A REPORT

ON

IMPLEMENTATION OF µITRON4.0 PORT OVER MicroC/OS-II

BY

YAMINI PRADEEPTHI ALLU   2001A7PS019

AT

NATSEM INDIA DESIGNS PVT LTD

BANGALORE.

A Practice School II Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI

14th DECEMBER 2004

A REPORT

ON

IMPLEMENTATION OF µITRON4.0 PORT OVER MicroC/OS-II

BY

NAME	ID.NO	DISCIPLINE
YAMINI PRADEEPTHI ALLU	2001A7PS019	B.E.(Hons) COMPUTER SCIENCE

Prepared in partial fulfilment of

Practice School II Course

AT

NATSEM INDIA DESIGNS PVT LTD

BANGALORE.

A Practice School II Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI

14th DECEMBER 2004

ACKNOWLEDGEMENTS

I would like to thank Prof S Venkateswaran Vice Chancellor, BITS Pilani and Prof. V.S.Rao, Dean Practice School Division, BITS-Pilani, for giving me an opportunity to work in the industry through Practice School-II. I acknowledge Mr. Ashok Kumar, Managing Director India Operations, Natsem India Designs Pvt. Ltd, Bangalore, for giving me an opportunity to work for the company and use its resources for successful completion of the project. I am extremely grateful to Mr. Sarma Kolluru, Software Manager, Displays Group for his direct involvement and leadership in the project. My sincere thanks to my Project guide Mr. Ramesh P.D.V.N for his guidance and able assistance throughout the project. I also thank my Instructor Dr.S.Rajashekharaiah for his constant support and encouragement. I finally thank all my friends at National Semiconductor for their support.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN)

Practice School Division

Station: Natsem India Designs Pvt Ltd. **Centre**: Bangalore

Duration: 5 ½ Months **Date of Start**: 5th July 2004

Date of Submission: December 14,2004

Title of Project: IMPLEMENTATION OF µITRON4.0 PORT OVER MicroC/OS-II.

IDNo.s /Names/Disciplines Of Students:

2001A7PS019, Yamini Pradeepthi Allu, B.E (Hons) Computer Science.

Name(s) and Designation(s) of Expert(s):

Mr. Sarma V Kolluru, Software Manager, Natsem India Designs.

Mr. Ramesh.P.D.V.N, Senior Software Engineer, Natsem India Designs.

Name(s) of PS Faculty: Dr.S.Rajashekharaiah.

Key Words: MicroC/OS-II, Itron, Real time kernel, Operating systems.

Project Areas: Real time Operating Systems.

Abstract: MicroC/OS-II is a real time kernel for embedded systems. This kernel is ported over a chip for digital processing in Natsem. The aim of the project is to develop a port over MicroC/OS-II to support µITRON API specification, a real time kernel specification, which is now a defacto standard in Japan. This report gives an overview of the two kernels and discusses the porting issues. It also provides details for the implementation of µITRON port.

Signature of Student: **Signature of PS faculty:**

Date: **Date:**

TABLE OF CONTENTS

Acknowledgements…………………………………………………………………..….i

Abstract…………………………………………………………………….. …..*ii*

1.INTRODUCTION…………………………………………………………..1

2.OVERVIEW OF KERNELS………………………………………………..2

2.1.MicroC/OS-II kernel-Overview…………………………………….…..2

2.2. μITRON4.0 Overview…………………………………………..…..3

3. PORTING ISSUES……………………………………………………..…..5

3.1.TaskStates……………………………………………………..…..5

3.1.1.Microc/OS-II Task States…………………………………... ….5

3.1.2. μITRON4.0 Task states…………………………………… ….7

3.1.3.Mapping ITRON task states over Micro/OS-II…………………9

3.2.TaskScheduling………………………………………………….….10

3.2.1.Task Scheduling under MicroC/OS-II……………………...…10

3.2.2.Task Scheduling under ITRON 4.0………………………… …10

3.2.3. Scheduling Issues…………………………………….. …10

4. MICROC/OS-II-KERNEL MODIFICATION……………………………………11

4.1.ImplementationDetails………………………………………….…..11

4.1.1.Changes made to the data structures of the kernel…………….11

4.1.2.Changes made in the functions of the kernel…………….…..13

4.2.Effect on execution times of various functions…………………………..15

5.ITRON PORT: IMPLEMENTATION…………………………………………. …15

5.1.Additional Data structures…………………………………………. …15

5.2.API………………………………………………………………..…15

5.2.1.Task Management Functions………………………………..16

5.2.2. Task Dependent Synchronization Functions……………….…20

Conclusion……………………………………………………………………… ..23
Appendix1………………………………………………………………………24
Appendix2………………………………………………………………………32
List of References…………………………………………………………… ..34

LIST OF ILLUSTRATIONS

Fig 3.1. MicroC/OS-II state Transition Diagram…………………………………………5

Fig 3.2. ITRON 4.0 State Transition Diagram ……………………………………………..7

Fig 4.1. Illustration of Data Structures…………………………………………….….12

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN)

PRACTICE SCHOOL DIVISION

Response Option Sheet

Station: Natsem India Designs Pvt Ltd. **Centre**: Bangalore

IDNo. and Name(s): 2001A7PS019, Yamini Pradeepthi Allu.

Title of Project: IMPLEMENTATION OF μITRON4.0 PORT OVER MicroC/OS-II.

Usefulness of the project to the on-campus courses of study in various disciplines. Project should be scrutinized keeping in view the following response options. Write Course No. and Course Name against the option under which the project comes.

Refer Bulletin for Course No. and Course Name.

Code No.	Response Options	Course No.(s) and Name
1.	A new course can be designed out of this project.	
2.	The project can help modification of course content of some existing courses.	
3.	The project can be directly used in some of the existing Compulsory Disciplinary Courses (CDC)/Discipline Courses Other than Compulsory (DCOC)/EA etc. Courses.	CS C372, Operating Systems.
4.	The project can be used in preparatory courses like Analysis and Application Oriented Courses(AAOC)/Engineering Science(ES)/Technical Art(TA) and Core Courses.	
5.	This project cannot come under any of the above mentioned options as it relates to the professional work of the Host Organization	

Signature of student **Signature of Faculty**

1. INTRODUCTION

MicroC /OS-II The Real-Time Kernel, by Mr. Jean.J.Labrosse is an open source kernel and is most widely used in the industry. At Natsem India Designs, the kernel is ported over CR-16, an embedded processor for digital video processing. ITRON is a de-facto standard operating system specification for small-scale embedded systems in Japan. The µITRON4.0 Specification is the latest version of the µITRON real-time kernel specification. The aim of this project is to develop a port a port (compatibility layer) to allow compatibility of MicroC/OS-II with ITRON based applications. This would enable an embedded processor (using MicroC/OS-II) to be used in other products for which applications were developed using ITRON specification.

