POWER AWARE SCHEDULING FOR ADHOC SENSOR NETWORK NODES

BY

ANKIT R. THAKKAR

07MCE022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2009

POWER AWARE SCHEDULING FOR ADHOC SENSOR NETWORK NODES

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

ANKIT R. THAKKAR

07MCE022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2009

Certificate

This is to certify that the Major Project entitled "Power Aware Scheduling For Adhoc Sensor Network Nodes" submitted by Ankit R. Thakkar (07MCE022), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. S.N. Pradhan
Guide and Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

Adhoc Sensor Networks are being considered for many novel applications. This thesis investigates into the power aware scheduling aspects of adhoc sensor networks with a real time test-bed environment. This will include the Periodic and Aperiodic Task Scheduling Issues, as well as power consumption by sensor nodes.

The architecture configured for the project work has been put forward with intricately defined requirements for each component. Texas Instruments's low power microprocessor, MSP430 is used to implement Modified Maximum Urgency First (MMUF), priority based scheduling algorithm to schedule periodic and aperiodic tasks generated by the MicroC/OS-II operating system or interrupt received by the MSP430. Aperiodic tasks always given higher priority compare to periodic tasks. The scheduler is configured to run on the MicroC/OS-II Real Time Operating System (RTOS). Out of the two Real Time Operating Systems for sensors (TinyOS and MicroC/OS-II), which were studied, MicroC/OS-II was selected for the project work, due to its extensibility, robustness and priority scheduling. Out of the five scheduling algorithms (RM, EDF, MLF, MUF and MMUF), which were studied, MMUF was selected for the project work, because it doesn't allow critical task to miss deadline. As compared to conventional Real Time Operating Systems for sensors, which are generally not open systems, MicroC/OS-II can be used as an Operating System for sensors and can be extended to implement new scheduling algorithms, which can be integrated and tested in real time working environment.

Another aspect of the project work is user priority based scheduling of the tasks. This is because; the Adhoc Sensor Network based Applications must adhere to stringent real-time constraints and Power Aware requirements. Therefore, a user priority based, Modified Maximum Urgency First (MMUF) is developed to orchestrate and guarantee the timely interaction between such applications. In this context, a kernel

level module is developed to switch the MSP430 into Low Power Mode (LPM) to reduce the power consumption by the sensors, when no user task is active.

Acknowledgements

With immense pleasure, I would like to present this report on the dissertation work related to "Power Aware Scheduling For Adhoc Sensor Network Nodes". I am very thankful to all those who helped me for the successful completion of the first phase of the dissertation and for providing valuable guidance throughout the project work.

I would first of all like to offer thanks to **Dr. S. N. Pradhan**, Guide & Programme Co-ordinator M.Tech. CS&E, Institute of Technology, Nirma University, Ahmedabad whose keen interest and excellent knowledge base helped me to finalize the topic of the dissertation work. His constant support and interest in the subject equipped me with a great understanding of different aspects of the required architecture for the project work. He has shown keen interest in this dissertation work right from beginning and has been a great motivating factor in outlining the flow of my work.

My sincere thanks and gratitude to **Prof. D.J. Patel**, Professor and Head, Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the Dissertation work.

I am thankful to Nirma University for providing all kind of required resources. I would like to thank The Almighty, my family, especially my wife, for supporting and encouraging me in all possible ways. I would also like to thank all my friends who have directly or indirectly helped in making this dissertation work successful.

- Ankit R. Thakkar

07MCE022

Abbreviation Notation and Nomenclature

ACLK ..... Auxiliary Clock
ADC ..... Analog-to-Digital Converter
BOR ..... Brown-Out
BSL ..... Bootstrap Loader
DAC ..... Digital-to-Analog Converter
DCO ..... Digitally Controlled Oscillator
dst ..... Destination
EDF ..... Earliest Deadline First
FLL ..... Frequency Locked Loop
GIE ..... General Interrupt Enable
INT(N/2) ..... Integer portion of N/2
I/O ..... Input/Output
ISR ..... Interrupt Service Routine
LSB ..... Least-Significant Bit
LSD ..... Least-Significant Digit
LPM ..... Low-Power Mode
MAB ..... Memory Address Bus
MCLK ..... Master Clock
MDB ..... Memory Data Bus
MLF ..... Maximum Laxity First
MSB ..... Most-Significant Bit
MSD ..... Most-Significant Digit
MMUF ..... Modified Maximum Urgency First
MUF ..... Maximum Urgency First
NMI ..... Non-Maskable Interrupt
PC ..... Program Counter
POR ..... Power-On Reset

PUC ..... Power-Up Clear
RM ..... Rate Monotonic
SCG ..... System Clock Generator
SFR ..... Special Function Register
SMCLK ..... Sub-System Master Clock
SP ..... Stack Pointer
SR ..... Status Register
src ..... Source
TOS ..... Top-of-Stack
WDT ..... Watchdog Timer

Contents

Certificate iii

Abstract iv

Acknowledgements vi

Abbreviation Notation and Nomenclature vii

List of Tables xii

List of Figures xiv

1 Introduction 1
- 1.1 Background . . . 2
- 1.2 Objective of Study . . . 3
- 1.3 Scope of Work . . . 4
- 1.4 Thesis Organization . . . 5

2 MSP430 Features 6
- 2.1 Features and Capabilities . . . 7
- 2.2 Interrupt Processing . . . 8
  - 2.2.1 Interrupt Control Bits in Special-Function Registers (SFRs) . 10
- 2.3 Operating Modes . . . 11
  - 2.3.1 Low-Power Modes 0 and 1 (LPM0 and LPM1) . . . 14
  - 2.3.2 Low-Power Modes 2 and 3 (LPM2 and LPM3) . . . 15
  - 2.3.3 Low-Power Mode 4 (LPM4) . . . 16
  - 2.3.4 Basic Hints for Low-Power Applications . . . 16
- 2.4 Summary . . . 17

3 Real Time Operating System for Sensor 18
- 3.1 Critical Sections . . . 18
- 3.2 Task States . . . 19
- 3.3 Task Control Blocks (OS_TCB) . . . 20
- 3.4 OS_TCBInit() . . . 24

3.5 Ready List . . . 24
3.5.1 Making a task ready to run . . . 25
3.5.2 Removing a task from the ready list . . . 26
3.5.3 Finding the highest priority task . . . 27
3.6 Task Scheduling . . . 28
3.7 Task Level Context Switch, OS_TASK_SW() . . . 30
3.8 Locking and Unlocking the Scheduler . . . 30
3.9 Starting Multitasking . . . 32
3.10 Creating a Task, OSTaskCreate() . . . 33
3.11 Summary . . . 35

4 Scheduling Algorithms for Sensor Nodes 36
4.1 Introduction . . . 36
4.2 Rate Monotonic Algorithm (RM) . . . 37
4.3 Earliest-Deadline-First Scheduling Algorithm (EDF) . . . 38
4.4 Minimum-Laxity-First Scheduling Algorithm (MLF) . . . 38
4.5 Maximum-Urgency-First scheduling algorithm (MUF) . . . 39
4.6 Modified Maximum-Urgency-First scheduling algorithm (MMUF) . . . 43
4.7 Summary . . . 46

5 Problem Definition and Existing Methodologies 47
5.1 Problem Definition . . . 47
5.2 Existing Methodologies . . . 48
5.2.1 Method One . . . 48
5.2.2 Method Two . . . 48
5.2.3 Method Three . . . 48
5.3 Summary . . . 49

6 The Proposed Algorithm 50
6.1 The Algorithm . . . 50
6.2 Summary . . . 51

7 Implementation 53
7.1 Implementation Environment . . . 53
7.1.1 Porting of MicroC/OS-II on MSP430 IAR Embedded Workbench IDE . . . 54
7.1.2 Data structures used to implement MMUF . . . 54
7.2 Results . . . 56
7.2.1 Case 1: All tasks are periodic . . . 57
7.2.2 Case 2: All tasks are periodic except one. Aperiodic task never becomes ready. . . . 61
7.2.3 Case 3: All tasks are periodic except one. Aperiodic task becomes ready once only. . . . 64

7.2.4 Case 4: New task will be added to the task list on the reception of the interrupt. 64
7.2.5 Analysis of Results 68
7.3 Summary 70

8 Conclusion and Future Scope **73**
8.1 Conclusion 73
8.2 Future Scope 74

A Architectural Overview of MSP430X44X **75**
A.1 Introduction 75
A.2 Central Processing Unit 76
A.3 Program Memory 77
A.4 Data Memory 77
A.5 Operation Control 78
A.6 Peripherals 78

B Peripheral Modules and Address Allocation **79**
B.1 Introduction 79
B.1.1 Peripheral Modules - Address Allocation 80

References **84**

Index **85**

List of Tables

I Low-Power Mode Logic Chart 14

I OSMapTbl[] 26

I Example of Task Set 44

I Functions contained in OS_CPU_A.S43 54
II Functions contained in OS_CPU.C 54
III Description about MMUF_DATA 55
IV Function contained in MMUF.C 56

I Peripheral File Address Map-Word Modules 81
II Peripheral File Address Map-Byte Modules 82

List of Figures

2.1 Interrupt Processing 9
2.2 Return From Interrupt 10
2.3 Status Register (SR) 10
2.4 Interrupt Control Bits in SFRs 11

3.1 Task States 20
3.2 List of free OS_TCB 23
3.3 Relationship between OSRdyGrp and OSRdyTbl 25

4.1 Example comparing RM, EDF, and MUF algorithms 41
4.2 Scheme to encode n-bit urgency value for MUF 43
4.3 Schedule generated by the MUF scheduling algorithm 44

6.1 Implementation of MMUF on MicroC/OS-II using MSP430 52

7.1 Simulation Input: All Six Tasks are Periodic 58
7.2 Schedule Generated by MMUF : All Six Tasks are Periodic 58
7.3 Status of the MicroC/OS-II : All Six Tasks are Periodic 59
7.4 Task list status at T = 40 : All are periodic 59
7.5 Simulation Input: Five tasks are Periodic and one is Aperiodic 62
7.6 Schedule Generated by MMUF : Five tasks are Periodic and one is Aperiodic 62
7.7 Status of the MicroC/OS-II : Five tasks are Periodic and one is Aperiodic 63
7.8 Task list status at T = 40 : Five tasks are Periodic and one is Aperiodic 63
7.9 Schedule Generated by MMUF : Five tasks are Periodic and one is Aperiodic 64
7.10 Status of the MicroC/OS-II : Five tasks are Periodic and one is Aperiodic 65
7.11 Task list status at T = 40 : Five tasks are Periodic and one is Aperiodic 65
7.12 Simulation Input: Three tasks are Periodic and one is Aperiodic. Among periodic, one task - Task 2, is created with the reception of interrupt. 66

7.13 Schedule Generated by MMUF : Three tasks are Periodic and one is Aperiodic.Among periodic, one task - Task 2, is not created because interrupt is not received. 66
7.14 Status of the MicroC/OS-II : Three tasks are Periodic and one is Aperiodic.Among periodic, one task - Task 2, is not created because interrupt is not received. 67
7.15 Task list status at T = 40 : Two Tasks are Periodic and one is Aperiodic. Third periodic task, Task 2, is not created due to the UARTRX0 interrupt is not received. 67
7.16 Schedule Generated by MMUF : Three tasks are Periodic and one is Aperiodic 68
7.17 Status of the MicroC/OS-II : Three tasks are Periodic and one is Aperiodic 69
7.18 Task list status at T = 40 : Three tasks are Periodic and one is Aperiodic 69
7.19 Nominal Power Consumption for CASE 1,2 & 3 up to T=40 71
7.20 Maximum Power Consumption for CASE 1,2 & 3 up to T=40 72
7.21 Power Saved for CASE 1,2 & 3 up to T=40 72

A.1 MSP430 System Configuration 76

B.1 Memory Map of Basic Address Space 80
B.2 Example of RAM/Peripheral Organization 81
B.3 Special Function Register Address Map 83

Chapter 1

Introduction

The dissertation work related to 'Power aware scheduling for adhoc sensor network nodes' demands the selection of Real Time Operating System (RTOS) for sensors, selection of scheduling algorithm and selection of the micro-controller. This requires to be done very meticulously, in order to meet the real time demands of the sensor nodes. The proposed architecture is inspired from a Texas Instrument's MSP430 based application. The system includes one MPS430 micro-controller, on which operating system and scheduling algorithm runs. The user can set the task-set, which can be a mixture of periodic and aperiodic tasks, can be generated by the reception of the interrupt by MSP430. This task-set is schedule by the scheduler to meet the real time and power aware demands of the user or application. MSP430 remains in the low power mode to reduce the power consumption by the sensor node, when no user task is active. This requirement is handled by the MicroC/OS-II, Real Time Operation System (RTOS) for sensor nodes. Whenever any interrupt is received by the MSP430 or any of the user task becomes active, MSP430 comes out from the low power mode within 6 μs. Most of the results derived are for single sensor node by scheduling periodic, aperiodic tasks which can be generated as per the user requirements and/or when interrupt is received. Also, it is assumed that aperiodic task has higher priority compared to periodic tasks.

1.1 Background

Adhoc sensor based network has proven itself to be an efficient and effective medium for information communication. Recently there have been many developments in technologies, which have made sensor nodes to consume very less power to extend their life. The prominent developments can be grouped in five parts.

First, the newer better protocols. While addressing the shortcomings of ancestors, newer protocols also provide novel ways for delivering better services between sensor nodes.

Second, the development of newer hardware technologies delivered faster processing powers that enabled faster data processing, the core requirement of sensor applications. Faster CPUs/MPUs, smaller & faster RAMs, ROMs made it possible to handle the sensor application data in real-time environments.

Thirdly, the alternate mediums. The existence of alternate mediums provided the platform for wider reach. Wireless technology, the mobile communications have grown to 4G age, providing the right set of Quality of Service for the sensor data communication.

Fourth, the newer RTOS. The development of excellent RTOS for sensors greatly supported the cause. TinyOS, FreeRTOS, MicroC/OS-II and many others could provide great flexibility to configure and shrink the component of RTOS, as per the requirement of the application, which reduces the requirement of RAM & ROM to deliver the desired performance and power consumption requirements of the sensors.

Fifth, the improving real time scheduling algorithms. The development of newer real time scheduling algorithms to support real-time requirements of the sensor ap-

plication. So, the critical tasks doesn't miss their respective deadlines.

The ball is rolling, but there are still many challenges to be met. The adhoc sensor network with its demanding Quality of Service and reducing power consumption, have always been providing the thrust/need for further developments. The periodic task set, aperiodic task set, critical task set, non-critical task set, task set containing mixture of all these types of task, environmental issues, response times as well as efficient handling of task through scheduler(s), compatibility issues among sensor nodes, protocols & to reduce power consumption by the sensors are few related ongoing areas of active research.

1.2 Objective of Study

Apart from the above mentioned goals and vision, this report addresses the specific aspect of power aware scheduling of mixed task set. The experimental setup has been created on a computer, with software based sensor node. Hence, the objective of the dissertation work can be summarized as:

Create mixed task set on a real time operating system and schedule them. The scheduling strategies should look into user priorities, real time requirement and power aware aspect related issues.

Study and selection of micro-controller unit (MCU) and configure them for the project work. Study related to design for sensor nodes and power aware requirements, have also been made.

The next objective is, study, selection and porting of RTOS on the selected micro-controller. The RTOS must small in size, support mixture of task set, switch the MCU between active and Low Power Mode (LMP) state and be modifiable to meet

requirements.

The next objective is, study, selection and implementation of scheduling algorithm on the selected RTOS. The scheduling algorithm must be preemptive, support mixture of task set, support user priorities and meet the real-time requirements of the sensor application.

The next objective is, to test the performance of the scheduler in scheduling of mixed task set, with the ported RTOS on the experimental test-bed to meet the real-time requirements. For this purpose, periodic and aperiodic tasks are generated using startup task of RTOS and through interrupts.

Finally, the objective is to measure the power consumption done by the MCU, with the ported RTOS on the experimental test-bed while meeting the real-time requirements.

1.3 Scope of Work

The experimental setup, prepared for the dissertation work, includes a simulator installed on a single computer in a Computer Lab. A part of the work is study and selection of micro-controller which can be used for the design of the senors. Texas Instrument's MSP430 is a very low power micro-controller and it can support five different low power modes. This is relatively a very new micro-controller and is like a boon for the design of the sensor fraternity. Thus study and porting of MicroC/OS-II, RTOS on MSP430 also become part of the thesis. Out of the various real-time scheduling algorithms, the thesis work includes a detailed implementation of MMUF in real time kernel MicroC/OS-II on MSP430 with and without using Low Power Modes (LPMs) and compare power consumption done by the MSP430.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *MSP430 Features*, describes the features and capabilities of the MSP430 and various low power modes it can support. It also outlines the basic hints for low power applications.

Chapter 3, *Real Time Operating System for Sensor*, describes MicroC/OS-II, which is a real time operating system. It also describes various kernel structures used in design of MicroC/OS-II.

Chapter 4, *Real Time Scheduling Algorithms*, describes various real-time scheduling algorithms and one which suites the needs for the sensor nodes.

Chapter 5, *Problem Definition and Existing Methodologies*, presents the problem definition and describes the existing methodologies to reduce power consumption in sensor nodes.

In **chapter 6**, *The Proposed Algorithm*, a new algorithm for performing the scheduling is presented. The algorithm suggests a new way of assigning priorities to task so critical tasks doesn't misses their respective deadlines. Also, this algorithms finds out when to switch micro-controller in LPM3.

Chapter 7, *Simulation Methodologies and Performance Evaluation*, describes in brief the procedure followed, to carry out the simulation. The simulation results along with the performance analysis of the proposed algorithm are presented.

Finally, in **chapter 8** concluding remarks and scope for future work is presented.

