Preemptive Multitasking on Atmel® AVR® Microcontroller

HABIBUR RAHMAN, Senthil Arumugam Muthukumaraswamy
School of Engineering & Physical Sciences
Heriot Watt University Dubai Campus
Dubai International Academic City
UNITED ARAB EMIRATES
hr69@hw.ac.uk; m.senthilarumugam@hw.ac.uk

Abstract: - This paper demonstrates the need for multitasking and scenario where multitasking is the only solution and how it can be achieved on an 8-bit AVR® microcontroller. This project explains how to create a simple kernel in a single C file, and execute any number of tasks in a multithreaded fashion. It first explains how the AVR® engine works and how it switches between different tasks using preemptive scheduling algorithm with the flexibility of blocking a task to allowing it more execution time based on their priority level. The code written for this project is basically in C, however the kernel code is mostly assembly functions called by C. The development environment is Atmel Studio®. The code is in such a way that it can be ported over any 8-bit AVR® microcontroller, however, this project demonstrates the results in both simulation and hardware chip on device Atmega8A.

Key-Words: - AVR®, Microcontroller, Threading, Preemptive Scheduling, Multitasking.

1 Introduction

Microcontroller development has been exponential over the past two decades. Development in terms of speed, transistor density, multi-core embedding as well as number of peripheral subsystem on SoC(System on Chip) is common, and development tools and compilers have been created that allows writing code for microcontroller on higher level of abstraction layer an easy task. The problem here laid out is that programmers working on low level language such as assembly have excellent understanding of the processor architecture but is limited to smart utilization of it capacities if they lack knowledge on high level language such as C/C++. On the other hand, programmers programming on high level language have to rely on toolchain libraries, unless they have understanding of the architecture and programming on low level assembly.

When it comes to writing different functions which can be multitasked, an understanding of the architecture is very important since the kernel, and scheduling algorithm can be different for different microcontroller engines. The easy solution to this problem is to look for commercial or open source RTOS (Real Time Operating System) , but again, the this will require research and understanding of the operating system, not the microcontroller architecture and will eventually lead to downloading and compiling hundreds of files according to the provided operating system instructions and also spend time learning how to use it.

But if the user who has basic understanding of C language and very good understanding on microcontroller wants to implement multitasking for the written functions, this is possible by writing a simple scheduling algorithm which can switch back and forth different functions, (here on threads) .

This paper will first describe the two types of task execution, single thread and multithread and also demonstrate scenario where multi-thread is the only solution. It will then explain in general how the processor switches between the tasks, and what is the hardware and time cost of using this scheduling method and also explain the scheduling model used. Then it will focus on the AVR® architecture, and how it can schedule task switching.

2 Problem Formulation

What is single threaded function?
Functions written in a single infinite loop executed in a linear fashion.

ISBN: 978-1-61804-276-7

```
int main(void)
{
        while(1)
        {
                function_1();
                function_2();
                function_3();
        }
}
```

In the above code, function_1() is executed followed by function 2, then function 3 after when the infinity loop ends, it goes back to function_1(). Here the functions are executed in a linear fashion.

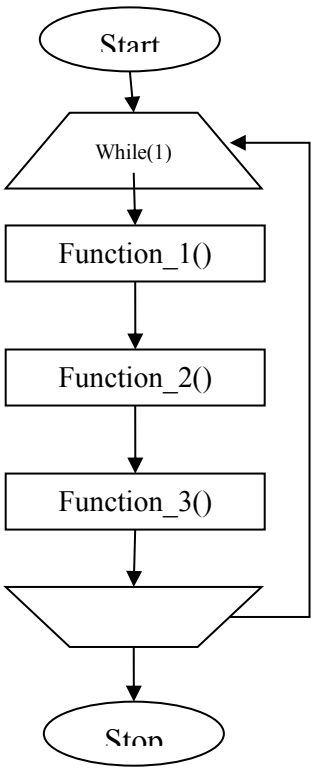

Figure 1 : flowchart showing how three functions are executed in an infinite loop in a single thread

If in the above example code of single thread, in any function consists of a delay, then the delay will be sum of delays in all the other functions as well.