The ITRON Project is promoted by the ITRON Specification Group in the TRON Association, as one of the subproject of the TRON Project(The Real-time Operating system Nucleus). Many series of ITRON (Industrial TRON) real-time kernel specifications have come up since the project has started. Of these, the µITRON (Micro ITRON) real-time kernel specification was designed for consumer products and other small-scale embedded systems. The µITRON4.0 Specification is the latest version of the µITRON real-time kernel specification.

The report as the title suggests gives documentation for implementation of ITRON port over MicroC OS-II. The implementation code including the modified kernel was compiled using Turbo C/C++ compiler V2.01, and options were selected to generate for Intel/AMD 80816 processor (large memory model). The code is run and tested on 3.0GHz Intel Pentium IV PC, running in a DOS window using Microsoft Windows XP version 2002.

This report on the Implementation of uITRON4.0 port for MicroC/OS-II includes six chapters with the first chapter being the Introduction. The second chapter gives an overview of the two kernels involved (MicroC/OS-II and ITRON 4.0). The third chapter deals with the issues involved in implementing the port. Chapter four includes the details

about the modifications made to Micro/OS-II to support the port and chapter 5 explains in detail the implementation of Itron API. Chapter 6 presents with Conclusions.

2. OVERVIEW OF KERNELS

2.1. MicroC/OS-II kernel Overview:

MicroC/OS-II is a small, portable real-time kernel. It has a fully preemptive prioritized task scheduler supporting up to 63 tasks, and mechanisms for passing signals and messages between tasks. Tasks can be dynamically created and deleted, and task priorities can be dynamically changed.

Features of MicroC/OS-II:

Portable: Most of MicroC/OS-II is written in highly portable ANSI C, with target microprocessor-specific code written in assembly language. Assembly language is kept to a minimum to make MicroC/OS-II easy to port to other processors.

ROMable:

MicroC/OS-II was designed for embedded applications. This means that if there is the proper tool chain (i.e. C compiler, assembler and linker/locator), µC/OS-II can be embedded as part of a product.

Scalable:

MicroC/OS-II was designed so that only the services that are needed in a application can be used this allows reduction in the amount of memory (both RAM and ROM) needed by µC/OS-II on a product per product basis.

Preemptive:

µC/OS-II is a fully preemptive real-time kernel. This means that µC/OS-II always runs the highest priority task that is ready.

Multi-tasking:

µC/OS-II can manage up to 64 tasks, however, the current version of the software reserves eight (8) of these tasks for system use. This leaves an application with up to 56 tasks. Each task has a unique priority assigned to it. This means that µC/OS-II cannot do round robin scheduling. There are thus 64 priority levels.

Deterministic:

Execution time of all µC/OS-II functions and services are deterministic i.e. the time needed for µC/OS-II to execute a function or a service is always known. Except for one service, execution time of all µC/OS-II services does not depend on the number of tasks running in your application.

Services:

µC/OS-II provides a number of system services such as mailboxes, queues, semaphores, fixed-sized memory partitions, time related functions, etc.

Interrupt Management:

Interrupts can suspend the execution of a task and, if a higher priority task is awakened as a result of the interrupt, the highest priority task will run as soon as all nested interrupts complete. Interrupts can be nested up to 255 levels deep.

Robust and reliable:

µC/OS-II is based on µC/OS, which has been used in hundreds of commercial applications since 1992. µC/OS-II uses the same core and most of the same functions as µC/OS yet offers many more features. µC/OS-II has attained FAA DO-178B level B certification and will soon attain level a certification.

2.2. µITRON4.0 Overview:

The µITRON4.0 Specification is the latest version of the µITRON real-time kernel specification. The µITRON4.0 specification offers many improvements over previous versions. The most important among them is the definition of the Standard Profile, which strictly specifies the standard set of kernel functions for improving application portability.

Features of ITRON specification:

Independent of Processor architecture:

ITRON specification is intended for OS implementation on many different processors. However, specific implementation and design is carried out independently for each processor. Specifications that were common to all processors are clearly isolated from those that were processor-dependent. Specifications which were difficult to standardize

were left alone, so as to prevent loss in performance due to excessive standardization and processor virtualization.

Service:

ITRON specification provides many system calls possessing various useful functions This allows for the selection of the mechanism best suited to the application and should lead to better performance and greater ease in programming.

High-speed response:

ITRON specification allows registers to be swapped out at dispatch time to be specified to achieve high-speed dispatching. Furthermore, it is possible to cause a user-defined interrupt handler to run any time an external interrupt occurs, without having to go through the OS

Built-in adaptability:

ITRON specification, an OS architecture, has been designed as a highly flexible and adaptable architecture A high level of freedom provided to the user by giving many choices such as selection of system calls etc.

Compatibility with hardware platforms:

There is a series of ITRON specification operating systems operating on many hardware platforms from 8-bit MCUs to 32-bit high-performance microprocessors

Ease of learning:

System call names and functions of ITRON specification are systematically defined according to uniform naming conventions, which make them easy to remember

3. PORTING ISSUES

Porting ITRON over MicroC/OS-II involves some complex issues that are discussed in detail below.

3.1.TaskStates:

3.1.1.Microc/OS-II Task States:

