# Implementation of RTOS Kernel in Hardware and the Scope of Hybridization of RTOS

Ponnaganti Sudhi Varun
M.Tech (EST), Dept. of ECE,
SRM University, Kattankulathur

## ABSTRACT
Real time operating systems have become an integral part of the embedded systems software. They play crucial role in allocating the available constrained resources efficiently. The resource can be anything like ALU in the processor, networking hardware, memory etc. The user program is broken into tasks and they will be competing for the resources. But in software RTOS, the RTOS itself has few tasks along with the user tasks which are resource hungry. So the resources are not confined to the user programs which reduce the efficiency of the overall system. The idea of implementing the RTOS in hardware itself comes here. Only the user program is to be programmed. The efficiency has increased drastically with some limitations. Here, a typical basic processor is developed. How the software RTOS effects the efficiency is discussed. Then the same processor is added to the hardware OS kernel. Scope of hybridization of RTOS is given which increases the efficiency to greater extent. The concept of hybridization is provided with an example and a programming environment for such hybrid RTOS is emphasized.

## General Terms
Real time operating system, Hardware kernel, Hybrid RTOS, Programming environment for hybrid RTOS, FPGA.

## Keywords
Real time operating system, Hardware kernel, Hybrid RTOS, Hardware scheduler, Hardware semaphore manager.

## 1. INTRODUCTION
Operating systems for embedded applications have following features. Encapsulates the hardware and provides easy to use style interfaces to the programmer. Schedules user tasks based on priorities assigned to them. Provides task synchronizing mechanisms. Provides communication services between tasks. These are the basic services provided by most of the RTOS kernels. The RTOS most of the time is a software by itself. So it shares the resources with the user tasks. Thus reducing the efficiency. Here efficiency is measured as the ratio of machine cycles expended on user tasks to the total number of machine cycles over a considerable period of time. In a time period driven scheduler, most of the machine cycles will be expended on updating the timer values, checking the timer outs, context switching, checking the priorities of the tasks whose timers are out and triggering them.

The basic processor which is shown in the figure 1 consumes three to four machine cycles for normal instruction. The operating system need to be invoked at the end of every instruction either by a special instruction or interrupt. For 4 user tasks which are to be scheduled based on timers and priority, the overhead imposed by the RTOS scheduler itself is very high.

The cycle requirement of the basic processor can be seen in the figure 1. It is capable of loading values in accumulator, memory, data transfer between memory, basic conditional and unconditional jumps in the program ROM by changing the value in the program counter, basic arithmetic operations, increment and decrement the accumulator. In the figure 1, the pcvalue shows the current program counter value. The value is changed abruptly because of jump instruction. The accumulator is loaded with the number 5, and then it is written to RAM. From RAM this value is read into regB and then addition is performed. Finally the result in the accumulator (regA) is outputted through outport. Such instructions are sufficient to build a simple scheduler, task switching and basic user programs.

Since there are four user tasks, each task will have its own accumulator, program counter, Timer etc at least as seen from the perspective of a user. Task switching means to save the contents of these on RAM and bring in the context of the task that is to be run. So 12 cycles are expended here. To decrement the timers of each task, 4 more cycles are used. To check the timer outs, another 4 more. And this continues further when the scheduler has to check priority and switch the task. The basic processor gave a clocking capability of 88 MHz theoretically when programmed on target device (Xilinx Spartan XC3S4000).

That means if the processor is programmed with just only user tasks without scheduler of RTOS, the user tasks straight away sees this clocking speed. If the scheduler is implemented, the user program part will only see frequency which is much less than 4.4 MHz (obtained by division, 88 MHz / 20 cycles) assuming I cycle per each instruction. Since each instruction will take four cycles, user instructions without scheduler will see a frequency of 22 MHz. And with scheduler it comes down to 1.1 MHz. This is the case only for the scheduler. This worsens when the kernel starts to have advanced features like semaphores, message queues, etc which heavily rely on data structures implemented in software.

## 2. THE HARDWARE KERNEL
### 2.1 Basic working
Hardware OS has been suggested by proceedings of several conferences. They rely on making most of the data structures used by RTOS available in hardware and switching of register banks. Such a system is developed and the performance is evaluated. But the approach used here is relatively simple and easier to implement.