Taking an example of a sample program consisting of 3 LEDs namely Red, Green and Blue. Function_1 toggles red led every 3 second, function_2, toggles green led every 5 second and function_3 toggles blue led every 1 second.

The execution in single thread, hence on function can be executed at a time, hence the sequence of sub functions will be as table below. (Note that execution of functions will also have certain delay, but so small so its negligible in this example).

Table 1 : table showing the different statements and delays in the functions executed in a single thread

		RED	GREEN	BLUE
	red_led_on();	**ON**		
	delay_sec(3);	3	3	3
	red_led_off();	**OFF**		
function_1();	**delay_sec(3);**	3	3	3
	green_led_on();		**ON**	
	delay_sec(5);	5	5	5
	green_led_off();		**OFF**	
function_2();	**delay_sec(5);**	5	5	5
	blue_led_on();			**ON**
	delay_sec(1);	1	1	1
	blue_led_off();			**OFF**
function_3();	**delay_sec(1);**	1	1	1

As a result, the blink in the LEDs will be as shown in Table 2.

Table 2 : The pattern in which the LEDs will blink if the code is written in a single thread.

		Delay(sec)		Delay
RED	ON	3	OFF	15
GREEN	ON	5	OFF	13
BLUE	ON	1	OFF	17

It can be observed that red led is intended to toggle every 3 seconds; however, it will toggle from on state to off state after 3 seconds, but from off state to on state every 15 seconds. (*3+5+5+2+2 = 15, all delays after the **red_led_off()** function and until the infinite loop ends*) Same goes for the green and blue led. This is the underlying problem of single linear execution.

As it can be observed that the number of functions inside an infinite loop will affect the time after which the function will get its execution. Worse case is when another function is stuck inside a loop waiting for a resource to be available. For example, what if function_2 was written to wait infinitely until a data becomes available through a serial mode of communication. In that case, function_2 would keep waiting, increasing delays for toggling the LEDs.

ISBN: 978-1-61804-276-7

This kind of problem cannot be eradicated even with the use of fast processor because here the delay is event related, such as time, or wait until event, etc.

Each column represents execution cycle and the blue cell represents the function being executed. After each execution cycle, the processor is given up and the function to be executed changes. The Execution time is determined by hardware timer interrupt as shall be explained later in the Scheduler Design Section.

3 Problem Solution

Each function should work independently therefore they are placed in a separate while(1) loop:

The solution to this problem is to create an algorithm by which each function is allocated certain time to execute after which, the processor switches to the next function. This method is called multi-threading, where functions execution time is sliced.

The following diagram will describe how time is sliced and certain execution time is allocated for the functions.

	1	2	3	4	5	6	7	8	9	10
task_1()	■			■			■			■
task_2()		■			■			■		
task_3()			■			■			■	

Figure 2: Time slicing of different tasks due to the termination of execution cycle

It can be seen that in cycle 1, **task_1()** was being executed, however, in cycle 2, **task_2()** began executing leaving the t**ask_1()** incomplete. **task_1()** will get its turn to continue later in cycle 4, and continue exactly where it had left. But after at the end of cycle 1, before switching to cycle 2, there has to be some sort of memory to remember where and in which state the **task_1()** was terminated and left incomplete and recall this information in cycle 4 to return exactly how **task_1()** was left abandoned.

3.1 How Does the Processor remember the point of return?

The AVR® CPU machine has 4 information to back up, to remember the point of return of each function. This information is called the 'context' of the task and it includes.

PC (Program Counter)
Location where the task is abandoned so it can return later.

