A New Real-time Kernel development on an embedded platform

CSC714: Real Time Systems Project – Final Report
Spring 2009

BALASUBRAMANYA BHAT
(bbhat@ncsu.edu)
SANDEEP BUDANUR RAMANNA
(sbudanu@ncsu.edu)

Table of contents

1. Overview

Real Time Operating Systems use specialized scheduling algorithms to provide predictable behavior in hard real-time systems. The scheduler has to guarantee that all periodic tasks meet their respective deadlines. Therefore the scheduler forms the core of any real-time kernel. As part of this project, we intend to implement a real time kernel which supports pre-emptive Earliest Deadline First(EDF) scheduling for periodic tasks and a simple pre-emptive static priority scheduling for aperiodic tasks. The real-time kernel also supports synchronization primitives like semaphores and mutex.

2. Introduction

Periodic tasks release their jobs at regular intervals each of which needs to be completed before their respective deadline. A hard real-time system has to ensure that all deadlines are met for all the jobs. In a soft real-time system, the deadlines can be missed occasionally. The deadlines can be less than, equal to or greater than the task period. The EDF scheduler always executes the job with the next earliest deadline at every scheduling instant. Aperiodic tasks can be scheduled when no periodic tasks are running that is aperiodic tasks are run in the available slack time.

The example below shows the operation of EDF scheduling algorithm on tasks T_1 (0, 4, 1, 3), T_2 (0, 8, 1, 5), T_3 (0, 10, 2, 6) and T_4 (0, 15, 4, 9) are periodic tasks with periods $P_1<P_2<P_3<P_4$.

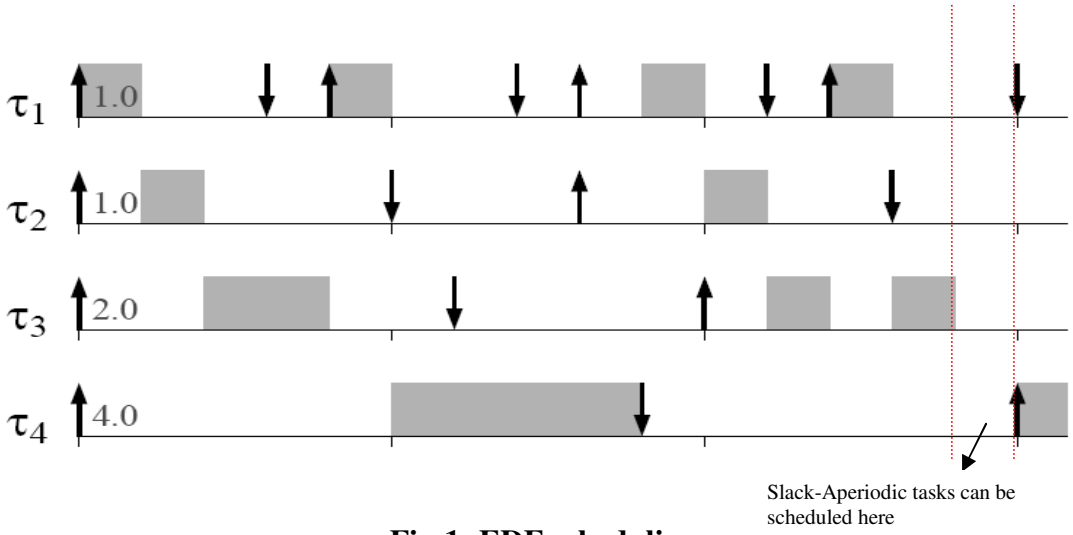


Fig-1: EDF scheduling

Non-preemptive EDF is not optimal and hence pre-emptive EDF was considered as the scheduling algorithm in the real-time kernel.

3. Features Supported

- EDF based scheduling for periodic tasks.

- Periodic tasks with deadlines equal to or less than the period.

- The scheduler shall also support aperiodic tasks.

- If two or more jobs have same deadline, then they are scheduled in the FIFO manner.

- The scheduler is capable of creating tasks based on phase, period, execution time and relative deadlines.