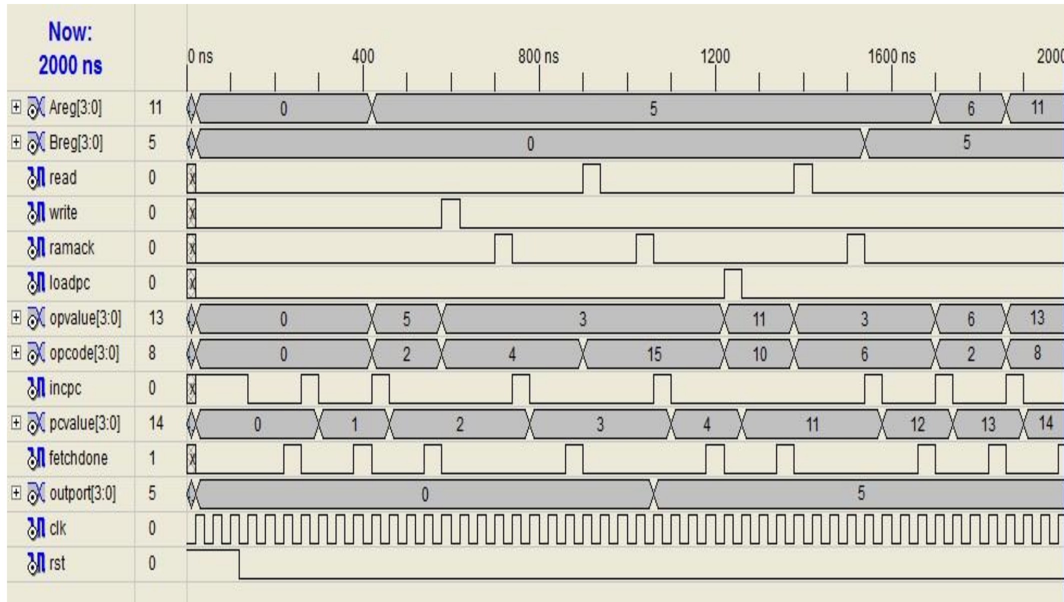


**Fig 1: Operation of the basic processor**

Further the advanced features are explored and scope of their existence in hardware is studied. The hardware RTOS designed here is capable of scheduling 4 tasks. The tasks can be timer driven, Interrupt driven or mix of both. And even the running tasks can trigger the tasks. Each task has a counter to schedule the timer based events. A trigger counter is dedicated for each task which will count the number of times the task is to be triggered or initiated. At this level, task is composed of the basic instruction set which is said earlier and a special system call that is implemented as an instruction which is used to indicate the task switcher that an iteration of the task is completed. This will decrement the trigger counter for that particular task. The other components of the developed hardware RTOS can be seen in the figure 2.

## 2.2 Functional Modules

All the blocks shown in the figure 2 are clocked from the same source without any gating. The system call interface will respond to the system call instructions and suitably updates the flags in the switching logic module. Selective task block responds to the system calls that are intended to suspend or resume some or all tasks (except the one that executes the instruction with some limitations). ROM, RAM, ALU and Control Logic Unit form the basic processor. Other shared resource can be anything like high precision floating point unit, signal or image processing unit, output ports, input ports, communication peripherals like UART, and other hardware data structures like queues, stacks etc intended for communication between tasks. These are connected by expanding the instruction set suitably. Some of these can be even system calls.

The register bank consists of the basic register set like accumulators banked for the use by each task. They are switched depending on the running task. Higher address lines of RAM and ROM are multiplexed and decoded by task switcher so as to assign separate code memory and data memory for each task. The principle can be extended to any other shared resource as well. The interrupt and task trigger controller takes care of interrupts, system calls use to trigger tasks and increments the trigger counter that corresponds to the particular task. There is a dedicated timer and a shadow register for each task. The shadow register consists of the period of the task if that task is periodic. The timer is counted down. Upon timer out, it reloads the value from the shadow register and increments the trigger counter value of that particular task. The switching logic switches the tasks depending on several criteria. The Hardware RTOS provides the functionalities of a basic RTOS kernel in three respects. They are scheduling the tasks, providing communication between tasks, synchronizing between tasks.