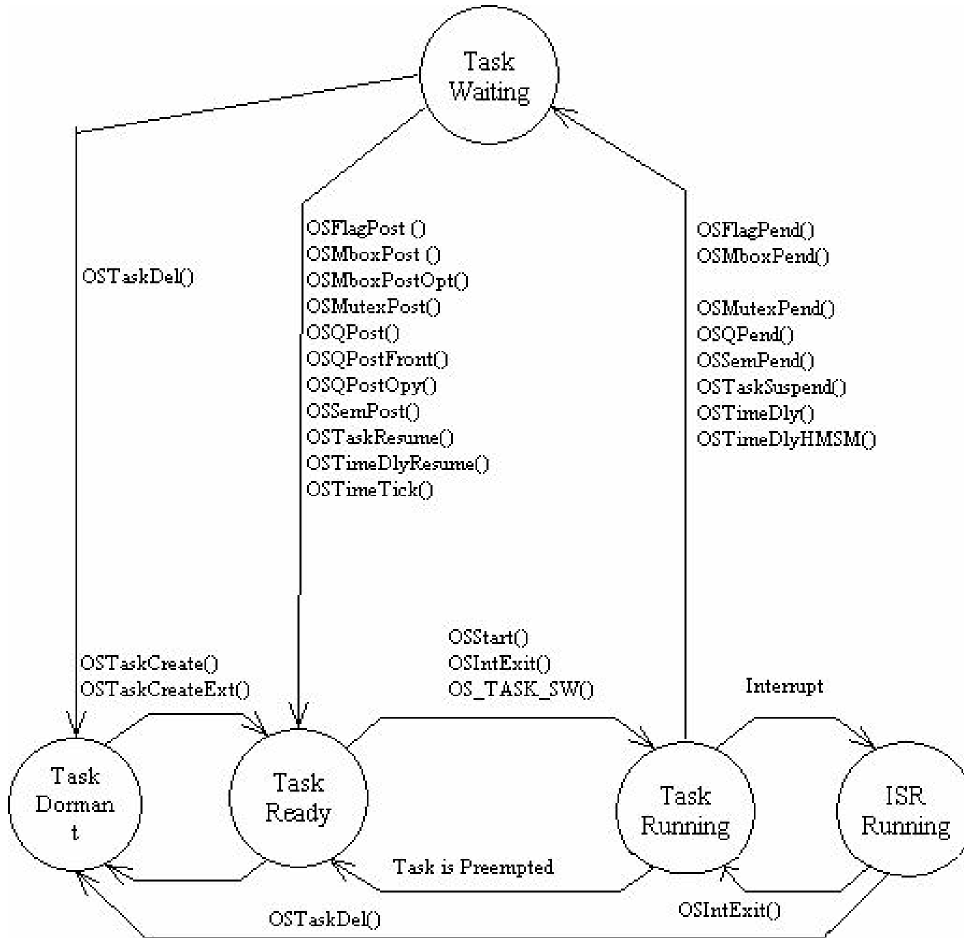

fig 3.1: MicroC/OS-II state Transition Diagram

In uC/OS-II at any given time, a task can be in any of the five states discussed below. The task state transition diagram(fig 3.1) shows the various transitions that are allowed in MicroC/OS-II

1.TASK DORMANT STATE:

This state corresponds to a task that resides in program space (ROM or RAM) but has not been made available to the kernel.

2.TASK READY :

A task is made available to uC/OS-II by creating the task. When a task is created it is made ready to run and placed in TASK READY state.

3.TASK RUNNING:

Multi tasking is started by calling OSStart(), which starts the highest priority task that has been created. The highest priority task is thus placed in the TASK RUNNING state.

4.TASK WAITING:

A task delaying itself or waiting for an event to occur is placed in the TASK WAITING state until the occurrence of the event or timeout.

5.ISR RUNNING:

A running task can always be interrupted, unless the interrupts are disabled. The task thus interrupted is put in ISR RUNNING state.

3.1.2. µITRON4.0 Task states:

fig 3.2: ITRON 4.0 State Transition Diagram

The above diagram (fig 3.2) shows the various states and transitions in ITRON.

1.RUNNING state

This state indicates the task is currently executing.

2.READY state

This state indicates that the task is ready to execute, but cannot because a task of higher priority (sometimes same priority) is already executing.

3. (General) Wait state

This state indicates that the task cannot execute because conditions necessary for execution have not yet been met. After a task wakes up from the general wait state, execution will restart from the location it was previously suspended. General wait state can be further classified into three types:

3.1.WAITING state

This state differs from SUSPENDED state, in that the task suspends execution due to a system call it has issued itself.

3.2 SUSPENDED state

This indicates a state where a task has been forcibly made to suspend execution by another task.

3.3 WAITING-SUSPENDED state

This state indicates that the conditions of both WAITING state and SUSPENDED state apply.

4 DORMANT state

This state indicates the task is not yet executing or has already exited. Newly created tasks always begin in this state. This state differs from a general wait state in several respects: all resources are free; the contexts of registers and program counter are initialized when the task starts; and any task management system calls (wup_tsk, ter_tsk, sus_tsk, chg_pri, etc.) issued to a task in this state will result in an error.

5 NON-EXISTENT state

This indicates a virtual state where the task in question does not exist on the system because it has not yet been created or has already been deleted.

3.1.3.Mapping ITRON task states over Micro/OS-II:

NON EXISTANT state of iTRON corresponds to the DORMANT state of MicroC/OS-II The DORMANT state of iTRON can be simulated by storing a pointer to the task creation information, i.e the state corresponds to DORMANT state of uC/OS-II except the task creation information of the task is still stored.

In MicroC/OS-II, the state of a task is stored for reference in the variable (OSTCBStat) in the Task Control Block (TCB).

The various states in uCOS are #defined as follows in the source code-

(Bit Definitions of OSTCBStat)

```
OS_STAT_RDY         0x00    // Ready to run
OS_STAT_SEM         0x01    // pending on semaphore
OS_STAT_MBOX        0x02    // pending on mailbox
OS_STAT_Q           0x04    // pending on queue
OS_STAT_SUSPEND     0x08    // Task is suspended
OS_STAT_MUTEX       0x10    // pending on mutual exclusion semaphore
OS_STAT_FLAG        0x20    // pending on event flag group
```

It can be seen that even though uC/OS-II doesn't define separate wait states SUSPENDED, WAITING and WAIT-SUSPENDED states, it defines different bit values for different waiting states, using which, the three different states of the WAIT state (iTRON) can be distinguished.

uCOS also maintains a flag OSTCBDly which is set if there is a delay on the task.

Hence if OSTCBDly is greater than zero and the task is in SUSPENDED state it is implied that the task is in wait-suspended state, waiting for the suspension to be removed as well as the time to expire.

This corresponds to the WAITING-SUSPENDED state of iTRON.

The state transitions for system calls are explained in detail in chapter 5 where the implementation details of the port are discussed.

3.2.TaskScheduling:

3.2.1.Task Scheduling under MicroC/OS-II:

μC/OS-II is a fully preemptive real-time kernel. This means that μC/OS-II always runs the highest priority task that is ready. No separate queues are maintained. It does bit-map scheduling. There can be only one task per priority and a maximum of 64 priority levels. This means that μC/OS-II cannot do Round Robin Scheduling.