SREG (Status Register)
the state of the ALU result , such as carry flag, zero flag, etc.,

GPR(General Purpose Registers) (R0 to R31)
The AVR® uses 32 general puspose registers for it arithmetic operations. These needs to be backed up.

Stack Pointer Backup (for each task at the kernel)

This stores the address of where the stack should point in the space for each task. This usually is the address of the stack pointer right after storing the context. So when the context is to be restored, reloading this stack pointer will be followed by restoring the GPR, SREG and PC.

The PC, SREG and GPR for each task are backed up in dedicated mapped data space for corresponding task and the Stack Pointer for each task is backed at the kernel space.

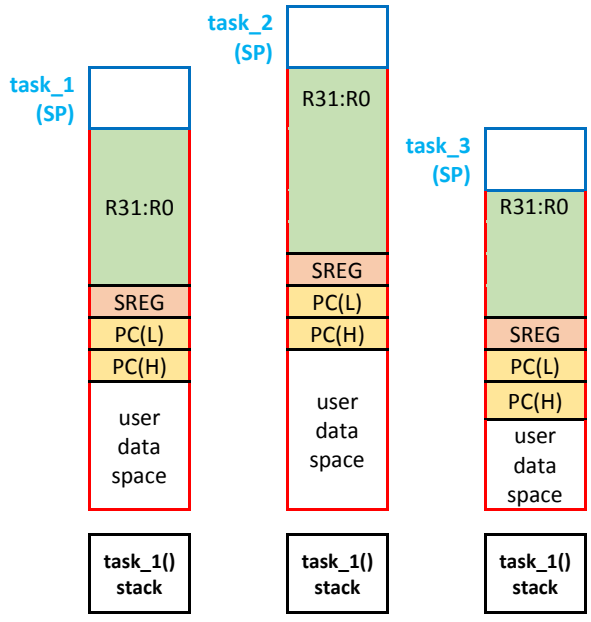

Figure 3 : the stack structure of each task in the data space

ISBN: 978-1-61804-276-7

The user data space should be pre-defined, as how much data in the stack the task will need. This needs user's estimation or calculation. Assigning extra space will render the space useless, and assigning less space will result in data of one task to overlap and corrupt that of another. The stack size for each task is defined as,

user defined data space per task + 32 (for GPR) + 1 register for SREG +2 register for PC.

Summing up to
= (user defined data space per task + 35)

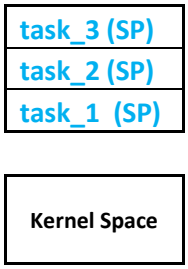

Figure 4 : the data in the kernel space which stores the most recent stack pointer address for each task.

3.2 How does the task Switching Work?

The scheduling algorithm used in this project is preemptive scheduling. In this scheduling method, execution cycle is determined by the use of an interrupt. The interrupt is timer based and is a hardware feature for the AVR® microcontroller.

The timer is an 8-bit counter, which increments every CPU clock frequency. Since the frequency used in this project is 1Mhz, the increment in the timer occurs every 1/1Mhz = 1us.

The timer increments from 0 to 255, and then returns back to zero but before that, it executes and interrupt service routine, due to its overflow. This interrupt is the Overflow Interrupt and will occur every 265 * 1us = 256 us.

This is the execution time that shall be allocated to the tasks. Hence Each task will be given 256us time to execute before its control is take over and given to the next task in queue.

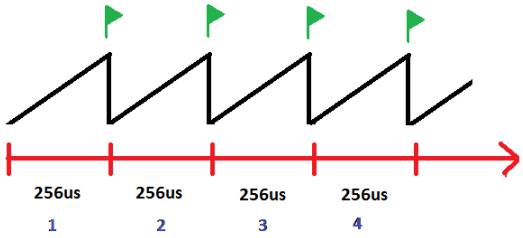

Figure-5: Timer causing overflow every 256 CPU cycle. The green flag shows the overflow flag triggering interrupt.