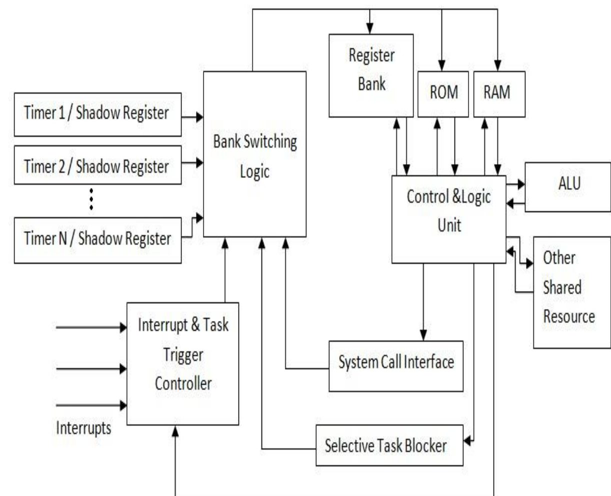


**Fig 2: Functional Modules**

# 3. SCHEDULING OF TASKS

The tasks can be in any of the states. They are blocked, ready and running. There is no difference between ready and running task as their corresponding flag says that the task is ready. Only the task switcher decides which task to run and the tasks priority is hardwired. Tasks can be triggered by timer outs of the timers, interrupts and by the tasks itself. From the figure 3, the runtaskid gives the id of the task that is running which is 0 (which has highest priority), 1, 2, and 3 (which has lowest priority). Siga, sigb, sigc, and sigd are the internal signals used to increment trigger counter. Here, task 2 runs first because it is only available at that time. Then tasks 0 and 1 are triggered via interrupts. Then task 0 completes execution followed by task1. During this time task 2 is

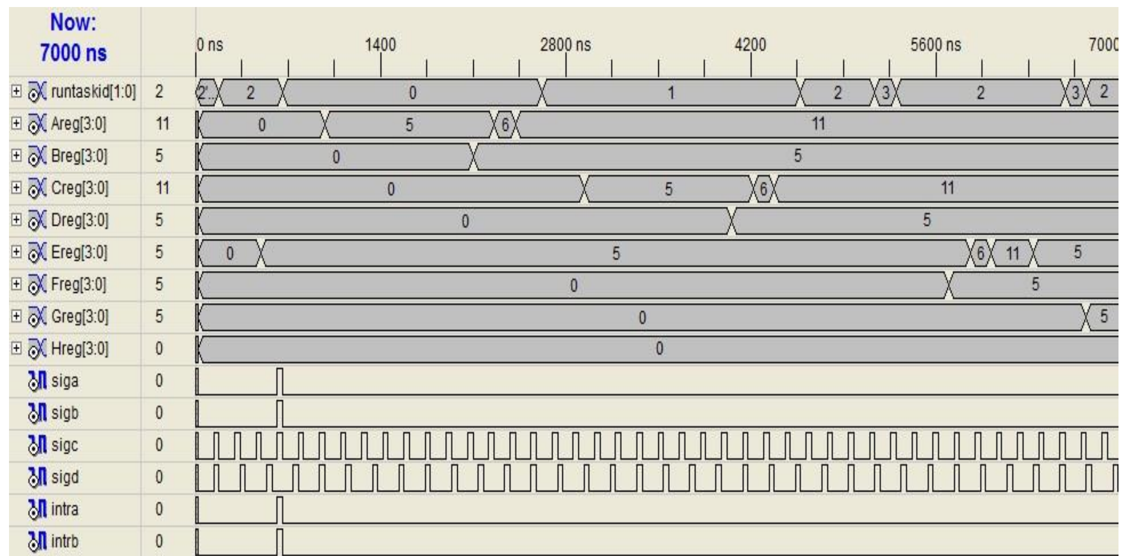


**Fig 3: Tasks competing for the processor**

preempted. It is to be noted that the preemption took only 1 machine cycle compared to many machine cycles in other RTOS cases. All the tasks on their completion will issue end of the task system call which is used to decrement the trigger counter of that particular task. Since tasks 0 and 1 are triggered by interrupts, their trigger counter is 0 after the completion. So they don’t execute again. Tasks 2 and 3 are periodic. So they take the processor again and again.