Chapter 2

MSP430 Features

This chapter outlines the features and capabilities of the Texas Instruments (TI) MSP430x4xx family of micro-controllers. The MSP430 employs a von-Neumann architecture; therefore, all memory and peripherals are in one address space. The MSP430 devices constitute a family of ultra low-power, 16-bit RISC micro-controllers with an advanced architecture and extensive peripheral set. The architecture uses advanced timing and design features, as well as a highly orthogonal structure, to deliver a processor that is both powerful and flexible. The MSP430 consumes less than 300 μA in active mode operating at 1 MHz in a typical 3-V system and can wake up from standby mode to fully synchronized operation in less than 6 μs. These exceptionally-low current requirements, combined with the fast wake-up time, enable a user to build a system with minimum current consumption and maximum battery life. Additionally, the MSP430 family has an abundant mix of peripherals and memory sizes enabling true system-on-a-chip designs. The peripherals include a 12-bit A/D, slope A/D, timers (some with capture/compare registers and PWM output capability), an LCD driver, on-chip clock generation, a hardware multiplier, USART, a Watchdog Timer, GPIO, and others.

2.1 Features and Capabilities

The TI MSP430x4xx family of controllers has the following features and capabilities:

- Ultralow-power architecture:
  - 0.1 - 300-μA nominal operating current at 1 MHz
  - 1.8-3.6-V operation
  - 6-μs wake-up from standby mode
  - Extensive interrupt capability relieves need for polling
- Flexible and powerful processing capabilities:
  - Seven source-address modes
  - Four destination-address modes
  - Only 27 core instructions
  - Prioritized, nested interrupts
  - No interrupt or subroutine level limits
  - Large register file
  - Ram execution capability
  - Efficient table processing
  - Fast hex-to-decimal conversion

The 44x device family includes:

- MSP430F447: 32-KB flash memory, 1-KB RAM
- MSP430F448: 48-KB flash memory, 2-KB RAM
- MSP430F449: 60-KB flash memory, 2-KB RAM

2.2 Interrupt Processing

The MSP430 programmable interrupt structure allows flexible on-chip and external interrupt configurations to meet real-time interrupt-driven system requirements. Interrupts may be initiated by the processor's operating conditions such as watchdog overflow; or by peripheral modules or external events. Each interrupt source can be disabled individually by an interrupt enable bit, or all maskable interrupts can be disabled by the general interrupt enable (GIE) bit in the status register.

Whenever an interrupt is requested and the appropriate interrupt enable bit and general interrupt enable (GIE) bit are set, the interrupt service routine becomes active as follows:

- CPU active: The currently executing instruction is completed.
- CPU stopped: The low-power modes are terminated.
- The program counter pointing to the next instruction is pushed onto the stack.
- The status register is pushed onto the stack.
- The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
- The appropriate interrupt request flag resets automatically on singlesource flags. Multiple source flags remain set for servicing by software.
- The GIE bit is reset; the CPUOff bit, the OscOff bit, and the SCG1 bit are cleared; the status bits V, N, Z, and C are reset. SCG0 is left unchanged, and loop control remains in the previous operating condition.
- The content of the appropriate interrupt vector is loaded into the program counter: the program continues with the interrupt handling routine at that address.

The interrupt latency is six cycles, starting with the acceptance of an interrupt request, and lasting until the start of execution of the appropriate interrupt-service routine first instruction, as shown in Figure 2.1. The interrupt handling routine

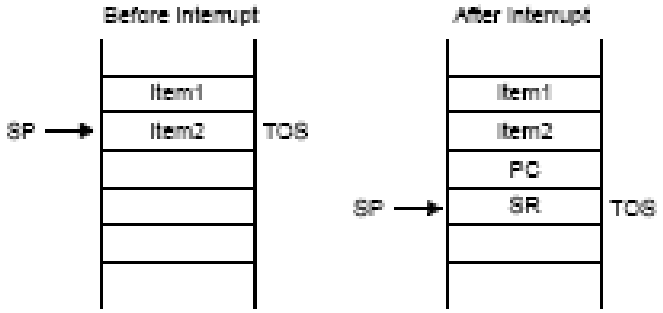

Figure 2.1: Interrupt Processing

terminates with the instruction:

```
RETI (return from an interrupt service routine)
```

which performs the following actions:

a. The status register with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings utilized during the interrupt service routine.

b. The program counter pops from the stack and begins execution at the point where it was interrupted.

The return from the interrupt is illustrated in Figure 2.2

A RETI instruction takes five cycles. Interrupt nesting is activated if the GIE bit is set inside the interrupt handling routine. The GIE bit is located in status register SR/R2, which is included in the CPU as shown in Figure 2.3.

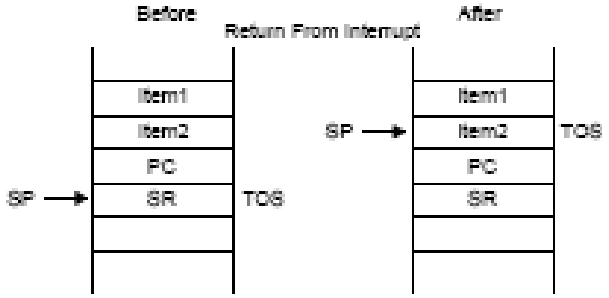

Figure 2.2: Return From Interrupt

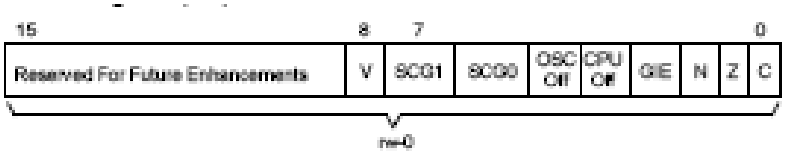

Figure 2.3: Status Register (SR)

Apart from the GIE bit, other sources of interrupt requests can be enabled/disabled individually or in groups. The interrupt enable flags are located together within two addresses of the special-function registers (SFRs). The program-flow conditions on interrupt requests can be easily adjusted using the interrupt enable masks. The hardware serves the highest priority within the empowered interrupt source.

2.2.1 Interrupt Control Bits in Special-Function Registers (SFRs)

Most of the interrupt control bits, interrupt flags, and interrupt enable bits are collected in SFRs under a few addresses, as shown in Figure 2.4. The SFRs are located in the lower address range and are implemented in byte format. SFRs must be accessed using byte instructions.

Address	7	0
000Fh	Not yet defined or implemented	
000Eh	Not yet defined or implemented	
000Dh	Not yet defined or implemented	
000Ch	Not yet defined or implemented	
000Bh	Not yet defined or implemented	
000Ah	Not yet defined or implemented	
0009h	Not yet defined or implemented	
0008h	Not yet defined or implemented	
0007h	Not yet defined or implemented	
0006h	Not yet defined or implemented	
0005h	Module enable 2 (ME2.x)	
0004h	Module enable 1 (ME1.x)	
0003h	Interrupt flag reg. 2 (IFG2.x)	
0002h	Interrupt flag reg. 1 (IFG1.x)	
0001h	Interrupt enable 2 (IE2.x)	
0000h	Interrupt enable 1 (IE1.x)	

Figure 2.4: Interrupt Control Bits in SFRs

2.3 Operating Modes

The MSP430 family was developed for ultralow-power applications and uses different levels of operating modes. The MSP430 operating modes, shown in Figure 2.3, give advanced support to various requirements for ultra-low power and ultralow-energy consumption. This support is combined with an intelligent management of operations during the different module and CPU states. An interrupt event wakes the system from each of the various operating modes and the RETI instruction returns operation to the mode that was selected before the interrupt event.

There are four bits that control the CPU and the system clock generator: CPUOff, OscOff, SCG0, and SCG1. These four bits support discontinuous active mode (AM) requests, to limit the time period of the full operating mode, and are located in the status register. The major advantage of including the operating mode bits in the status register is that the present state of the operating condition is saved onto the stack during an interrupt service request. As long as the stored status register information

is not altered, the processor continues (after RETI) with the same operating mode as before the interrupt event. Another program flow may be selected by manipulating the data stored on the stack or the stack pointer. Being able to access the stack and stack pointer with the instruction set allows the program structures to be individually optimized, as illustrated in the following program flow:

- Enter interrupt routine

  The interrupt routine is entered and processed if an enabled interrupt awakens the MSP430:

  - The SR and PC are stored on the stack, with the content present at the interrupt event.
  - Subsequently, the operation mode control bits OscOff, SCG1, and CPUOff are cleared automatically in the status register.

- Return from interrupt

  - Return with low-power mode bits set. When returning from the interrupt, the program counter points to the next instruction. The instruction pointed to is not executed, since the restored low-power mode stops CPU activity.
  - Return with low-power mode bits reset. When returning from the interrupt, the program continues at the address following the instruction that set the OscOff or CPUOff-bit in the status register. To use this mode, the interrupt service routine must reset the OscOff, CPUOff, SCGO, and SCG1 bits on the stack. Then, when the SR contents are popped from the stack upon RETI, the operating mode will be active mode (AM).

The software can configure five operating modes:

- Active mode AM; SCG1=0, SCG0=0, OscOff=0, CPUOff=0: CPU clocks are active

- Low-power mode 0 (LPM0); SCG1=0, SCG0=0, OscOff=0, CPUOff=1:
  CPU is disabled
  '44x: ACLK and SMCLK remain active. MCLK is disabled Loop control for MCLK remains active

- Low-power mode 1 (LPM1); SCG1=0, SCG0=1, OscOff=0, CPUOff=1:
  CPU is disabled
  Loop control for MCLK is disabled
  '44x: ACLK and SMCLK remain active.
  MCLK is disabled

- Low-power mode 2 (LPM2); SCG1=1, SCG0=0, OscOff=0, CPUOff=1:
  CPU is disabled
  MCLK and loop control for MCLK are disabled
  DCO's dc-generator remains enabled
  ACLK remains active

- Low-power mode 3 (LPM3); SCG1=1, SCG0=1, OscOff=0, CPUOff=1:
  CPU is disabled
  MCLK and loop control for MCLK are disabled
  DCO oscillator is disabled
  DCO's dc-generator is disabled
  ACLK remains active

- Low-power mode 4 (LPM4); SCG1=X, SCG0=X, OscOff=1, CPUOff=1:
  CPU is disabled
  ACLK is disabled
  MCLK and loop control for MCLK are disabled
  DCO oscillator is disabled
  DCO's dc-generator is disabled
  Crystal oscillator is stopped

Table I: Low-Power Mode Logic Chart

	SCG1	SCG0	OscOff	CPUOff
LPM0	0	0	0	1
LPM1	0	1	0	1
LPM2	1	0	0	1
LPM3	1	1	0	1
LPM4	X	X	1	1

2.3.1 Low-Power Modes 0 and 1 (LPM0 and LPM1)

Low-power mode 0 or 1 is selected if bit CPUOff in the status register is set. Immediately after the bit is set the CPU stops operation, and the normal operation of the system core stops. The operation of the CPU halts and all internal bus activities stop until an interrupt request or reset occurs. The system clock generator continues operation, and the clock signals MCLK/SMCLK and ACLK stay active depending on the state of the other three status register bits, SCG0, SCG1, and OscOff.

The peripherals are enabled or disabled according with their individual control register settings, and with the module enable registers in the SFRs. All I/O port pins and RAM/registers are unchanged. Wake-up is possible through all enabled interrupts.

The following are examples of entering and exiting LPM0. The method shown is applicable to all low-power modes.

The following example describes entering into low-power mode 0.

```
;===Main program flow with switch to CPUOff Mode==============

BIS     #18h,SR     ;Enter LPM0 + enable general interrupt GIE
                    ;(CPUOff=1, GIE=1). The PC is incremented
                    ;during execution of this instruction and
                    ;points to the consecutive program step.
```

```
......              ;The program continues here if the CPUOff
                    ;bit is reset during the interrupt service
                    ;routine. Otherwise, the PC retains its
                    ;value and the processor returns to LPM0.
```

The following example describes clearing low-power mode 0.

```
;===Interrupt service routine================================= .....
;CPU is active while handling interrupts

BIC     #10h,0(SP)  ;Clears the CPUOff bit in the SR contents
                    ;that were stored on the stack.

RETI                ;RETI restores the CPU to the active state
                    ;because the SR values that are stored on
                    ;the stack were manipulated. This occurs
                    ;because the SR is pushed onto the stack
                    ;upon an interrupt, then restored from the
                    ;stack after the RETI instruction.
```

2.3.2 Low-Power Modes 2 and 3 (LPM2 and LPM3)

Low-power mode 2 or 3 is selected if bits CPUOff and SCG1 in the status register are set. Immediately after the bits are set, CPU, and MCLK operations halt and all internal bus activities stop until an interrupt request or reset occurs.

Peripherals that operate with the MCLK signal are inactive because the clock signal is inactive. Peripherals that operate with the ACLK signal are active or inactive according with the individual control registers and the module enable bits in the SFRs. All I/O port pins and the RAM/registers are unchanged. Wake-up is possible by enabled interrupts coming from active peripherals or RST/NMI.

2.3.3 Low-Power Mode 4 (LPM4)

In low-power mode 4 all activities cease; only the RAM contents, I/O ports, and registers are maintained. Wake-up is only possible by enabled external interrupts.

Before activating LPM4, the software should consider the system conditions during the low-power mode period. The two most important conditions are environmental (that is, temperature effect on the DCO), and the clocked operation conditions.

The environment defines whether the value of the frequency integrator should be held or corrected. A correction should be made when ambient conditions are anticipated to change drastically enough to increase or decrease the system frequency while the device is in LPM4.

2.3.4 Basic Hints for Low-Power Applications

There are some basic practices to follow when current consumption is a critical part of a system application:

- Switch off analog circuitry when possible.
- Switch off the MCLK source for the CPU when not required. Use interrupts to activate the CPU. Program execution starts in less than 6 μs.
- Select the lowest possible operating frequency for the individual peripheral module. Disable unused peripherals.
- Select the weakest drive capability if an LCD is used or switch the drive off.
- Tie all unused inputs to an applicable voltage level.

2.4 Summary

This chapter describes features, interrupt processing and various Low Power Modes (LPMs) of MSP430. For more detail about MSP430F449 refer to [1], Appendix - A and Appendix - B.

Chapter 3

Real Time Operating System for Sensor

This chapter describes the topics those are required to be taken into the consideration, while designing the Kernel for the real-time system. Since, in sensor nodes, RT tasks must meet their respective deadline, scheduling in sensor nodes can be considered as a real time system. Since, sensors are very small in size, memory available to them is also very less. So, we can say that the, operating system in sensors should support real time architecture and should be small in size.

3.1 Critical Sections

All real-time kernels needs to disable interrupts in order to access critical section of code and to re-enable interrupts when done. OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() are the two macros used to disable and enabled the interrupts and these two are always used in pair.

The way to implement OS_ENTER_CRITICAL() is to save the interrupt status on to the stack and then disable interrupts. OS_EXIT_CRITICAL() is simply implemented by restoring the interrupt status from the stack. The pseudo code for these macros

is:

```
#define OS_ENTER_CRITICAL()
    asm(" PUSH SR ")
    asm(" DINT ")

#define OS_EXIT_CRITICAL()
    asm(" POP SR ")
```

The PUSH SR instruction pushes the processor status register (SR) onto the stack. The DINT instruction stands for disable interrupts. Finally, the POP SR instruction is used to restore the original state of the status register (SR) from the stack.

3.2 Task States

At any given time task is in one of the five states as shown in the Figure 3.1.
The **Task Dormant** state correspond to a task that resides in the program space but has been not made available to OS. A task is made available to OS by either calling OSTaskCreate(). This calls simply provides the starting address of your task to OS, the priority you want to assign to the task, the stack limit of the task, the starting address of the stack etc. A task can return to the dormant state by simply calling OSTaskDel().

When multiple task are ready to run only the highest priority task is allowed to run and remaining task are thus kept in a queue of ready to run state. If a task is in a running state and an interrupt occurs, then CPU is given to the ISR and when ISR completes its execution CPU is allocated to the same task or other high priority task depending upon the type of the kernel and action carried out by the ISR.

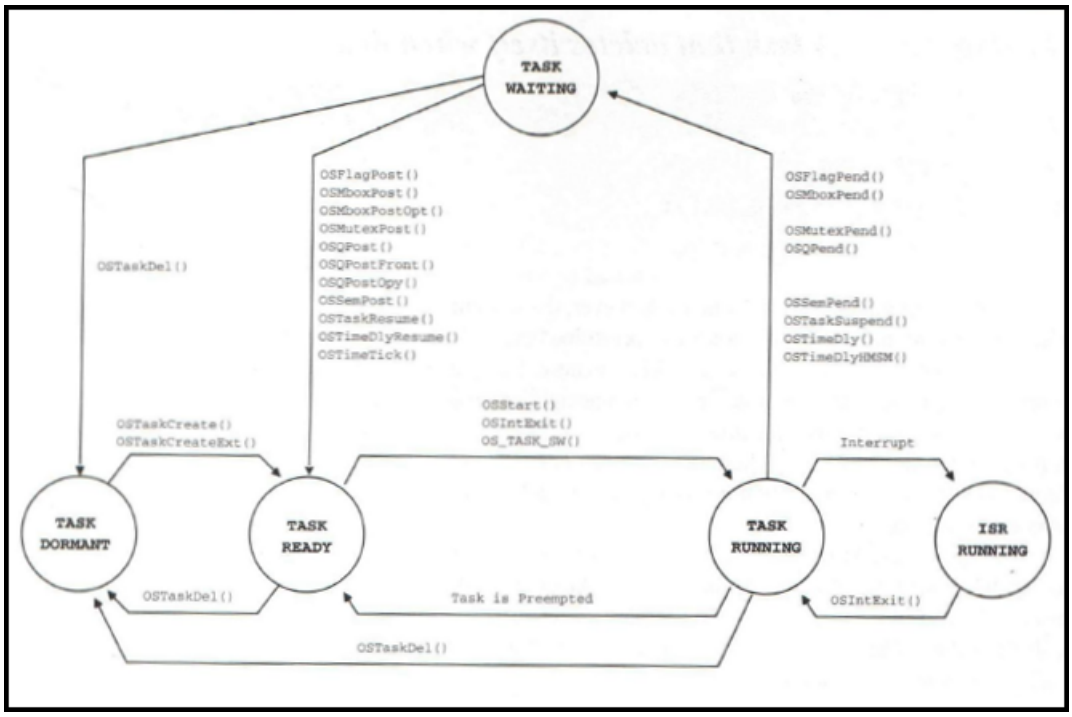

Figure 3.1: Task States

3.3 Task Control Blocks (OS_TCB)

When a task is created, it is assigned a task control block, OS_TCB. A task control block is a data structure that is used by OS to maintain the state of the task when it is preempted. When the task regains the control of the CPU, the TCB allows the task to resume execution exactly where it is left off. All OS_TCB resides in RAM.

```
typedef struct os_tcb{

    OS_STK  *OSTCBStkPtr;
    OS_STK  *OSTCBStkBottom;
    INT32U  OSTCBStkSize;
    INT16U  OSTCBId;

    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
```

```
    INT16U OSTCBDly;
    INT8U OSTCBStat;
    INT8U OSTCBPrio;

    INT8U OSTCBX;
    INT8U OSTCBY;
    INT8U OSTCBBitX;
    INT8U OSTCBBitY;

    #if OS_TASK_DEL_EN > 0
        BOOLEAN  OSTCBDelReq;
    #endif

}OS_TCB;
```

OSTCBStkPtr

contains a pointer to the current top-of-stack for the task.