3.3 The Scheduling Process

Whenever the overflow triggers the interrupt, there is when the scheduling and task switching occurs. Hence we can safely say that the scheduling occurs every 256us at the overflow interrupt.

The scheduling has 6 basic steps:

1. Save the context of current task
2. Back up the stack pointer of current task in the kernel
3. Decrement corresponding task timer variable
4. Change to next task
5. Load the stack pointer of the next task from kernel
6. Restore the context of the next task

3.4 How Does the Stack Pointer Work?

Stack Pointer for Register Data
The stack pointer works exactly like a stack following Last in First Out (LIFO). Data can be pushed into the stack and popped out from the stack using the instructions **PUSH** and **POP** respectively. Every time the data byte is pushed in, the address of the stack pointer decrements by 1, and when data is popped out, stacks pointer increments by 1.

Stack Pointer for Program Counters Data Storage
When a function/subroutine is called using the **RCALL** instruction or jumping to ISR(interrupt service routine) , the stack pointer pushes the current PC address (Program Counter) automatically which stacks 2 bytes into the stack i.e. PC (High) and PC(Low), hence the stack pointer decrements by two.

The return from a function/subroutine using **RET** instruction, or return from interrupt using the **RETI**

ISBN: 978-1-61804-276-7

instruction will jump to the loaded PC, and increment the Stack Pointer address by 2.

The example below shows the execution of the following code and the effect on the stack and change in address of the stack pointer.

The Red Block shows how the stack pointer increments and decrements based on the instructions

PUSH R16;
RCALL label;
RET;
POP R16;

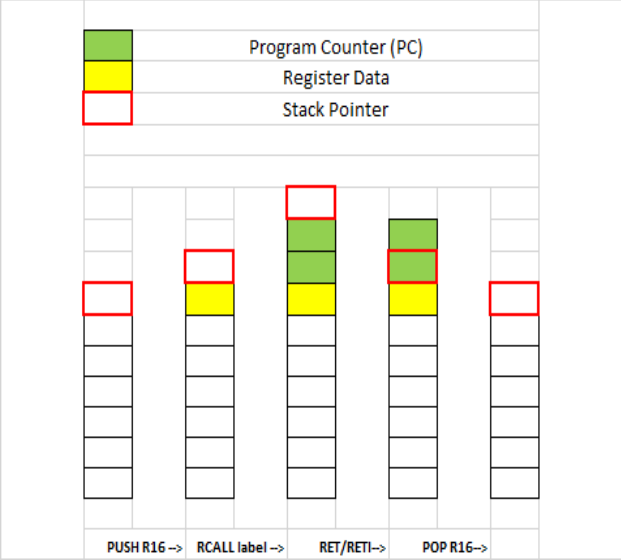

Figure 6 : The stack pointer address incrementing on and decrementing on different instructions and how data and program counter is pushed and returned

Note that the stack address starts from 0 at the top and increments on the way down. So when the stack pointer moves up means it is decremented and vice versa.

As can be observed, PUSH instruction pushed the R16 register (yellow) value onto the stack and decrements the stack pointer by one. On the use of RCALL instruction (same happens when interrupt being called), the Program Counter (green) is pushed onto the stack the stack pointer decrements by 2. The RET/RETI instruction will increment the counter and load the two bytes of Program Counter (green) and hence jump back to where it had left before the RCALL instruction, and finally restore the R16 register using the POP instruction which pops the yellow byte from the stack into R16 and increments the counter by one .

4. Task Switching

Since each task has its own data space to back up the context, and the SP pointing to the the top of the context ready, it's only a matter of loading the backed up stack address of the corresponding task, into the stack pointer followed by the restoring the context which will restore all the GPR, SREG and finally jump to the PC of that task where it had left off. Since the kernel backs up the stack address of each task as an array, it can be loaded easily.

If first task's sp backup at the kernel is stored at address x, then the nth task's sp backup will be at x+2*n.