3.2.2.Task Scheduling under ITRON 4.0:

Under ITRON specification, task scheduling is conducted based on task priority. ITRON specification requires at least 16-levels of priorities and allow more than one task per priority. Each task is uniquely identified by its ID.If there are more than one tasks of the same priority, scheduling is conducted on "first come, first served" (FCFS) basis. Round Robin Scheduling can be achieved by rotating the precedence of the tasks with same priority at the intervals of TIME SLICE.

3.2.3. Scheduling Issues:

Since the basic function of creating a task with already existing priority (Hence Time Slicing) is not possible in MicroC/OS-II, a port over the existing kernel would add a very high overhead, and affects the response time, interrupt latency etc. Hence the uCOS kernel should be changed to allow more than one task per priority. Porting ITRON over this changed kernel would be more economical in terms of overhead. The next chapter deals with the issues involved in modification of MicroC/OS-II kernel and the implementation details.

4.MICROC/OS-II-KERNEL MODIFICATION

MicroC/OS-II is a preemptive kernel and always executes the highest priority task ready to run. It does not support multiple tasks with same priority. The maximum number of tasks that can be created is 64 including system tasks.

Modified MicroC/OS-II:

MicroC/OS-II is modified to a kernel in which

- Multiple tasks (any number) can be created with same priority.
- Multilevel queue scheduling implemented, i.e. different levels of queues (priority levels), tasks with same priority and that are ready to run are queued at that level.
- Only 64 levels of priority.
- The scheduling policy adopted in this kernel for each queue is time slicing (also known as round-robin scheduling); this means that a task can be moved from RUN state to READY state at any time, in order that one of its peers may run. This is not strictly conformant to the µITRON specification, which states that time slicing may be implemented by periodically issuing a rot_rdq(0) call from within a periodic task or cyclic handler; otherwise it is expected that a task runs until it is pre-empted in consequence of synchronization or communication calls it makes, or the effects of an interrupt or other external event on a higher priority task cause that task to become READY

4.1.ImplementationDetails:

4.1.1.Changes made to the data structures of the kernel

- An array of pointers to TCBs (ID_TBL[ ]) indexed by the Id is added. This structure can be used to refer to the TCB of the task when the id of task is known.
- An array of pointers to ready queues indexed by the priority (RDY_TBL[ ]) is added. This is the multi level queue structure for ready tasks used for scheduling.
- Each element in the queue (RDY_NODE) has three pointers. Two pointing to next and previous RDY_NODEs (To form a doubly linked list of RDY_NODEs) and one pointing to the TCB of the task, which is ready.

- An array of RDY_NODE structures ITRdyNodeList [MAX_NO_TASKS] is created which acts as a pool of nodes from which a node is allocated when a task is created or made ready. The node allocated to a task would be the one indexed by its id.
- Two pointers Nextp and Prevp are added to the TCB. These pointers are used to point to the next and previous TCBs of tasks with the same priority to form a circular doubly linked list
- An array of integers (EventCntTbl [OS_ LOWEST _ PRIO]) is added to the Event Control Block. This is used to keep a count of the number of tasks with same priority waiting on that event.

The following figure illustrates clearly the above mentioned data structures.

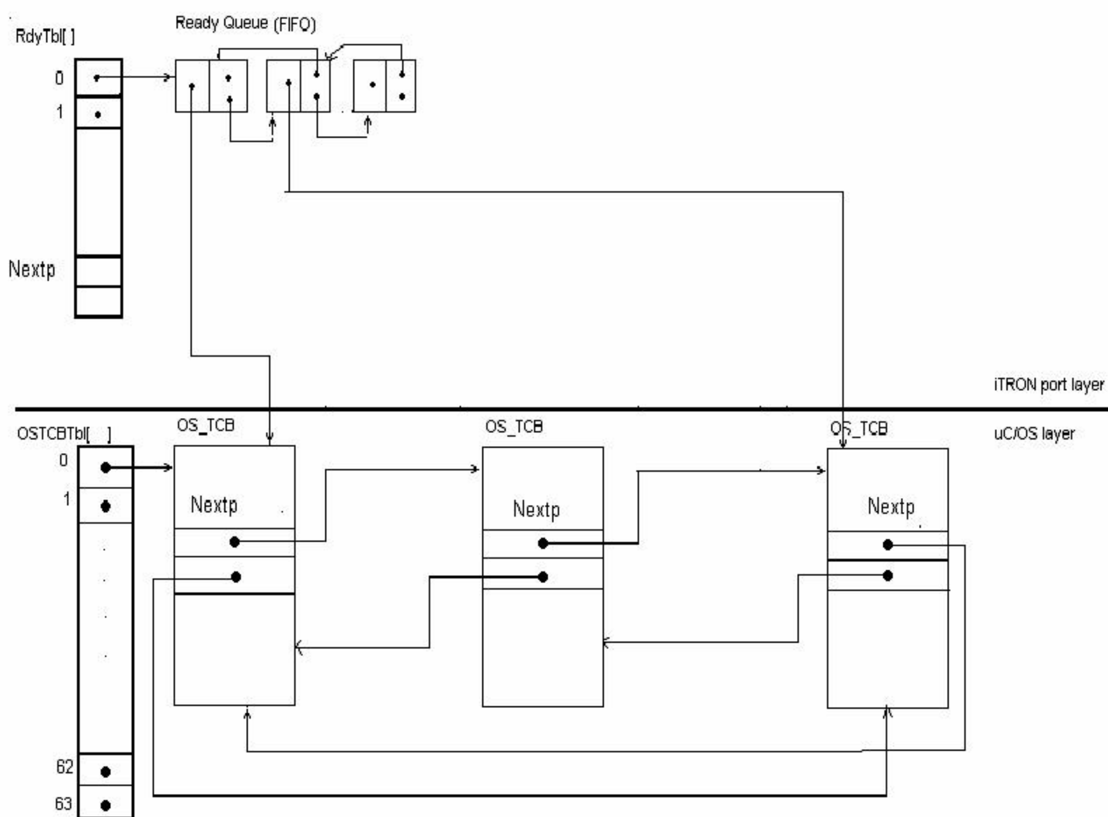

Fig 4.1.Illustration of Data Structures

4.1.2.Changes made in the functions of the kernel:

Changes are made to almost every function in MicroC/OS-II to accommodate for multiple tasks with same priority. Since no more a task is identified by its priority but is identified by its ID all API that involves priority of a task as a parameter are changed to ID of the task. Modifications made in some of the very important functions that actually deal with the creation and scheduling of tasks is discussed in detail below.

OSTaskCreateExt ( )

- A task is created even if there exists another task with the same priority.
  The Corresponding check is removed.
- TCBInit( ) is changed so that the tcb that was allocated is linked in the doubly linked list. And also the task is added in the ready list corresponding to that priority.

