("$8"); 5. 6. inline unsigned long crTR_GET(void){ 7. //load reg $0(crTR) value into _reg 8. //variable 9. asm("mfc2 $8, $0"); 10. return __reg; 11. }	1. 00000e24 <crTR_GET>: 2. e24: 48080000 mfc2 t0,$0 3. e28: 03e00008 jr ra 1. e2c: 01001021 move v0,t0

E. Interfacing nHSE_lr with Scheduler

The interaction between the nHSE_lr and the Scheduler is absolutely necessary because the COP2 instructions get to be very powerful. The mtc2 instruction can be used to create more software custom instructions that can be used to:

- Create an atomic instruction that can be executed in 2 machine cycles. nHSE_lr is used to write directly, at a specified address the data, into nHSE_gr peripheral.
- Create an atomic instruction that can force a SCPUi thread to enter into a true sleep mode. nHSE_lr will inform the Scheduler that the task entered into sleep mode in order not to promote the current task to long task queue (LTQ [3]) .

The custom instructions are created in software by writing the instructions mnemonic into a 32 bit register which is then used to be send to the nHSE_lr. The register used is $31.

1) Instruction used to signal the finish of a task

Table 4 - Instruction used to signal the finish of a task

Instruction	0 3	4 31
taskFinishExecution	Opcode 0	n/a

The instruction will alert the Scheduler, by hardware that the task has finished successfully and can be restarted again (Table 4).

2) Instruction used to force a SCPUi to enter sleep mode

Table 5 - Instruction used to enter sleep mode

Instruction	0 3	4 31
taskSleep	Opcode 1	n/a

To decode the instruction nHSE_lr need only the opcode because the behavior is predefined in the module. The instruction will stall all the pipeline registers, in this way the SCPUi state will be freeze (Table 5).

3) Instruction used to Lock/Release a mutex

Table 6 - Instruction used to Lock/Release a mutex

Instruction	0 3	4 11	12 19
mutexLock	Opcode 2	Task Id	Mutex Id

A mutex can be released only by the task that locked it. If another task try to lock a mutex that is already locked nothing will happened. The same instruction is used to release the mutex when is called the second time.

Mutex Id (Table 6) will be used to compute the address and the Task Id for data that are going to be written into grMutex registers from nHSE_gr.

4) Instruction used to stop a SCPUi from execution

Table 7 - Instruction used to stop a SCPUi from execution

Instruction	0 3	4 27	28 31
CRMSTOP	Opcode 3	CR0MSTOP	CR0_STOP_ADDR

Each bit from the register CR0MSTOP will cause the SCPUi to be stopped from the execution immediately (Table 7).

5) Instruction used to reset a SCPUi

Table 8 - Instruction used to reset a SCPUi from execution

Instruction	0 3	4 27	28 31
CRMRESET	Opcode 3	CR0RESET	CR0_RESET_ADDR

III. CONCLUSIONS

In this paper we created the development environment for the nHSE architecture that is functional and can be used to create applications. The drawback of this architecture is a more restrictive environment that the one with a single core, because the gcc compiler will generate code only for single core microcontrollers. The programming parading is not that easy to understand because of the knowledge in embedded and hardware that is needed to handle this environment.

Despite the fact that we need big amount of programming experience in embedded, the architecture along with the C programming language can create really fast applications that can handle a much more amount of external interrupts and internal events than an ordinary microcontroller.

Also with this architecture we can create more secure and hard to crack applications, when it comes to third party, because no SCPUi can alter the internal state of another SCPUi.

IV. FUTURE WORK

A point that must be taken in consideration for future development is to create an easier framework for the programmer or create a custom compiler for this processor architecture, in order to make this approach more popular.

V. ACKNOWLEDGMENT

This paper was supported by the project "Sustainable performance in doctoral and post-doctoral research PERFORM - Contract no. POSDRU/159/1.5/S/138963", project co-funded from European Social Fund through Sectorial Operational Program Human Resources 2007-2013.