Here the n is multiplied by 2 since the stack pointer uses 2 registers in the stack.

5. The Tick Delay

The scheduler interrupts the delay function and switching to different task. Since each task has its own infinite loop, if they want to implement a wait function, it can no longer be associated with the main CPU clock. For example, if a delay for 1 second is used in a task, then when the execution cycle is over and the CPU switches to another task, even the delay function of that task is halted and continued when its turn in the queue comes back again hence the delay will not be 1 and accurate one second.

This solution to this is to create a timer variable for each task at the kernel level. This timer is decremented every time the overflow interrupt triggers the scheduler.

Since the execution time is knowns to use 256us, the timer variable of each task will decrement the timer variable until it reaches down to zero. Then at the task the delay function can safely rely on its corresponding timer variable.

Example, if a delay of 1 second is required, 1 sec = 1000000us, which means 1000000/256 = 3906.25 execution cycles. Therefore the timer variable should be set to 3906 which will start decrementing every execution cycle, regardless of which task is being executed. Whenever the task gets its turn in the execution cycle, it will check to see if its timer variable has reached zero, so it can continue to the next code statement or else execute not instruction.

ISBN: 978-1-61804-276-7

5.1 Priority levelling of the tasks

The scheduler dedicates one execution cycle per task. However, biasing the number of execution cycle before switching can dedicate more time to one task than another. Hence the priority level of each task can be modified based on a variable that loads the priority level value of the current task and starts decrementing until it reaches zero. Once it reaches zero, only then it will switch task, otherwise load the same task.

5.2 Code Fragment for the Scheduler

```
ISR(TIMER0_OVF_vect,__attribute__((naked)))
{
	backup_context();
	current_task_sp_backup();

	//task_timer_variable_decrementer
	for(int i=0;i<total_tasks;i++)
	{
		task_tick_delay_timer[i]--;
	}

	task_switch();

	next_task_sp_restore();
	restore_context();

	asm volatile ( "reti" );
}
```

5.3. Task Switch Timing Cost

Switching between tasks will call the scheduler every time and execute the scheduler code to backup context, change task, and restore task. But scheduling itself will consume a lot of CPU cycles. As tested in the project by using the Atmel Studio ® simulation tool, by inserting breakpoint at the first statement i.e. `backup_context();`and another at the last statement, i.e. `asm volatile ("reti");` the number of CPU cycle can be simulated. Once the simulation reaches the first breakpoint, the cycle counter can be resettled.

Cycle Counter	220
Frequency	1.000 MHz
Stop Watch	220.00 µs

Therefore currently 220 CPU cycles are used for the scheduling, which on a 1 MHz Clock frequency will consume 220us. This is the time cost of preemptive multithreading. However faster CPU clock will result in faster task switch time.