OSSched( ) and **OSINTExit**( )

- The OSPrioHighRdy is obtained, and OSTCBHighrdy is made to point the TCB of the first task in the ready queue corresponding to the priority OSPrioHighRdy.
- The two pointers OSTCBCur and OSTCBHighRdy are compared and if they differ task switching is done.

  In the previous kernel the two priorities OSPrioCur and OSPrioHighRdy are compared and only if they differ a context switch is called.

OSTimeTick( ) (Implementation Of Round Robin Scheduling)

- A count of the execution time(in ticks) of a task is maintained in a variable (RRCnt) declared in the header of the ready queue.
- For each tick this count in the ready queue of the Current task is incremented
- When the count equals RR_TIME_SLICE (value to be declared in OS_CFG.h) the ready queue of the task currently running is rotated, and count is re initialized to zero.
- The task, which was second in the queue, would now become the first.
- i.e. the precedence is rotated.
- OSINTExit( ) makes this task to run. hence round robin is achieved.

OSEventTaskRdy( )

- The priority of the task that is highest among the tasks waiting for event is obtained through the OSEventTbl [].
- The linked list of tcbs corresponding to this priority is traversed till a task waiting on that event is found.
- This task is made ready by adding into the ready list, if it is not suspended and the count of tasks waiting is decremented for that priority.
- The bit in OSEventtbl [] is cleared only if there is no other task waiting on the event with same priority.

OSMutexCreate, Pend and Post:

- A mutex is created even if there is a mutex with the same PIP (priority inheritance priority) that is already created.
- When a task pends on a mutex that is owned by a task, which is of lesser priority, its priority, is changed to that of PIP.
- I.e. The lesser priority task is removed from the TCB list at that priority and linked to the TCB list at priority PIP.
- If it was ready at previous priority it is removed from the ready queue at old priority and added to the queue at PIP.
- When a task posts a mutex and if it was raised to PIP the same actions are taken as above(PIP to old prio).
- Thus all the tasks that are raised to PIP run in round robin.

These are some of the few functions that are changed, in most othere cases the code corresponding to making the task ready or making the task wait for an event or for time out is changed. the following lines explain the steps taken for this.

- Whenever a tasks need to be delayed it is removed from the ready queue(dequeue) and the bit in the OSRdyTbl is cleared only if there are no more tasks ready with that priority.I.e RDY_TBL[prio]->N_tsks is zero.
- If the task is delayed waiting on an event the count of EventCntTbl[prio] corresponding to that priority is incremented.
- Whenever a task is made ready its tcb is enqueued in the ready list of its priority. And N_tsks incremented.

- Ü If OSEventTaskRdy makes the task ready, the bit in the OSEventTbl is cleared only if there are no more tasks waiting on that event with that priority. I.e. if EventCntTbl [prio] is zero.

Additional functions to handle the ready queue like enqueue( ) dequeue() rot_rdq() are added which puts a task, removes a task from ready queue and rotate the ready queue respectively.

4.2.Effect on execution times of various functions:

The execution time of all the functions except OSEventTaskRdy ( ) remain unchanged or increased by constant time, irrespective of the number of tasks created.

Execution of OSEventTaskRdy is O (n) where n is the maximum number of tasks that are created with the same priority.

5. ITRON PORT IMPLEMENTATION

5.1.Additional Data structures:

A file (uit_port.h) declares all the data structures and data types as required by Itron 4.0 specification.

An array of pointers to packets containing Task Creation Information (TSK_INFO []) is added. This table is used to retrieve the initial task parameters when the task is again activated.

Arrays of pointers to Event Control Blocks (MTX_TBL, SEM_TBL etc) and to Queue control blocks (MSQ_TBL []) are added. These tables are used to retrieve the pointer to corresponding element when indexed by the corresponding Event Id

5.2.API

Functions are written according to the uITRON API specification. These functions include a wrap up code in addition to calls to appropriate functions from MicroC/OS-II API. The implementation details for Task Management Functions and Task Dependant Synchronization functions are discussed below.

5.2.1.Task Management Functions:

cre_tsk: Create Task

This system call creates the task specified by tskid. Specifically, a TCB (Task Control Block) is allocated for the task to be created, and initialized according to accompanying parameter values of itskpri, task, stksz, etc. A stack area is also allocated for the task based on the parameter stksz.

The basic difference between uCos and iTRON is that in uC/OS when a task is created it is immediately moved from DORMANT to READY state. (The dormant state is a state where the task does not exist).

In iTRON the task is moved from NON-EXISTENT state to DORMANT state when a task is created. If TA_ACT is specified as an attribute task is moved to READY state.

In ITRON NON-EXISTENT state is similar to the DORMANT state of uCos and the DORMANT (iTRON) state is a state where the task is just created and not activated or it has terminated. Once the task is terminated its context is lost and is moved to DORMANT state so when it is activated again by act_tsk () it starts executing from the beginning.

C language Interface: ER ercd =cre_tsk(ID tskid,T_CTSK pk_ctsk)

Implementation Details:

The following actions are taken when a call for Cre_tsk( ) is made.

1. If attr== TA_ACT, make a call to OSCreateTaskExt( ) with the given task data.
2. .Store the task creation information in TSK_INFO[tskid] indexed by the id.
3. All the port related variables are initialized.(act_cnt,sus_cnt,wup_cnt etc.

del_tsk: Delete Task

This system call deletes the task specified by tskid. Specifically, it changes the state of the task specified by tskid from DORMANT into NON-EXISTENT (a virtual state not existing on the system.

C Language Interface: ER ercd = del_tsk (ID tskid) ;

Implementation details:

1. It is checked if the task is in DORMANT state,if not an error is returned.(A task is in dormant state if ID_TBL[tskid] is NULL)
2. The task creation information for that ID is deleted.TSK_INFO[tskid] is made point to NULL.(Hence the task is moved to NON-EXISTANT state)

act_tsk: Activate a task

This system call puts the task with specified ID in READY state from DORMANT state. If the task is not in the DORMANT state the activation request is queued.

C Language Interface: ER ercd=act_tsk(ID tskid);

Implementation details:

1. If task is in DORMANT state the system call for uCOS task creation
2. OSTaskCreateExt() is called with parameters as the data stored in TSK_INFO[tskid].

3. If the task is not in DORMANT state the act_cnt is incremented by one. If it is greater than the maximum act_cnt an error is returned.

can_act: Cancel Activation

This system call makes the activation count zero and returns the cancelled count.