The theoretical clocking frequency of the basic processor is 88 MHz. When the scheduler is added, because of the overhead of the logic even after the synthesizer (here Xilinx XST is used) optimizing, the speed dropped down to 71 MHz. This can be justified. The decoder in the processor should be able to handle additional instructions. The switching logic should decide which task to run. Since this should happen in a single clock cycle, propagation delay between the components synthesized to handle the logic had caused this drop in frequency. This is far better than software scheduler and context switching where the user tasks see a clocking frequency of only 2.2 MHz. But this is at the cost of increased hardware. But when we want to target such approaches on FPGAs and other programmable logic devices, the modern versions of them are suitable of handling such requirements.

## 4. COMMUNICATION

Communication between the tasks rely heavily on the data structures like queues, stacks, message pipes, shared memory etc. When these are provided by software RTOS, this is made possible by using software data structures. Reading from them, writing to them, checking error conditions, will take a lot of machine cycles. To implement in hardware is simpler. In software, RAM is primarily used as it is by encapsulating it within the operating system services for such data structures. The control of the RAM operations will be taken by OS which in turn consumes the processor. This would further degrade the performance of the RTOS itself.

In the figure 2, the shared resource can be the data structures implemented in hardware. These data structures can be accessed very easily just as reading and writing RAM. And the error conditions are given by special flags. They consume as many cycles as a RAM would consume. The additional logic would definitely add overhead which influences the clocking speed. But the task switching logic had influenced the clocking speed the most. Since all blocks are clocked parallel, the overhead can be accommodated within the time that task switching takes place. Thus showing no additional influence on the clocking frequency.

Here in this paper no hardware data structure that provides communication between tasks is given as it cannot influence the crucial parameter ( here it is clocking speed) and hardware resources (in a sense that hardware structures also uses RAM with very little additional logic just like software structures uses RAM for data and additional RAM for control entries). As many data structures as needed can be created. And they are interfaced to the processor just as addressable locations like RAM. Even many of such things together cannot influence the clocking speed as all of them are clocked parallel and are independent of each other. Thus the advantage of using hardware data structures lies in the fact that they can be used with lesser machine cycles (equal to machine cycles taken by RAM) even though they consume similar resources as consumed by software data structures.

## 5. SYNCHRONIZATION

Synchronization in simple is implemented using two methods. That is by blocking the tasks at a stretch, and triggering the tasks at a stretch. Here at a stretch means it can be from one task to the total number of available tasks within the same cycle by use of some masks. To suspend selected task or tasks, a mask is issued as a system call to the task switching module. The MSB of the mask corresponds to the first or highest priority task. The LSB corresponds to last or lowest priority task. The task that is to be suspended is given 1 in its bit position. Then the mask is given to the switching unit through a system call. The switching unit reads the mask and updates the suspend mask register. If a tasks bit position in the suspend mask register is 1, the task will be blocked until it’s brought again to 0. This is achieved by framing the mask again and sending it to the switching unit through a system call. In the switching module, the values in the suspend mask register are retained until it is changed by another system call. Similarly there is trigger mask register in the switching module. This is also updated using a system call. The MSB corresponds to the highest priority task and the LSB corresponds to lowest priority task. Whenever the mask is