- Schedulability test for each periodic task when the task is created. The task is successfully created only if the schedulability test passes.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

- Methods for periodic and aperiodic tasks to sleep for given amount of time.

- Methods related to task synchronization like creation of mutex/semaphores.

- An Idle task which is executed when there are no tasks in ready queue.

- Tracking the current CPU utilization.

- All time values have one micro second resolution.

- The execution time for each job is strictly monitored and shall not be allowed to exceed.

- The aperiodic tasks have lesser priority than all periodic tasks.

- The aperiodic tasks are scheduled using static priority based preemptive

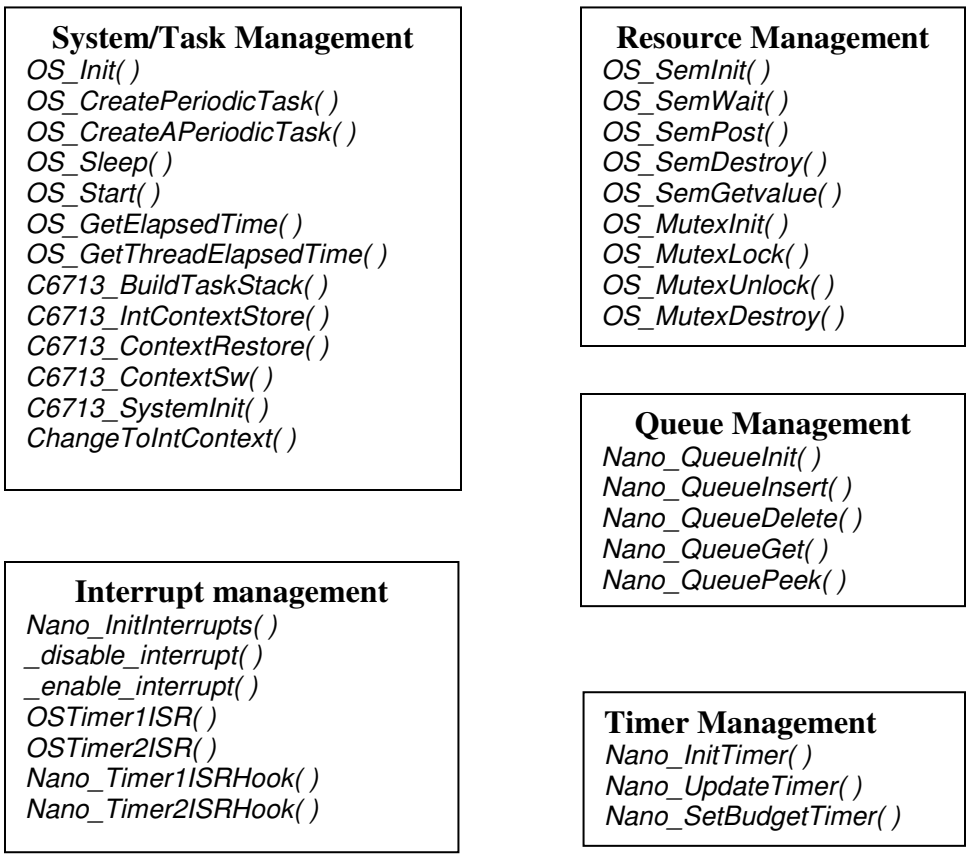
scheduling during slack time .

- The scheduler is easily portable to different platforms.

- The scheduler is quite efficient as context switching routines and ISRs are written in assembly code.

- The scheduler has a small memory footprint.

4. Modules



The real-time kernel contains several modules as listed above and the corresponding methods supported by each module are shown within the respective block.

5. Design & Implementation

The real-time kernel is written partly in C and partly in Assembly. Various parts of the Kernel are described below:

5.1. Coding Standards

We attempted to standardize coding standards before we started coding. See 'CodingStandards.txt' for the coding standard specification.