C Language Interface:UINT cnt= can_act(ID tskid);

Implementation details.

1.act_cnt is stored in a temporary variable and made zero.

2.the value in temporary variable is returned.

ext_tsk: Exit Issuing Task

This system call causes the issuing task to exit, changing the state of the task into the DORMANT state. When a task is terminated it loses its context, so when activated again it starts from the beginning.

C Language Interface: void ext_tsk ( ) ;

Implementation details:

1.The task is removed from all the existing states by making a call to OSTaskDelete()

2.If act_cnt is greater than zero , task is created again by calling OSTaskCreateExt() with the values stored in TSK_INFO[OSTCBCur->OSTCBId].

Hence this makes the task to start from its initial context again.

Sta_tsk : Activate task with given start code.

Activates a task with given start code.

C language interface: ER ercd=act_tsk (ID tsk id)

Implementation details:

1.If the task is in dormant state, create the task.

2.else, return error (The request is not queued)

exd_tsk: Exit and Delete Task

This system call causes the issuing task to exit and then delete itself. The state of the issuing task changes into the NON-EXISTENT (a virtual state not existing on the system).

C Language Interface: void exd_tsk ( );

Implementation details:

1. Calling OSTaskDelete (OS_PRIO_SELF) terminates the task.
2. The task creation information is also deleted, i.e TSK_INFO [OSTCBCur->OSTCBId] is made NULL.

ter_tsk: Terminate Task

This system call forcibly terminates the task specified by tskid. That is, it changes the state of the task specified by tskid into DORMANT. It is similar to ext_tsk except that here the task with specified ID is terminated.

C Language Interface: ER ercd = ter_tsk (ID tskid);

Implementation details:

1.The task is removed from all the existing states by making a call to OSTaskDelete()

2.If act_cnt is greater than zero, task is created again by calling OSTaskCreateExt () with the values stored in TSK_INFO [tskid].

chg_pri: Change Task Priority

This system call changes the current priority of the task specified by tsk_id to the value specified by tsk_pri.

C Language Interface: ER ercd = chg_pri (ID tskid, PRI tskpri) ;

Implementation Details:

1.A Call is made to OSTaskChangePrio( )

get_pri, ref_tsk ref_tst are functions to refer to the task priority, and its state.

5.2.2.Task Dependent Synchronization Functions:

slp_tsk: Sleep Task

tslp_tsk: Sleep Task with Timeout

Both these system calls cause the issuing task (which is in RUN state) to sleep until wup_tsk is invoked.

C Language Interface:

ER ercd = slp_tsk ( );

ER ercd = tslp_tsk (TMO tmout);

Implementation Details:

1.The sleep_flag of the task is set (This flag should be maintained because wup_tsk can only resume a sleeping task, but not any other waiting task).

2. OSTimeDly ( ) is called to delay the task.(incase of slp_tsk this is called in a loop).

wup_tsk: Wakeup Task

This system call releases the WAIT state of the task specified by tsk_id caused by the execution of slp_tsk or tslp_t

C Language Interface: ER ercd = wup_tsk (ID tskid)

Implementation details:

1.if sleep_flag of task is set, call OSTimeDlyResume ();

2.else, return error.

Rel_wai: Release task from waiting.

C language Interface: ER rel_wai (ID tskid)

Implementation Details:

OSTimeDlyResume( ) releases tasks waiting on any events event flags.(in this case the task waiting on the event assumes that it had timed out waiting for the event).

1.OSTimeDlyResume is called

sus_tsk: Suspend Task

Moves the runnable task to suspended state and waiting task to wait-suspended state.

C Language Interface: ER ercd = sus_tsk (ID tskid)

Implementation Details:

1. OSTaskSuspend( ) is called.
2. susp_cnt is incremented by 1.if sus_cnt > max_sus_cnt, error is returned.

This will suffice because in ucos task suspension is additive.

rsm_tsk: Resume Task

frsm_tsk: Forcibly Resume Task

Both these system calls release SUSPEND state of the task specified by tsk_id. Specifically, they cause SUSPEND state to be released and the execution of the specified task to resume when the task has been suspended
by the prior execution of sus_tsk.

C Language Interface

ER ercd = rsm_tsk (ID tskid) ;

ER ercd = frsm_tsk (ID tskid) ;

Implementation details:

Rsm_tsk

1. susp_cnt is Decremented by one.
2. If susp_cnt is greater than zero the function is returns. (TRON requires sus_cnt to be atleast one)
3. Else OSTaskResume( ) is called.

Frsm_tsk( )

1. susp_cnt is cleared.
2. OSTaskResume( ) is called.

dly_tsk: delay calling task

C language Interface: ER ercd= dly_tsk

Implementation Details:

1. OSTimeDly() is called.

Synchronization and communication functions:

Semaphores, Event flags and Data queues and Mail Boxes are all supported.

Extended Synchronization and communication functions:

Mutexes and Message queues are supported.

6. Conclusion:

The aim of the project was to develop a port for support of uItron 4.0 specification API by MicroC/OS-II. Porting an application from MicroC/OS-II, which is a conventional RTOS to an advanced real time operating specification μITRON4.0, involved various issues that were to be considered. Modifying the existing MicroC/OS-II to have multiple tasks with same priority had removed the kernels limitation of being able to support only a limited number of tasks, which was a serious problem when the application involved mutexes also. . The run time of most of the functions still remains independent of the number of tasks created. The resulting (modified) kernel with the port though needed extra storage has not added much overhead to the response time of the kernel

Appendix1

List of ITRON4.0 System Calls

List of μITRON4.0 System Calls		
Task management function	CRE_TSK	Create Task (Static API)
	cre_tsk	Create Task
	acre_tsk	Create Task (ID Number Automatic Assignment)
	del_tsk	Delete Task
	act_tsk	Activate Task
	iact_tsk	Activate Task
	can_act	Cancel Task Activation Requests.
	sta_tsk	Activate Task (with a Start Code)
	ext_tsk	Terminate Invoking Task
	exd_tsk	Terminate and Delete Invoking Task
	ter_tsk	Terminate Task
	chg_pri	Change Task Priority
	get_pri	Reference Task Priority
	ref_tsk	Reference Task State
	ref_tst	Reference Task State (Simplified Version)
Task-Dependent Synchronization Function	slp_tsk	Put Task to Sleep
	tslp_tsk	Put Task to Sleep (with Timeout)
	wup_tsk	Wakeup Task
	iwup_tsk	Wakeup Task with Interrupt Handler
	can_wup	Cancel Task Wakeup requests
	rel_wai	Release Task from Waiting
	irel_wai	Release Task from Waiting with Interrupt Handler
	sus_tsk	Suspend Task
	rsm_tsk	Resume Suspended Task
	frsm_tsk	Forcibly Resume Suspended Task
	dly_tsk	Delay Task
Task Exception Handing Function	DEF_TEX	Define Task Exception Handling Routine (Static API)