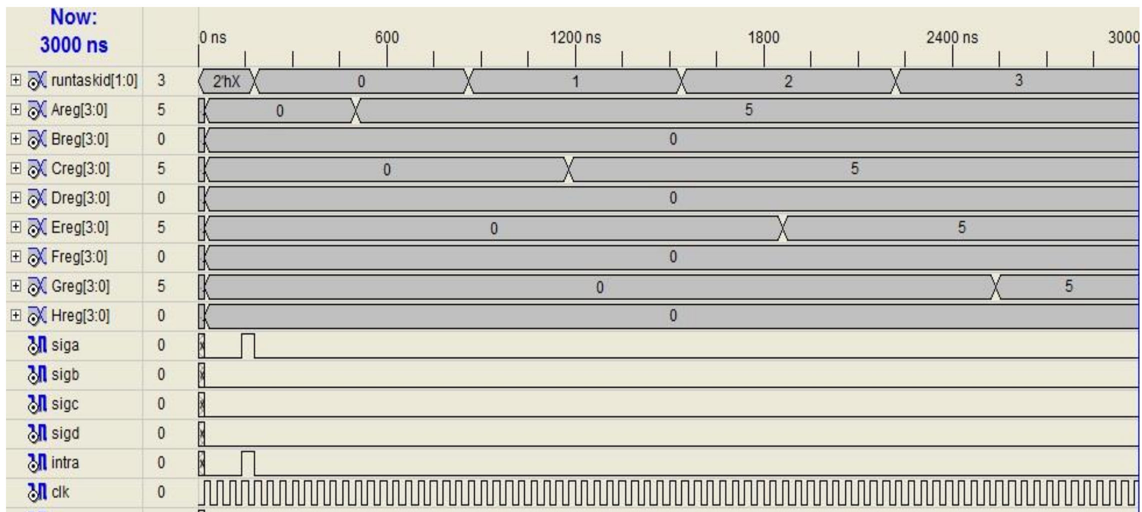


**Fig 4: Tasks getting triggered in series**

updated, the tasks whose corresponding positions are 1 will be triggered by incrementing the trigger register. The values are not retained and brought to 0 as the triggers are already registered in the trigger register of the corresponding task.

In figure 4, the triggering system call implemented can be seen. Here, task 0 triggers task1. Task 1 triggers task 2 which further triggers task 3. The task 0 is triggered by interrupt. This simple flow can be seen in the figure 4. In figure 5, even though all tasks are triggered via interrupts, only task 0 ran. At the end it had issued a block system call imposing suspension on all the other three tasks and then finally issued an end task call. Though the trigger register of task 0 is now 0, the remaining tasks are not run because the suspend register has the corresponding positions 1 and are not brought down to 0. That means not resumed.

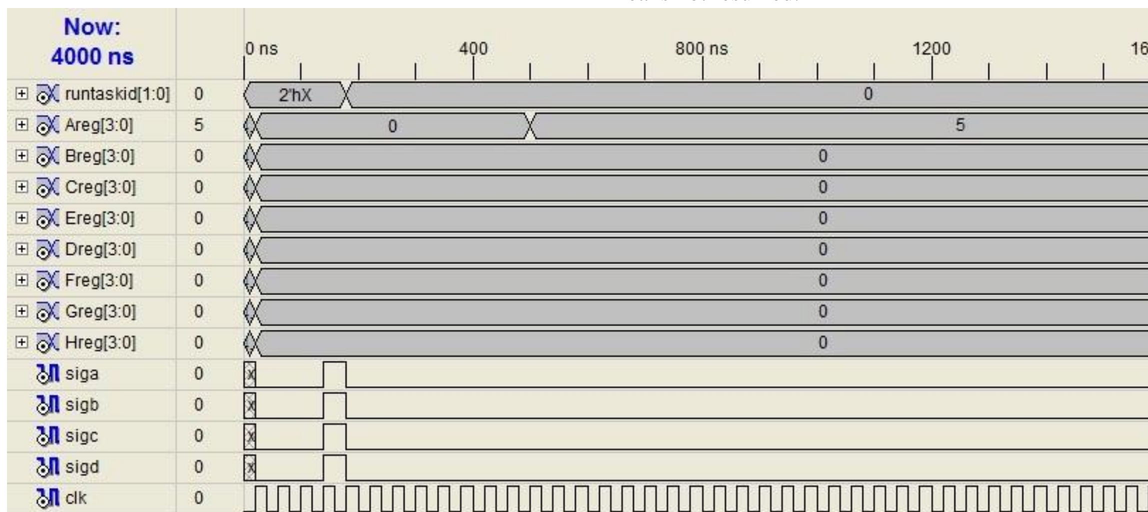


**Fig 5: All the other three tasks are blocked by task 0**

Task triggering along with the scheduler brought down the theoretical operational speed to 67 MHz. Task suspending brought it down further to 66 MHz. But when such things are incorporated in software RTOS the user programs will only see a clock which is less than 2.2 MHz. The total logic of such hardware RTOS consumed only 2 percent of the available resources on the target platform (here it is Xilinx Spartan XC3S4000).

These techniques seem to be simple, but offer a great tool to control the synchronization process. These can be used to achieve the advantages of advanced features like semaphores, mutexes, wait on a task, and suspend a task etc, with little overhead to the user tasks. That means little code is to be added for the user tasks to make use of the above features. This may seem to waste the machine cycles, but relatively better results are achieved using this approach rather than using complete software. This approach will lead us to a new concept, known as hybridization of RTOS.