OSTCBStkBottom

is a pointer to the bottom of the task's stack. If the processor's stack grows from high to low memory locations then OSTCBStkBottom points at the lowest valid memory location for the stack. Similarly, if the processor's stack grows from low to high memory locations, then OSTCBStkBottom points at the highest valid stack address.

OSTCBStkSize

holds the size of the stack in number of elements instead of bytes.

OSTCBId

is used to hold an identifier for the task.

OSTCBNext and OSTCBPrev

are used to doubly link OS_TCBs.

OSTCBDly

is used when a task needs to be delayed for a certain number of clock ticks or the task needs to be pend for an event to occur with a timeout. When this variable is 0, the task is not delayed or has no timeout when waiting for an event.

OSTCBStat

contains the status of the task. When OSTCBStat is OS_STAT_READY, the task is ready to run.

OSTCBPrio

contains the task priority. A high priority task has low OSTCBPrio value.

OSTCBX, OSTCBY, OSTCBBitX and OSTCBBitY

are used to accelerate the process of making a task ready to run or make a task wait for an event. The values for these field are computed when the task is created or when the task priority is changed.

```
.OSTCBY          = priority >> 3;

.OSTCBBitY       = OSMapTbl[priority>>3];

.OSTCBX          = priority & 0X07;

.OSTCBBitX          = OSMapTbl[priority & 0X07];
```

OSTCBDelReq

is a boolean used to indicate weather or not a task has requested that the current task be deleted. This field is present in the OS_TCB only when OS_TASK_DEL_EN in OS_CFG.H is set to 1.

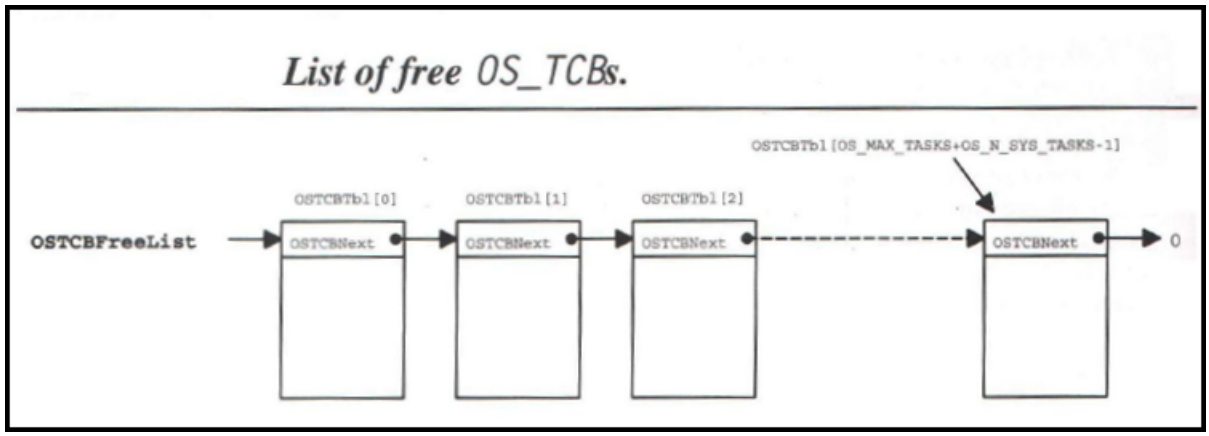

Figure 3.2: List of free OS_TCB

The minimum number of tasks(OS_MAX_TASKS) that an application can have is specified in OS_CFG.H and it is assumed to be 64. It determines the number of OS_TCBs allocated for your application. All OS_TCB are placed in OSTCBTbl[]. When a task is created, the OS_TCB to which OSTCBFreeList points is assigned to the task, and OSTCBFreeList is adjusted to point to the next OS_TCB in the chain. When the task is deleted, its OS_TCB is returned to the list of free OS_TCBs.

3.4 OS_TCBInit()

An OS_TCB is initialized by the function OS_TCBInit(), when a task is created. OS_TCBInit() is called by OSTaskCreate(). OS_TCBInit() receives five arguments.

a. prio is the task prioroty.

b. ptos is a pointer to the top of stack after the stack frame has been build by OSTaskStkInit() and is stored in the .OSTCBStkPtr field of the OS_TCB().

c. pbos is a pointer to the bottom of the stack and is stored in the .OSTCBStk-Bottom field of the OS_TCB.

d. id is the task identifier and is saved in the OSTCBId field.

e. stk_size is the total size of the stack and is saved in the .OSTCBStkSize filed of the OS_TCB.

3.5 Ready List

Each task is assigned a unique priority level between 0 and OS_lowest_PRIO, inclusive. Task priority OS_lowest_PRIO is always assigned to the idle task when OS is initialize. Each task ready to run is placed in a ready list consisting of two variables,OSRdyGrp and OSRdyTbl[]. Task pointers are grouped (eight tasks per group) in OSRdyGrp. Each bit in OSRdyGrp indicates when a task in a group is ready to run. When a task is ready to run, it also sets its corresponding bit in the ready table, OSRdyTbl[]. The relationship between OSRdyGrp and OSRdyTbl[] is shown in the Figure 3.3 and is given by the following rules:

- Bit 0 in OSRdyGrp is 1 when any bit in OSRdyTbl[0] is 1.
- Bit 1 in OSRdyGrp is 1 when any bit in OSRdyTbl[1] is 1.
- Bit 2 in OSRdyGrp is 1 when any bit in OSRdyTbl[2] is 1.

- Bit 3 in OSRdyGrp is 1 when any bit in OSRdyTbl[3] is 1.
- Bit 4 in OSRdyGrp is 1 when any bit in OSRdyTbl[4] is 1.
- Bit 5 in OSRdyGrp is 1 when any bit in OSRdyTbl[5] is 1.
- Bit 6 in OSRdyGrp is 1 when any bit in OSRdyTbl[6] is 1.
- Bit 7 in OSRdyGrp is 1 when any bit in OSRdyTbl[7] is 1.

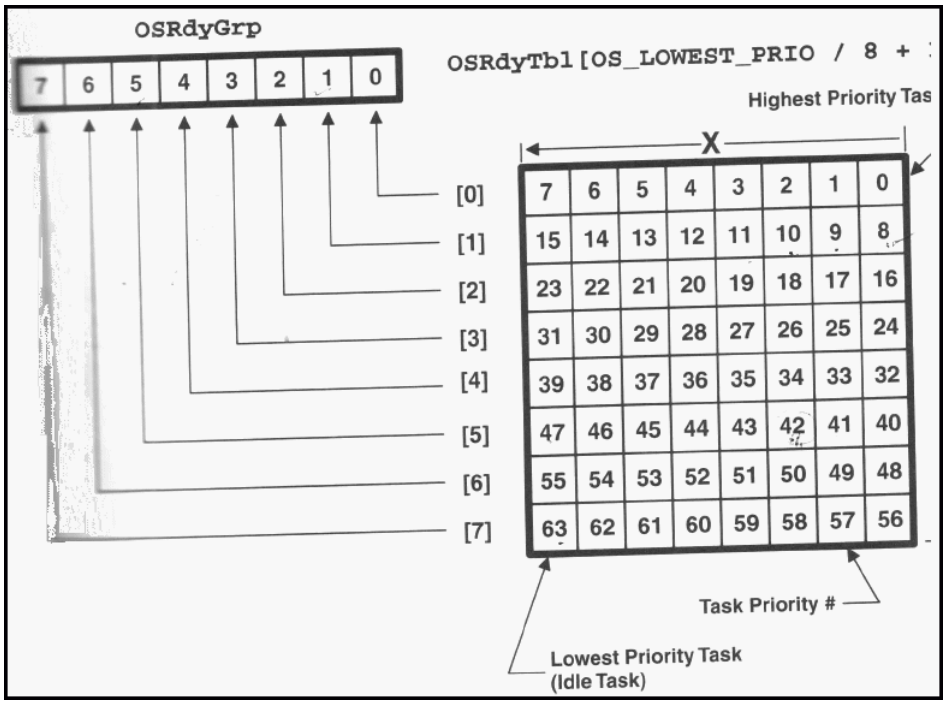

Figure 3.3: Relationship between OSRdyGrp and OSRdyTbl

3.5.1 Making a task ready to run

To determine which priority (and thus which task) will run next, the scheduler in OS determines the lowest priority number that has its bit set in OSRdyTbl[].
The following code is used to place a task in the ready list. prio is the task priority.

```
OSRdyGrp              |= OSMapTbl[prio >> 3];
```

```
OSRdyTbl[prio>>3]    |= OSMapTbl[prio & 0X07];
```

As we can see from the figure, the lower three bits of the task's priority are used to determine the bit position in OSRdyTbl[], and next three most significant bits are used to determine index into OSRdyTbl[]. OSMapTbl[] is in ROM and is used to equate index (0 to 7) to a bit mask. The OSMapTbl[] is as follows:

Table I: OSMapTbl[]

Index	Bit Masks(Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

3.5.2 Removing a task from the ready list

```
if((OSRdyTbl[Prio >>3]  &= ~OSMapTbl[prio & 0X07])==0)
OSRdyGrp                &= ~OSMapTbl[prio>>3];
```

The above code clears the ready bit of the task in the OSRdyTbl[] and clears the bit into the OSRdyGrp only if all the tasks in the group are not ready to run, that is, all bits in the

```
OSRdyTbl[prio>> 3]
```

are 0.

3.5.3 Finding the highest priority task

```
y       = OSUnMapTbl[OSRdyGrp];
x       = OSUnMapTbl[OSRdyTbl[y]];
prio    = (y << 3)+x;
```

The above code finds the highest priority task among all the ready tasks, which are ready to run. The table lookup method is performed speedup the operation. The OSUnMapTbl[256] is a priority resolution table.

```
INT8U  const  OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x00 to 0x0F*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x10 to 0x1F*/

    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x20 to 0x2F*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x30 to 0x3F*/

    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x40 to 0x4F*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x50 to 0x5F*/

    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x60 to 0x6F*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x70 to 0x7F*/

    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x80 to 0x8F*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x90 to 0x9F*/
```

```
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xA0 to 0xAF*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xB0 to 0xBF*/

    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xC0 to 0xCF*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xD0 to 0xDF*/

    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xE0 to 0xEF*/

    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0   /* 0xF0 to 0xFF*/
};
```

For example, if OSRdyGrp contains 01101000(binary) or 0X68, then the table lookup OSUnMapTbl[OSRdyGrp] yields a value of 3, which is coorespond to bit 3 in OSRdyGrp. Note that the bit positions are assumed to start on the right with bit 0 being the right most bit. Similarly, if OSRdyTbl[3] contains 11100100(Binary) or 0XE4, then OSUnMapTbl[OSRdyTbl[3]] returns in a value of 2(int 2). The Task priority prio, is then 26 (i.e., 3 * 8 +2). Getting a pointer to the OS_TCB for the corresponding task is done by indexing into the OSTCBPrioTbl[] using task's priority.

3.6 Task Scheduling

MUF always execute highest priority task ready to run. The determination of which task has the highest priority, thus which task will be next to run, is determined by the scheduler. Task level scheduling is done by OS_Sched(). ISR level scheduling is handled by OSInitExit(). The code for OS_Sched() is as below.

Code for Task Scheduler

```
void OS_Sched(){

    INT8U y;
    OS_ENTER_CRITICAL();
    if((OSIntNesting == 0) && (OSLockNesting == 0)){

        y=OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy=(INT8U)((y << 3)+OSUnMapTbl[OSRdyTbl[y]]);

        if(OSPrioHighRdy != OSPrioCur){

            OSTCBHighRdy=OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASk_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

A context switch consists of saving the processor register on the stack of the task being suspended and restoring the register of the higher priority task from its stack. In a micro-OS, the stack frame for a ready task always looks as if an interrupt has just occurred and all processor register were saved on to it. In other words, all micro-OS has to do to run a ready task is restore all processor register from the task's stack and execute a return from the interrupt. To switch context, I thought to use OS_TASK_SW() to simulate interrupt. Most processor provides a software interrupt of TRAP instruction to accomplish this task. The ISR or trap handler must vector to the assembly language function OSCtxSw(). OSCtxSw() expects to have OSTCBHighRdy point to the OS_TCB of the task to be switch in and to have OSTcbCur point to the OS_TCB of the task being suspended. OS_TASk_SW()

suspends the execution of the current task and start execution of the more important task.

All of the code in OS_Sched() is considered a critical section. Interrupts are disabled to prevent ISR from setting the ready bit of one or more tasks during the process of finding the highest priority task ready to run.

3.7 Task Level Context Switch, OS_TASK_SW()

As discussed in the previous section, after the scheduler has determine that a more important task needs to run, OS_TASK_SW() is called to perform a context switch. The context of a task is generally the contents of all of the CPU registers. The context switch code simple needs to save the register value of the task being preempted and load into the CPU the value of the registers for the task to resume.

Context-switch psuedocode

```
void OSCtxSw(void){

    PUSH R1,R2,R3,R4,...,R15  on to the current stack;
    OSTCBCur->OSTCBStkPtr   = SP;
    OSTCBCur                = OSTCBHighRdy;
    SP                      = OSTCBHighRdy->OSTCBStkPtr;
    POP R15,...,R4,R3,R2 and R1 from the new stack;
    ASM(RETI)
}
```

3.8 Locking and Unlocking the Scheduler

The OSScheLock() function is used to prevent the task reshceduling until its counterpart, OSSchedUnlock(), is called. The task that calls OSScheLock() keeps control

of the CPU even though other higher priority task are ready to run. Interrupts are still recognized and serviced (assuming interrupts are enabled). OSSchedLock() and OSSchedUnlock must be used in pairs. The variable OSLockNesting keeps track of the number of times OSSchedLock() has been called. Nested functions can thus contain critical code that other tasks cannot access. I thought to allows nesting up to 255 levels deep. Scheduling is re-enabled when OSLockNesting is 0. OSSchedLock() and OSSchedUnlock() must be used with caution because it affects response time of the system.

Locking the Scheduler

```
void OSScheLock(void){

    if(OSRunning == TRUE){                                    (1)

        OS_ENTER_CRITICAL();
        if(OSLockNesting < 255){                              (2)
            OSLockNesting++;
        }
        OS_EXIT_CRITICAL();
    }
}
```

(1) It only make sense to lock the scheduler if multitasking has started (i.e. OSStart() was called).
(2) Before incrementing OSLockNesting, we need to make sure that we have not exceeded the allowable number of the nesting levels.

Unlocking the Scheduler

```
void OSSchedUnlock(void){
```

```
    if(OSRunning == TRUE){                                          (1)

        OS_ENTER_CRITICAL();
        if(OSLockNesting > 0){                                      (2)
            OSLockNesting--;                                        (3)
            if((OSLockNesting ==0 ) && (OSInitNesting == 0)){
                OS_EXIT_CRITICAL();
                OS_Sched();                                         (4)
            } else{
                OS_EXIT_CRITICAL();
           }
        }else{
            OS_EXIT_CRITICAL();
        }
    }
}
```

(1) It only make sense to unlock the scheduler only if multitasking has started.
(2) We make sure OSLockNesting is not already 0. If it were, it would be an indication that you called OSSchedUnlock() too many times. In other words, you would not have the same number of OSSchedLock() as OSSchedUnlock().
(3)OSLockNesting is decremented.
(4)We only want to allow the scheduler to execute when all nesting function are complete. OSSchedUnlock() is called from a task because event could have made higher priority task ready to run while scheduling was locked.

3.9 Starting Multitasking

Code for Starting Multitasking

```
void OSStart(void){

    INT8U y;
    INT8U x;

    if(OSRunning == FALSE){

        y             = OSUnMapTbl[OSRdyGrp];
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3)+ x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCbPrioTbl[OSTCBHighRdy];          (1)
        OSTCBCur      = OSTCBHighRdy;
        OSSartHighRdy();                                     (2)
    }
}
```

(1)When called OSStart() finds the OS_TCB() (from the ready list) of the highest priority task that you have created.
(2)Then OSStart() calls OSStartHighRdy(). Basically, OSStartHighRdy()restores the CPU registers by popping them off the task's stack and then executing return from the interrupt instruction, which forces the CPU to execute your task's code. Note that OSStartHighRdy() never returns to OSStart().

3.10 Creating a Task, OSTaskCreate()

To create a task, you required to pass its address and other arguments to OSTaskCreate() function. A task can be created prior to the start of the multitasking or by another task. The code for the OSTaskCreate() is shown below. OSTaskCreate() requires four arguments: *task* is pointer to the task code, *pdata* is a pointer to the

arguments that is passed to the task when it starts executing, *ptos* is a pointer to the top of stack that is assigned to the task and *prio* is the desired task priority.