5.2. Interrupt Vectors

The interrupt vectors are setup using C6000 Assembly. The file C6713_Vectors.asm has all the interrupt vectors in it. Mainly we are interested in reset interrupt, Timer 0 Interrupt and Timer 1 Interrupts. All remaining interrupts are made to end up in a infinite loop. The reset vector has calls to Initialization routines described below. The Timer 0/1 vectors make calls to appropriate handler in C files (Nano_Timer.c)

5.3. Initialization routines

When the system starts, certain initializations need to be made before the control can be transferred to the 'main()' function. They are:

5.3.1. Clear BSS Section:

Before the control can go to 'main', we need to set all the data in BSS sections to zero as the programmer expects all global & static variables to be initialized to zero. The code for this can be found in C6713_OS_Init.asm.

5.3.2. CPU Initializations:

We also need to perform certain CPU register initializations like ISTP (interrupt vector table address register), CSR, IER initialization etc. The code for this can be found in _C6713_SystemInit function within C6713_OS_Init.asm.

5.3.3. System Stack Setup:

Our kernel is designed to work with or without the CCS provided CRT libraries. When we are not using ready CRT libraries, we need to do certain initializations ourselves. Especially all CRT libraries have 'c_int00' function which sets up stack. In the absence of CRT, we need to do this our selves. The function 'c_int00' in file 'autoinit.c' initializes the CPU SP register to point to the bottom of the system stack region declared in 'autoinit.c'. Till this is done, we cannot make any C

function calls (use branch instructions instead). We also setup the DP (register B14, data pointer) register to point to the beginning of the BSS section. All global data are addressed relative to this register.

5.3.4. Initialization of global variables with assigned values:

In the absence of CRT, we ourselves need to take care of assigning initialized global variables to their initialization values. The compiler puts all initialized variable addresses and their values in 'cinit' memory section. As part of system initialization, we need to take these values and assigned them to their corresponding addresses. The '_auto_init()' function in 'autoinit.c' file has the code for doing this.

5.4. Invocation of user 'main() function

After all the system initializations described above, the user main() function is called. By this time all global/static variables are set to zero, initialized variables are set to their initialized values. The system stack is setup. The main() function is called under the system context itself, which is currently setup with 1024 bytes of stack. The programmer has to take care that he doesn't exceed this limit within main function or the functions called by main.

The programmer is expected to call 'OS_Init()' function before he calls any other OS functions. He can create few periodic & aperiodic tasks within main as he likes. And at the end he should call 'OS_Start()' function. This function actually starts the OS scheduling. It is important to note that this function never returns.

The picture 1 shows a sample of how main() function looks like. In this picture it creates 4 periodic tasks with different periods, phase, deadlines and execution times.

5.5. Kernel APIs

All APIs provided by the kernel are listed in 'NanoOS.h' file. This basically provides APIs for creating periodic & aperiodic tasks, semaphore management, getting information on task CPU usage, global CPU usage, Sleep etc. Please see 'NanoOS.h' for details of all the APIs and their semantics.

We can see that the creation function for periodic tasks take all 4 parameter of the periodic task model (phase, period, deadline & execution

budget). All these parameters can be specified with up to 1 micro second resolution. We can also see that for periodic tasks, we do not take priority as one of the inputs as periodic tasks are scheduled based on the pre-emptive EDF scheduling which is a dynamic priority algorithm. Where as the priority is one of the arguments for creating aperiodic tasks.

```
#include "SOURCES\NanoOS.h"

#define LED_on(x)       *(int *)0x90080000 |= 0x01 << x
#define LED_off(x)      *(int *)0x90080000 &= ~(0x01 << x)
#define LED_toggle(x)   *(int *)0x90080000 ^= (0x01 << x)

OS_PeriodicTask task1, task2, task3, task4;

void task_fn(int * ptr)
{
    LED_toggle(*ptr);
}

UINT32 stack1 [0x100], stack2 [0x100], stack3 [0x100], stack4 [0x100];

int a = 0, b = 1, c = 2, d = 3;

void main(int argc, char *argv[])
{
    OS_Init();
    OS_CreatePeriodicTask(100000, 5000, 3000, 0, stack1,
        sizeof(stack1), &task1, task_fn, &a);
    OS_CreatePeriodicTask(200000, 20000, 2000, 0, stack2,
        sizeof(stack2), &task2, task_fn, &b);
    OS_CreatePeriodicTask(500000, 50000, 2000, 0, stack3,
        sizeof(stack3), &task3, task_fn, &c);
    OS_CreatePeriodicTask(1000000, 90000, 2000, 0, stack4,
        sizeof(stack4), &task4, task_fn, &d);
    OS_Start();

    return;
}
```