## 6. HYBRIDIZATION OF RTOS

### a. Need for Hybridization

It is not always possible to scale the above approach to as many tasks as needed. Even though there are only 4 user tasks, Basic kernel has brought down the clocking speed from 88 MHz to 66 MHz. When the number of tasks is increased or RTOS features are increased they would degrade it. But when it is compared with the software counterpart, it is always better. The solution is not in making everything in hardware as at this point, the resources used will start to get increase. For example if we want to create a semaphore manager, the manger has to keep the track of all the running tasks, tasks

that are waiting on it and then set the flags of the task to ready on the release of semaphore. There may be number of semaphores used in a system. Each should be properly registered in the semaphore manager. Even the logic used to implement starts getting complex. Either the clocking speed should be reduced, or the entire manager should be isolated and should be brought into use whenever required by suspending the other operations. But a better result can be achieved using another approach. That is to divide the available RTOS in terms of hardware and software. The semaphore manager has become very complex as said above. It gets further complicated when mutex has to be implemented.

## b. Mutex in hybrid RTOS

Now as a new approach, the mutex management system can be partitioned as follows. One part is the one which suspends and resumes the tasks. Another one will check the semaphore variables. The first one will be in hardware. In the presented hardware RTOS system, this is possible by issuing a system call. The second one is implemented in software. The semaphore variables are checked by conditions that are programmed in the RAM. A task can be specially created to encapsulate the semaphore and can be triggered whenever the semaphore is needed. This will check the semaphore variable and suspends or resumes the task. Or even this can be done by user task itself. Here the software part relies on the single cycle instruction in hardware (In case of software RTOS, such system instruction will not exist and suspending a task requires updating the task control block, switching the context etc). And the hardware part relies on the conditional checks performed by software part (Actually, comparing as a software instruction is easy to implement in the existing processor, rather than to duplicate the hardware to check conditions in special semaphore hardware and the delay in checking conditions in both cases is same and unavoidable). The effect is that the hardware is consumed little. The efficiency is not compromised (In fact, it improved). This can be demonstrated in the system presented in this paper.

To implement mutex, when a task wants to acquire a mutex it should check a RAM entry which acts like a mutex. Then if it is available, it will update it as locked and suspends all the tasks that are likely to use the same mutex. After using the shared resource, the mutex is unlocked and suspension mask is updated again to resume tasks. Here resume means to make them ready. This straight away solves the priority inversion problem. If the RAM entry is already locked, it suspends itself by framing a suspension mask and updating the suspension mask register. Before that, it should use a flag to indicate itself to continue from the point it has suspended itself. Here after resuming, triggering the task triggering the task should be done. On triggering, the resumed task will check the status of its flag and then jumps to the semaphore instruction or runs from start.

The same protocol should be followed by all tasks that use the mutex. So when one task suspends itself because of locked mutex, other tasks that are using the semaphore will trigger it. The task on resume and being triggered will know from where it should start again. That means the tasks will have break points (which are actually conditional jumps and the conditions are set as flags by the same or other tasks, in the RAM entries). This approach has drastically reduced the need of special semaphore manager in hardware.

Instead of break points, the program counter value can be changed before triggering. And instead of suspension, an end of the task can be issued by the running task. On resume the task will start running from where it has to start. This would require a special instruction that would change the value of program counter. In this way the semaphore manager is partitioned into hardware and software. This is efficient compared to complete software semaphore management (Many machine cycles are expended) or complete hardware semaphore management (consumes heavily resources, basically replicating the same that are in processor).

## c. Wait on task in hybrid RTOS

The task that has to wait on other task will communicate with the other task to tell it that it is waiting on it. Then it will trigger it. Then waiting task will set a flag to know its break point. The other task after performing its function will try to identify which task is waiting on it based on the communication received from it. Then it will trigger that task. In this way, the wait on task is implemented.

## d. Comments on hybrid RTOS

The approach is similar to hardware software co-design. This is a recent emerging trend and the standardizations are not completed yet. This paper confines the development of system to RTOS. Embedded designers make use of software RTOS libraries to program their user programs and finally a ROM image of the total system is obtained. This is dumped to the specific processor or controller. Here neither the processor is developed keeping RTOS in mind, nor is the RTOS developed keeping processor in mind. Thus a single processor can run many operating systems and a single operating system can be ported on several processors. So there is a chance that the systems developed using such hardware and software are always sub optimal. So the hybrid RTOS would be the ideal and optimal solution for real time applications.

