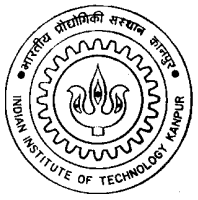
μITRON Interface for a Generic Modular Embedded Operating System Platform

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Sridhar Akula



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

July, 1999

Certificate

This is to certify that the work contained in the thesis entitled "*μITRON Interface for a Generic Modular Embedded Operating System Platform*", by *Sridhar Akula*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

July, 1999

(Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Advances in microprocessor technology continue to open up new application fields for embedded systems. Originally they were used mainly for factory production line control and other industrial applications. Their use spread to communications and office equipment, then on to automotive systems, audio and video products, TVs, cellular phones, synthesizers, game machines, and household appliances such as washing machines, airconditioners and lighting systems. Today nearly all the electrical and electronic products around us are controlled by embedded systems. The growing scale and complexity of software and the need for fast development turnaround time have made improving software productivity a pressing need. The use of an operating system has become increasingly important for this reason.

In this thesis, we design and implement the μITRON interface for a modular embedded operating system platform. ITRON (Industrial - The Real-time Operating system Nucleus) is a real-time, multitasking OS specification intended for use in industrial embedded systems. The implementation supports level E(Extended) of μITRON 3.0, the latest ITRON real-time kernel specification. Operating systems compatible with interfaces like μITRON, POSIX, etc., contribute to improved software productivity, as existing software components and development support tools that are compatible with the standard can be used and it would be easier to train system designers and programmers.

Contents

List of Figures

Chapter 1

Introduction

Microcomputers being small and inexpensive are being used in various devices ranging from automated telephone switching devices, cars, medical equipment, industrial robots, missiles and spaceships to home appliances such as air conditioners, washers, cameras, cell phones and televisions. Often these microcomputers are *embedded* in larger systems and are hence termed as *embedded computer systems* or *embedded systems* in short.

The application programs written for embedded systems are not developed from scratch but rather around an *embedded* operating system(EOS). This makes the applications simple, reliable, portable, and quick to develop. Further, an application can use an existing code written for a different operating system(OS), if both OSes are compatible with one or more industry standards like μITRON[Sak93], POSIX[POS92] etc.

Embedded operating systems differ from general purpose operating systems in the sense that an EOS has to be multi-threaded and preemptible; the notion of thread priority has to exist; predictable thread synchronization mechanisms should be supported; the OS behavior should be known; Interrupt latency (i.e. time from interrupt to task run), the maximum time for each system call, and the maximum time the OS and drivers mask the interrupts should be predictable; and the OS should have extremely small memory and computation requirements.

In this thesis we design and implement a generic modular embedded OS platform compatible with level E (Extended) μITRON 3.0[Sak93] standard. The current implementation, however, doesn't support the connection function required in a

loosely-coupled network of ITRON based machines.

1.1 μITRON 3.0 Specification

ITRON (Industrial - The Real-time Operating system Nucleus) is a real-time, multitasking OS specification intended for use in industrial embedded systems. μITRON 3.0[1] is the latest ITRON real-time kernel specification[Sak93].

1.1.1 Levels of μITRON 3.0 specification

μITRON 3.0 specification is divided into three system call levels: Level R (Required), Level S (Standard) and Level E (Extended). In addition to these three levels, there is also Level C for CPU-dependent system calls.

level R (Required): The functions in this level are mandatory for all implementations of μITRON specification. These functions can be implemented even without a hardware timer.

level S (Standard): This level includes basic functions for achieving a real-time, multitasking OS.

level E (Extended): This level includes several additional and extended functions. Specifically, this level includes object creation and deletion functions, rendezvous functions, memory-pools and the timer handler.

level C (CPU dependent): This level provides implementation-dependent functions required due to the CPU or hardware configuration.

The support level of the connection function is indicated by appending an 'N' to the end of the level. For example, connectivity supported at level S would be referred to as level SN.

The specification levels outlined above indicate the system calls supported at each level. It is sometimes possible, however, to introduce extended features into some system calls for which no compatibility is guaranteed. These functions are called level X functions. These extended features include nested *suspend_task()* calls, queued *wakeup_task()* calls, added information for *refer_task()* etc.

[1] The 'μ' of μITRON is read as *micro*

1.1.2 Compatibility under μITRON 3.0 specification

The following conditions must be met for implementation of μITRON 3.0 specification to be compliant.

1. The implementation must provide at least the three task states: RUN, READY and WAIT.
2. Interrupt handlers may be defined. A method must be available for waking up tasks from an interrupt handler, may be through a system call.
3. All level R and level RN system calls are available. The connection function does not need to be supported. Level RN system calls (such as *signal_semaphore()*) can only be used provided that all the functions are available except for connectivity functions.

1.2 Motivation

The equipments controlled by embedded systems are becoming more and more sophisticated, often incorporating many functions in a single product. Embedded systems have grown in scale and complexity as a result. The growing scale and complexity of software and the need for fast development turnaround time have made improving software productivity a pressing need. The use of C and other high-level languages, along with the use of an Operating System(OS), have become increasingly common for this reason.

1.2.1 Need for a generic and configurable OS

Most often, the OS itself may not be well suited for all types of applications, as the demand of such an OS can vary greatly depending on the scale and nature of the embedded system in which it is used. An OS equipped with advanced functions that are of little use in a small-scale embedded system, where memory in particular is severely limited, will only increase the size of the system and lower its performance. Specifically, the OS carries extra baggage not needed by the application. Thus there is a need for an OS which is generic enough, to suite both large-scale and small-scale embedded systems, and configurable such that it would not carry any extra baggage not needed by the application.

1.2.2 Need for OS Compatibility with an industrial standard

OSes compatible with standard interfaces like μITRON[Sak93], POSIX[POS92], etc., contribute to improved software productivity, as existing software components and development support tools can be used, that are compatible with the standard. Moreover, the expanding application of embedded systems means that an increasing number of software engineers are coming into contact with an embedded OS, making it highly important to train system designers and programmers in the requisite skills. The training becomes costlier, due to the large differences in specifications from one OS to another. Thus there is a need for the OS intended for embedded OS development to be compatible with an industrial standard.

In this thesis, our motivation has been to come up with an OS design that is reliable, generic, highly configurable, and modular. The OS provides a compatible interface with an industrial standard. Moreover, it is a collection of modules, which can be plugged in or out as per the needs of the application. It is hence possible to support any standard API with no extra cost.

1.3 Existing Operating Systems

There are many embedded kernels available[ES] today, most of them being highly configurable and powerful, are only targeted to applications of a fixed range of complexities. We present here, a brief description[2] of some of these operating systems.

1.3.1 CRTX Real-time Micro Kernel

CRTX[CRT] is a compact, simple to use, high performance off-the-shelf Micro Kernel intended for use in small 8 and 16 bit embedded applications. CRTX was designed primarily for portability. However, it doesn't offer a standard API.

[2] Any comparisons made here are purely my personal beliefs based on the information I could gather. They are presented here to bring out the specific areas in which these proprietary systems lag and are not intended to jeopardize anybody's reputation.

1.3.2 OS/9 (Microware Systems Corporation)

OS-9[OS/] is a real-time, multiuser, multitasking operating system. It's modular architecture allows individual modules to be included or deleted in the operating system during configuration for a specific application. This modularity makes OS/9 extremely scalable and powerful to fit most application needs. However, it doesn't support any standard API.

1.3.3 Harmony (National Research Council of Canada)

Harmony[oCH] is a multitasking, multiprocessing operating system for realtime control, developed at the National Research Council Laboratories. Harmony is a portable, extensible and configurable system. Primarily developed for realtime control of robotics experiments, for the development of experimental robot controllers and for other applications of embedded systems where predictable temporal performance is a requirement, Harmony fulfills the needs of only a fixed range of embedded applications. However, Harmony doesn't provide a standard API and comes with huge overhead for small-scale embedded applications.

1.3.4 RTEMS (Redstone Military Arsenal)

Real-Time Executive for Multiprocessor Systems (RTEMS)[RTE] is a non-commercial real-time operating system for embedded computer systems. RTEMS implementations are available in either the Ada or C programming language providing functionality equivalent to that of commercial products. RTEMS is designed for the best use in military control devices. The POSIX standard API is being added to RTEMS 3.6.0.

1.3.5 VxWorks (Wind River Systems)

VxWorks[VxW] offers a development and execution environment for complex real-time and embedded applications on a wide variety of target processors. It supports POSIX 1003.1b[POS92] real-time extensions, ANSI C (including floating point support) and complete TCP/IP networking across various media. VxWorks has been used in many successful projects including the Mars *path-finder*, Virtual Reality, and Traffic control.

Due to the reason that it is targeted mostly towards solving large and complex problems, it is rather unsuitable for small-scale embedded systems, where most of its extended features may not be used.

1.4 Organization of the Report

The rest of the thesis is organized as follows. In chapter 2, we discuss the overall system design at block level and the relationships between different blocks. We also give a brief overview of the blocks, that have been already implemented. We discuss the design and implementation of the μITRON Task Manager Module in chapter 3, the design and implementation of the μITRON Communication Manager Module in chapter 4, the design and implementation of the μITRON Interrupt Manager Module in chapter 5, the design and implementation of the μITRON Memory Manager Module in chapter 6, the design and implementation of the μITRON Time Manager Module in chapter 7, the design and implementation of the μITRON System Manager Module in chapter 8. In chapter 9, we discuss an example application using this OS. This application is a car dashboard controller for which design and implementation details are given in this chapter. Finally, we conclude this thesis in chapter 10 and suggest some future extensions.