Fig-2: Sample test application

5.6. Kernel Objects

The task control blocks for tasks, semaphore objects etc form the kernel objects. The basic philosophy of our kernel is not use any dynamic memory at the kernel level. Hence we rely on the programmer to provide memory for the

kernel objects. Since in our embedded platform we do not have separation of kernel space and user space this should be fine. As shown in the picture 1, the task/semaphore structure are declared globally or created dynamically and passed to task/semaphore creation routines.

5.7. Global Time

We have two 32 bit counters in our embedded platform which counts once for every 4 CPU clock cycles. Using 32 bit counters for keeping track of the current time is not sufficient. We would need a 64 bit counter for this purpose. We achieve this by using 64 bit software timer. When the scheduler is running we set the timer0 to interrupt at the immediate next pending deadline / release time of some task. This time is always short enough to fit within 32 bits. When the interrupt occurs, we update the software global timer with the time that has elapsed since the last timer0 interrupt. Thus we maintain a continuously running 64 bit global time. All periodic tasks maintain one 'alarm_time' attribute within their TCB which stand for when does the task requires next attention (either because of a deadline expiry or to introduce new job for that task).

5.8. Periodic Task Scheduling

We use **Preemptive EDF** policy to schedule periodic tasks. The Fig-3 shows how this scheduling is done. The following steps summarize the basic scheduling for periodic tasks:

1. When a new task is created, an entry level schedulability test is performed before the task is accepted.
2. Once the task is accepted, it is added to the 'WaitQ' with the next wakeup time as its 'phase'. Also the 'Reschedule()' function is called to make sure that this task is taken for scheduling if it has a nearest deadline.
3. When the timer0 interrupts for the first time after OS_Start(), we start the software global timer with 0 value.
4. Timer0 Interrupt Handler
    - Whenever the timer0 interrupt occurs, we first check the tasks front of the ReadyQ which have their 'alarm_time' as the current global time. All these tasks had their deadline occurring at this time. Hence we move all those tasks from the 'ReadyQ' to 'WaitQ'. Since all these tasks missed their deadlines, we increment the 'deadline_miss' count for the corresponding tasks.

- We do not need to iterate through all tasks in the 'ReadyQ'. We only need to iterate through the tasks at the front of the queue as long as the 'alarm_time' for those tasks is = the current global time. As the 'ReadyQ' is a priority queue, this would suffice.
- When the tasks are added to 'WaitQ' from the 'ReadyQ' we need to set their alarm time as the next release time for that task. Based on the requested alarm time for a given task, it gets added at a proper location within the 'WaitQ'.
- If certain tasks had deadline = period, we reintroduce the task back to 'ReadyQ' instead of 'WaitQ' as we needed to create a new job for this task.
- We then iterate through the tasks at the front of the 'WaitQ' which have their alarm_time as the current global time. All these tasks need to reintroduce new jobs at this time. Hence we move these tasks from 'WaitQ' to 'ReadyQ' with the alarm time as the next deadline for the new job.
- Once we move tasks b/w Ready/WaitQ, we should now take the front most task from the ReadyQ and switch to that task. This task starts running as soon as the Timer0 ISR returns.
- If there was no task in ReadyQ, we should take the first task from the Aperiodic task queue and switch to that task.

5. Budget Tracking using Timer1.
   - Whenever a new job is introduced (task moves from WaitQ to ReadyQ), we set the remaining_time parameter for the task as its execution time specified at the time of task creation.
   - Whenever we switch to a periodic task (it has to be at the front of the ReadyQ), we set the timer1 to with a time out as the current task's 'remaining_time'.
   - Whenever a periodic task is switched out (due to a new task with a smaller deadline), we see how much of the remaining_time is remaining in Timer1, and set this value in the outgoing task's TCB. This value will be later used when that task is switched back in.
   - Whenever the timer1 is triggered because the current task has exceeded its deadline, we get a timer1 interrupt. At this point we increment the 'TBE_count' for the task for having exceeded its budget and move that task from 'ReadyQ' to the 'WaitQ' with the alarm_time

as its next release time. We also keep track of total execution time given to a task in the 'accumulated_budget' parameter in the TCB.