		def_tex	Define Task Exception Handling Routine
		ras_tex	Raise Task Exception Handling
		iras_tex	Raise Task Exception Handling with Interrupt Handler
		dis_tex	Disable Task Exceptions
		ena_tex	Enable Task Exceptions
		sns_tex	Reference Task Exception Handling State
		vesns_tex	Reference Task Exception Handling for CPU exception case (TJ original function)
		ref_tex	Reference Task Exception Handling State
Synchronization and communication function	Semaphore	CRE_SEM	Create Semaphore (Static API)
		cre_sem	Create Semaphore
		acre_sem	Create Semaphore (ID Numbre Automatic Assignment)
		del_sem	Delete Semaphore
		sig_sem	Release Semaphore Resource
		isig_sem	Release Semaphore Resource
		wai_sem	Acquire Semaphore Resource
		pol_sem	Acquire Semaphore Resource (Poling)
		twai_sem	Acquire Semaphore Resource (with Timeouta)
		ref_sem	Reference Semaphore State
	Event Flags	CRE_FLG	Create Eventflag (Static API)
		cre_flg	Create Eventflag
		acre_flg	Create Eventflag (ID Number Automatic Assignment)
		del_flg	Delete Eventflag
		set_flg	Set Eventflag
		iset_flg	Set Eventflag
		clr_flg	Clear Eventflag
		wai_flg	Wait for Eventflag
		pol_flg	Wait for Eventflag (Polling)
		twai_flg	Wait for Eventflag (with Timeout)
		ref_flg	Reference Eventflag Status

	Data Queues	CRE_DTQ	Create Data Queue (Static API)
		cre_dtq	Create Data Queue
		acre_dtq	Create Data Queue (ID Number Automatic Assignment)
		del_dtq	Delete Data Queue
		snd_dtq	Send to Data Queue
		psnd_dtq	Send to Data Queue (Polling)
		ipsnd_dtq	Send to Data Queue (Polling)
		tsnd_dtq	Send toData Queue (with Timeout)
		fsnd_dtq	Forcibly Send to Data Queue
		ifsnd_dtq	Forcibly Send to Data Queue
		rcv_dtq	Receive from Data Queue
		prcv_dtq	Receive from Data Queue (Polling)
		trcv_dtq	Receive from Data Queue (with Timeout)
		ref_dtq	Reference Data Queue State
	Mail Boxes	CRE_MBX	Create Mailbox (Static API)
		cre_mbx	Create Mailbox
		acre_mbx	Create Mailbox (ID Number Automatic Assigment)
		del_mbx	Delete Mailbox
		snd_mbx	Send to Mailbox
		isnd_mbx	Send to Mailbox
		rcv_mbx	Receive from Mailbox
		prcv_mbx	Receive from Mailbox (Polling)
		trcv_mbx	Receive from Mailbox (with Timeout)
		ref_mbx	Reference Mailbox State
Extended Synchronization and communication function	Mutexes	CRE_MTX	Create Mutex (Static API)
		cre_mtx	Create Mutex
		acre_mtx	Create Mutex (ID Number Automatic Assigment)
		del_mtx	Delete Mutex
		loc_mtx	Lock Mutex
		ploc_mtx	Lock Mutex (Polling)
		tloc_mtx	Lock Mutex (with Timeout)
		unl_mtx	Unlock Mutex
		ref_mtx	Reference Mutex State

	Message Buffers	CRE_MBF	Create Message Buffer (Static API)
		cre_mbf	Create Message Buffer
		acre_mbf	Create Message Buffer (ID Number Automatic Assignment)
		del_mbf	Delete Message Buffer
		snd_mbf	Send to Message Buffer
		psnd_mbf	Send to Message Buffer (Polling)
		tsnd_mbf	Send to Message Buffer (with Timeout)
		rcv_mbf	Receive from Message Buffer
		prcv_mbf	Receive from Message Buffer (Polling)
		trcv_mbf	Receive from Message Buffer (with Timeout)
		ref_mbf	Reference Message Buffer State
	Rendezvous	CRE_POR	Create Rendezvous Port (Static API)
		cre_por	Create Rendezvous Port
		acre_por	Create Rendezvous Port (ID Number Automatic Assignment)
		del_por	Delete Rendezvous Port
		cal_por	Call Rendezvous
		tcal_por	Call Rendezvous (with Timeout)
		acp_por	Accept Rendezvous
		pacp_por	Accept Rendezvous (Polling)
		tacp_por	Accept Rendezvous (with Timeout)
		fwd_por	Forward Rendezvous
		rpl_rdv	Terminate Rendezvous
		ref_por	Reference Rendezvous Port State
		ref_rdv	Reference Rendezvous State
Memory Pool Management functions	Fixed-Sized Memory Pools	CRE_MPF	Create Fixed-Sized Memory Pool (Static API)
		cre_mpf	Create Fixed-Sized Memory Pool
		acre_mpf	Create Fixed-Sized Memory Pool (ID Number Automatic Assignment)
		del_mpf	Delete Fixed-Sized Memory Pool
		get_mpf	Acquire Fixed-Sized Memory Block
		pget_mpf	Acquire Fixed-Sized Memory Block (Polling)

		ipget_mpf	Acquire Fixed-Sized Memory Block
		tget_mpf	Acquire Fixed-Sized Memory Block (with Timeout)
		rel_mpf	Release Fixed-Sized Memory Block
		ref_mpf	Reference Fixed-Sized Memory Pool State
	Variable-Sized Memory Pools	CRE_MPL	Create Variable-Sized Memory Pool (Static API)
		cre_mpl	Create Variable-Sized Memory Pool
		acre_mpl	Create Variable-Sized Memory Pool (ID Number Automatic Assignment)
		del_mpl	Delete Variable-Sized Memory Pool
		get_mpl	Acquire Variable-Sized Memory Block
		pget_mpl	Acquire Variable-Sized Memory Block (Polling)
		tget_mpl	Acquire Variable-Sized Memory Block (with Timeout)
		rel_mpl	Release Variable-Sized Memory Block
		ref_mpl	Acquire Variable-Sized Memory Pool State
Time Management Functions	System Time Management	set_tim	Set System Time
		get_tim	Reference System Time
		isig_tim	Supply Time Tick
	Cyclic Handlers	CRE_CYC	Create Cyclic Handler (Static API)
		cre_cyc	Create Cyclic Handler
		acre_cyc	Create Cyclic Handler (ID Number Automatic Assignment)
		del_cyc	Delete Cyclic Handler
		sta_cyc	Start Cyclic Handler Operation
		stp_cyc	Stop Cyclic Handler Operation
		ref_cyc	Reference Cyclic Handler State
	Alarm Handler	CRE_ALM	Create Alarm Handler (Static API)
		cre_alm	Create Alarm Handler
		acre_alm	Create Alarm Handler (ID Number Automatic Assignment)
		del_alm	Delete Alarm Handler
		sta_alm	Start Alarm Handler Operation