```
INT8U OSTaskCreate(void(*task)(void *pd), void *pdata, OS_STK
*ptos,INT8U prio){

    void *psp;
    INT8U err;

    #if OS_ARG_CHK_EN > 0
        if(Prio > OS_LOWEST_PRIO){
            return (OS_PRIO_INVALID);
        }
    #endif

    OS_ENTER_CRITICAL();

    if(OSTCBPrioTbl[prio] == (OS_TCB*)0){

        OSTCBPrioTbl[prio] = (OS_TCB*)1;
        OS_EXIT_CRITICAL();
        psp=(void*)OSTaskStkInit(task, pdata, ptos, 0);
        err=OS_TCBInit(prio, psp, (void*)0, 0, 0, (void *)0, 0);
        if(err == OS_NO_ERR){

            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if(OSRunning == TRUE){
                OS_Sched();
```

```
            }
        }else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB*)0
            OS_EXIT_CRITICAL();
        }
        return(err);
        OS_EXIT_CRITICAL();
        return(OS_PRIO_EXIST);
    }
}
```

3.11 Summary

The presented, kernel structures for MicroC/OS-II are described here are the ones that used for current research work. More details about the MicroC/OS-II can found at [2].

Chapter 4

Scheduling Algorithms for Sensor Nodes

This chapter describes, selection of scheduling algorithm for sensor nodes. Since sensor has most of the task real-time, in order to find best algorithm among the RT algorithms. I have considered RM, EDF and MUF and shown that how MUF is better than RM or EDF. Then, I have considered MMUF, which reduces the deficiency of MUF by considering the case in which critical task misses its respective deadline if scheduled with MUF, but if same is scheduled with MMUF critical task never misses its respective deadline. Also, number of context switches are also reduced by MMUF [3].

4.1 Introduction

Sensor-based Adhoc Networks are dynamic in nature and having limited battery power, selection of the scheduling algorithm is a very important task. Scheduling algorithm has to meet respective deadlines for critical task and it should be energy efficient.

Here I have given the comparison of RM, EDF, MUF and MMUF algorithm. MLF

is explained because it is a part of MUF.

4.2 Rate Monotonic Algorithm (RM)

The *Rate Monotonic Algorithm* is a fixed priority scheduling algorithm, which consists of assigning the highest priority to the highest frequency task in the system, and the lowest priority to the lowest frequency tasks. At any time the scheduler chooses to execute the task with the highest priority. By specifying the period and the computational time required by the task, the behavior of the system can categories *apriori*.

One problem with the RM is that the schedulers bound is less than 100%. The *schedulable bound* of a task set is defined as the maximum CPU utilization for which the set of the tasks can be guaranteed to meet their deadlines.The CPU utilization of task P_i is computed as the ratio of worst-case computing time C_i to the period T_i. The total utilization U_n for n tasks is calculated as follows[4]:

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{4.1}$$

Main problem with the RM is that it does not support dynamically changing priority very well, a feature required by the some sensor based control system. For example, a task set with three tasks, P1, P2 and P3 with periods T1=30ms, T2=50ms and T3=100ms has the following fixed priority assignment (from highest to lowest): P1, P2, P3. Suppose period of P1 changes to 75ms. Under the RM algorithm we would require that the priority of each task can be reassigned to the order P2,P1, P3, which violates the condition that the priorities are static.

The problems with RM encourage the use of dynamic priority algorithms.Although many such algorithms exist, I restrict my attention in this thesis to EDF and MLF.

4.3 Earliest-Deadline-First Scheduling Algorithm (EDF)

As the name implies, the *earliest-deadline-first* algorithm uses the deadline of a task as its priority. The task with the earliest deadline has the highest priority, while the task with the latest deadline has the lowest priority. One advantage of this algorithm is that the schedulable bound is 100% for all task sets. Secondly, because priorities are dynamic, the periods of tasks can be changed at any time.

A major problem with the EDF algorithm is that there is no way to guarantee which tasks will fail in a transient overload situation. In many systems, although the average case utilization is less than 100%, it is possible that the worst-case utilization is above 100%, leaving the possibility of one or more tasks failing. In such cases, it is desirable to control which tasks should fail and which one succeeds during such a transient overload. In the RM algorithm, low priority tasks will always be the first to fail. However, no such priority assignment exists with EDF, and thus there is no control of which task fails during a transient overload. As a result, it is possible that a very critical task may fail at the expense of a lesser important task.

4.4 Minimum-Laxity-First Scheduling Algorithm (MLF)

The minimum-laxity-first algorithm assigns a laxity to each task in a system, then selects the task with the minimum laxity to execute next. Laxity is defined as follows:

$$laxity = \text{ deadline} - \text{ current time} - \text{ CPU time needed} \tag{4.2}$$

Laxity is a measure of the flexibility available for scheduling a task. A laxity of tl means that even if the task is delayed by time units, it will still meet its deadline. A

laxity of zero means that the task must begin to execute now or it will risk failing to meet its deadline. The main difference between MLF and EDF is that MLF takes into consideration the execution time of a task, which EDF does not do. Like EDF, MLF has a 100% schedulable bound, but there is no way to control which are guaranteed to execute during a transient overload.

4.5 Maximum-Urgency-First scheduling algorithm (MUF)

MUF algorithm, which allows the control of task failures during transient overload, while maintaining the flexibility of a dynamic scheduler, and 100% schedulable bound for the critical set.The maximum-urgency-first scheduling algorithm is a combination

of fixed and dynamic priority scheduling, also called mixed priority scheduling. With this algorithm, each task is given an urgency. The urgency of a task is defined as a combination of two fixed priorities, and a dynamic priority. One of the fixed priorities, called the criticality, has higher precedence over the dynamic priority. The other fixed priority, which we call user priority, has lower precedence than the dynamic priority. The dynamic priority is inversely proportional to the laxity of a task.

The MUF algorithm consists of two parts. The first part is the assignment of the criticality and user priority, which is done apriori. The second part involves the actions of the MUF scheduler during run-time The steps in assigning the criticality and user priority are the following:

a. As with RM, order the tasks from shortest period to longest period.

b. Define the critical set as the first N tasks such that the total worst-case CPU utilization does not exceed 100%. These will be the tasks that do not fail, even during a transient overload of the system. If a critical task does not fall within

the critical set, then period transformation, as used with RM, can also be used here.

c. Assign high criticality to all tasks in the critical set, and low criticality to all other tasks.

d. Optionally assign a unique user priority to every task in the system.

The static priorities are defined once, and do not change during execution. The dynamic priority of each task is assigned at run-time, inversely proportional to the laxity of the task. Before its cycle, each task must specify its desired start time, deadline time, and worst-case execution time.

Whenever a task is added to the ready queue, a reschedule operation is performed. The MUF scheduler is used to determine which task is to be selected for execution, using the following algorithm:

a. Select the task with the highest criticalness.

b. If two or more tasks share highest criticalness, then select the task with the highest dynamic priority (i.e. minimum laxity). Only tasks with pending deadlines have a non-zero dynamic priority. Tasks with no deadlines have a dynamic priority of zero.

c. If two or more tasks share highest criticalness, and have equal dynamic priority, then the task among them with the highest user priority is selected.

d. If there are still two or more tasks that share highest criticalness, dynamic priority, and highest user priority, then they are serviced in a first-come-firstserve manner.

The optional assignment of unique user priorities for each task ensures that the scheduler never reaches step d, thus providing a deterministic scheduling algorithm.

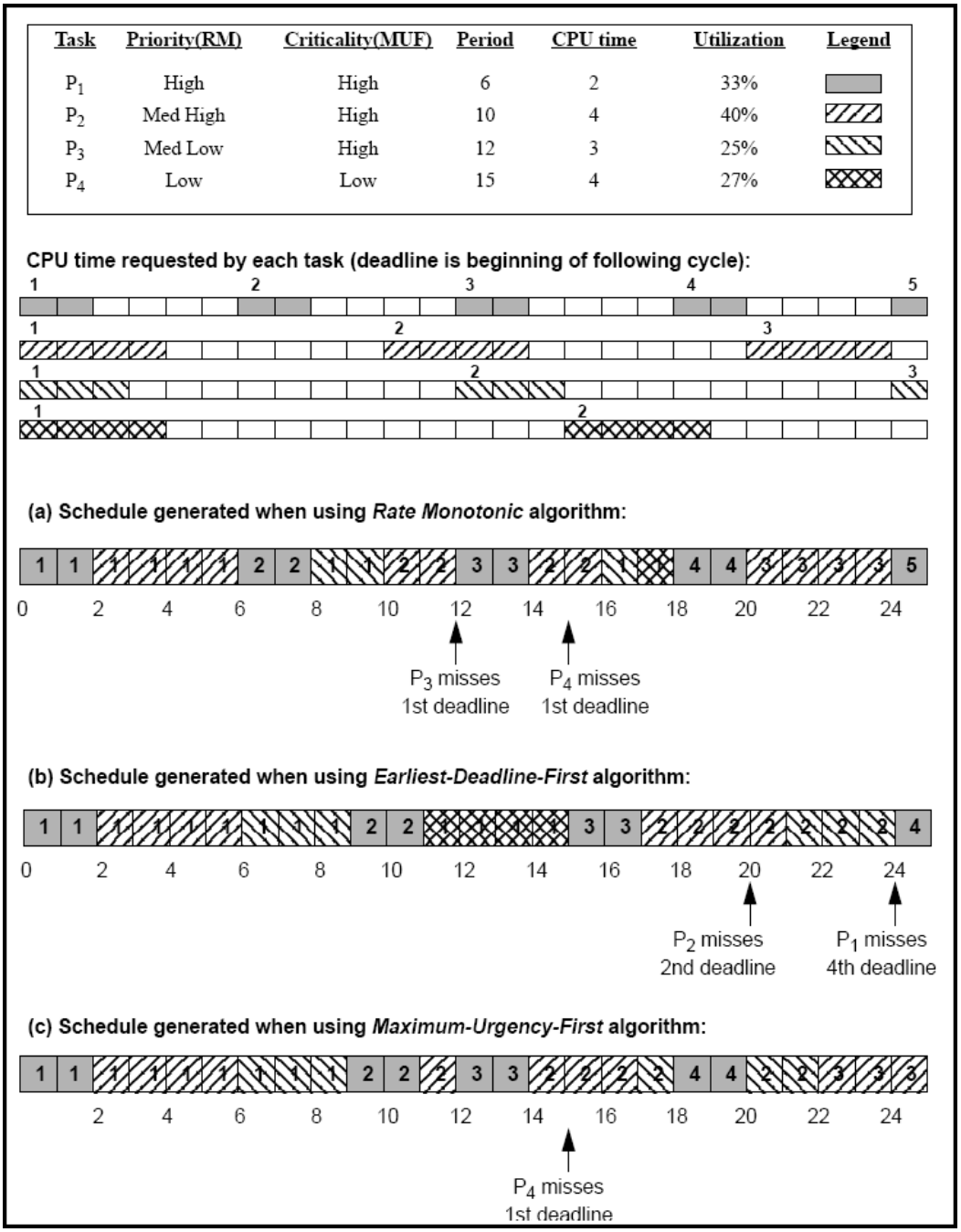

Figure 4.1: Example comparing RM, EDF, and MUF algorithms

To demonstrate the advantage of MUF over RM and EDF, consider the task set shown in Figure 4.1. We assume that the deadline of each task is the beginning of the next cycle. Four tasks are defined, with a total worst-case utilization of over 100%, thus in the worst-case, missed deadlines are inevitable.Figure 4.1(a) shows the schedule produced by a static priority scheduler when priorities are assigned using the RM algorithm. In this case, only P1 and P2 are in the critical set, and are guaranteed not to miss deadlines. Expectably, both P3 and P4 miss their deadlines. When using the EDF algorithm, as in Figure 4.1(b), tasks P1 and P2 fail. However, any task may have failed, since with EDF there is no way to predict the failure of tasks during a transient overload of the system. With the MUF algorithm, all tasks in the critical set are guaranteed not to miss deadlines. In our example, the combined worst-case utilization of P1, P2, and P3 is less than 100%, and thus they form the critical set. Only task P4 can miss deadlines, because it is not in the critical set. Figure 4.1(c) shows the schedule produced by the MUF scheduler. Note the improvement over RM: because of a higher schedulable bound for the critical set, task P3 is also in the critical set and thus does not miss any deadlines. Also, unlike EDF, we are able to control that the only task that may fail is P4.

The choice of using MLF to calculate the dynamic priority instead of EDF enables the scheduler to detect missed deadlines. There are three failures which the MUF scheduler can detect:

a. A task has not completed its cycle when the deadline time has been reached;

b. A task was given as much CPU time as was requested in the worst-case, yet it still did not meet its deadline;

c. The task will not meet its deadline because the minimum CPU time requested cannot be granted. This case also requires that the minimum amount of CPU time required by a task is specified.

The first case is the standard notion of a missed deadline. The second case will detect bad worst-case estimates of execution time. The third case allows the MUF scheduler to make the most of its CPU time, and it will not start executing a task if that task has no possibility to finish before its deadline, thus providing the early detection of missed deadlines. Instead, the CPU time can be reclaimed for ensuring that other tasks do not miss deadlines, or to call alternate, shorter threads of execution.

The implementation of the urgency value for the MUF is shown in the Figure 4.2

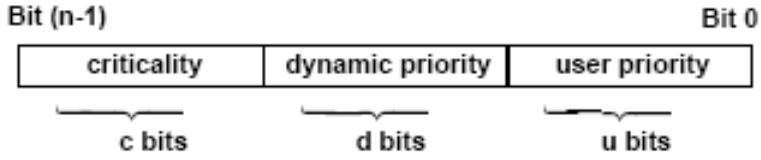

Figure 4.2: Scheme to encode n-bit urgency value for MUF

4.6 Modified Maximum-Urgency-First scheduling algorithm (MMUF)

Although MUF is an efficient algorithm, it has a major disadvantage. Since the rescheduling operation is performed whenever a task is arrived to the ready queue, there is the possibility of failing a critical task in certain situations. In these situations, a task with minimum laxity may be selected whose remaining execution time is greater than remaining time to another tasks laxity. This problem is due to performing the rescheduling operation whenever a new task is added to the ready queue. The scheduling should be performed at any given instant and the scheduler will choose the highest priority task to run among all available tasks. Therefore, the schedule should be produced in such a way that the task having the highest priority always be running.