6. Writing the Code Preemptive Multi-tasking

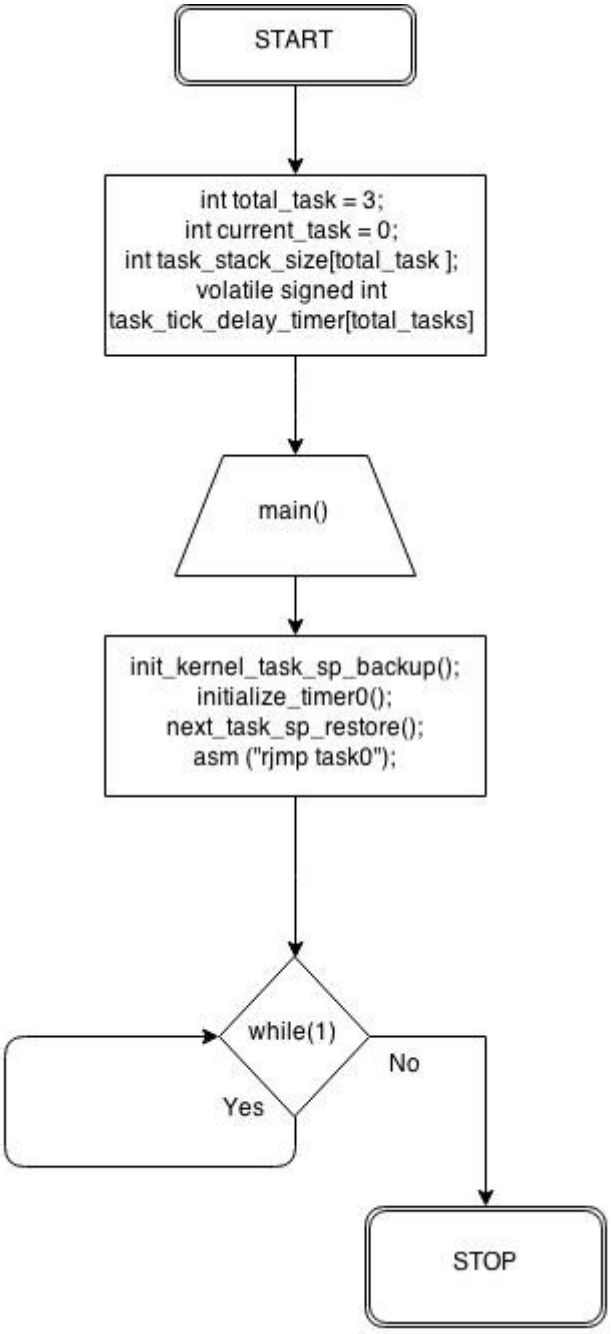

Figure 7: the flowchart of the main Program initializing the variables, timer, and stack pointer of each task.

init_kernel_task_sp_backup();