	stp_alm	Stop Alarm Handler Operation
	ref_alm	Reference Alarm Handler State
System State Management Functions	rot_rdq	Rotate Task Procedence
	irot_rdq	Rotate Task Procedence
	get_tid	Reference Task ID in the RUNNING State
	iget_tid	Reference Task ID in the RUNNING State
	loc_cpu	Lock the CPU
	iloc_cpu	Lock the CPU
	unl_cpu	Unlock the CPU
	iunl_cpu	Unlock the CPU
	dis_dsp	Disable Dispatching
	ena_dsp	Enable Dispatching
	sns_ctx	Reference Contexts
	vesns_ctx	Reference Contexts for CPU exception case (TJ original function)
	sns_dsp	Reference Dispatching State
	vesnd_dsp	Reference Dispatching State for CPU exception case (TJ original function)
	sns_dpn	Reference Dispatch Pending State
	vesnd_dpn	Reference Dispatch Pending State for CPU exception case (TJ original function)
	sns_loc	Reference CPU state
	vesns_loc	Reference CPU state for CPU exception case (TJ original function)
	ref_sys	Reference System State
Interrupt Management Functions	DEF_INH	Define Interrupt Handler (Static API)
	def_inh	Define Interruput Handler
	dis_int	Disable Interrupt
	idis_int	Disable Interrupt
	ena_int	Enable Interrupt
	iena_int	Enable Interrupt
	chg_ims	Change Interrupt Mask
	ichg_ims	Change Interrupt Mask
	get_ims	Reference Interrupt Mask
	iget_ims	Reference Interrupt Mask

	chg_ilv	Change Interrupt Level
	ichg_ilv	Change Interrupt Level
	get_ilv	Reference Interrupt Level
	iget_ilv	Reference Interrupt Level
System Configration Management Functions	DEF_EXC	Define CPU Exception Handler (Static API)
	def_exc	Define CPU Exception Handler
	ref_cfg	Reference Configration Information
	ref_ver	Reference Version Information
	ATT_INI	Attach Initialization Routine (Static API)

Appendix 2:

List of MicroC/OS-II System Calls

Task Management

INT8U OSTaskChangePrio(INT8U oldprio, INT8U newprio);

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

INT8U OSTaskCreateExt(void (*task)(void *pd),

INT8U OSTaskDel(INT8U prio);

INT8U OSTaskDelReq(INT8U prio);

INT8U OSTaskResume(INT8U prio);

INT8U OSTaskSuspend(INT8U prio);

INT8U OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);

INT8U OSTaskQuery(INT8U prio, OS_TCB *pdata);

Semaphores

INT16U OSSemAccept(OS_EVENT *pevent);

OS_EVENT *OSSemCreate(INT16U cnt);

OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U OSSemPost(OS_EVENT *pevent);

INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);

Mutual Exclusion Semaphores

INT8U OSMutexAccept(OS_EVENT *pevent, INT8U *err);

OS_EVENT *OSMutexCreate(INT8U prio, INT8U *err);

OS_EVENT *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err);

void OSMutexPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U OSMutexPost(OS_EVENT *pevent);

INT8U OSMutexQuery(OS_EVENT *pevent, OS_MUTEX_DATA *pdata);

Event Flags

OS_FLAGS OSFlagAccept(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *err);

OS_FLAG_GRP *OSFlagCreate(OS_FLAGS flags, INT8U *err);

OS_FLAG_GRP *OSFlagDel(OS_FLAG_GRP *pgrp, INT8U opt, INT8U *err);

OS_FLAGS OSFlagPend(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *err);

OS_FLAGS OSFlagPost(OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U operation, INT8U *err);

OS_FLAGS OSFlagQuery(OS_FLAG_GRP *pgrp, INT8U *err);

Message Mailboxes

void *OSMboxAccept(OS_EVENT *pevent);

OS_EVENT *OSMboxCreate(void *msg);

OS_EVENT *OSMboxDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U OSMboxPost(OS_EVENT *pevent, void *msg);

INT8U OSMboxPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

INT8U OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);

Message Queues

void *OSQAccept(OS_EVENT *pevent);

OS_EVENT *OSQCreate(void **start, INT16U size);

OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *err);

INT8U OSQFlush(OS_EVENT *pevent);

void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

INT8U OSQPost(OS_EVENT *pevent, void *msg);

INT8U OSQPostFront(OS_EVENT *pevent, void *msg);

INT8U OSQPostOpt(OS_EVENT *pevent, void *msg, INT8U opt);

INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);

Memory Management

OS_MEM *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err);

void *OSMemGet(OS_MEM *pmem, INT8U *err);

INT8U OSMemPut(OS_MEM *pmem, void *pblk);

INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);

Time Management

void OSTimeDly(INT16U ticks);

INT8U OSTimeDlyHMSM(INT8U hr, INT8U min, INT8U sec, INT16U ms);

INT8U OSTimeDlyResume(INT8U prio);

INT32U OSTimeGet(void);

void OSTimeSet(INT32U ticks);

Miscellaneous

void OSInit(void);

```
void OSIntEnter(void);
void OSIntExit(void);
void OSSchedLock(void);
void OSSchedUnlock(void);
void OSStart(void);
void OSStatInit(void);
INT16U OSVersion(void);
```

List of References

1. Jean J. Labrosse, *"MicroC/OS-II The Real-Time Kernel*", Second Edition, CMP Books,2002.

E-books:
1.mitron-400e *(*µITRON4.0 specification document available at http://www.ertl.jp/ITRON/SPEC/FILE/ mitron-400e.pdf *)*

References on web
http://www.ertl.jp/ITRON/SPEC/mitron4-e.html

http://www.micrium.com/