This work was partially supported from the project Integrated Center for research, development and innovation in Advanced Materials, Nanotechnologies, and Distributed Systems for fabrication and control, Contract No. 671/09.04.2015, Sectoral Operational Program for Increase of the Economic Competitiveness co-funded from the European Regional Development Fund.

References

[1] Zhikao Ren, Coll. of Inf., Qingdao Univ. of Sci. & Technol., Qingdao, China, Chen Ye, Guozu Liu, “Application and Research of C Language Programming Examination System Based on B/S”, ISBN 978-1-4244-8627-4.

[2] Wu, Z., Comput. Lab., Cambridge Univ., UK, "Making C++ a distributed programming language", ISBN 0-8186-4430-3 http://en.wikipedia.org/wiki/GNU_toolchain

[3] Gaitan, V.G.; Gaitan, N.C.; Ungurean, I., "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.PP, no.99, pp.1,1, ISSN : 1063-8210, doi: 10.1109/TVLSI.2014.2346542.

[4] Andries, L.; Gaitan, G., "Dual priority scheduling algorithm used in the nMPRA microcontrollers," *System Theory, Control and Computing (ICSTCC), 2014 18th International Conference* , vol., no., pp.43,47, 17-19 Oct. 2014, doi: 10.1109/ICSTCC.2014.6982388

[5] Gaitan, N.C.; Gaitan, V.G.; Moisuc, E.-E.C., "Improving interrupt handling in the nMPRA," *Development and Application Systems (DAS), 2014 International Conference on* , vol., no., pp.11,15, 15-17 May 2014 doi: 10.1109/DAAS.2014.6842419.

[6] Gaitan, N.C.; Andries, L., "Using Dual Priority scheduling to improve the resource utilization in the nMPRA microcontrollers," *Development and Application Systems (DAS), 2014 International Conference on* , vol., no., pp.73,78, 15-17 May 2014, doi: 10.1109/DAAS.2014.6842431.

[7] Moisuc, E.-E.C.; Larionescu, A.-B.; Gaitan, V.G., "Hardware event treating in nMPRA," *Development and Application Systems (DAS), 2014 International Conference on* , vol., no., pp.66,69, 15-17 May 2014 doi: 10.1109/DAAS.2014.6842429.

[8] Moisuc, E.-E.C.; Larionescu, A.-B.; Ungurean, I., "Hardware event handling in the hardware real-time operating systems," *System Theory, Control and Computing (ICSTCC), 2014 18th International Conference*, vol., no., pp.54,58, 17-19 Oct. 2014, doi:10.1109/ICSTCC.2014.6982390

Submission number	**138**
uthors or proposers	**Andries*, Lucian (Stefan Cel Mare University of Suceava) (59273)** **Gaitan, Vasile Gheorghita (Stefan cel Mare University of Suceava) (58655)** **Moisuc (Ciobanu), Elena-Eugenia (Stefan cel Mare University of Suceava) (58709)**
Title	**Programming Paradigm of a Microcontroller with Hardware Scheduler Engine and Independent Pipeline Registers – a Software Approach**
R for supplementary information	
Type of submission	Contributed paper
Recei ed on	une 4, 2015
Code	
ey ords	Embedded Systems
bstract at initial submission	In computer science, for embedded exists only two types of microcontrollers that can be used to develop a working system. You can use a single core or a multicore which is much faster but will not be the equivalent of a single core multiple with the numbers of cores, because a small part from the power will be used for inter process communications. Our approach is a little bit different because we use a single core CPU that have a number of finite task that act like different CPU's. In this new architecture there is no need for inter process communication because the processor is a single core and the hardware tasks use the same resources as the others. The peripherals of this architecture will improve interrupt latencies and task switching times, what makes this microcontroller the best choice when it comes in interrupt response time.
ttachments	Copyright Transfer orm (attachment to final submission). Status Recei ed
rofile	Contributed apers
Status	**Final version received**
ate of latest decision or action	ugust 5, 2015
bstract	In computer science, for embedded field only two types of microcontrollers exists, that can be used to develop a working system. You can use a single core or a multicore which is much faster but will not be the equivalent of a single core multiple with the numbers of cores, because a small part from the power will be used for inter process communications. Our approach is a little bit different because we use a single core CPU that have a number of finite task that act like different CPU's. In this new architecture there is no need for inter process communication because the processor is a single core and the hardware tasks use the same resources as others. The peripherals of this architecture will improve interrupt latencies and task switching times, what makes this microcontroller the best choice when it comes in interrupt response time.
umber of pages in final manuscript	6
Copyright transferred	
Session	Embedded Systems (Regular Session)
Schedule code	Th 5.6
Scheduled time of presentation	Thursday October 15, 2015 10:50−12:50 Room 5 12:30−12:50
	The session, schedule code and scheduled time of presentation, if present, are tentati e. lease refer to the final program