Chapter 2

The System Design

Typical embedded system applications range from very simple ones, which need minimal services from the kernel, to complex ones that need highly extended and sophisticated services. Also, an application might need the kernel to be compatible with any of the industrial standards like μITRON[Sak93], POSIX[POS92] etc. The design of our system is aimed to fulfill the requirements of all such applications.

2.1 The Problem

To meet the specific requirements of an embedded application, ideally, the kernel should offer exactly only those services that are needed by the application, and should carry no extra code. In practice, this can be achieved by constructing modules, each module offering a particular class of service needed by the application, and the kernel being able to *hold* only those modules that are needed by the application. This focuses on the need to define a well defined intermodular interface, such that new modules can be developed and added to the existing system with ease.

2.2 The Solution

The solution suggested in this thesis is to build a *modular* kernel. The modular architecture allows the kernel to be modified and configured to meet the specific needs of an application. Specifically, modules can be added or subtracted from the system as per the application need. The result is a highly modular and reconfigurable

platform.

The kernel consists of two parts: the set of modules, or the configurable portion, which offer various services to the application, and the compulsory portion, called the *nucleus*, or the *nano-kernel*, which acts as a *glue* between the modules. Typical modules range from physical memory manager, interrupt handler, thread manager, etc. to μITRON memory manager[1], μITRON timer manager, μITRON communication manager, etc. Thus, there could be an application using a bare interrupt handler, a μITRON timer manager, and a POSIX thread manager[2].

2.2.1 Nucleus Design

The specified design results in the nucleus code being pretty small, as it carries only the start-up code and the code needed to interface one module with another. Specifically, its functionality includes initialization of the kernel modules at system start-up and communication among different modules, and between the application and the kernel modules. The nucleus was developed by Kshitiz Krishna[Kri97] along with various other modules.

2.2.2 Modules' Design

Each module is designed such that, it makes very little or no assumptions about the existence of other modules. However, a module can use the services of one of more existing modules, with the help of the *nanokernel*. The modules being used are called the *pre-requisites* of the module that uses them. For instance, a module like μITRON task manager, could be using the services of the physical memory manager module and the thread manager module.

Pre-requisites can be either *hard* or *soft*. A hard pre-requisite module must be present and initialized before the module that uses it. In case of soft pre-requisites, the pre-requisite module may or may not be present. If it is not present, nano-kernel provides it's own service (a null function) but in case the module is present, then it must be initialized before this module.

[1] μITRON memory manager refers to the module compatible with the μITRON industrial standard[Sak93], that offers memory management functions

[2] POSIX interrupt handler refers to the module compatible with the POSIX industrial standard[POS92], that offers interrupt handling functions

A brief description of the modules developed prior to this thesis, namely, the interrupt manager module and the thread manager module, are given in Appendix B.

2.2.3 Intermodule Interface

The design standardizes the intermodule interface[Kri97] to be used for all service requests and results between various modules, and the application. Communication between the modules/application is achieved by using the *module-id*s, or ports[3]. A module or an application willing to communicate with another module would request the nucleus to *give* the address of the service providing routine of that module, mentioning the destination module's id, or port.

2.2.4 Application Program Interface(API) Design

The modular design of the kernel facilitates new modules to be *plugged-in* without any changes to the existing modules or the nucleus. The job of providing an API to the existing kernel is to just add a new API module, or a set of API modules, which make use of the services of the existing modules. For instance, to provide μITRON standard API to the existing kernel, one can either develop a single μITRON API module, or a set of modules like μITRON memory manager, μITRON task manager, μITRON interrupt manager, etc. which make use of existing modules like physical memory manager, bare interrupt handler, thread manager, etc. The latter approach is better, since, all the modules are *pluggable* and the application can choose only those modules that are needed.

In this thesis, we add μITRON API for the existing kernel. Specifically, various functionally distinguished μITRON modules are designed and implemented to provide an application with μITRON compatible interface. These modules use the services of the existing modules, which provide the basic services, and are hence very small and conserve space as the application in most cases has to be put in a ROM.

[3] Though these are not exactly ports, the term is being used on the basis of closest similarity.

2.3 The Kernel Architecture

The block diagram of the system is given in figure 2.1. The nucleus is the compulsory portion of the kernel, and all the modules, including the API modules form the configurable portion. Any of these modules may be added or removed from the system, as per the application requirements, at the system build time.

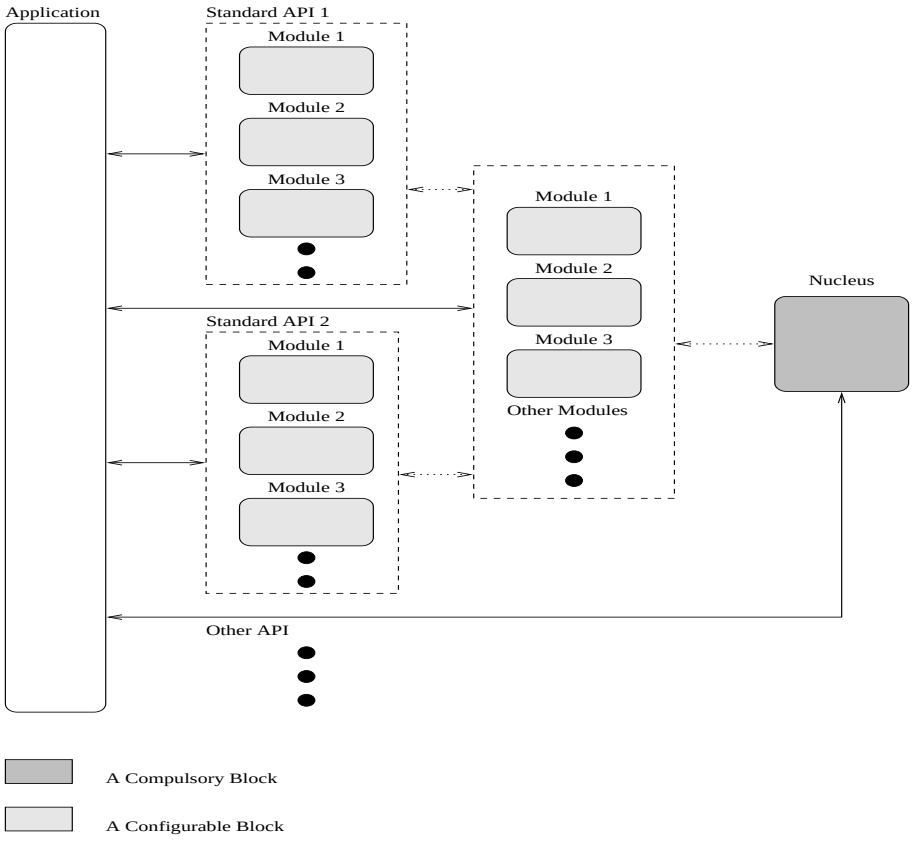


Figure 2.1: Block Diagram of the System

At system startup, the nucleus initializes the modules, one by one, informing each of them, it's own service handler's address. It will then pass the control of execution to the application, informing it again, the address of the nucleus service provider. From then onwards, the application can query the nucleus for the address of any module's service handler and use the corresponding module services. Modules requiring services from other modules go through the same procedure.

In this chapter, the overall design of the system has been presented. The kernel

architecture has been shown at the block level and the design of individual blocks has been presented. Specifically, the relationship between the nucleus, the modules, and the application has been described. A brief overview of the existing work and the work done in this thesis has been presented.

Chapter 3

μITRON Task Manager Module

The μITRON Task Manager Module provides *task management* functions,(to access and control the state of tasks), *task-dependent synchronization* functions,(which include functions that suspend tasks for a while, and associated functions that release a task from the SUSPEND state), and the inter-task synchronization functions. This Module uses the services of the Thread Manager Module[Section B.2] for task management and the services of the Memory Manager Module for memory allocation and deallocation for each task.

3.1 The Design

A *thread* is an independent flow of execution. While the nano-kernel supports a single thread of execution, the thread manager provides multiple threads executing concurrently in the system, which are transparent to nano-kernel and other modules. The μITRON task manager module uses the services provided by the thread manager module while supporting multiple tasks, each task corresponding to a unique thread of the thread manager module. Task scheduling is governed by the task manager by requesting the thread manager to use the task manager's scheduling function to schedule threads. This is done by having the address of the task manager's scheduling function sent to the thread manager during the initialization phase of the task manager module. Hence, the task manager manages the task states; maintains it's own *ready queue* to guide the thread manager through the μITRON specified scheduling policies and provides it's own task-dependent synchronization functions.

3.1.1 Basic Task Management

Tasks can be created and destroyed dynamically. Creation of a task involves a request sent to the Memory Manager Module for allocation of memory specified by the *stack size*. A newly created task is not started unless an explicit call is made to do so. This call, *sta_tsk()*, requests the Thread Manager Module to create a new thread and *mark* it READY. When a task exits, it enters the DORMANT state and the corresponding thread is killed, but the allocated stack area is still held. This task can be re-started by issuing the *sta_tsk()* system call. The stack space is released only when the task is deleted.