Consider two tasks, T1 and T2, shown in Table I. Figure 4.3 shows the schedule

which is produced by the MUF algorithm for the task set in Table I.

Table I: Example of Task Set

	Remaining Execution Time	Deadline	Remaining Time to Laxity
T1	4	6	2
T2	1	4	3

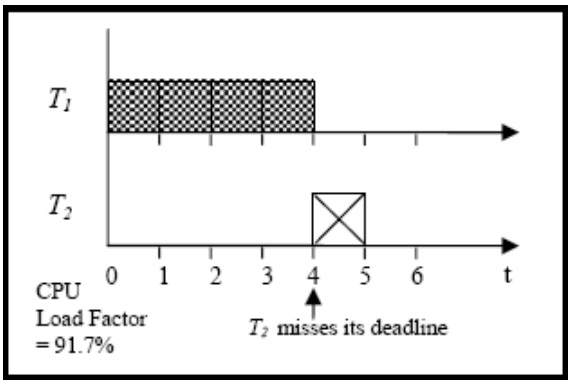

Figure 4.3: Schedule generated by the MUF scheduling algorithm

As it is shown in Figure 4.3 the MUF will select the task with minimum laxity (T1) at time zero. The remaining execution time of task T1 is greater than remaining time to T2's laxity. This selection will cause task T2 to miss its deadline.

The modified maximum urgency first algorithm we propose in this paper is a modified version of MUF algorithm which resolves the mentioned MUF algorithms deficiency. In addition it has some extra advantages which will be explained later

The modifications are as follows: With this algorithm, we use a unique importance parameter, instead of using tasks request intervals, to create the critical set. The importance parameter is a fixed priority which can be defined as user priority or any other optional parameter which expresses the degree of the tasks criticalness. It is trivial that the task with the shortest request period is not necessarily the most important one. Furthermore, using the importance parameter, it is not needed to use

period transformation, as it is done in MUF algorithm. With the MMUF algorithm, either EDF or MLLF can be used to define the dynamic priority. Another optimization made in this algorithm is the elimination of unnecessary context switches which in turn reduced the overall system overhead. This is done firstly by using MLLF instead of LLF and secondly, by letting the currently running task to keep running while there are some other tasks with the same priority.

The MMUF algorithm consists of two phases with the following details:

Phase 1: In this phase fixed priorities are defined only once as follows. These fixed priorities will not change during execution time.

a. Order the tasks from the most importance to the least importance

b. Add the first N tasks to the critical set such that the total CPU load factor does not exceed 100%

Phase 2: This phase calculates the dynamic priorities at every scheduling event and selects the task to be executed next.

a. If there is at least one critical task in the ready queue

   (1) Select the *critical* task with the earliest deadline (EDF algorithm) if there is no tie

   (2) If there are two or more *critical* tasks with the same earliest deadline

      i. If any of these *critical* tasks is already running select it to continue running

      ii. Otherwise, select the *critical* task with the highest importance

b. If there is no critical task in the ready queue

   (1) Select the task with earliest deadline (EDF algorithm) if there is no tie

(2) If there are two or more tasks with the same earliest deadline

i. If any of these tasks is already running select it to continue running

ii. Otherwise, select the task with the highest importance

4.7 Summary

The presented, modified version of MUF scheduling algorithm called MMUF which resolves the deficiency of the MUF algorithm in which a critical task may miss its deadline in certain situations. Moreover, some additional optimizations are applied in the MMUF algorithm. The performance of the MMUF was compared to MUF algorithm and showed to be superior [3]. It usually has less task preemption and hence, less related overhead. It also leads to less failed non-critical tasks in overloaded situations in which the CPU load factor is greater than 100% [3].

Chapter 5

Problem Definition and Existing Methodologies

In the previous chapters, we have seen various ways to save power for sensor nodes from hardware point of view, kernel point of view and scheduling algorithm point of view.

5.1 Problem Definition

In previous chapters (**Chapter 2**, **Chapter 3** and **Chapter 4**), we have seen the various ways to save the power in the sensor nodes. By switching MSP430 into the various Low Power Modes (LPMs) can reduce a huge amount of the power consumption. MicroC/OS-II is a real-time operating system, which can freely available for education purpose and it has various configuration modes to be set by the user as per the application requirements, in order to reduce the size and load time for the Kernel. MMUF - the real time scheduling algorithm can reduces the amount of context switching between the task and meets the respective deadline of the critical task. In this work, an effort is being made in the area to put all these things together to get the best result. *The intention is to improve the power consumption of the sensor nodes, so that the total power consumption by each sensor node can be reduced to extend*

their lives. Also, the intention is to extend MMUF algorithm to support Aperiodic tasks.

5.2 Existing Methodologies

5.2.1 Method One

The first method, relates to save the power from the hardware point of view i.e. select the lowest power consumption micro-controller unit in the design of the sensor nodes. The chosen micro-controller, MSP430, is a world's lowest power consumption micro-controller[5]. Details about its architecture, various low power modes that it can support and guidelines for the low power application can be found in (**Chapter 2**, **Appendix A**, **Appendix B**, [1], [5] and [6].

5.2.2 Method Two

The real-time operating system which can be configured as per the requirement and must be small in size. Due to reduction in the size of the RTOS, the power required to load the RTOS is also reduced. Some details about the MicroC/OS-II and its kernel structures is explained in **Chapter 3**. More details about the MicroC/OS-II can be found at [2] and [7].

5.2.3 Method Three

The real-time scheduling algorithm, which schedules the critical tasks in a way such that critical tasks can meet their respective deadline. Also, it must reduce the context switching between the task in order to meet the desired goal. By reducing the context switching, CPU utilization and Memory/Stack read/write can be reduce, which in turn save the power consumption done by scheduler. The brief about real-time scheduling algorithms is explained in **Chapter 4** and more details can be found at [3],[4] and [8].

5.3 Summary

This chapter presented work already done to solve the problem described in section 5.1. Methods described in section 5.2, are the various innovations done in the different direction to save the power consumption. Method 5.2.1, saves the power from the hardware point of view. Method 5.2.2,saves the power from the Operating System point of view and Method 5.2.3 saves the power from the scheduler point of view.

Chapter 6

The Proposed Algorithm

The proposed algorithm Algorithm 6.1, combines the idea presented in 5.2.1, 5.2.2 and 5.2.3. Thus, the proposed algorithm is more efficient than the any of the methods parented in section 5.2. The proposed algorithm is as follows:

6.1 The Algorithm

Algorithm 6.1 The Proposed Algorithm

1 Initialize Operating System which does all necessary initialization and creates the idle task with the lowest priority

2 Create Startup task with assigned user priority

3 Start Multitasking

4 Execute the code for Startup task

5 When new task creation function is called with the user priority, startup time, worst case execution time, periodicity for periodic task and Maximum Interrupt latency for Aperiodic task then

6 if new task is Aperiodic then convert it to periodic by setting periodicity = Maximum Interrupt latency (Periodicity) + CPU Time Needed

7 insert new task into the task linked list and sort task list according to user priority

8 create a task list ***Critical*** of first N tasks such that total CPU utilization is $\leq$

100%. Set criticality bit of all the tasks which falls into task list ***Critical*** and reset criticality bit for all other tasks which falls into task list ***Non-Critical***.

9 Rearrange the task list ***Critical*** and ***Non-Critical*** by sorting them according to earliest deadline.

10 Delay the newly created task until OSTime = startup time.

11 if new task is aperiodic then set its status to Ready + Suspend.

12 Execute the scheduler to select highest priority task.

13 If highest priority task = Idle task then

14 switch micro-controller to Low Power Mode 3.

15 else

16 if micro-controller in Low Power Mode 3 then

17 exit from the Low Power Mode 3

18 execute the highest priority task.

repeat steps 12-18 with each tick interrupt

repeat steps 5-18 when new task creation function is called

The flow chart for above algorithm is shown in Figure 6.1.

6.2 Summary

This chapter describes, the algorithm and its flow chart to implement power aware scheduling on MicroC/OS-II using MSP430 which can support both periodic and aperiodic tasks. Also, this algorithm takes care of switching MSP430 into LPM3 when only idle task is active.

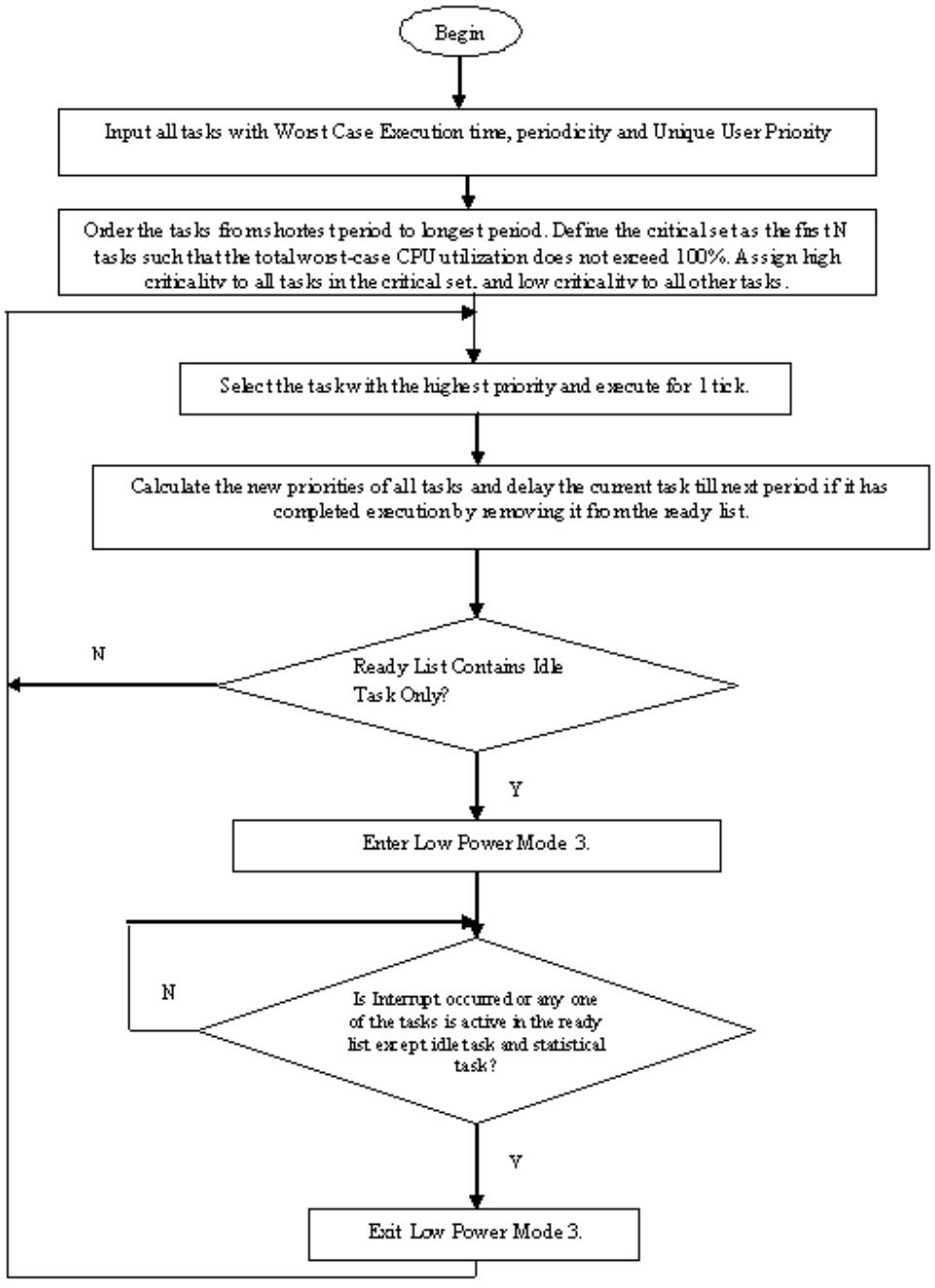

Figure 6.1: Implementation of MMUF on MicroC/OS-II using MSP430

Chapter 7

Implementation

To implement MMUF on MicroC/OS-II and MSP430, first it is required to port MicroC/OS-II on MSP430. For this, first install "MSP430 IAR Embedded Workbench IDE" as per steps found in [9]. Download MicroC/OS-II from [7] and modify OS_CPU.H, OS_CPU_A.S43 and OS_CPU.C to port it on MSP430 [2]. Create necessary data structures to implement MMUF and do necessary modification in MicroC/OS-II to support power aware scheduling using MSP430.

Assumption: Aperiodic Tasks always assigned higher priority by the user over periodic tasks.

7.1 Implementation Environment

Regarding how to install and use "MSP430 IAR Embedded Workbench IDE" will be found from [9]. To port MicroC/OS-II on "MSP430 IAR Embedded Workbench IDE" requires to modify three kernel files listed above. OS_CPU.H contains the function prototypes and data types definition. OS_CPU_A.S43 contains the kernel functions in assembly routine for the faster execution. OS_CPU.C contains the hook function to extend the functionality of MicroC/OS-II in C syntax.

7.1.1 Porting of MicroC/OS-II on MSP430 IAR Embedded Workbench IDE

OS_CPU.H contains the definition for OS_ENTER_CRITICAL and OS_EXIT_CRITICAL functions. OS_CPU_A.S43 contains the function/Macro in assembly routine as listed in Table I.

Table I: Functions contained in OS_CPU_A.S43

Macro/Function Name	Purpose
PUSHALL	Save all registers
POPALL	Restores all registers
OSStartHighRdy	Starts highest priority task which is ready to run
OSCtxSw	Task level context switch
OSIntCtxSw	Interrupt level context switch
WDT_ISR	Watch dog interrupt level service routine
USARTX0_ISR	USART0 Transmit Interrupt Service Routine
USARTR0_ISR	USART0 Receive Interrupt Service Routine

OS_CPU.C contains the hook functions written in C syntax as listed in Table II

Table II: Functions contained in OS_CPU.C

Function Name	Purpose
OSTaskStkInit	To initialization of the task stack
OSTaskIdleHook	To switch CPU in Low Power Mode when only Idle task is running
OSTCBInitHook	To initialize OSTCB with user specified data
OSTimeTickHook	To extend the functionality of OSTimeTick

7.1.2 Data structures used to implement MMUF

To implement MMUF on MicroC/OS-II using MSP430, two files are needed MMUF.H and MMUF.C. These two files are taken into the consideration during compilation of the application only when **OS_MUF_EN is set** into OS configuration file OS_CFG.H Otherwise the MicroC/OS-II can run its default scheduler which comes with it.

MMUF.H contains the data structure mmuf_data of type MMUF_DATA which is described below.