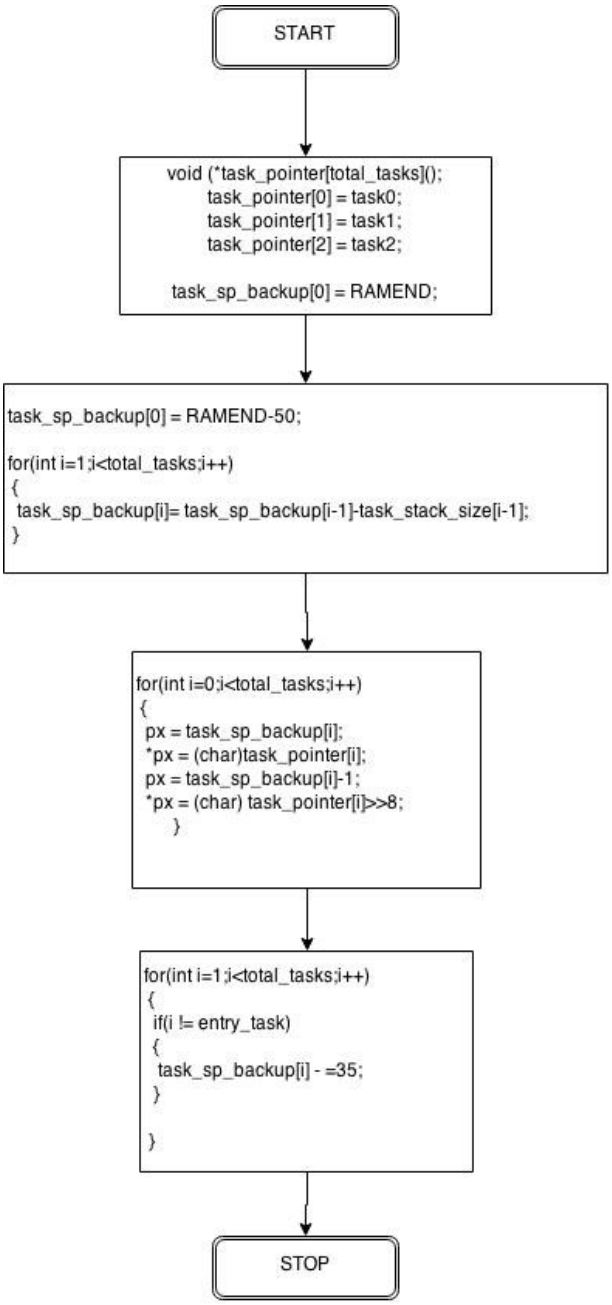

Figure 8 : this flowchart shows how the stack pointer is initialized and backed up in the kernel for the first call of the tasks. Notice tasks2 and task3 has stack pointer shifted up by 35. This is because when the task is first called after the switch from task 1 to task 2, it will restore context, but it was never backed up before.

Task_delay_cycles(int cycles);

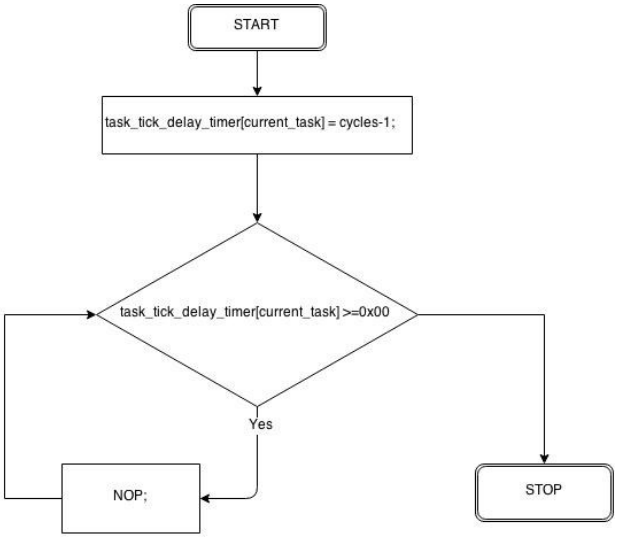

Figure 9 : The flow chart for the tick delay. The delay timer for each task follows its own specific corresponding timer variable.

The Tasks:

```
void task0(void)
{
	DDRB|= (1<<PINB0);
	while(1)
	{
		PORTB^=(1<<PINB0);
		task_delay_cycles(3906);
	}
}

void task1(void)
{
	DDRB|= (1<<PINB1);
	while(1)
	{
		PORTB^=(1<<PINB1);
		task_delay_cycles(1953);
	}
}

void task2(void)
{
	DDRB|= (1<<PINB2);
	while(1)
	{
		PORTB^=(1<<PINB2);
		task_delay_cycles(977);
	}
}
```

ISBN: 978-1-61804-276-7

For this example, 3 tasks are created and the purpose for task0, task1 and task2 is to toggle PINB0, PINB1 and PINB2 respectively at intervals of 1 second, 0.5 seconds, and 0.25 seconds respectively.

The task delay cycle for 1 sec = 3906.
(1000000us/256us = 3906.)

The task delay cycle for 0.5 sec = 1953.
500000us/256us = 1953

The task delay cycle for 025 sec = 977.
250000us/256us = 977

7. Results on Proteus ISIS simulation

The simulation on the cad software includes device atmega8 where its PINB0 toggles every 1 second, PINB1 toggles every 0.5 seconds and PINB2 toggles every 0.25 seconds. All the three pins are connected to a simulated oscilloscope with grid size of 0.2 seconds. The microcontroller is running at 1 MHz as decided for this testing purpose.

The simulation CAD connection can be seen as Figure 10. The Oscilloscope results are shown in figure 11.