3.1.2 Task Scheduling

Task scheduling is conducted based on task priority. If there are multiple tasks of the same priority, scheduling is conducted on *first come, first served (FCFS)* basis. It is, however, possible to dynamically change the priority of a task. On occurrence of an external interrupt, tasks are re-scheduled after the interrupt handler is finished. The task which was executing gets back its execution privileges, unless a task of higher priority was brought to READY state by the interrupt handler.

3.1.3 Task Dependent Synchronization

A task can *sleep* or it can *suspend*, *resume* or *wake-up* other tasks. If the task specified to be suspended is already in WAIT state, it will be put in the combined WAIT-SUSPEND state. If wait conditions for the task are later fulfilled, it will enter SUSPEND state. If the task is resumed, it will return to the WAIT state before the suspension. It is possible to nest the pairs of suspension and resumption requests i.e., a task must be resumed the same number of times for which it was suspended in order to return the task to its original state before the suspension. A task can sleep with timeout, or forever, until another task issues a wakeup call. Wakeup calls can be queued, i.e., a wakeup call could be issued to a task that is not sleeping, and the request will be queued, unless the queue is full.

3.2 Module Prerequisites

- **Hard Prerequisites:** Memory Manager Module, Thread Manager Module. The Memory Manager Module services are used each time a task is created or deleted, to allocate or deallocate the stack space. The Thread Manager services are used for the basic task management.
- **Soft Prerequisites:** None

3.3 Module configuration parameters

The module should be provided with the following static configuration parameters.

1. The maximum number of user tasks.
2. The maximum number of times suspend requests may be nested.
3. The maximum allowable number for the wakeup request queuing count.

3.4 Services provided by the Task Manager

The module provides Task Management functions and Task-Dependent Synchronization functions, as specified by the μITRON specifications. It also provides services to be used by other μITRON modules that deal with task management.

3.4.1 Task Management Functions

- **Create Task** This system call creates the task specified by *task-id*. Specifically, a TCB (Task Control Block) is allocated for the task to be created, and initialized according to accompanying parameter values of *task priority, task handler, stack size* etc. A stack area is also allocated for the task based on the parameter *stack size*.
- **Delete Task** This system call deletes the task specified by *task-id*. Specifically, it changes the state of the task specified by *task-id* from DORMANT into NON-EXISTENT, and then clears the TCB and releases stack.

- **Start Task** This system call starts the task specified by *task-id*. Specifically, it changes the state of the task specified by *task-id* from DORMANT into RUN/READY.

- **Exit Task** This system call causes the issuing task to exit, changing the state of the task into the DORMANT state.

- **Exit and Delete Task** This system call causes the issuing task to exit and then delete itself.

- **Terminate Other Task** This system call forcibly terminates the task specified by *task-id*. That is, it changes the state of the task specified by *task-id* into DORMANT.

- **Disable Dispatch** This system call disables task dispatching. Dispatching will remain disabled after this call is issued until a subsequent call to *ena_dsp()* is issued.

- **Enable Dispatch** This system call enables task dispatching, that is, it finishes dispatch disabled state caused by the execution of *dis_dsp()*.

- **Change Task Priority** This system call changes the current priority of the task specified by *task-id* to the value specified by *task priority*.

- **Rotate Tasks on the Ready Queue** This system call rotates tasks on the ready queue associated with the priority level specified by *task priority*. Specifically, the task at the head of the ready queue of the priority level in question is moved to the end of the ready queue, thus switching the execution of tasks having the same priority.

- **Release Wait of Other Task** This system call forcibly releases WAIT state (not including SUSPEND state) of the task specified by *task-id*.

- **Get Task Identifier** This system call gets the ID of the issuing task.

- **Reference Task Status** This system call refers to the state of the task specified by *task-id*, and returns its current priority, its task state, and its extended information.

3.4.2 Task-Dependent Synchronization Functions

- **Suspend Other Task** This system call suspends the execution of the task specified by *task-id* by putting it into SUSPEND state.
- **Resume Suspended Task / Forcibly Resume Suspended Task** Both these system calls release SUSPEND state of the task specified by *task-id*. *Rsm_tsk()* only releases one suspend request from the suspend request nest. Accordingly, if more than one *sus_tsk()* has been issued on the task in question, that task will remain suspended even after the execution of *rsm_tsk()* is completed. In contrast, *frsm_tsk()* will clear all suspend requests even if more than one *sus_tsk* has been issued on the same task.
- **Sleep Task / Sleep Task with Timeout** Both these system calls cause the issuing task (which is in RUN state) to sleep until *wup_tsk()* is invoked.
- **Wakeup Other Task** This system call releases the WAIT state of the task specified by *task-id* caused by the execution of *slp_tsk()* or *tslp_tsk()*.
- **Cancel Wakeup Request** This system call returns the wakeup request queuing count for the task specified by *task-id* while canceling all associated wakeup requests. Specifically, it resets the wakeup request queuing count to 0.

3.4.3 Services provided for other modules

- **Query/Update system status** This system call can be used to check or update the current system status flags, which indicate the current system execution to be in *dispatch disabled* state, or in *task independent portion* or in *CPU locked* state.
- **Release wait state** This system call releases the wait state of the task specified by *task-id* and returns it an error value specified.
- **Wait and release control** This system call makes the current running task wait and release control.

3.5 Implementation

3.5.1 Module Initialization

At initialization the task manager module sets-up the TCB (task control block) tables and initializes the ready queue. It marks the current thread of execution to be the first user task running, and schedules it. It *queries* the nano-kernel for the service handlers of the physical memory manager module and the thread manager module. It then intimates the thread manager, the address of the `next_to_schedule()` function to be used in future. Further, it requests the thread manager to *unhook* the thread manager's `check_sleepers()` function from the timer interrupt and *hook* the task manager's `check_sleepers()` function.

3.5.2 Data Structures

The fields of the TCB are given below.

```
typedef struct task_control_block {
    VP  exinf;    /* Extended Information */
    ATR tskatr;   /* Task Attribute */
    PRI itskpri;  /* Initial Task Priority */
    PRI priority; /* Current Task Priority */
    FP  task;     /* Task Start Address */
    INT state;    /* Current Task State */
    INT suspend_count;  /* Number of pending suspend requests */
    INT wakeup_count;   /* Number of pending wake-up requests */
    UINT cause_of_wait; /* Cause of wait, if the task is waiting */
    TMO  timeout_count; /* Wait-time left if the task is waiting */
    int_2b thread_id;   /* The task's corresponding thread-id */
    MEM_BLK_ID_TYPE stack_id; /* Id of memory allocated for stack */
} TCB;
```

The pointer `exinf` is provided for the user to include extended information about the task to be created. `tskatr` specifies whether the task is written in assembly or high level language.

The user tasks and system tasks are maintained using two static arrays, both of type `TCB` and of sizes `max_usr_tasks` (known at system configuration time) and `max_sys_tsks` (system known) respectively. These tables are indexed by the `task-id` for efficient retrieval of information.

The ready queue is a singly-linked list simulated on an array of size `(max_usr_tasks + max_sys_tsks)`. The linked list representation is needed for efficient dispatching (the task at the head of the list is the one to be scheduled next), and during insertion/deletion operations, as tasks frequently enter/leave the ready queue, which is to be maintained sorted on task priority.

The boolean variables `dispatch_enabled`, `task_independent_portion_entered` and `CPU_locked` represent the current system state.

3.5.3 Task Management and Scheduling

The thread manager's scheduler, which is hooked to the timer interrupt, calls the `next_to_schedule()` function of the task manager. This function finds the task to be scheduled next and returns the corresponding thread id. The `check_sleepers()` function of the task manager decrements the `timeout_count` of all tasks that are waiting with timeout, by one and once the counter reaches zero, it releases the wait state for the corresponding task and returns it an error value `E_TMOUT`, through `timeout_count`.

3.5.4 μITRON Implementation-dependent Specifications for Task Management

- **Specification:** *It is implementation dependent whether or not SUSPEND state, WAIT-SUSPEND state, DORMANT state, and NON-EXISTENT state are supported.*
  Implementation: All these states are supported, along with the mandatory ones, RUN, READY, and WAIT.

- **Specification:** *Priority levels outside the range 1 to 8 (including negative values) may also be specified depending on the implementation.*
  Implementation: Priority levels outside the range 1 to 8 are not allowed.

- **Specification:** *Depending on the implementation, specifying tskpri =* TPRI_INI *may cause a task's priority to be reset to the initial task priority which was defined when it was first created.*
  Implementation: This feature is supported.

- **Specification:** *It is implementation dependent where a task which has been forced to enter SUSPEND state and is later resumed by the rsm_tsk system call will enter the ready queue among tasks of corresponding priority.*
  Implementation: Such a task is placed to the end of the ready queue among the tasks of the same priority. Since the suspended tasks are removed from the ready queue, for better dispatching performance in cases where many suspended tasks can be expected, maintaining the scheduling order for suspended tasks would be inefficient.

- **Specification:** *It is implementation dependent whether rotation of the ready queue including the running task is supported.*
  Implementation: This feature is supported and can be used by the user, for instance, to implement round robin scheduling of same priority tasks when this method is preferred over FCFS scheduling. Specifically, round robin scheduling may be implemented by using an interrupt handler invoked by timer interrupt periodically, or using a cyclic handler, to issue the *rot_rdq* system call.