```
typedef struct mmuf_data {
  INT16U StartTime;
  INT16U Period;
  INT16U CPUTimeNeeded;
  INT16U CPUTimeUsed;
  BOOLEAN Critical;
  INT16U  Cntr;
  INT16U  OriginalPrio;
  BOOLEAN IsPeriodic;
  INT8U   *TaskName;
}MMUF_DATA;
```

The purpose of each field is listed in Table III It also contains the signature of the

Table III: Description about MMUF_DATA

Field Name	Purpose
StartTime	Specifies the start time of the task
Period	Specifies the periodicity for the periodic task For aperiodic task period refers to Maximum Interrupt Latency for that task and is set by period = period + CPUTimeNeeded
CPUTimeNeeded	Specifies the worst case execution time of the task
CPUTimeUsed	Specifies the CPU Time Used by the task and it is always reset to zero when task completes its execution.
Critical	Set by the OS and specifies criticality of the task.
Cntr	Set by the OS and indicates how many times task has completed its execution
OriginalPrio	Priority of the task specified by the User when submitted to OS
IsPeriodic	Specified by the user; indicates that task is periodic or Aperiodic.
TaskName	Contains Name of the task given by the user.

functions listed in Table IV and its definitions contained in MMUF.C

Table IV: Function contained in MMUF.C

Function Signature	Purpose
void Increment_Time(void)	To increment time of the task
void SetCritical (void)	To set criticality of the task
void SET_APERIODIC_TASK_EN(void)	To enable Aperiodic tasks
void SetEDF (void)	To sort the task according to EDF
void MUFPrio (void)	To change the state of the task from ready to delay/ready + Suspend when task completes its execution.

MMUF.H also contains defined constant named OS_CRITICAL, OS_NON_CRITICAL, OS_A_PERIODIC and OS_PERIODIC which indicates task is critical or non-critical and periodic or aperiodic.

If user wants that MMUF can support Aperiodic tasks then he/she has to set MAX_APERIODIC_TASK_PRIO to some value say n then maximum n Aperiodic tasks, user can create with their priorities lies in the interval [0,n). Also, user has to specify MAX_INTERRUPT_LATENCY for Aperiodic tasks by setting some value to Period field. Aperiodic Tasks can be converted to periodic tasks by using the formula referred in 7.1

$$Period = \text{Period} + \text{CPUTimeNeeded} \tag{7.1}$$

MAX_APERIODIC_TASK_PRIO defined constant can be found in Application Configuration file APP_CFG.H.

7.2 Results

Various simulation runs were conducted for different set of parameters as mentioned in the following cases. Here, performance is tested with a task list containing six different tasks for total 40 ticks.

7.2.1 Case 1: All tasks are periodic

Tasks are created with the input parameters mentioned in Figure 7.1 and schedule generated by MMUF is shown in Figure 7.2. At T = 40, status of the MicroC/OS-II is shown in Figure 7.3 and task list status is shown in Figure 7.4.

The first frame in the Figure 7.3 shows the green emoticon indicating that MicroC/OS-II is running. When it is not running then it shows the emoticon in the red color. In the same frame at the right most side, V2.86 is shown, which is the version number of the MicroC/OS-II. In the same figure the second frame named *Statistics:Ready*, is shows the statistics of the OS in terms of CPU Usage, Number of Tasks, Idle Counter and number of context switching done up to the ticks mentioned in the right most bottom frame. The frame below it named *Timers*, shows the statistics about the number of timers used, idle and timer time. MicroC/OS-II, V2.86, allows maximum 16 timers. Timers are used to periodically call back a function when time specified in the timer expires [2]. The frame named *Nesting*, contains the information about the interrupt and multi-task lock. The interrupt indicates the number of interrupts received but yet not serviced [2]. The multi-task lock when set to 1, the scheduler is locked and when reset it indicates that scheduler is unlocked. Whenever scheduler is locked, the task which is in execution, continues to execute even though high priority task becomes ready [2].

Figure 7.4 contains the various columns which indicates the status of the task at T=40. 7.4,

First Column the one of the task names is pointed by a sign >, which indicates the task selected by the scheduler at T = 40.

Name specifies the name of the task assigned by the user or the kernel it self.

Ref The order in which the task is created and internally used by the kernel.

Prio Priority of the task at time T = 40.

Task Name	User Priority	Periodicity	Worst Case Execution Time	Is Periodic?	CPU Usage (in %)
Start Up	1	11	4	Yes	36
Task 2	2	15	1	Yes	7
Task 3	0	15	2	Yes	13
Task 4	3	11	1	Yes	9
Task 5	4	18	2	Yes	11
Task 6	5	16	1	Yes	6
Total CPU Usage (in %):					82

Figure 7.1: Simulation Input: All Six Tasks are Periodic

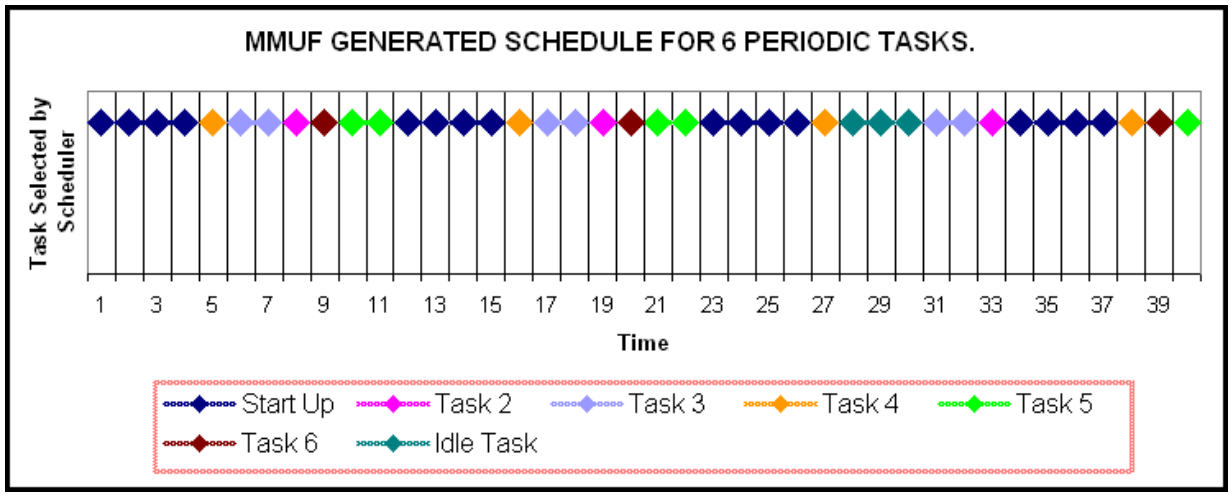

Figure 7.2: Schedule Generated by MMUF : All Six Tasks are Periodic

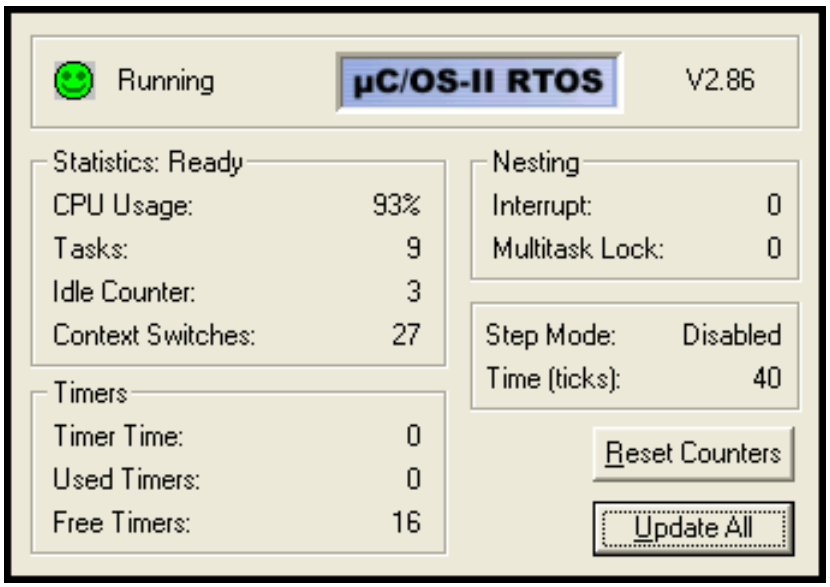

Figure 7.3: Status of the MicroC/OS-II : All Six Tasks are Periodic

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Up	3	0	Dly	4			4	1142	0%	0%	0	0	128	0000	1102
uC/OS-II Idle	0	63	Ready	0			3	1F14	19%	18%	50	48	256	1F44	1E44
uC/OS-II Stat	1	62	Ready	0			3	1E0C	21%	20%	54	52	256	1E40	1D40
uC/OS-II Tmr	2	59	Sem	0	OS-TmrSig		1	2BBA	24%	23%	62	60	256	2BF6	2AF6
Task 2	4	3	Dly	5			3	1342	0%	0%	0	0	512	0000	1182
Task 3	5	2	Dly	5			3	1542	0%	0%	0	0	512	0000	1382
Task 4	6	1	Dly	4			4	1742	0%	0%	0	0	512	0000	1582
Task 5	7	5	Ready	0			3	1942	0%	0%	0	0	512	0000	1782
Task 6	8	4	Dly	8			3	1B42	0%	0%	0	0	512	0000	1982

Figure 7.4: Task list status at T = 40 : All are periodic

State State of the task at time T = 40. *Dly* indicates delayed task, *Ready* indicates ready task, *Sem* indicates a task waiting on semaphore.

Dly Amount of time (in ticks) the task has been delayed (if the State column indicates 'Dly') or, the amount of time left that the task will be waiting for either the semaphore, the mutex, the event flag group, the mailbox or the queue (if the State column indicates an object type). The value is 0 if the task will wait forever for one of the objects. (.OSTCBDly).

Waiting On Name of the object (if any) for which the task is waiting. This can be either an Event Flag Group or an Event (Semaphore, Mutex, Mailbox, or Queue).

Msg Message sent by a Task to other task.

Ctx Sw Number of times the task was 'switched-in'. This counter can be reset to 0 by selecting Reset Counters from the context menu, or by clicking the Reset Counters button in the Status window. This counter is only available if you set the configuration constant OS_TASK_PROFILE_EN to 1 which should be done when you are using the kernel awareness feature of C/OS-II. (.OSTCBCtxSwCtr).

Stk Ptr Current value of the task's stack pointer (in hexadecimal)..

Max% Maximum stack space used by the task expressed as a percentage. For example, a value of 47% means that, during execution of the task, the total stack space used never exceeded 47%. This value is reset to 0 by the Reset StkUsed feature of the context menu..

Cur% Current stack usage of the task expressed as a percentage. For example, a value of 39% means that the stack pointer is currently located 39

Max Maximum stack space used by the task (in bytes). This value is reset to 0 by the Reset StkUsed feature of the context menu. (.OSTCBStkUsed)

Cur Current stack usage of the task (in bytes).

Size Number of bytes allocated for the task stack.

Starts @ Address of the beginning of the stack. If the stack, on the processor you are using, grows downwards (i.e. OS_STK_GROWTH set to 1 in OS_CPU.H) then this indicates the highest address that the stack pointer can take, otherwise (i.e. OS_STK_GROWTH set to 0), this indicates the lowest address the stack pointer can take. (.OSTCBStkBase)

ends @ Address of the end of the stack. If the stack, on the processor you are using, grows downwards (i.e. OS_STK_GROWTH set to 1 in OS_CPU.H) then this indicates the lowest address that the stack pointer can take, otherwise (i.e. OS_STK_GROWTH set to 0), this indicates the highest address the stack pointer can take.

For more detail about the Micrium, Inc. MicroC/OS-II Kernel Awareness for C-SPY refer to [10]

7.2.2 Case 2: All tasks are periodic except one. Aperiodic task never becomes ready.

Tasks are created with the input parameters mentioned in Figure 7.5 and schedule generated by MMUF is shown in Figure 7.6. At T = 40, status of the MicroC/OS-II is shown in Figure 7.7 and task list status is shown in Figure 7.8. In this simulation Task 3 is aperiodic. It is created and remains in the ***Ready + Suspend*** state until it receives UART0TX (UART0 Transmit) Interrupt.

Task Name	User Priority	Periodicity	Worst Case Execution Time	Is Periodic?	CPU Usage (in %)
Start Up	1	11	4	Yes	36
Task 2	2	15	1	Yes	7
Task 3	0	15	2	No	13
Task 4	3	11	1	Yes	9
Task 5	4	18	2	Yes	11
Task 6	5	16	1	Yes	6
Total CPU Usage (in %):					82

Figure 7.5: Simulation Input: Five tasks are Periodic and one is Aperiodic

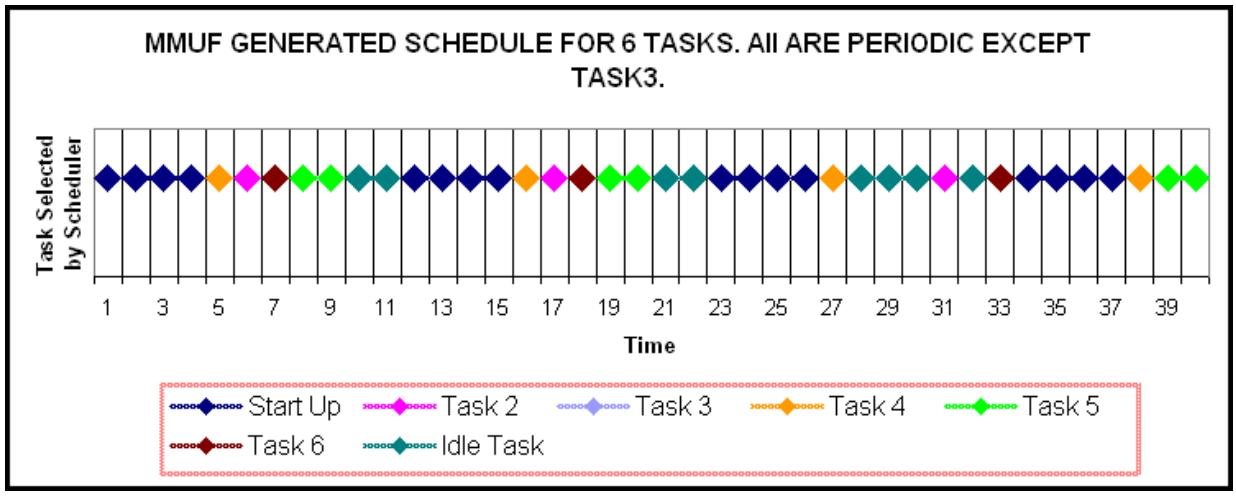

Figure 7.6: Schedule Generated by MMUF : Five tasks are Periodic and one is Aperiodic

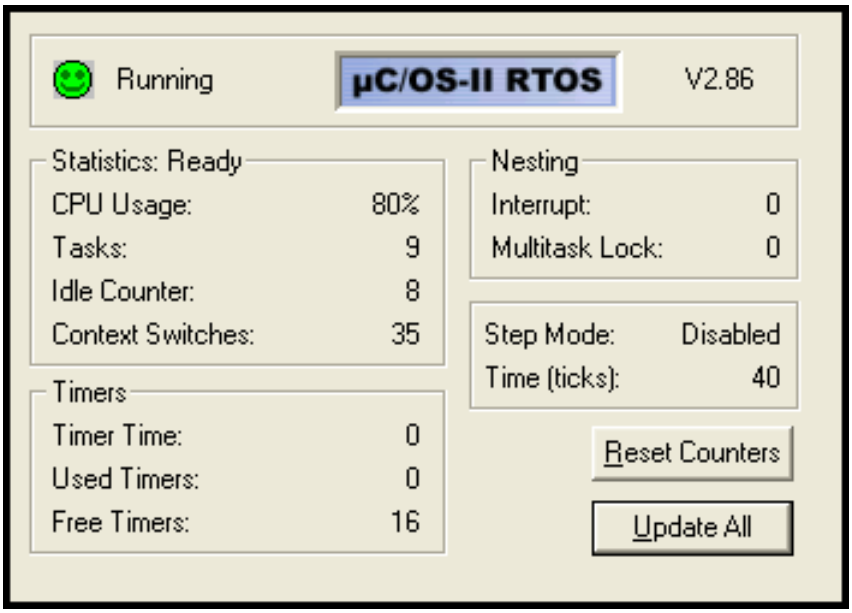

Figure 7.7: Status of the MicroC/OS-II : Five tasks are Periodic and one is Aperiodic

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Up	3	1	Dly	3			3	1142	0%	0%	0	0	128	0000	1102
› uC/OS-II Idle	0	63	Ready	0			9	1F14	19%	18%	50	48	256	1F44	1E44
uC/OS-II Stat	1	62	Ready	0			9	1E0C	21%	20%	54	52	256	1E40	1D40
uC/OS-II Tmr	2	59	Sem	0	OS-TmrSig		1	2BBA	24%	23%	62	60	256	2BF6	2AF6
Task 2	4	3	Dly	4			3	1342	0%	0%	0	0	512	0000	1182
Task 3	5	0	Ready+Suspended	0			0	1562	0%	0%	0	0	512	0000	1382
Task 4	6	2	Dly	3			4	1742	0%	0%	0	0	512	0000	1582
Task 5	7	5	Dly	13			3	1942	0%	0%	0	0	512	0000	1782
Task 6	8	4	Dly	7			3	1B42	0%	0%	0	0	512	0000	1982

Figure 7.8: Task list status at T = 40 : Five tasks are Periodic and one is Aperiodic

7.2.3 Case 3: All tasks are periodic except one. Aperiodic task becomes ready once only.

Tasks are created with the input parameters mentioned in Figure 7.5 and schedule generated by MMUF is shown in Figure 7.9. At T = 40, status of the MicroC/OS-II is shown in Figure 7.10 and task list status is shown in Figure 7.11. In this simulation Task 3 is aperiodic. It is created and remains in the ***Ready + Suspend*** state until it receives UART0TX (UART0 Transmit) Interrupt. At T = 2, MSP430 generates interrupt which makes Task 3 ready. So, at T =3, a context switching occurs, and Task 3 is selected by the scheduler since it is the highest priority task among all the ready tasks.

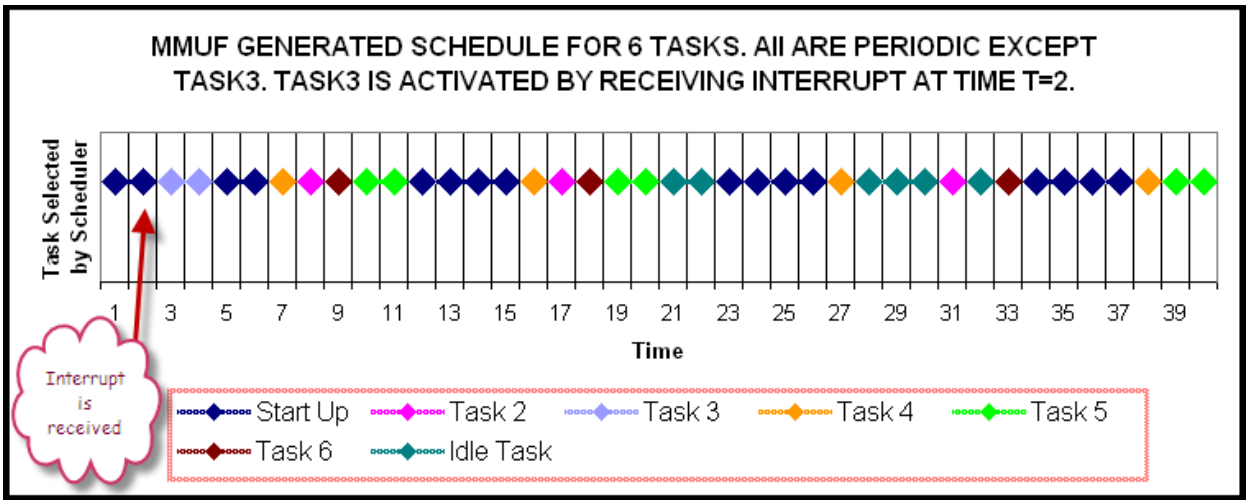

Figure 7.9: Schedule Generated by MMUF : Five tasks are Periodic and one is Aperiodic

7.2.4 Case 4: New task will be added to the task list on the reception of the interrupt.