6. Scheduling Aperiodic task:

When timer0 interrupt occurs after the tasks are moved between 'WaitQ' and 'ReadyQ', the control will be transferred to the first task in the 'ReadyQ'. If there were no task in the 'ReadyQ', we would switch to the first task in the 'AperiodicQ'.

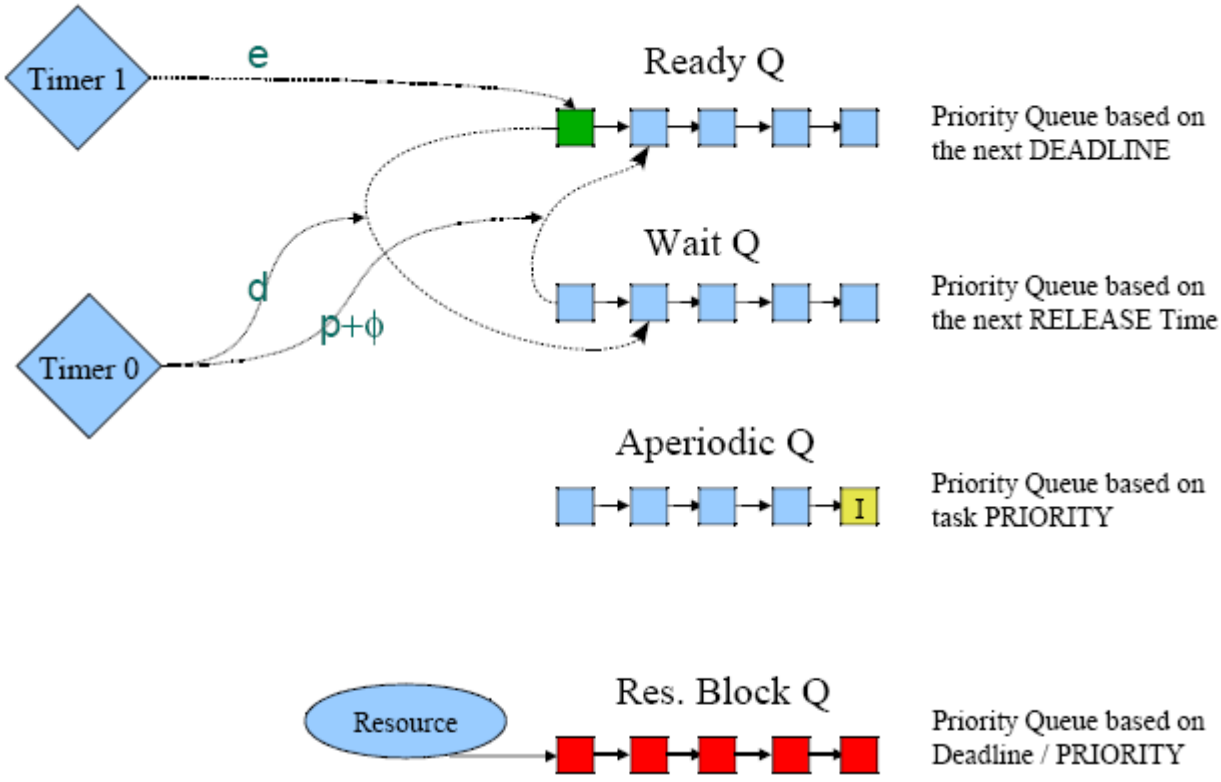


Fig-3: High level design of the scheduler

5.9. Idle Task

When the main() function calls 'OS_Start()' function, it starts the Timer1 interrupt which will then take care of scheduling. The main thread then turns itself into an 'idle' task by creating a 'Aperiodic' task TCB for itself and inserting itself into the 'AperiodicQ' with the lowest priority. Whenever no other tasks are ready to run, this idle task will be run. See the 'OS_Start()' in 'Nano_Sched.c' file.