- **Specification:** *The maximum number of times suspend requests may be nested, and even whether or not suspend request nesting (the ability to issue sus_tsk on the same task more than once) is even allowed, is implementation dependent.*
  Implementation: Suspend request nesting is allowed and the maximum number of suspend requests is known at the time of system configuration.

- **Specification:** *It is always possible to queue at least one wakeup request. The maximum allowable number for the wakeup request queuing count is implementation dependent, and may be any number higher than or equal to one.*
  Implementation: Queuing more than one wakeup requests is supported and the maximum number of wakeup requests that can be queued is known at the time of system configuration.

Chapter 4

Communication Manager

The μITRON Communication Manager supports task synchronization, mutual exclusion, and communication functions. These functions are completely independent of tasks and include semaphores, eventflags and mailboxes.

4.1 The Design

The module uses the services of the Task Manager Module to deal with the task WAIT states. Specifically, the μITRON task manager's service, *release wait state*, is used to release the wait state of tasks waiting for an event, and the service, *wait and release control*, is used to make the current task wait on an event. These services are briefly described in section 3.4.3.

The module uses the services of the Memory Manager Module during inter-task communication, to allocate/deallocate memory for the buffers. Specifically, memory is allocated or de-allocated when a mailbox is created or deleted, respectively.

4.1.1 Semaphores

The module provides a generic semaphore scheme to provide mutual exclusion and synchronized access to resources. The task notifying other tasks of an event increases the number of resources held by the semaphore by one, and the task waiting for the event decreases the number of resources held by the semaphore by one. If the number of resources held by a semaphore is 0, the task requiring resources will wait until the

next time resources are returned to the semaphore. If there is more than one task waiting for a semaphore, the tasks will be placed in the queue.

4.1.2 Eventflags

The module provides eventflags for task synchronization which use bit correspondence flags to represent the availability of events. A task notifying other tasks of an event can set and clear certain bits of the associated eventflag. Tasks waiting for the event will continue to wait until certain conditions, represented by the bit pattern of the eventflag, have been met.

4.1.3 Mailboxes

Mailboxes provide both task synchronization and communication by passing messages. The task notifying other tasks of an event (the task which are sending a message) can place messages on a message queue. Tasks waiting for the event (tasks which will receive the message) can retrieve messages from the message queue. If there is no message on the message queue yet, the task will wait until the next message arrives. If there is more than one task waiting for a message the tasks will be placed in a queue. The contents of the messages are in shared memory and only the corresponding first address is actually sent or received. The message contents are not copied.

4.1.4 Level X functions supported by the module

- The initial semaphore count and the maximum semaphore count can be specified at the time of semaphore creation.
- The ordering of tasks based on task priority level, on a semaphore's/a mailbox's queue is supported.
- Multiple tasks can wait at the same time for the same eventflag. Hence, a single *set_flg()* system call could result in the release of multiple waiting tasks.
- When a semaphore or an eventflag or a mailbox is *referred*, the system call returns the *task-id* at the head of the waiting queue (along with the other

information), rather than just returning a boolean value stating whether some task is waiting.

- Priority-ordered queuing of messages in a mailbox buffer is supported.

4.2 Module Prerequisites

- **Hard Prerequisites:** Memory Manager Module, μITRON Task Manager Module. The Memory Manager Module services are used to allocate/deallocate memory for the buffers used by mailboxes. The μITRON Task Manager Module services are used to deal with the task WAIT states.
- **Soft Prerequisites:** None

4.3 Module Configuration Parameters

The module needs to be provided with the following static configuration parameters:

1. The maximum number of user semaphores.
2. The maximum number of user event flags.
3. The maximum number of user mailboxes.

4.4 Module Services

- **Create Semaphore** This system call creates the semaphore specified by *semid*.
- **Delete Semaphore** This system call deletes the semaphore specified by *semid*.
- **Signal Semaphore** This system call returns one resource to the semaphore specified by *semid*. Specifically, if there are any tasks waiting for the specified semaphore, the task at the head of the queue becomes READY. if there are no tasks waiting, the associated semaphore count is incremented by one.
- **Wait on Semaphore** This system call obtains one resource from the semaphore specified by *semid*. Specifically, if the semaphore count is one or greater, it is

decremented by one and the issuing task continues to execute. If the count is 0, the issuing task will enter the WAIT state and will be put in the queue associated with the specified semaphore. This function can be used with timeout.

- **Reference Semaphore Status** This system call refers to the state of the semaphore specified by *semid*, and returns its current semaphore count, waiting task information, and its extended information.

- **Create Eventflag** This system call creates the eventflag specified by *flgid* and initializes the associated flag pattern.

- **Delete Eventflag** This system call deletes the eventflag specified by *flgid*.

- **Set Eventflag** This system call sets the bits specified by *setptn* of the one word eventflag specified by *flgid*.

- **Clear Eventflag** This system call clears the bits of the one word eventflag based on the corresponding zero bits of *clrptn*.

- **Wait for Eventflag** This system call waits for the eventflag specified by *flgid* to be set to satisfy the wait release condition specified by *wfmode*. This function can be used with timeout.

- **Reference Eventflag Status** This system call refers to the state of the eventflag specified by *flgid*, and returns its current flag pattern, waiting task information, and its extended information.

- **Create Mailbox** This system call creates the mailbox specified by *mbxid*. A buffer area of size *bufcnt* is also allocated for the mailbox.

- **Delete Mailbox** This system call deletes the mailbox specified by *mbxid*.

- **Send Message to Mailbox** This system call sends the message packet whose start address is given by *pk_msg* to the mailbox specified by *mbxid*.

- **Receive Message from Mailbox** This system call receives a message from the mailbox specified by *mbxid*. If there is no message in the specified mailbox the issuing task will enter the WAIT state, and be put on the queue for waiting for arriving messages. This function can be used with timeout.

- **Reference Mailbox Status** This system call refers to the state of the mailbox specified by *mbxid*, and returns the next message to be received, waiting task information, and its extended information.

4.5 Implementation

While the control information for managing semaphores, eventflags and mailboxes are statically allocated tables, the message queues used by the mailboxes, however, use dynamic allocation of memory. Message queues are implemented as ring buffers. Depending the buffer size specified during mailbox creation, memory is allocated for the ring buffer and the buffer data structures are set-up. With the message queue implemented as a ring buffer, the task issuing the *snd_msg()* call will not enter a WAIT state even if the message cannot be queued because the ring buffer is full. An *E_QOVR* error will be returned immediately to the issuing task if this situation arises.

To maintain wait queues, for instance, the one used by semaphores, a single wait queue is statically allocated of size `(max_no_of_semaphores * max_no_of_tasks)`, and each semaphore dynamically *picks up* individual nodes from this pool, as needed.

4.5.1 Module Initialization

At initialization the μITRON communication manager *queries* the nano-kernel for the service handlers of the physical memory manager module and the thread manager module. It then sets-up the control tables and initializes the wait queues for managing the semaphores, eventflags, and the mailboxes.

4.5.2 Data Structures

```
typedef struct semaphore_node {
VP exinf; /* Extended Infromation */
ATR sematr; /* Semaphore Attributes */
INT semcnt; /* Semaphore Count */
INT maxsem; /* Max Semaphore Count */
BOOL status; /* States whether the node is valid or not */
```

```
} semaphore_node;
```

The semaphore attributes specify the manner in which waiting tasks are put on the semaphore's queue.

```
typedef struct eventflag_node {
VP exinf; /* Extended Infromation */
ATR flgatr; /* Event Flag Attributes */
UINT flgptn; /* Event Flag Pattern */
BOOL status; /* States whether the node is valid or not */
} eventflag_node;
```

The eventflag attributes specify whether multiple tasks waiting on the eventflag is allowed.

```
typedef struct mailbox_node {
VP exinf; /* Extended Information */
ATR mbxatr; /* Mailbox Attributes */
T_MSG* buffer; /* Buffer to hold messages */
INT bufcnt; /* Buffer Message Count */
INT head, tail; /* Positions in the buffer where messages are
                   deleted and inserted, respectively */
INT msg_count; /* # of msgs currently stored in the buffer */
BOOL status; /* States whether the node is valid or not */
} mailbox_node;
```

The mailbox attributes specify the manner in which tasks receiving messages are put on the mailbox's queue and the manner in which messages are put on the message queue (the buffer).

4.5.3 μITRON Implementation-dependent Specifications

- **Specification:** *When a semaphore/an eventflag/a mailbox being waited for by more than one tasks is deleted, the order of tasks on the ready queue after the WAIT state is cleared is implementation dependent in the case of tasks having*

the same priority.

Implementation: Tasks with same priority enter the ready queue in the same order as they were earlier in the semaphore/eventflag/mailbox wait queue.

Chapter 5

μITRON Interrupt Manager Module

The μITRON Interrupt Manager Module services include defining interrupt handlers and disabling/enabling of external interrupts and task dispatching.

5.1 The Design

Interrupt handlers are considered task-independent portions. Task switching (dispatching) is not performed while a task-independent portion is executing and even if the result of a system call issued inside a task-independent portion is a dispatching request, that dispatching is delayed until the control leaves the task-independent portion. This is called *delayed dispatching*.

The module leaves the interrupt handling strategies to the bare interrupt manager module, which is its hard prerequisite. Thus the module provides the μITRON interrupt management functions independent of the interrupt handling strategies that could be provided by various interrupt management modules, of which, any of them could be plugged into the system as per the application need.