Tasks are created with the input parameters mentioned in Figure 7.12 and schedule generated by MMUF is shown in Figure 7.13. At T = 40, status of the MicroC/OS-II is shown in Figure 7.14 and task list status is shown in Figure 7.15. In this simulation

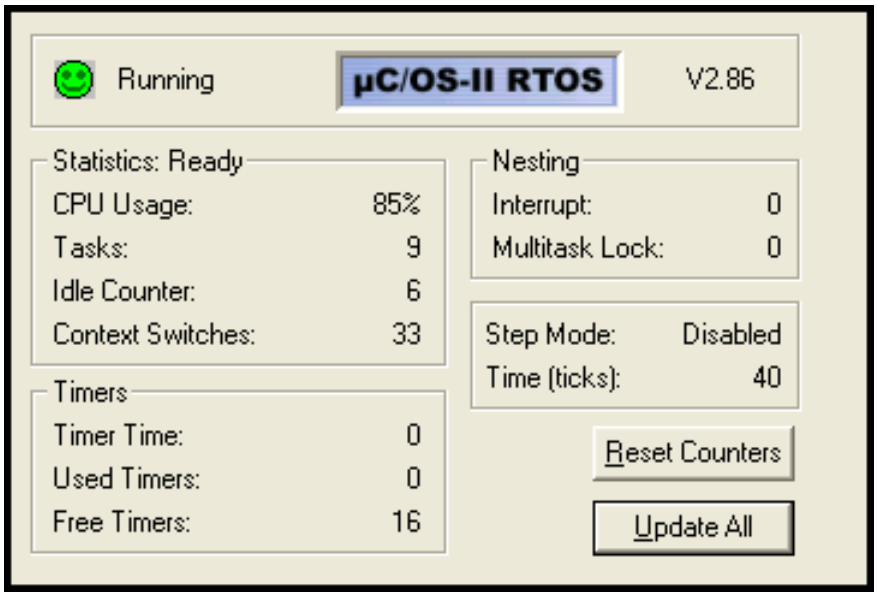

Figure 7.10: Status of the MicroC/OS-II : Five tasks are Periodic and one is Aperiodic

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Up	3	1	Dly	3			4	1142	0%	0%	0	0	128	0000	1102
› uC/OS-II Idle	0	63	Ready	0			7	1F14	19%	18%	50	48	256	1F44	1E44
uC/OS-II Stat	1	62	Ready	0			7	1E0C	21%	20%	54	52	256	1E40	1D40
uC/OS-II Tmr	2	59	Sem	0	OS-TmrSig		1	2BBA	24%	23%	62	60	256	2BF6	2AF6
Task 2	4	3	Dly	4			3	1342	0%	0%	0	0	512	0000	1182
Task 3	5	0	Ready+Suspended	0			1	153E	0%	0%	0	0	512	0000	1382
Task 4	6	2	Dly	3			4	1742	0%	0%	0	0	512	0000	1582
Task 5	7	5	Dly	13			3	1942	0%	0%	0	0	512	0000	1782
Task 6	8	4	Dly	7			3	1B42	0%	0%	0	0	512	0000	1982

Figure 7.11: Task list status at T = 40 : Five tasks are Periodic and one is Aperiodic

Task 3 is aperiodic. It is created and remains in the ***Ready + Suspend*** state until it receives UART0TX (UART0 Transmit) Interrupt. Here, *Task 2* is created on the reception of the UART0RX (UART0 Receive) Interrupt if it is not created before.

Task Name	User Priority	Periodicity	Worst Case Execution Time	Is Periodic?	CPU Usage (in %)
Start Up	1	6	3	Yes	50
Task 2	2	4	2	Yes	50
Task 3	0	17	1	No	6
Task 4	3	5	1	Yes	20
Total CPU Usage (in %):					126

Task2 is created on the reception of the USAR0RX interrupt if it is not created.

Figure 7.12: Simulation Input: Three tasks are Periodic and one is Aperiodic. Among periodic, one task - Task 2, is created with the reception of interrupt.

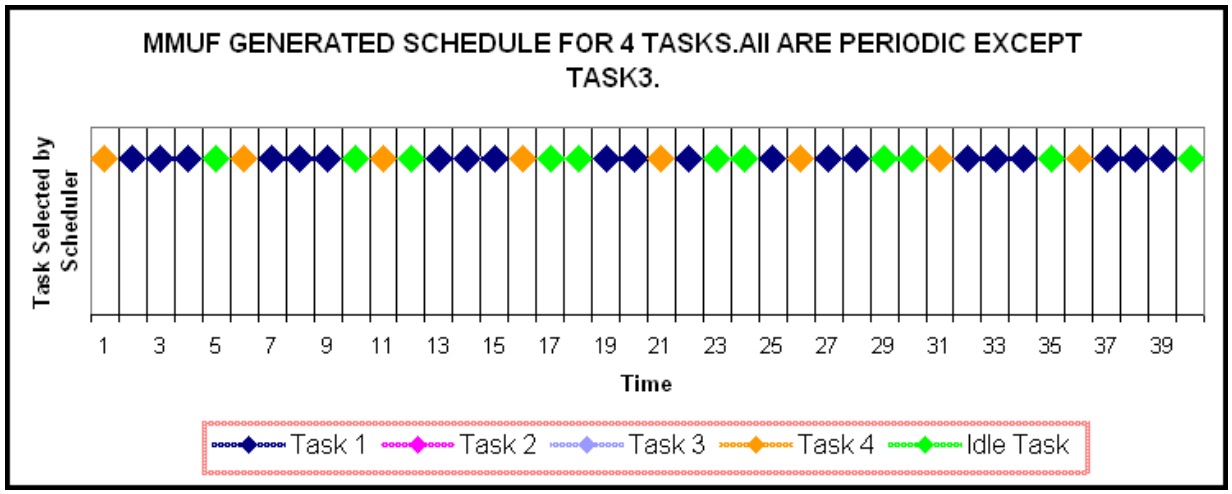

Figure 7.13: Schedule Generated by MMUF : Three tasks are Periodic and one is Aperiodic.Among periodic, one task - Task 2, is not created because interrupt is not received.

Now as shown in Figure 7.16, UART0RX interrupt is received at T = 4. So, Task 2 is created and periodically scheduled. Since, here total CPU Usage is 126%, only critical tasks i.e. *Start Up, Task 1 and Task 2* never misses their respective deadlines.

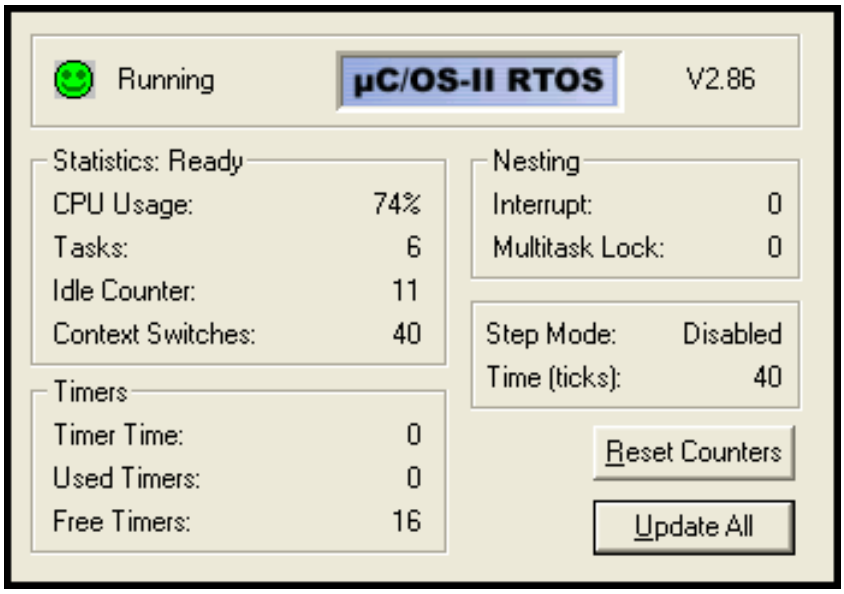

Figure 7.14: Status of the MicroC/OS-II : Three tasks are Periodic and one is Aperiodic.Among periodic, one task - Task 2, is not created because interrupt is not received.

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Up	3	3	Dly	2			9	1142	0%	0%	0	0	128	0000	1102
uC/OS-II Idle	0	63	Ready	0			11	1B14	19%	18%	50	48	256	1B44	1A44
uC/OS-II Stat	1	62	Ready	0			11	1A0C	21%	20%	54	52	256	1A40	1940
uC/OS-II Tmr	2	59	Sem	0	OS-TmrSig		1	27BA	24%	23%	62	60	256	27F6	26F6
Task 4	4	1	Ready	0			8	1342	0%	0%	0	0	512	0000	1182
Task 3	5	0	Ready+Suspended	0			0	1562	0%	0%	0	0	512	0000	1382

Figure 7.15: Task list status at T = 40 : Two Tasks are Periodic and one is Aperiodic. Third periodic task, Task 2, is not created due to the UARTRX0 interrupt is not received.

As per the input mentioned in 7.12, task 4 must be scheduled for at most 1 tick for every five tick. But, as seen from the Figure 7.16, task 4 scheduled only for 1 tick upto 10 ticks. This is because after the creation of the task 2, task 4 does not remains critical task.

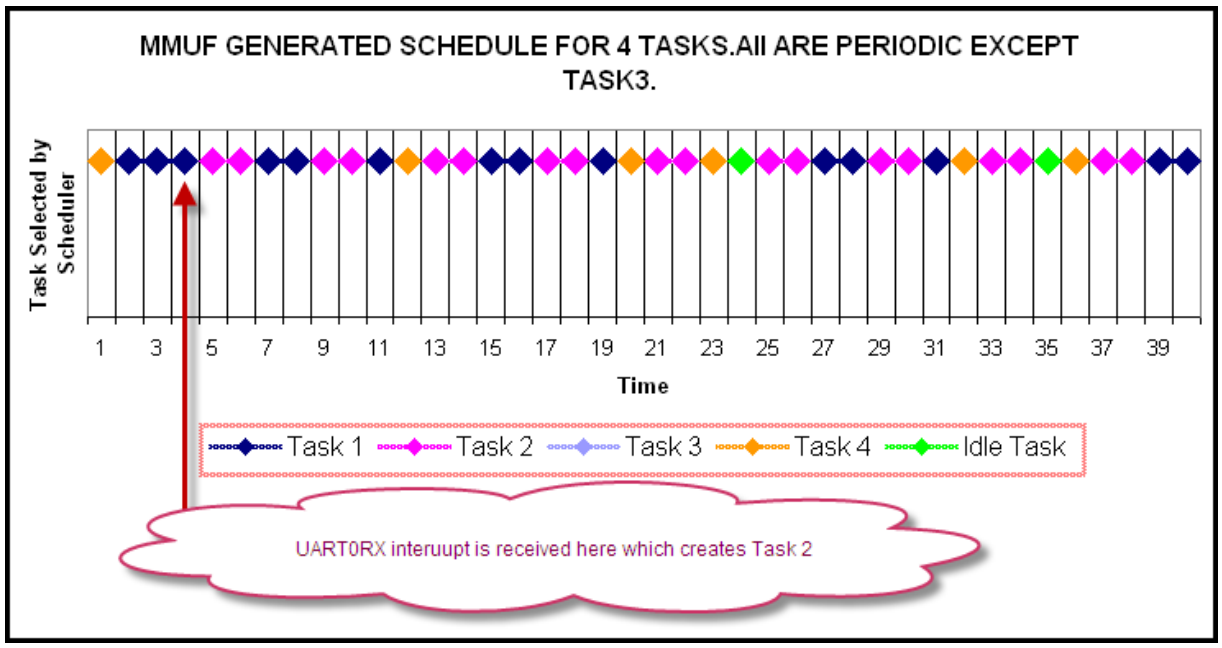

Figure 7.16: Schedule Generated by MMUF : Three tasks are Periodic and one is Aperiodic

7.2.5 Analysis of Results

From the scheduling graphs (Figure 7.2,Figure 7.6, Figure 7.9, Figure 7.13 and Figure 7.16), it is verified that no critical task will miss its respective deadline. Also, when idle task idle task is selected by the schedule, it switches MSP430 in LPM3. The nominal and maximum power consumption done by MSP430F449 is shown in Figure 7.19 and Figure 7.20 respectively, when operating voltage VCC is 2.2V and 3V and environment temperature is between -40 to 80 ° C [6]. Also power saved for cases 7.2.1, 7.2.2 and 7.2.3 is shown in Figure 7.21. So, power consumption by the Micro-controller is reduced drastically as many time idle task is selected by scheduler because no other task is ready for execution. Here, MMUF takes care of the critical tasks and

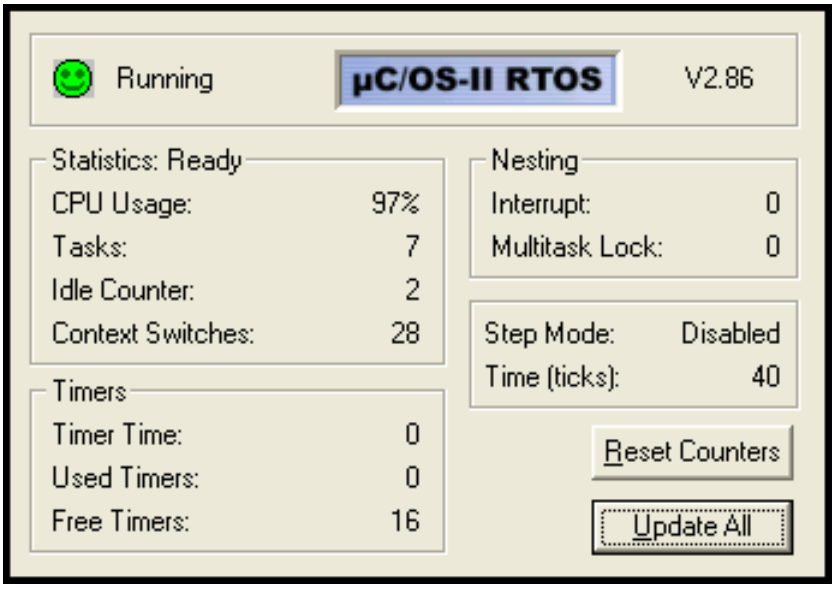

Figure 7.17: Status of the MicroC/OS-II : Three tasks are Periodic and one is Aperiodic

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Up	3	2	Ready	0			8	1142	0%	0%	0	0	128	0000	1102
uC/OS-II Idle	0	63	Ready	0			2	1B14	19%	18%	50	48	256	1B44	1A44
uC/OS-II Stat	1	62	Ready	0			2	1A0C	21%	20%	54	52	256	1A40	1940
uC/OS-II Tmr	2	59	Sem	0	OS-TmrSig		1	27BA	24%	23%	62	60	256	27F6	26F6
Task 4	4	3	Ready	0			6	1342	0%	0%	0	0	512	0000	1182
Task 3	5	0	Ready+Suspended	0			0	1562	0%	0%	0	0	512	0000	1382
Task 2	6	1	Ready	0			9	1742	0%	0%	0	0	512	0000	1582

Figure 7.18: Task list status at T = 40 : Three tasks are Periodic and one is Aperiodic

selects idle task when no other task is active and switch the MSP430F449 to LPM3 to save the power, which indirectly extend the life of the sensor nodes.

7.3 Summary

This chapter describes how to set up the implementation environment for the thesis work. It also describes the various cases 7.2.1, 7.2.2, 7.2.3 and 7.2.4 and detailed analysis of the result. From the result, it is concluded that no critical task will miss its respective deadline. It is also concluded that the algorithm presented in Section 6.1 is power aware and it is best suited for the design of the adhoc sensor nodes using MicroC/OS-II - Real Time Kernel and MSP430F449.

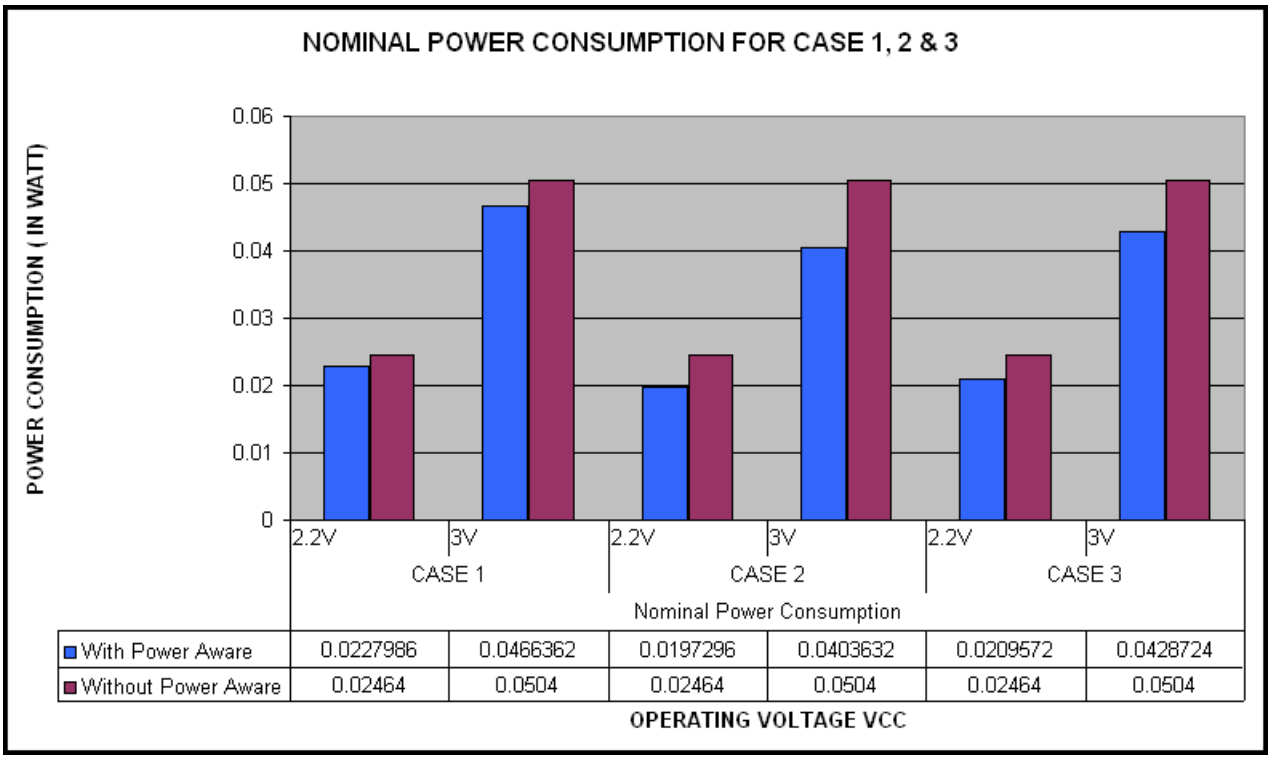

Figure 7.19: Nominal Power Consumption for CASE 1,2 & 3 up to T=40

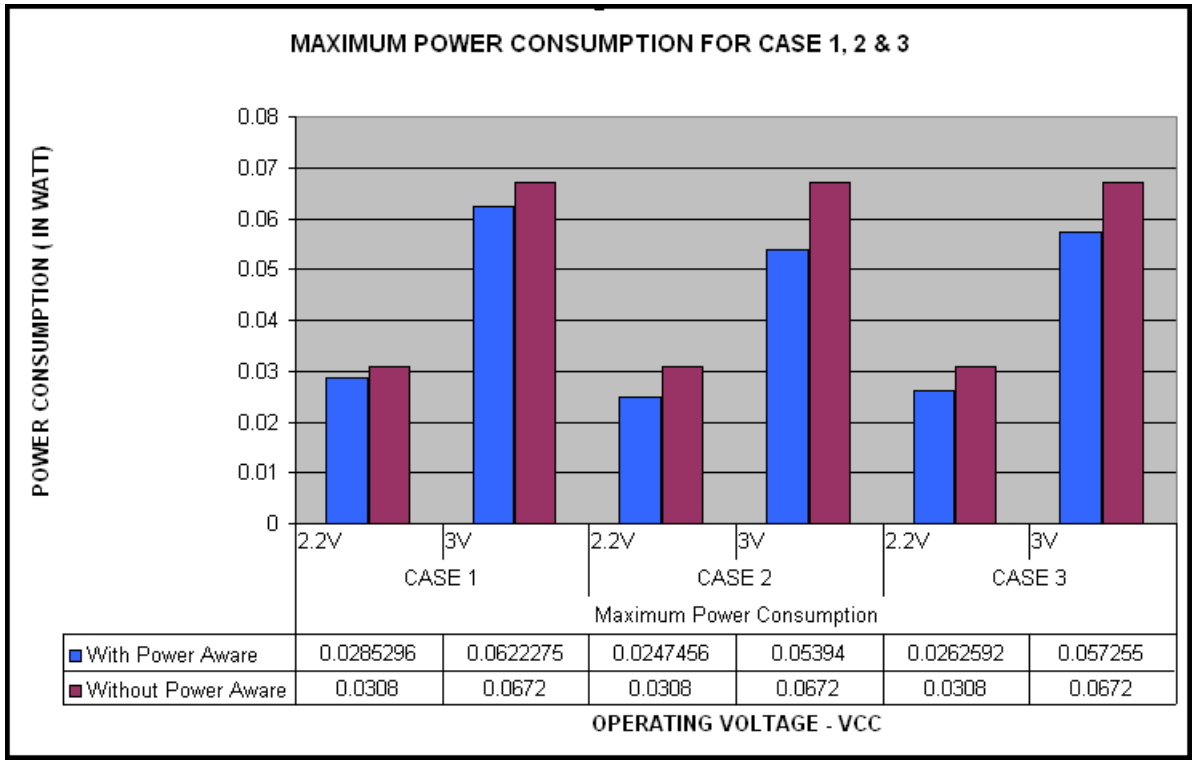

Figure 7.20: Maximum Power Consumption for CASE 1,2 & 3 up to T=40

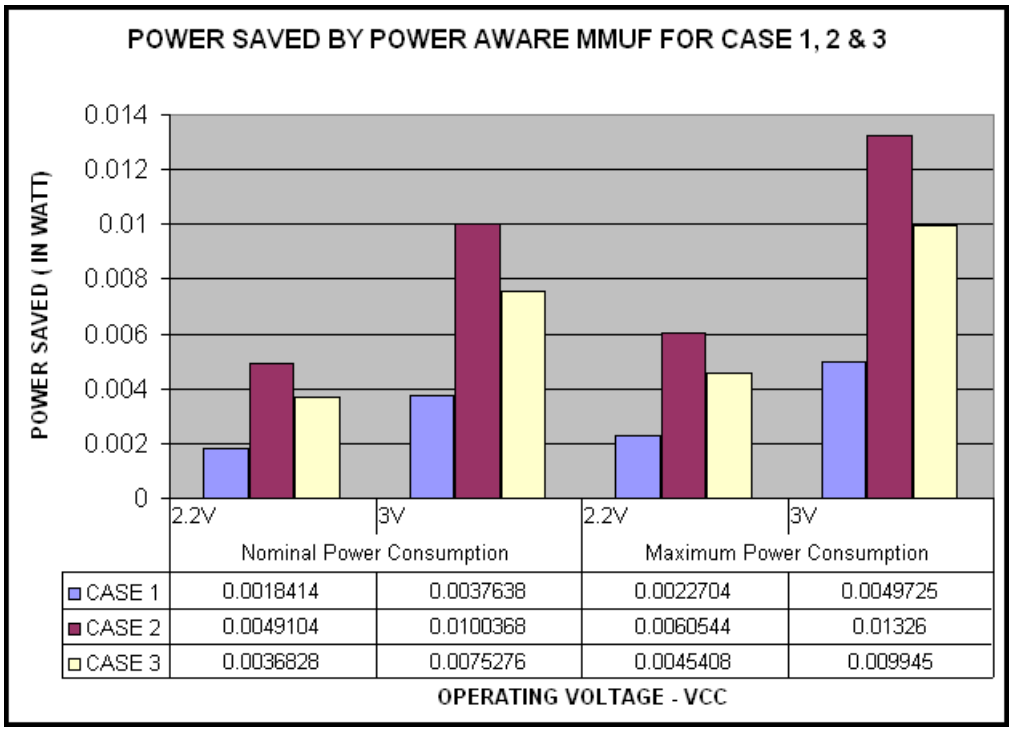

Figure 7.21: Power Saved for CASE 1,2 & 3 up to T=40

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

In this dissertation, I have proposed a method to implement power aware scheduling for adhoc sensor nodes. It doesn't allow critical tasks to miss its respective deadlines. Also, the proposed method supports aperiodic task under the assumption mentioned in Assumption 7. This method also reduces the number of context switching compared to RM, EDF, MLF or MUF [3]. Aperiodic tasks are converted into the periodic tasks by adding maximum interrupt latency to CPUTimeNeeded by the task. Whenever aperiodic task is created, it is converted into the periodic task and immediately its state is changed to (Ready + Suspend) state. As soon as the interrupt is received by the Kernel for an aperiodic task, interrupt service routine made the appropriate aperiodic task Ready and as per the assumption mentioned in 7, it is scheduled without missing its respective deadline.

Since Idle task has assigned the lowest priority by the Kernel, it is selected by the scheduler only when no other task is in ready state. So, Idle task switches the MSP430F449 into LPM3 to save the power. Whenever any interrupt is received by the MSP430F449 or any other task becomes ready except Idle Task, MicroC/OS-II

switches the MSP430F449 into active mode within $< 6\mu$s. So, response time is very small to react the interrupt.

Limitation: *Here, MicroC/OS-II version 2.86 is used which can support maximum 128 tasks. Among 128 tasks 3 priorities are reserved for Idle Task, Statistical Task and to swap the priorities between two tasks. So, maximum 125 tasks are supported by each sensor node.*

From the results we can see that goal of thesis - "Power Aware Scheduling For Adhoc Sensor Nodes" is achieved.

It may be noted that power can be saved by switching MSP430F449 into LMP3 only when CPU is idle i.e. Idle task is scheduled. So, as the CPU load increases, the amount of time for which MSP430F449 remains into LMP3 reduces and power consumption done by sensor node increases.

8.2 Future Scope

In future, the idea presented over here can be extended by creating adhoc network of the sensor nodes which are made of MSP430F449. Also, some low power RF transreceiver can be used to reduce the required power for communicate with other nodes in the network. Also, some new protocol can be designed or may be existing one can be used, to periodically turn off some nodes from the group of nodes which monitors the same region.

Appendix A

Architectural Overview of MSP430X44X

This section describes the basic functions of an MSP430-based system.
The MSP430 devices contain the following main elements:

- Central processing unit
- Program memory
- Data memory
- Operation control
- Peripheral modules
- Oscillator and clock generator

A.1 Introduction

The architecture of the MSP430 family is based on a memory-to-memory architecture, a common address space for all functional blocks, and a reduced instruction set applicable to all functional blocks as illustrated in Figure A.1

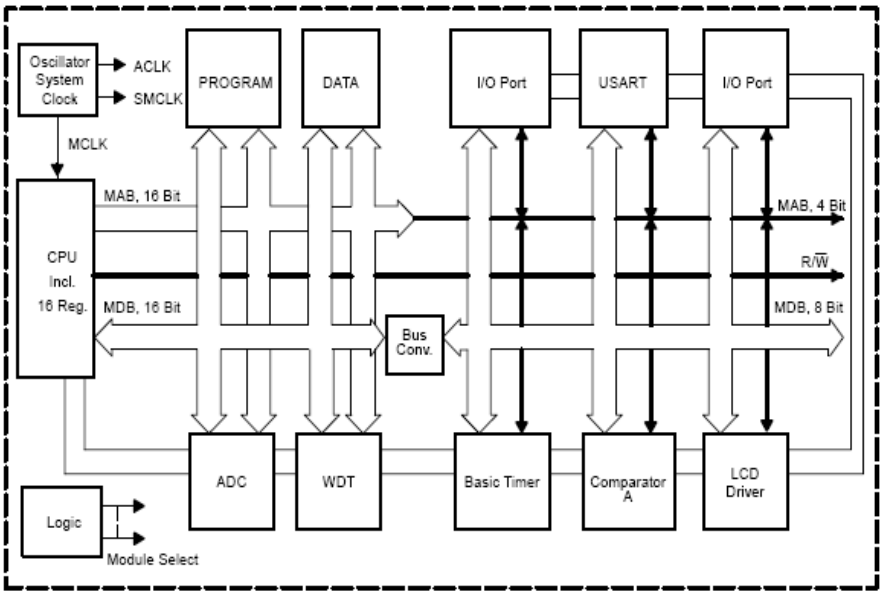

Figure A.1: MSP430 System Configuration

A.2 Central Processing Unit

The CPU incorporates a reduced and highly transparent instruction set and a highly orthogonal design. It consists of a 16-bit arithmetic logic unit (ALU), 16 registers, and instruction control logic. Four of these registers are used for special purposes. These are the program counter (PC), the stack pointer (SP), the status register (SR), and the constant generator (CGx). All registers, except the constant-generator registers R3/CG2 and part of R2/CG1, can be accessed using the complete instruction set. The constant generator supplies instruction constants, and is not used for data storage. The addressing mode used on CG1 separates the data from the constants.

The CPU control over the program counter, the status register, and the stack pointer (with the reduced instruction set) allows the development of applications with sophisticated addressing modes and software algorithms.

A.3 Program Memory

Instruction fetches from program memory are always 16-bit accesses, whereas data memory can be accessed using word (16-bit) or byte (8-bit) instructions. Any access uses the 16-bit memory data bus (MDB) and as many of the least-significant address lines of the memory address bus (MAB) as required to access the memory locations. Blocks of memory are automatically selected through module-enable signals. This technique reduces overall current consumption. Program memory is integrated as programmable or mask-programmed memory.

In addition to program code, data may also be placed in the code memory section of the memory map and may be accessed using word or byte instructions; this is useful for data tables, for example. This unique feature gives the MSP430 an advantage over other microcontrollers because the data tables do not have to be copied to RAM for usage.

Sixteen words of memory are reserved for reset and interrupt vectors at the top of the 64-kilobytes address space from 0FFFFh down to 0FFE0h.

A.4 Data Memory