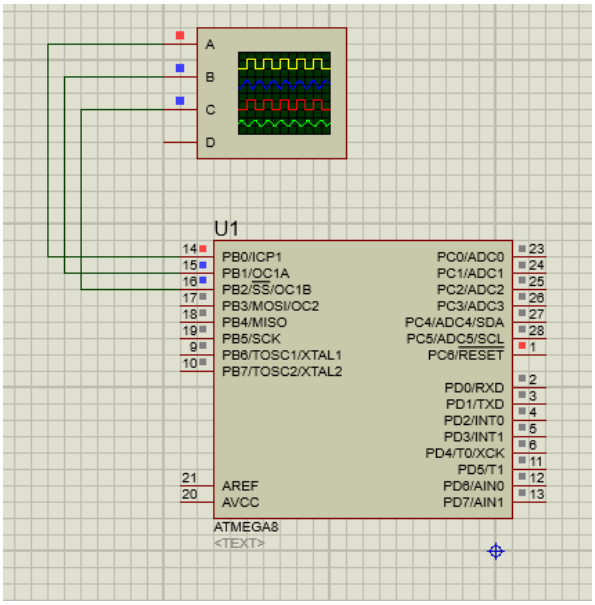

Figure 10 : Oscilloscope connection of the corresponding pins on the AVR microcontroller atmega8.

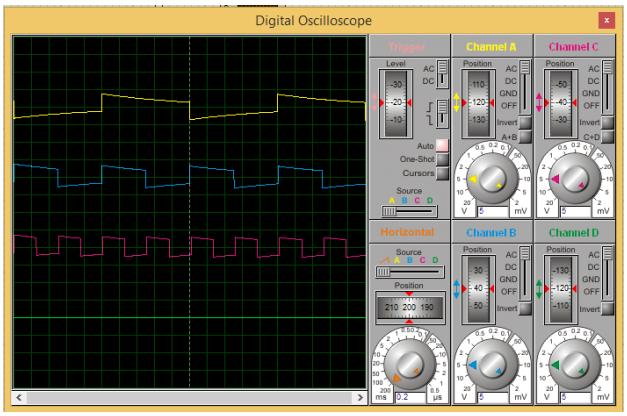

Figure 11 : The oscilloscope result as seen when the simulation is run. The Yellow signal represents PINB0, Blue represents PINB1 and red represents PINB2.

Yellow graph for PINB0 which toggles every 5 grids = 5* 0.2 = 1 seconds, just as expected.

Blue graph for PINB1 which toggles every 2.5 grids = 2.5* 0.2 = 0.5 seconds.

Red graph for PINB2 which toggles every 1.25 grids = 1.25* 0.2 = 0.25 seconds.

8 Conclusions

As seen in the results, the paper demonstrates how to create a simple kernel in a single C file, and execute any number of tasks in a multithreaded fashion. The Scheduler in the ISR switches tasks effectively with 220 clock cycles and each task having its own timer variable can implement real time delays which follows the timer clock regardless of which task is running. The results are exactly as expected as seen on the simulation.

References:

[1] Atmel® Corporation, ATmega8A datasheet Rev.: 8159E–AVR–02/2013.
[2] Atmel® Corporation, *AVR Instruction Set*, Rev: 0856I–AVR–07/10.
[3] Han-Way Huang, The Atmel AVR Microcontroller Mega and XMega in Assembly and C
[4] Elliot Williams, Make: AVR Programming, First Release

ISBN: 978-1-61804-276-7

[5] Avr-libc user manual [online] : http://www.nongnu.org/avr-libc/user-manual/index.html .
[6] FreeRTOS® website [online]: http://www.freertos.org/implementation/a00009.html .
[7] AVRFreaks® website [online] http://www.avrfreaks.net/
[8] CSC560: Design and Analysis of Real Time Systems, Nithin Goyal, Rayhan Rahman website [online] http://web.uvic.ca/~rayhan/csc560/projects/project3/context_switch.html
[9] Wikipedia : Thread (Computing) [online] http://en.wikipedia.org/wiki/Thread_%28computing%29 .
[10] Wikipedia : Preemption (computing) [online] http://en.wikipedia.org/wiki/Preemption_%28computing%29
[11] Proteus® ISIS® software : http://www.labcenter.com .