Proceedings of 2015 19th International Conference on System Theory, Control and Computing (ICSTCC)

Joint conference SINTES 19, SACCS 15, SIMSIS 19

October 14 - 16, 2015

CheileGradistei - Fundata Resort, ROMANIA

Editors:

Sergiu Caraman

Marian Barbu

Răzvan Şolea

IEEE Catalog Number: CFP1536P-USB
ISBN: 978-1-4799-8480-0

Copyright and Reprint Permission: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limit of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid

pp. 693-698

Postolache, Mihai Gheorghe Asachi Tech. Univ. of Iasi

CAN (Controller Area Network) and TIA-485 are two of the most used standards in fieldbus systems. While CAN ISO IS-11898 includes complete data link layer specifications on top of its physical layer, TIA-485 only addresses the physical layer of the 7-layer OSI model. Other communication parameters like speed, format, and data transmission protocol are not specified by RS-485 in order to provide interoperability of similar devices from different manufacturers. After a brief introduction and comparison of the two communication protocols, the paper investigates to what extent the CANopen specification - an application level protocol developed primarily for CAN networks - should be used in TIA-485-based fieldbus networks in order to provide the user an application programming interface (API) independent of the physical layer. Selected and customized CANopen services have been implemented and tested on a network of microcontroller-based stations equipped with CAN ISO 11898 and TIA-485 communication interfaces.

12:10-12:30 ThA5.5

Future House Automation, pp. 699-704

Florea, Adrian Lucian Blaga Univ. of Sibiu
Bancioiu, Iosif Lucian Blaga Univ. of Sibiu

In this paper, we propose the future house automation, a PLC-based embedded system that aims reducing the house energy consumption by optimizing the entire hardware assembly and software algorithms. The project started from the idea of designing a self-controlled house, to increase user's comfort in his daily environment, reducing the cost and optimizing the energy consumption. Our embedded application represents a green solution into a growing number of environmentally aware consumers, very suitable for the market of energy-efficient control systems. We provide a cheap solution for developing by everyone its own automation system control house. Therefore, our project contributes for helping the elderly, which represents another social challenge with global character.

12:30-12:50 ThA5.6

Programming Paradigm of a Microcontroller with Hardware Scheduler Engine and Independent Pipeline Registers – a Software Approach, pp. 705-710

Andries, Lucian Stefan Cel Mare Univ. of Suceava
Gaitan, Vasile Gheorghita Stefan Cel Mare Univ. of Suceava
Moisuc (Ciobanu), Elena-Eugenia Stefan Cel Mare Univ. of Suceava

In computer science, for embedded field only two types of microcontrollers exists, that can be used to develop a working system. You can use a single core or a multicore which is much faster but will not be the equivalent of a single core multiple with the numbers of cores, because a small part from the power will be used for inter process communications. Our approach is a little bit different because we use a single core CPU that have a number of finite task that act like different CPU's. In this new architecture there is no need for inter process communication because the processor is a single core and the hardware tasks use the same resources as others. The peripherals of this architecture will improve interrupt latencies and task switching times, what makes this microcontroller the best choice when it comes in interrupt response time.