The module offers a special service to *lock* the CPU. Basically this system call disables external interrupts and task dispatching. All possibility that a task might be preempted (have its CPU privileges taken away) by an interrupt handler or another task is suppressed. This system call internally informs the μITRON task manager module that the CPU has been locked, and the status of the system is updated

(CPU locked). The system status at any instant can be obtained by the *Reference_System_Status()* system call, provided by the μITRON System Manager Module.

5.2 Module Prerequisites

- **Hard Prerequisites:** Interrupt Manager Module.
  The Interrupt Manger Module services are used to define interrupt handlers.
- **Soft Prerequisites:** μITRON Task Manger Module.
  The μITRON Task Manager Module services are used to inform the task manager if CPU has been locked/unlocked and to wakeup tasks.

5.3 Configuration Parameters

The module does not have any static configuration parameters.

5.4 Services provided by the Interrupt Manager

The μITRON interrupt manager module provides the following services.

- **Define Interrupt Handler** This system call defines an interrupt handler for the given interrupt number and makes that interrupt handler ready to use.
- **Return from Interrupt Handler** This system call causes the invoked interrupt handler to finish.
- **Return and Wakeup Task** This system call releases the SLEEP state of the task specified and causes the issuing interrupt handler to finish.
- **Lock CPU** This system call disables external interrupt and task dispatching.
- **Unlock CPU** This system call enables external interrupt and task dispatching.
- **Disable Interrupt** This system call disables the interrupt, specified by the interrupt number.

- **Enable Interrupt** This system call enables the interrupt, specified by the interrupt number.
- **Change Interrupt Mask** This system call changes the interrupt mask to that specified.
- **Reference Interrupt Mask** This system call returns the interrupt mask.

5.5 Implementation

At initialization, the μITRON interrupt manager module *queries* the nano-kernel for the service handlers of the bare interrupt handler module and the μITRON task manager module.

The module uses the service, *add_handler()* of the interrupt handler module to define a new interrupt handler for the given interrupt number.

```
add_handler (intr_no, handler_priority, handler_addr);
```

The *add_handler()* system call adds the given service routine to the list of handlers, sorted by handler priority, corresponding to the interrupt number. The μITRON interrupt manager module does not use the service handlers' priority feature provided by the interrupt Manager Module, but instead specifies a default priority.

The module uses the service *update_system_status()* of the μITRON Task Manager Module to update the system status, when the CPU is locked or unlocked. The μITRON System Manager Module gets the system status information from the μITRON Task Manager Module, using the *Reference_System_Status()* system call.

Chapter 6

μITRON Memory Manager Module

The μITRON Memory Manager Module offers services to manage memorypools and to allocate/deallocate memory blocks from the memory pools. There are two types of memorypools: fixed-size memorypools and variable-size memorypools. Both are considered separate objects and require different system calls for manipulation. While the size of memory blocks allocated from fixed-size memorypools are all fixed, blocks of any size may be specified when allocating memory blocks from variable-size memorypools.

6.1 The Design

The μITRON Memory Manager Module uses the services of the Physical Memory Manager Module, to allocate or deallocate memory for the memorypools. The μITRON Memory Manger then manages the pool memory to allocate and deallocate blocks of memory to various tasks.

If the memory block cannot be issued to the requesting task, due to lack of memory in the memorypool, the task is placed on the memory allocation queue of the specified memorypool, and will wait until it can get the memory it requires. The manner in which tasks are put into this queue can be specified to be either FIFO-ordered or Task Priority ordered, at the time of creation of the memorypool. However, a task requesting for a memory block can always poll for memory in the memory pool or

wait for memory, with timeout.

The μITRON Memory Manager uses the services of the μITRON Task Manager Module either to make the current task wait, or to release the wait state of a task waiting for memory. The module makes the current task wait, if the request can not be satisfied as there is not enough memory in the memorypool. The module releases the wait state of a task either when its memory request is satisfied successfully, or if the memorypool it is waiting on, is deleted. If a task waits with timeout, for a memory block, its wait state is released directly by the μITRON Task Manager Module.

6.1.1 Level X functions supported by the module

- The placement of tasks on the memorypool wait queue based on the task priority level is supported.

6.2 Module Prerequisites

- **Hard Prerequisites:** Physical Memory Manager Module.
  The Physical Memory Manager Module services are used to allocate and de-allocate memory for the memorypools at the time of their creation and deletion respectively.

- **Soft Prerequisites:** μITRON Task Manager Module.
  The μITRON Task Manager Module services are used to make the current task wait (and release control) when it is going to wait for memory, and to release the wait state of a waiting task which was waiting for memory.

6.3 Configuration Parameters

The module needs to be provided with the following static configuration parameters:

1. The maximum number of fixed size memorypools used by the application.

2. The maximum number of variable size memorypools used by the application.

6.4 Services provided by the Memory Manager

The module provides the following services to the application and the other modules.

- **Create Variable-Size Memorypool** This system call creates a variable-size memorypool of the given size.
- **Delete Variable-Size Memorypool** This system call deletes the specified variable-size memorypool.
- **Get Variable-Size Memory Block** A memory block of the given size is allocated from the specified variable-size memorypool. The system call supports both polling and timeout features.
- **Release Variable-Size Memory Block** This system call releases the specified memory block to the variable-size memorypool.
- **Reference Variable-Size Memorypool Status** This system call refers to the state of the specified variable-size memorypool, and returns the total and the maximum continuous free memory available, waiting task information, and its extended information.
- **Create Fixed-Size Memorypool** This system call creates a fixed-size memorypool of size given by memory block size and memory pool block count (the maximum number of blocks that can be allocated from the pool at a time).
- **Delete Fixed-Size Memorypool** This system call deletes the specified fixed-size memorypool.
- **Get Fixed-Size Memory Block** A fixed size memory block is allocated from the specified fixed-size memorypool. The system call supports both polling and timeout features.
- **Release Fixed-Size Memory Block** This system call releases the specified memory block to the fixed-size memorypool.
- **Reference Fixed-Size Memorypool Status** This system call refers to the state of the specified fixed-size memorypool and returns the current number of free blocks, waiting task information, and its extended information.

6.5 Implementation

6.5.1 Module Initialization

At initialization the μITRON Memory Manager Module sets-up the memory pool tables and initializes their wait queues. It *queries* the nano-kernel for the service handlers of the Physical Memory Manager Module and the μITRON Task Manager Module.

6.5.2 Data Structures

Each entry of the variable-size memory pool and the fixed-size memory pool tables have the following fields, respectively.

```
struct var_sz_mem_pool_cntrl_block {
   VP               exinf;       /* Extended Information */
   ATR              mplatr;      /* Memory Pool Attributes */
   MEM_BLK_ID_TYPE  blkid;       /* Physical Mem mgr's blkid */
   INT              maxblks;     /* Max blks that can be allocated */
   var_sz_block_node var_sz_blocks_head_node;
      /* Head node of linked list which maintains block assignments */
};
struct fixed_sz_mem_pool_cntrl_block {
   VP                exinf;       /* Extended Information */
   ATR               mpfatr;      /* Memory Pool Attributes */
   INT               blfsz;       /* Memory Block Size */
   INT               mpfcnt;      /* # of blocks in the pool */
   MEM_BLK_ID_TYPE   blkid;       /* Physical Mem mgr's blkid */
};
```

The memory pool attributes specify the manner in which tasks waiting for memory allocation are put on the memorypool's queue. The `blkid` is a tag to the memory allocated for the pool by the Physical Memory Manager Module.

6.5.3 Memory Allocation

Memory is allocated for creation of memorypools, using the service, *get_mem_blk()*, of the Physical Memory Manager Module. Allocation of memory blocks in each of the pools is done as follows.

- *Variable-size Memorypools*

A doubly linked list is maintained to manage the block allocation. Specifically, at the time of pool creation, additional amount of memory is requested, which is used for the linked list. Given the size of the pool, it is assumed that the maximum number of blocks that can be requested from the pool are (pool_size/FACTOR). The FACTOR used was 10, as it is fair enough for most of the cases. This results in faster allocation and deallocation of memory blocks, as there is no need to contact the Physical Memory Manager Module again, for more memory. The doubly linked list representation simplifies merging of adjacent free memory blocks, and is faster. However, it conserves more memory.

The following algorithm is used when a request for a memory block of a specific size arrives.

```
1. Refer the status of memorypool;
2. if (the memorypool wait queue is not empty) OR
   if (the continuous free space available is lesser than required)
   then
        if (the task is just polling) return NO_MEM;
        else
            Add the task-id and the memory requested to the queue;
            Make the current task wait and Release Control;
3. Allocate memory block;
4. Check out the free space left now.
   Try allocating memory to the tasks at the head of the queue.
```

- *Fixed-size Memorypools*

As the maximum number of blocks that can be requested from a fixed-size memorypool is fixed and known, additional memory (a control block) of size *number-of-blocks/8* is requested at the time of pool creation, to manage the allocation/deallocation of blocks. Each bit in the control block corresponding to a unique block in the pool and determines whether the block is allocated.