The data memory is connected to the CPU through the same two buses as the program memory (flash): the memory address bus (MAB) and the memory data bus (MDB). The data memory can be accessed with full (word) data width or with reduced (byte) data width.

Additionally, because the RAM and flash are connected to the CPU via the same busses, program code can be loaded into and executed from RAM. This is another unique feature of the MSP430 devices, and provides valuable, easy-to-use debugging

capability.

A.5 Operation Control

The operation of the different MSP430 members is controlled mainly by the information stored in the special-function registers (SFRs). The different bits in the SFRs enable interrupts, provide information about the status of interrupt flags, and define the operating modes of the peripherals. Total current consumption can be reduced by disabling peripherals that are not needed during an operation.

A.6 Peripherals

Peripheral modules are connected to the CPU through the MAB, the MDB, and the interrupt service and request lines. The MAB is usually a 5-bit bus for most of the peripherals. The MDB is an 8-bit or 16-bit bus. Most of the peripherals operate in byte format. Modules with an 8-bit data bus are connected by bus-conversion circuitry to the 16-bit CPU. The data exchange with these modules must be handled with byte instructions. The SFRs are also handled with byte instructions.

Appendix B

Peripheral Modules and Address Allocation

It has code memory, data memory, and peripherals in one address space. As a result, the same instructions are used for code, data, or peripheral accesses. Also, code may be executed from RAM.

B.1 Introduction

All of the physically separated memory areas (ROM, RAM, SFRs, and peripheral modules) are mapped into the common address space, as shown in Figure B.1 for the MSP430 family. The addressable memory space is 64KB. Future expansion is possible. The memory data bus (MDB) is 16- or 8-bits wide. For those modules that can be accessed with word data the width is always 16 bits. For the other modules, the width is 8 bits, and they must be accessed using byte instructions only. The program memory (ROM) and the data memory (RAM) can be accessed with byte or word instructions.

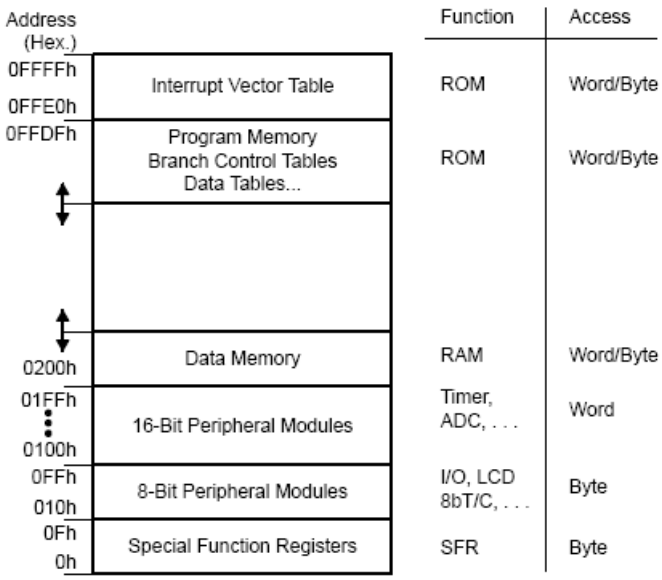

Figure B.1: Memory Map of Basic Address Space

B.1.1 Peripheral Modules - Address Allocation

Some peripheral modules are accessible only with byte instructions, while others are accessible only with word instructions. The address space from 0100 to 01FFh is reserved for word modules, and the address space from 00h to 0FFh is reserved for byte modules.

Peripheral modules that are mapped into the word address space must be accessed using word instructions (for example, MOV R5,&WDTCTL). Peripheral modules that are mapped into the byte address space must be accessed with byte instructions (MOV.B #1,&TCCTL).

The addressing of both is through the absolute addressing mode or the 16-bit working registers using the indexed, indirect, or indirect autoincrement addressing mode. ***See Figure B.2*** for the RAM/peripheral organization.

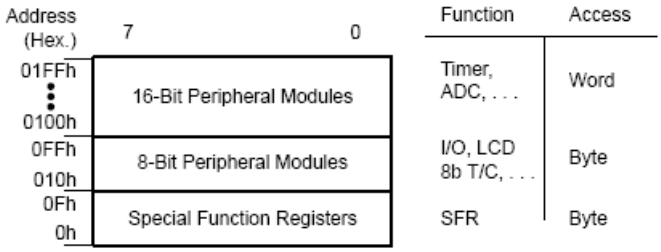

Figure B.2: Example of RAM/Peripheral Organization

Word Modules

Word modules are peripherals that are connected to the 16-bit MDB.

Word modules can be accessed with word or byte instructions. If byte instructions are used, only even addresses are permissible, and the high byte of the result is always 0.

The peripheral file address space is organized into sixteen frames with each frame representing eight words as described in Table I

Table I: Peripheral File Address Map-Word Modules

Address	Description
1F0h - 1FFh	Reserved
1E0h - 1EFh	Reserved
1D0h - 1DFH	Reserved
1C0h - 1CFH	Reserved
1B0h - 1BFH	Reserved
1A0h - 1AFH	ADC12 control and interrupt
190h - 19FH	Reserved
180h - 18FH	Reserved
170h - 17FH	Timer_A
160h - 16FH	Timer_A
150h - 15FH	ADC12 conversion memory
140h - 14FH	ADC12 conversion memory
130h - 13FH	Multiplier
120h - 12FH	Watchdog Timer, flash control
110h - 11FH	Reserved
100h - 10FH	Reserved

Byte Modules

Byte modules are peripherals that are connected to the reduced (eight LSB) MDB. Access to byte modules is always by byte instructions. The hardware in the peripheral byte modules takes the low byte (the LSBs) during a write operation.

Byte instructions operate on byte modules without any restrictions. Read access to peripheral byte modules using word instructions results in unpredictable data in the high byte. Word data is written into a byte module by writing the low byte to the appropriate peripheral register and ignoring the high byte.

The peripheral file address space is organized into sixteen frames as described in Table II.

Table II: Peripheral File Address Map-Byte Modules

Address	Description
00F0h - 00FFh	Reserved
00E0h - 00EFh	Reserved
00D0h - 00DFh	Reserved
00C0h - 00CFh	Reserved
00B0h - 00BFh	Reserved
00A0h - 00AFh	LCD
0090h - 009Fh	LCD
0080h - 008Fh	ADC12 memory control
0070h - 007Fh	USART0
0060h - 006Fh	Reserved
0050h - 005Fh	System clock generator, Comparator_A, brownout/SVS
0040h - 004Fh	Basic timer, 8-Bit Timer/Counter, Timer/Port
0030h - 003Fh	Digital I/O port P5 and P6 control
0020h - 002Fh	Digital I/O port P1 and P2 control
0010h - 001Fh	Digital I/O port P3 and P4 control
0000h - 000Fh	Special function

Address	Data Bus 7 0
000Fh	Not yet defined or implemented
000Eh	Not yet defined or implemented
000Dh	Not yet defined or implemented
000Ch	Not yet defined or implemented
000Bh	Not yet defined or implemented
000Ah	Not yet defined or implemented
0009h	Not yet defined or implemented
0008h	Not yet defined or implemented
0007h	Not yet defined or implemented
0006h	Not yet defined or implemented
0005h	Module enable 2; ME2.2
0004h	Module enable 1; ME1.1
0003h	Interrupt flag reg. 2; IFG2.x
0002h	Interrupt flag reg.1; IFG1.x
0001h	Interrupt enable 2; IE2.x
0000h	Interrupt enable 1; IE1.x

Figure B.3: Special Function Register Address Map

Peripheral Modules-Special Function Registers (SFRs)

The system configuration and the individual reaction of the peripheral modules to the processor operation is configured in the SFRs as described in Figure B.3. The SFRs are located in the lower address range, and are organized by bytes. SFRs must be accessed using byte instructions only. The system power consumption is influenced by the number of enabled modules and their functions. Disabling a module from the actual operation mode reduces power consumption while other parts of the controller remain fully active (unused pins must be tied appropriately or power consumption will increase; see Basic Hints for Low Power Applications in Section 2.3.4

References

[1] T. Instruments, "Msp430x4xx - flash/rom lcd products from texas instruments." Website, 2009. http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?familyId=914&sectionId=95&tabId=1530&family=mcu.

[2] J. J. Labrosse, *MicroC/OS-II - The Real Time Kernel (Second Edition)*. 600 Harrison Street, San Francisco, CA 94107 USA: CMPBooks, 2002.

[3] S. T. Zargar, V. Salmani, and M. Naghibzadeh, "A modified maximum urgency first scheduling algorithm for real-time tasks," in *World Academy Of Science, Engineering And Technology Volume 9*, NOVEMBER 2005.

[4] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in Real-Time Programming, ed. by W. Halang and K. Ramamritham,(Tarrytown, New York: Pergamon Press Inc.), July 1992.

[5] T. Intsruments, "Msp430 microcontroller (mcu), low power mixed signal processor," 2009.

[6] T. Instruments, "Electrical charactristics of msp430f449." Website, 2009. http://focus.ti.com/lit/ds/symlink/msp430f449.pdf.

[7] micrium, "Download microc/os-ii." Website, 2009. http://www.micrium.com/products/rtos/ucos-ii_download.html.

[8] W. Kalfa, "Proposal of an external processor scheduling in micro-kernel based operating systems," tech. rep., The Tenet Group, Computer Science Division, Department of EECS, University of California, Berkeley, and International Computer Science Institute. On sabbatical leave from Dresden University of Technology, Germany, 1992.

[9] I. Systems, "Msp430 iar embedded workbench ide user guide." Website, 2009. ftp://ftp.iar.se/WWWfiles/msp430/guides/ou430-6.pdf.

[10] micrium, "Micrium, inc. microc/os-ii kernel awareness for c-spy." Website, 2009. http://www.micrium.com/downloads/support/uCOS-II-KA-CSpy-UserGuide.pdf.

Index

Abbreviation Notation and Nomenclature, vii
Abstract, iv
Acknowledgements, vi
Analysis of Results, 68
Architectural Overview of MSP430X44X, 75

Background, 2
Basic Hints for Low-Power Applications, 16
Byte Modules, 82

Case 1: All tasks are periodic, 57
Case 2: All tasks are periodic except one. Aperiodic task never becomes ready., 61
Case 3: All tasks are periodic except one. Aperiodic task becomes ready once only, 64
Case 4: New task will be added to the task list on the reception of the interrupt, 64
Central Processing Unit, 76
Certificate, iii
Creating a Task, OSTaskCreate(), 33
Critical Sections, 18

Data Memory, 77
Data structures used to implement MMUF, 54

Earliest-Deadline-First Scheduling Algorithm, 38
Existing Methodologies, 48

Features and Capabilities, 7

Implementation, 53
Implementation Environment, 53
Interrupt Control Bits in Special-Function Registers (SFRs), 10
Interrupt Processing, 8
Introduction, 1
Introduction about the scheduling algorithms, 36
Introduction of Architectural Overview of MSP430X44X, 75
Introduction of Peripheral Modules and Address Allocation, 79

Locking and Unlocking the Scheduler, 30

Low-Power Mode 4 (LPM4), 16
Low-Power Modes 0 and 1 (LPM0 and LPM1), 14
Low-Power Modes 2 and 3 (LPM2 and LPM3), 15

Making a task ready to run, 25
Maximum-Urgency-First scheduling algorithm, 39
Method One, 48
Method Three, 48
Method Two, 48
Minimum-Laxity-First Algorithm, 38
Modified Maximum-Urgency-First scheduling algorithm, 43
MSP430 Features, 6

Objective of Study, 3
Operating Modes, 11
Operation Control, 78
OS_TCBInit(), 24

Peripheral Modules - Address Allocation, 80
Peripheral Modules and Address Allocation, 79
Peripheral Modules-Special Function Registers (SFRs), 83
Peripherals, 78
Porting of MicroC/OS-II on MSP430 IAR Embedded Workbench IDE, 54
Problem Definition, 47
Problem Definition and Existing Methodologies, 47
Program Memory, 77

Rate Monotonic Algorithm, 37
Ready List, 24
Real Time Operating System for Sensor, 18
Results, 56

Scheduling Algorithms for Sensor Nodes, 36
Scope of Work, 4
Starting Multitasking, 32
Summary of Implementation, 70
Summary of MSP430 Features, 17
Summary of Problem Definition and Existing Methodologies, 49
Summary of Real Time Operating System for Sensor, 35
Summary of scheduling algorithms for sensor nodes, 46
Summary of the proposed algorithm, 51

Task Control Blocks (OS_TCB), 20
Task Level Context Switch, OS_TASK_SW(), 30
Task Scheduling, 28
Task States, 19
The Algorithm, 50

The Proposed Algorithm, 50
Thesis Organization, 5

Word Modules, 81