5.10. Resources

As discussed earlier, we support creating semaphores. The space for the semaphore object is created within the user space and passed onto the kernel. As part of semaphore structure, it maintains a single queue of blocked tasks. This is also a priority queue. The periodic tasks are inserted at the front ordered by their alarm_time. The aperiodic tasks are inserted from the tail ordered by their priority. At some later point, we intend to create separate block queue for periodic and aperiodic tasks within the resource object for efficiency reasons.

5.11. Context Switch functions

We have several routines for actually swapping context. See file 'C6713_OS_Context.asm' for all context switching functions.

The function, _C6713_BuildTaskStack builds initial stack for a new periodic & aperiodic task.

The function _C6713_IntContextStore function stores the context of the current task into it's stack when an interrupt occurs. Once all registers are stored in the stack, it updates the latest stack pointer in the current task's TCB.

The function _C6713_ContextRestore, restores the current context to a new task passed as an argument. It also changes the current task pointer in 'g_current_task' global variable.

The function _C6713_ContextSw switches the context from one thread to another thread. It first stores the current context in the current task's stack, update its TCB and the new task, restore the context to the new thread, update g_current_task variable.

Note that all these functions have portions which need to be executed within the critical section and portions which can be executed outside the critical section. Effort is taken to minimize the time spent within the critical section.

5.12. Idle Task

When the main() function calls 'OS_Start()' function, it starts the Timer1 interrupt which will then take care of scheduling. The main thread then turns itself into an 'idle' task by creating a 'Aperiodic' task TCB for itself and inserting itself into the 'AperiodicQ' with the lowest priority. Whenever no other tasks are ready to run, this idle task will be run. See the 'OS_Start()' in 'Nano_Sched.c' file.

5.13. Resources

As discussed earlier, we support creating semaphores. The space for the semaphore object is created within the user space and passed onto the kernel. As part of semaphore structure, it maintains a single queue of blocked tasks. This is also a priority queue. The periodic tasks are inserted at the front ordered by their alarm_time. The aperiodic tasks are inserted from the tail ordered by their priority. At some later point, we intend to create separate block queue for periodic and aperiodic tasks within the resource object for efficiency reasons.

5.14. Context Switch functions

We have several routines for actually swapping context. See file 'C6713_OS_Context.asm' for all context switching functions.

The function, _C6713_BuildTaskStack builds initial stack for a new periodic & aperiodic task.

The function _C6713_IntContextStore function stores the context of the current task into it's stack when an interrupt occurs. Once all registers are stored in the stack, it updates the latest stack pointer in the current task's TCB.

The function _C6713_ContextRestore, restores the current context to a new task passed as an argument. It also changes the current task pointer in 'g_current_task' global variable.

The function _C6713_ContextSw switches the context from one thread to another thread. It first stores the current context in the current task's stack, update its TCB and the new task, restore the context to the new thread, update g_current_task variable.

Note that all these functions have portions which need to be executed within the critical section and portions which can be executed outside the critical section. Effort is taken to minimize the time spent within the critical section.

5.15. Interrupt Context

Whenever an interrupt occurs we store the context of the currently running thread using the function _C6713_IntContextStore. We then need some other context to run the ISR. This is accomplished by having a separate stack called 'Interrupt Stack'. Every ISR changes the context to interrupt context before it executes rest of the interrupt handling. Changing to interrupt context just requires us to set the SP to the bottom of interrupt stack. When the ISR is about to return it always returns to some task by switching to that thread's context. Note that we do not store the registers in the interrupt stack when we return to a

task. And every time an interrupt occurs we start from the bottom of interrupt stack. This is highly efficient. At this point of time we do not allow nested interupts.

5.16. Critical Section

For all critical sections within the kernel we use enable/disable interrupts. However, we make every effort to make sure that this critical section is kept very small.

As discussed earlier, we support creating semaphores. The space for the semaphore object is created within the user space and passed onto the kernel. As part of semaphore structure, it maintains a single queue of blocked tasks. This is also a priority queue. The periodic tasks are inserted at the front ordered by their alarm_time. The aperiodic tasks are inserted from the tail ordered by their priority. At some later point, we intend to create separate block queue for periodic and aperiodic tasks within the resource object for efficiency reasons.