The following algorithm is used when a request for a memory block arrives.

```
1. Refer the status of memorypool;
2. if (the memorypool wait queue is not empty)
   then
        if (the task is just polling) return NO_MEM;
        else
            Add the task-id to the queue;
            Make the current task wait and Release Control;
3. Allocate memory block;
```

6.5.4 Memory De-allocation

Memory is de-allocated when a memorypool is deleted, using the service, *free_mem_blk()*, of the Physical Memory Manager Module. When a pool is deleted, however, if there are any tasks waiting to get memory blocks from the memorypool, a specific error will to returned to each waiting task.

Deallocation of memory blocks in the pools is done as follows.

- *Variable-size Memorypools*

The address of the memory block to be freed, indirectly gives the address of the control node, in the pool linked list. The status of the node is set to be free, and using the doubly linked feature of the list, the neibhoring nodes, if already FREE, are merged to form a single node with larger continuous space.

- *Fixed-size Memorypools*

De-allocation in fixed-size memorypools is simply retrieving and resetting the corresponding control bit of the memory block.

6.5.5 μITRON Implementation-dependent Specifications

- **Specification:** *When tasks form a queue to compete for a memory, it is implementation dependent whether priority is given to tasks requesting the smaller size of memory or those at the head of the queue.*
  Implementation: No priority is given to the tasks requesting smaller size of memory but allocation is strictly done in the order specified (FIFO or task-priority based) while creating the pool.

- **Specification:** *When a memorypool being waited for by more than one tasks is deleted, the order of tasks on the ready queue after the WAIT state is cleared is implementation dependent in the case of tasks having the same priority.*
  Implementation: The tasks are placed in the same order in the ready queue as they were in the memorypool wait queue.

Chapter 7

μITRON Time Manager Module

The μITRON Time Manager Module provides services for time-dependent processing. These services include functions for setting and referring the system clock, delaying tasks, manipulating handlers invoked cyclically (cyclic handlers), and manipulating handlers started at specified time (alarm handlers).

7.1 The Design

Time can be either *absolute* or *relative*, and is managed and expressed in terms of milliseconds, seconds, minutes, hours, days(or date), months and years. Using the services of the Interrupt Manager Module, the module maintains the system time and manages the cyclic handlers and alarm handlers. This is done by defining highest priority handlers(a service provided by the Interrupt Handler Module) for the hardware timer interrupt, which maintain the system time and manage the cyclic handlers and the alarm handlers.

Cyclic handlers and alarm handlers are generally called timer handlers. Timer handlers are executed as task-independent portions and the user must save any registers used by the timer handler. Even if dispatching is required while a timer handler is executing, it is not processed immediately, but rather that dispatching is delayed until the timer handler finishes. This is called *delayed dispatching*.

A cyclic handler is invoked first exactly after the time interval has elapsed. The handler will run cyclically until either its definition is cancelled or it is deactivated. Different cyclic handlers are identified by *cyclic handler-id* and when redefining a

cyclic handler, it is not necessary to first cancel the handler definition which has that id.

Alarm handlers are similarly identified by *alarm handler-id*s but the time for alarm handler invocation could be either absolute or relative. If an absolute time is specified, the handler will be invoked at the clock time specified. If relative time is specified, the alarm handler will be invoked after the amount of time specified has elapsed. In any case, the definition of the alarm handler is cancelled automatically when the specified time comes and that the handler is invoked. When redefining an alarm handler, it is, however, not necessary to first cancel the handler definition which has that id.

The module offers services to delay (keep in WAIT state) the running task, in real-time. This is done by using the services of the μITRON Task Manager Module to make the current task wait, with timeout(the delay value). The task WAIT state is released as soon as the specified time expires, but if a higher priority task is executing by this time, the task is kept in the ready queue, and it waits further for it's turn to come.

7.2 Prerequisites for the Time Manager Module

- **Hard Prerequisites:** Interrupt Handler Module.
  The Interrupt Handler Module services are used to maintain system time, and to manage cyclic handlers and alarm handlers.

- **Soft Prerequisites:** μITRON Task Manager Module.
  The μITRON Task Manager Module timeout service is used to delay the running task in real-time.

7.3 Configuration parameters

The module needs to be provided with the following static configuration parameters:

1. The maximum number of cyclic handlers defined by the application.

2. The maximum number of alarm handlers defined by the application.

7.4 Services provided by the Time Manager

The μITRON Time Manager Module provides the following services.

- **Set System Clock** This system call sets the system clock to the time specified.
- **Get System Clock** This system call returns the current value of the system clock.
- **Delay Task** This system call temporarily halts the execution of the task issuing the call, and makes it enter the time elapse wait state. The task halts execution for the amount of time specified.
- **Define Cyclic Handler** This system call defines a cyclic handler.
- **Activate Cyclic Handler** This system call changes the activation of the specified cyclic handler (OFF/ON/Re-initialize the interval counter).
- **Reference Cyclic Handler Status** This system call refers to the state of the specified cyclic handler and returns the cyclic handler's activation state, the remaining time until the cyclic handler is invoked next, and its extended information.
- **Define Alarm Handler** This system call defines an alarm handler.
- **Reference Alarm Handler Status** This system call refers to the state of the specified alarm handler, and returns the remaining time until the alarm handler is invoked, and its extended information.
- **Return from Timer Handler** This system call causes the invoked timer handler (cyclic or alarm) to finish.

7.5 Implementation

7.5.1 Module Initialization

At initialization the μITRON Time Manager Module sets the system time to a default value and initializes the cyclic handler and alarm handler tables. It also initializes the 8253 timer(Counter 0) to generate an interrupt, 100 times per second. It *queries*

the nano-kernel for the service handlers of the Interrupt Manager Module and the μITRON Task Manager Module. It then adds handlers to the timer interrupt to maintain system clock and manage cyclic handlers and alarm handlers.

7.5.2 System Time

System time is absolute and contains the fields milli-seconds, seconds, minutes, hours, date, month and year. The system time is incremented by 10 milli-seconds on each timer interrupt and other fields like seconds, minutes, etc., are modified accordingly. Leap years are taken care of. The system call, *set_system_call()*, re-initializes the system time.

7.5.3 Cyclic Handlers

Relative time is specified for invocation of a cyclic handler, in terms of milli-seconds, seconds, minutes, hours, days, months(1 month = 30 days), and years (1 year = 365 days). The handler that manages the cyclic handlers goes through all the active cyclic handlers, each time it is invoked, and decrements the waiting count by 10 milli-seconds, and if the count reaches zero, the module informs the μITRON Task Manger Module, that a *task independent* portion is being entered into. It then invokes the cyclic handler. Once the cyclic handler exits, the μITRON Task Manager Module is re-informed about the current system status and the wait counter is re-initialized.

7.5.4 Alarm Handlers

Absolute or relative time can be specified for invocation of an alarm handler. Alarm handler specified by relative time are processed in the same way as a cyclic handler, except that their definition is cancelled once the handler is invoked. However, if the time specified is absolute, a comparison is made with the current system time, instead of maintaining a waiting counter.

7.5.5 μITRON Implementation-dependent Specifications

- **Specification:** *If more than one timer handler and/or interrupt handler are to be invoked at the same time, it is implementation dependent whether they may*

either be run serially or they may be run nested.

Implementation: Timer handlers are invoked by an interrupt handler, which invokes them serially. Interrupt handlers, however, can be nested.

- **Specification:** *In an implementation allowing* define_cyclic_handler() *or* define_alarm_handler() *to be issued by a timer handler, it is possible to redefine a timer handler with the same timer handler number/id in the handler.*

  Implementation: *define_cyclic_handler()* and *define_alarm_handler()* are allowed inside any timer handler and hence the redefinition of the same timer handler number/id.

Chapter 8

μITRON System Manager Module

System management functions are used to set and refer the overall system environment. These system calls include functions for getting the OS version, referencing the system's dynamic status, and referencing configuration information (the system's static status).

8.1 The Design

The information about the OS version, maker code, μITRON specification version number implemented (3.02), internal implementation version number (ver 1.0), CPU information (Intel 8086, 8088), available functions (Level E) etc., is hard-coded, and can be obtained from the *get_version()* system call.

To refer the system's dynamic status, the μITRON System Manager Module uses the services of the μITRON Task Manager Module. The μITRON Task Manager Module manages the three flags, DISPATCH-DISABLED, TASK-INDEPENDENT-PORTION-ENTERED, and CPU-LOCKED. Any module that performs an operation that would modify the system status, informs the μITRON Task Manger Module about the change, and the latter updates the flag. The μITRON System Manger Module simply refers to these flags for the current system status.

8.2 Prerequisites for the System Manager Module

- **Hard Prerequisites:** None.

- **Soft Prerequisites:** μITRON Task Manager Module.
  The μITRON Task Manager Module services are used to refer to the current system status - if the CPU has been locked, or if a task-independent portion is running, or if the dispatching is disabled.

8.3 Configuration parameters

The module does not have any static configuration parameters.

8.4 Services provided by the System Manager

The μITRON System Manager Module provides the following services.

- **Get Version Information** This system call gets informations of the maker of the μITRON specification OS currently executing, the identification number of the OS, the μITRON specification version number which the OS is based on, and the version number of the OS product.
- **Reference System Status** This system call refers the execution state of the CPU and OS, and returns information such as whether dispatching is disabled and whether a task-independent portion is executing.
- **Reference Configuration Information** This system call refers static information regarding the system and information specified at its configuration.