# 7. RESULTS

This system tried to partition RTOS into hardware and software manually. For limited number of tasks, mutexes, and breakpoints, it worked well. But to scale it to a larger size, such manual partitioning will not help. Even the components in the RTOS in hardware and software cannot be as straight as they are now. They get more complex. So a programming environment is needed which readily forms the hardware and software components on programming like a normal software RTOS. Some system calls may be implemented in hardware and some in software. Some user code may be implemented in software and some in hardware, like for example user data structures can be in hardware. So this has opened a new area of research in the domain of hardware software co-design. Here the optimization is left to the programming environment itself.

# 8. FUTURE WORK

FPGAs and other programmable logic arrays are highly configurable. A single programming environment will be developed. In this environment, the programmer programs using RTOS of his choice. The only thing is that he will be unaware of the target processor. Then the environment will analyze it and decides which components will go to hardware and which will go to software. In such approach the operating system is broken down into hardware parts and software parts. The user will not know which parts are of hardware, which are of software. Even some system calls are implemented in software and some in hardware which are highly dependent on the user program. Then finally, as a part of hardware, a HDL code is generated and as a part of software ROM image

is created. HDL (can be Verilog, VHDL etc) is used to synthesize the hardware on FPGA or other programmable logic devices. ROM image is put in ROM and both of them are interfaced. The advantage is that the hardware will become application specific (here, it is mostly RTOS) and the software will become hardware specific. So the result is optimal.

# REFERENCES

[1]. P. Kuacharoen, M. A. Shalan and V. J. Mooney III, “A Configurable Hardware Scheduler for Real-Time Systems,” in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, June 2003.

[2]. A.Parisoto, A. Souza, L. Carro, M. Pontremoli, C. Pereira, and A.Suzim, “F-Timer: dedicated FPGA to real-time systems design support.In Real-Time Systems”, In Proceedings of the Ninth Euromicro Workshop, June, 1997.

[3]. T. Samuelsson, M. Akerholm, P. Nygren, J. Starner and L. Lindh, “A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software.” in proceedings of International Workshop on Advanced Real-Time Operating System Services, Porto,Portugal, 2003.

[4]. Vetromille M., Ost L., Marcon C. A. M., Reif C. and Hessel F. “RTOS Scheduler Implementation in Hardware and Software for Real Time Applications,” Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006.

[5]. T.Nakan, M.Itabash A.Shiomi and M.Imai "Hardware Implementation of a Real-time Operating System"in Proceedings of the Twelwth TRON Project International Symposium, IEEE Computer Society Press,Nov 1995.

[6]. P. Kohout, B. Ganesh, and B.Jacob, "Hardware support for real-time operating systems," in Proceedings of the first IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, Newport Beach, CA, USA, 2003.

[7]. G. Bloom, G. Parmer, B. Narahari, and R. Simha, “Real-Time Scheduling with Hardware Data Structures”, IEEE Real-Time Systems Symposium, 2010, December 2010.

[8]. H. Jamal, and Z. A. Khan, "Hardware IP for Scheduling of Periodic Tasks in Multiprocessor Systems", in WSEAS Transactions on Computer Research, Issue 3, Volume 3, March 2008.

[9]. S. Chandra, F. Regazzoni, and M. Lajolo, “Hardware/Software partitioning of operating systems: a behavioral synthesis approach”, in Proceedings of ACM GLSVLSI, 2006.

[10]. N. Gupta, S.K. Mandal, J. Malave, A. Mandal, R.N. Mahapatra, "A Hardware Scheduler for Real Time Multiprocessor System on Chip", in 23rd International Conference on VLSI Design, 2010.

[11]. J. Hildebrandt, F. Golatowski, and D. Timmermann, “Scheduling coprocessor for enhanced Least-Laxity-First scheduling in hard Real-Time systems,” in Real-Time Systems, in proceedings of Euromicro Conference, Los Alamitos, CA, USA ,1999.

[12]. S. Saez, J. Vila, A. Crespo, and A. Garcia, “A hardware scheduler for complex real-time systems,” in Proceedings of the IEEE International Symposium on Industrial Electronics, 1999.