5.17. Context Switch functions

We have several routines for actually swapping context. See file 'C6713_OS_Context.asm' for all context switching functions.

The function, _C6713_BuildTaskStack builds initial stack for a new periodic & aperiodic task.

The function _C6713_IntContextStore function stores the context of the current task into it's stack when an interrupt occurs. Once all registers are stored in the stack, it updates the latest stack pointer in the current task's TCB.

The function _C6713_ContextRestore, restores the current context to a new task passed as an argument. It also changes the current task pointer in 'g_current_task' global variable.

The function _C6713_ContextSw switches the context from one thread to another thread. It first stores the current context in the current task's stack, update its TCB and the new task, restore the context to the new thread, update g_current_task variable.

Note that all these functions have portions which need to be executed within the critical section and portions which can be executed outside the critical section. Effort is taken to minimize the time spent within the critical section.

5.18. Interrupt Context

Whenever an interrupt occurs we store the context of the currently running thread using the function _C6713_IntContextStore. We then need some other context to run the ISR. This is accomplished by having a separate stack called 'Interrupt Stack'. Every ISR changes the context to interrupt context before it executes rest of the interrupt handling. Changing to interrupt context just requires us to set the SP to the bottom of interrupt stack. When the ISR is about to return it always returns to some task by switching to that thread's context. Note that we do not store the registers in the interrupt stack when we return to a task. And every time an interrupt occurs we start from the bottom of interrupt stack. This is highly efficient. At this point of time we do not allow nested interupts.

5.19. Critical Section

For all critical sections within the kernel we use enable/disable interrupts. However, we make every effort to make sure that this critical section is kept very small. Whenever an interrupt occurs we store the context of the currently running thread using the function _C6713_IntContextStore. We then need some other context to run the ISR. This is accomplished by having a separate stack called 'Interrupt Stack'. Every ISR changes the context to interrupt context before it executes rest of the interrupt handling. Changing to interrupt context just requires us to set the SP to the bottom of interrupt stack. When the ISR is about to return it always returns to some task by switching to that thread's context. Note that we do not store the registers in the interrupt stack when we return to a task. And every time an interrupt occurs we start from the bottom of interrupt stack. This is highly efficient. At this point of time we do not allow nested interupts.

6. Results

We implemented our new kernel on C6713 DSK from Texas Instruments. This has a TMS320C6713 DSP Processor which is running at 150MHz (Fig-4). This processor has VLIW Architecture (with 8 instructions / cycle) where a single instruction extending up to 256 bits.

Fig-4: C6713 DSK

Our implementation resulted in about 2400 SLOC with about 1000 SLOC in Assembly.

We created multiple periodic tasks with different (f, p, e, D) parameters and they get scheduled properly. Currently we used 4 user LEDs being controlled from 4 different tasks at different periods/phases. It works properly.

We tested d parameter by having code similar to what is given below, this code basically toggles one LED and waits for the deadline to pass. Once the deadline is expired, it gets preempted. In the next period when the task comes up again, it quits the while loop and returns without doing anything else. In the next period again the same thing repeats. By looking at the LEDs blinking and the dline_miss_count and exec_count (# of jobs executed for that task) we can infer the correct behavior.

```
void task_fn(int * ptr)
{
        OS_PeriodicTask * task = (OS_PeriodicTask *) g_current_task;
        UINT32 dm = task->dline_miss_count;
        LED_toggle(*ptr);
        while(dm == task->dline_miss_count)
        {}
}
```

Similar test with TBE_count also works but has a bug which need to be fixed.

We also are planning for more tests like running standard benchmark tests on multiple tasks and comparing the performance against plain Priority based scheduling of MicroC OS and RMA scheduler developed for the FREEDM project to get a feel of the performance of our current implementation.