8.5 Implementation

At initialization, the μITRON System Manager Module *queries* the nano-kernel for the service handler of the μITRON Task Manager Module. The system call *Reference System Status* uses the service *system_status_info()* of the μITRON Task Manager Module and the current system state is informed to the user, in terms of whether dispatching is disabled, whether the cpu has been locked and whether a task-independent portion is executing. The configuration information returned by the module is implementation dependent, and has the following structure.

```
typedef struct t_rcfg {
     nucleus_node nucleus;
     u_mem_m_node u_mem_m;
     u_task_m_node u_task_m;
     u_time_m_node u_time_m;
     u_comm_m_node u_comm_m;
} T_RCFG;
typedef struct nucleus_node {
     INT max_no_of_ports;
} nucleus_node;
typedef struct u_mem_m_node {
     INT max_app_var_sz_pools;
     INT max_app_fixed_sz_pools;
} u_mem_m_node;
typedef struct u_task_m_node {
     INT max_user_tasks;
     INT max_suspend_requests;
     INT max_wakeup_requests;
} u_task_m_node;
typedef struct u_time_m_node {
     INT max_cyclic_handlers;
     INT max_alarm_handlers;
} u_time_m_node;
typedef struct u_comm_m_node {
     INT max_user_eventflags;
     INT max_user_mailboxes;
     INT max_user_semaphores;
} u_comm_m_node;
```

Hence we see that the μITRON System Manager Module provides valuable information such that flexible code can be written.

Chapter 9

Car Dashboard Controller - An Application Program

The kernel design and various modules presented in this thesis, offer services that can be used for a wide range of applications. The OS platform is compatible with level E(extended) μITRON 3.0 industrial standard. In this chapter, we discuss the design of a car dashboard controller, an application developed using the current implementation of the kernel.

9.1 The Model

Consider a car dashboard which shows the speed, total distance traveled, traveled distance in a trip, fuel status, battery status, car temperature, pressures of all the four tyres and alarms for an unlocked door. The car parameters are monitored continuously by an embedded controller. Warnings are shown on the dashboard so that the driver can take appropriate action. For implementing such a car dashboard, we model the application program as follows.

- The CPU is interrupted on completion of each rotation of the car tyre. A photo sensor, that is stationary with respect to the rotating tyre, can be used for this purpose. It would detect the light from an emitter, again stationary with respect to the wheel. The movement of the wheel interrupts the light path between emitter and detector. This signal is used to identify the wheel rotation. Once the CPU is interrupted, the corresponding task would update the distance

traveled and the speed.

- A switch on the car body can be used to detect whether a door is locked. This switch is pressed closed by the door. Door lock status is displayed periodically, and is checked when the car is moving, to warn the driver if any door is unlocked.

- An object floating in the fuel tank, connected to a *rheostat* can produce a voltage corresponding to the fuel level. This value, through an A/D converter, is read periodically by the controller to update the fuel display.

- The output of the temparature sensor, sensing the temparature inside the car, is read using A/D converter to periodically update the temparature display.

- Battery voltage is directly read and converted to digital form, to be read by the CPU, periodically.

- Tyre pressure can be measured in the following way. A metal cylinder connected to the nozzle of the tyre having a disc obstructing the air, which can be moved away from the nozzle by the air pressure in the tyre and by a spring trying to push it towards the nozzle. Hence, the higher the tyre pressure, the farther the disc moves away from the nozzle, and a *rheostat* connected to the other end can produce a corresponding voltage. This voltage can be continuously read by the pads connected on the stationary part on either side of the tyre.

9.2 The Design

The application needs various data to be displayed and updated on the dashboard constantly. Specifically, data is received by the two tasks - **Update_Display** and **Show_Warning** from their respective mailboxes used for data communication - one for display requests, and one for warnings. Both the tasks are of high priority and their main function is to wait on a mailbox (other tasks are scheduled during this time) as long as there are no messages, and once a message arrives, they retrieve it and display. The task, *Show_Warning*, is the highest priority task.

The interrupt handler, **UpdateDistancePerOneRotationOfTheWheel**, invoked on each rotation of the car tyre, updates the distance traveled (both the total distance and the recently traveled distance) by the circumference of the tyre.

The following cyclic handlers constantly monitor the input data at regular intervals and append the information to the *display* mailbox. If an event is detected that needs the driver to be warned, an appropriate warning message is also sent (to the *warning* mailbox).

1. **Speed Handler:** The speed handler computes the new speed depending on the last known distance, current distance and the time lapse between two successive invocations of the handler. It then sends the new speed data as a message to the *display* mailbox. Depending on the speed, it also generates the following warnings:

   (a) *Doors unlocked.* (if speed > 0)

   (b) *You are going too fast.* (If speed $>$ SPEED_LIMIT)

2. **Distance Handler:** The display handler simply sends a *display* message to the *display* mailbox for both the distance values maintained.

3. **Fuel Handler:** The fuel handler sends the current fuel value to the *display* mailbox and depending on the fuel value, the following warnings are sent to the *warning* mailbox.

   (a) *Fuel is below the safe amount.*

   (b) *Fuel too low.*

4. **Battery Handler:** The battery handler sends the current battery voltage to the *display* mailbox. Depending on this value the following warning is sent to the *warning* mailbox.

   (a) *Battery Voltage too low.*

5. **Temperature Handler:** The temperature handler sends the current temperature inside the car to the *display* mailbox.

6. **Tyre Pressure Handler:** The tyre pressure handler sends four messages to the *display* mailbox giving the tyre pressures of the four tyres. It also sends the following warning message, depending on the tyre pressures.

   (a) *Tyre pressure low in tyre #*

7. **Door-lock Handler:** The Door-lock handler sends the current Door-lock status of the four doors of the car to the *display* mailbox.

All the above mentioned cyclic handlers, tasks, and the interrupt handler together make the car dashboard controller simple and complete.

9.3 Results

The design and implementation(a simulation) of this application, took about twenty man hours. The application is completely written in 'C' language[Appendix C], and the size of the source code is about 500 lines. It can be clearly seen that using the μITRON API we are able to design the application at a very high level of abstraction, making the design and implementation of the application very simple.

Chapter 10

Conclusion

In this thesis we presented a generic modular embedded OS platform compatible with level E(Extended) μITRON 3.0 standard. The OS is well suited for both the small-scale and large-scale embedded systems, as the modules are *pluggable* in nature and hence, no extra baggage needs to be carried by the application. Further the modular architecture allows the OS to provide a compatible interface with an industrial standard like μITRON, contributing to improved software productivity.

The example application of a car dashboard controller, presented in Chapter 9 and Appendix C, clearly shows that the embedded OS platform presented in this thesis reduces the development time; reduces the design complexity; reduces the chances of error and makes the application compatible with an industrial standard.

The current implementation supports the basic modules: Physical Memory Manager, Interrupt Handler, Thread Manager and the μITRON API modules: Task Manager, Communication Manager, Interrupt Manager, Memory Manager, Time Manager and System Manager. These modules would suffice for both the simple applications and most of the complicated ones. For other applications, modules like Network Manager, may be necessary.

10.1 Existing Work

The work presented here is an extension of the existing work, which includes the design and implementation of the nucleus and the following modules [Appendix: B]

1. Interrupt Handler Module and

2. Thread Manager Module

These two modules are sufficient for small applications, but do not provide any standard API.

10.2 Future Work

The following tasks could be taken up, as part of future work, to take the work towards its logical completion.

Modules Several other modules like Network Manager, etc., and modules supporting other standard API can be developed. Further, modules serving the same purpose but using different strategies can be developed too, and any of those modules can be picked up by the application, depending on the need.

Applications More applications can be developed using this kernel to verify the correctness and suitability of the kernel.

Tools Several tools can be developed for simulating and debugging sample applications and modules, leading to rapid development of the OS and the embedded applications.

Porting Porting of the *machine-dependant* parts of the modules for different hardware architectures and processors can also be taken up.

Appendix A

API

This chapter discusses the application program interface provided by various μITRON modules. Though the modules and the application communicate using the stardardized intermodule interface[Kri97], given below are the library function prototypes that can be used by the applications using the help of the standard library.