7. Summary

- Implemented on C6713 DSK
- 320C6713 DSP Processor
- IW Architecture (with 8 instructions / cycle)
- Tested for all parameters (f, p, e, D)
- Keeps track of Deadline miss & TBE counts for every thread
- Also keeps track of thread wise execution time upto 1us resolution.
- About 2400 SLOCs of source code (1000 lines assembly)

References

[1] Real-Time Systems, Jane W.S. Liu, Pearson Education

[2] Lecture notes

[3] http://www.ti-estore.com/Merchant2/merchant.mvc?Screen=PROD&Product_Code=TMDSDSK6713

Appendix

API Description

```
// macro for entering the critical section
#define  OS_ENTER_CRITICAL( )  _disable_interrupt()

//macro for leaving the critical section
#define  OS_EXIT_CRITICAL( )   _enable_interrupt()

////////////////////////////////////////////////////////////////////////
// Task creation APIs
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to create periodic tasks
//@period_in_us - period of the task in micro seconds
//@deadline_in_us - deadline of the task in micro seconds
//@budget_in_us - WCET of the task in micro seconds
//@phase_shift_in_us - phase of the task in micro seconds
//@stack - pointer to the stack
//@stack_size_in_bytes - size of the stack in bytes
//@task - pointer to the task contol block (user allocated memory)
//@periodic_entry_function - function pointer where the execution begins on creation of a job
//@pdata - parameters passed to 'periodic_entry_function'
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_CreatePeriodicTask(
UINT32 period_in_us,
UINT32 deadline_in_us,
UINT32 budget_in_us,
UINT32 phase_shift_in_us,
UINT32 * stack,
UINT32 stack_size_in_bytes,
OS_PeriodicTask *task,
void (*periodic_entry_function)(void *pdata),
void *pdata);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to create aperiodic tasks
//@priority - priority of the aperiodic task (0 - HIGHEST, 255 - LOWEST)
//@stack - pointer to the stack
//@stack_size_in_bytes - size of the stack in bytes
//@task - pointer to the task contol block (user allocated memory)
//@periodic_entry_function - function pointer where the execution begins on creation of a job
//@pdata - parameters passed to 'periodic_entry_function'
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_CreateAperiodicTask(
UINT16 priority,
UINT32 * stack,
UINT32 stack_size_in_bytes,
OS_AperiodicTask *task,
void (*task_entry_function)(void *pdata),
void *pdata);
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The following function Initializes the OS data structures
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void OS_Init();

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The following function starts the OS scheduling
// Note that this function never returns
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void OS_Start();

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The below function, gets the total elapsed time since the beginning
// of the system in microseconds.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
UINT64 OS_GetElapsedTime();

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The following function gets the total time taken by the current
// thread since the thread has begun in microseconds. Note that this is not the global
// time, this is just the time taken from only the current thread.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
UINT64 OS_GetThreadElapsedTime();

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Semaphore functions
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to create semaphore
//@sem - pointer to semaphore control structure (user allocated memory)
//@pshared - semaphore is shared between processes (not used currently)
//@value - initial value of the semaphore
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_SemInit(OS_Sem *sem, INT16 pshared, UINT32 value);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to wait on a semaphore
//@sem - pointer to semaphore control structure created using OS_SemInit
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_SemWait(OS_Sem *sem);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to signal a task waiting on a semaphore
//@sem - pointer to semaphore control structure created using OS_SemInit
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_SemPost(OS_Sem *sem);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to destroy a semaphore
//@sem - pointer to semaphore control structure created using OS_SemInit
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_SemDestroy(OS_Sem *sem);
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to read the value of a semaphore
//@sem - pointer to semaphore control structure created using OS_SemInit
//@val - pointer to an integer where semaphore's value is returned
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_SemGetvalue(OS_Sem *sem, INT32 *val);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to create a mutex
//@mutex - pointer to mutex control structure (user allocated memory)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_MutexInit(OS_Mutex *mutex);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to lock a  mutex
//@mutex - pointer to mutex control structure created using OS_MutexInit()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_MutexLock(OS_Mutex *mutex);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to unlock  a mutex
//@mutex - pointer to mutex control structure created using OS_MutexInit()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_MutexUnlock(OS_Mutex *mutex);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//API to destroy a mutex
//@mutex - pointer to mutex control structure created using OS_MutexInit()
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
OS_Error OS_MutexDestroy(OS_Mutex *mutex);
```