A.1 μITRON Task Manager Module

1. **Create Task**

```
ER cre_tsk ( ID tskid, T_CTSK *pk_ctsk ) ;
 -(pk_ctsk members)-
  VP      exinf   ExtendedInformation
  ATR     tskatr  TaskAttribute
  FP      task    TaskStartAddress
  PRI     itskpri InitialTaskPriority
  INT     stksz   StackSize (in bytes)
```

2. **Delete Task**

```
ER del_tsk ( ID tskid ) ;
```

3. **Start Task**

```
ER sta_tsk ( ID tskid, INT stacd ) ;
```

4. **Exit Issuing Task**

```
void ext_tsk ( ) ;
```

5. **Exit and Delete Task**

```
void exd_tsk ( ) ;
```

6. **Terminate Other Task**

```
ER ter_tsk ( ID tskid ) ;
```

7. **Disable Dispatch**

```
ER dis_dsp ( ) ;
```

8. **Enable Dispatch**

```
ER ena_dsp ( ) ;
```

9. **Change Task Priority**

```
ER chg_pri ( ID tskid, PRI tskpri ) ;
```

10. **Rotate Tasks on the Ready Queue**

```
ER rot_rdq ( PRI tskpri ) ;
```

11. **Release Wait of Other Task**

```
ER rel_wai ( ID tskid ) ;
```

12. **Get Task Identifier**

```
ER get_tid ( ID *p_tskid ) ;
```

13. **Reference Task Status**

```
ER ref_tsk ( T_RTSK *pk_rtsk, ID tskid ) ;
 -(pk_rtsk members)-
  VP      exinf   ExtendedInformation
  PRI     tskpri  TaskPriority
  UINT    tskstat TaskState
```

14. **Suspend Other Task**

```
ER sus_tsk ( ID tskid ) ;
```

15. **Resume Suspended Task/Forcibly Resume Suspended Task**

```
ER rsm_tsk ( ID tskid ) ;
ER frsm_tsk ( ID tskid ) ;
```

16. **Sleep Task/Sleep Task with Timeout**

```
ER  slp_tsk ( ) ;
ER tslp_tsk ( TMO tmout ) ;
```

17. **Wakeup Other Task**

```
ER wup_tsk ( ID tskid ) ;
```

18. **Cancel Wakeup Request**

```
ER can_wup ( INT *p_wupcnt, ID tskid ) ;
```

A.2 μITRON Communication Manager Module

1. **Create Semaphore**

```
ER cre_sem ( ID semid, T_CSEM *pk_csem ) ;
 -(pk_csem members)-
  VP      exinf   ExtendedInformation
  ATR     sematr  SemaphoreAttributes
  INT     isemcnt InitialSemaphoreCount [level X]
  INT     maxsem  MaximumSemaphoreCount [level X]
```

2. **Delete Semaphore**

```
ER del_sem ( ID semid ) ;
```

3. **Signal Semaphore**

```
ER sig_sem ( ID semid ) ;
```

4. **Wait on Semaphore/Poll and Request Semaphore/Wait on Semaphore with Timeout**

```
ER wai_sem ( ID semid ) ;
ER preq_sem ( ID semid ) ;
ER twai_sem ( ID semid, TMO tmout ) ;
```

5. **Reference Semaphore Status**

```
ER ref_sem ( T_RSEM *pk_rsem, ID semid ) ;
 -(pk_rsem members)-
  VP      exinf   ExtendedInformation
  BOOL_ID wtsk    WaitingTaskInformation
  INT     semcnt  SemaphoreCount
```

6. **Create Eventflag**

```
ER cre_flg ( ID flgid, T_CFLG *pk_cflg ) ;
 -(pk_cflg members)-
  VP      exinf   ExtendedInformation
  ATR     flgatr  EventFlagAttributes
  UINT    iflgptn InitialEventFlagPattern
```

7. **Delete Eventflag**

```
ER del_flg ( ID flgid ) ;
```

8. **Set Eventflag/Clear EventFlag**

```
ER set_flg ( ID flgid, UINT setptn ) ;
ER clr_flg ( ID flgid, UINT clrptn ) ;
```

9. **Wait for Eventflag/Wait for Eventflag (Polling)/Wait for Eventflag with Timeout**

```
ER wai_flg ( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
ER pol_flg ( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode);
ER twai_flg ( UINT *p_flgptn, ID flgid, UINT waiptn, UINT wfmode,
              TMO tmout ) ;
```

10. **Reference Eventflag Status**

```
ER ref_flg ( T_RFLG *pk_rflg, ID flgid ) ;
 -(pk_rflg members)-
  VP      exinf   ExtendedInformation
  BOOL_ID wtsk    WaitingTaskInformation
  UINT    flgptn  EventFlagBitPattern
```

11. **Create Mailbox**

```
ER cre_mbx ( ID mbxid, T_CMBX *pk_cmbx ) ;
 -(pk_cmbx members)-
  VP      exinf   ExtendedInformation
  ATR     mbxatr  MailboxAttributes
  (the use of the following information is implementation dependent)
  INT     bufcnt  BufferMessageCount
```

12. **Delete Mailbox**

```
ER del_mbx ( ID mbxid ) ;
```

13. **Send Message to Mailbox**

```
ER snd_msg ( ID mbxid, T_MSG *pk_msg ) ;
```

14. **Receive Message from Mailbox/Poll and Receive Message from Mailbox/Receive Message from Mailbox with Timeout**

```
ER rcv_msg ( T_MSG *ppk_msg, ID mbxid ) ;
ER prcv_msg ( T_MSG *ppk_msg, ID mbxid ) ;
ER trcv_msg ( T_MSG *ppk_msg, ID mbxid, TMO tmout ) ;
```

15. **Reference Mailbox Status**

```
ER ref_mbx ( T_RMBX *pk_rmbx, ID mbxid ) ;
 -(pk_rmbx members)-
  VP      exinf   ExtendedInformation
  BOOL_ID wtsk    WaitingTaskInformation
  T_MSG   *pk_msg Start Address of Message Packet to be Received
```

A.3 μITRON Interrupt Manager Module

1. **Define Interrupt Handler**

```
ER def_int ( UINT dintno, T_DINT *pk_dint ) ;
 -(pk_dint members)-
  ATR     intatr  InterruptHandlerAttributes
  FP      inthdr  InterruptHandlerAddress
```

2. **Return from Interrupt Handler**

```
void ret_int ( ) ;
```

3. **Return and Wakeup Task**

```
void ret_wup ( ID tskid ) ;
```

4. **Lock CPU**

```
ER loc_cpu ( ) ;
```

5. **Unlock CPU**

```
ER unl_cpu ( ) ;
```

6. **Disable Interrupt**

```
ER dis_int ( UINT eintno ) ;
```

7. **Enable Interrupt**

```
ER ena_int ( UINT eintno ) ;
```

8. **Change Interrupt Mask (Level or Priority)**

```
ER chg_iXX ( UINT iXXXX ) ;
```

9. **Reference Interrupt Mask (Level or Priority)**

```
ER ref_iXX ( UINT *p_iXXXX ) ;
```

A.4 μITRON Memory Manager Module

1. **Create Variable-Size Memorypool**

```
ER cre_mpl ( ID mplid, T_CMPL *pk_cmpl ) ;
 -(pk_cmpl members)-
  VP      exinf   ExtendedInformation
  ATR     mplatr  MemoryPoolAttributes
  INT     mplsz   MemoryPoolSize (in bytes)
```

2. **Delete Variable-Size Memorypool**

```
ER del_mpl ( ID mplid ) ;
```

3. **Get Variable-Size Memory Block/Poll and Get Variable-Size Memory Block/Get Variable-Size Memory Block with Timeout**

```
ER get_blk ( VP *p_blk, ID mplid, INT blksz );
ER pget_blk ( VP *p_blk, ID mplid, INT blksz );
ER tget_blk ( VP *p_blk, ID mplid, INT blksz, TMO tmout ) ;
```

4. **Release Variable-Size Memory Block**

```
ER rel_blk ( ID mplid, VP blk ) ;
```

5. **Reference Variable-Size Memorypool Status**

```
ER ref_mpl ( T_RMPL *pk_rmpl, ID mplid ) ;
 -(pk_rmpl members)-
  VP      exinf   ExtendedInformation
  BOOL_ID wtsk    Waiting TaskInformation
  INT     frsz    FreeMemorySize (in bytes)
  INT     maxsz   MaximumFreeMemorySize (in bytes)
```

6. **Create Fixed-Size Memorypool**

```
ER cre_mpf ( ID mpfid, T_CMPF *pk_cmpf ) ;
 -(pk_cmpf members)-
  VP      exinf   ExtendedInformation
  ATR     mpfatr  MemoryPoolAttributes
  INT     mpfcnt  MemoryPoolBlockCount
  INT     blfsz   MemoryBlockSize (in bytes)
```

7. **Delete Fixed-Size Memorypool**

```
ER del_mpf ( ID mpfid ) ;
```

8. **Get Fixed-Size Memory Block/Poll and Get Fixed-Size Memory Block/Get Fixed-Size Memory Block with Timeout**

```
ER  get_blf ( VP *p_blf, ID mpfid );
ER pget_blf ( VP *p_blf, ID mpfid );
ER tget_blf ( VP *p_blf, ID mpfid, TMO tmout ) ;
```

9. **Release Fixed-Size Memory Block**

```
R rel_blf ( ID mpfid, VP blf ) ;
```

10. **Reference Fixed-Size Memorypool Status**

```
ER ref_mpf ( T_RMPF *pk_rmpf, ID mpfid ) ;
 -(pk_rmpf members)-
  VP      exinf   ExtendedInformation
  BOOL_ID wtsk    WaitingTaskInformation
  INT     frbcnt  FreeMemoryBlockCount
```

A.5 μITRON Time Manager Module

1. **Set System Clock**

```
ER set_tim ( SYSTIME *pk_tim );
typedef struct {
H  msecs; /* Milli-seconds */
B  secs;  /* Seconds */
B  mins;  /* Minutes */
B  hrs;   /* Hours */
B  date;  /* Date */
B  month; /* Month */
H  year;  /* Year */
} SYSTIME, CYCTIME, ALMTIME;
```

2. **Get System Clock**

```
ER get_tim ( SYSTIME *pk_tim );
```

3. **Delay Task**

```
ER dly_tsk ( DLYTIME dlytim );
```

4. **Define Cyclic Handler**

```
ER def_cyc ( HNO cycno, T_DCYC *pk_dcyc ) ;
 -(pk_dcyc members)-
  VP      exinf   ExtendedInformation
  ATR     cycatr  CyclicHandlerAttributes
  FP      cychdr  CyclicHandlerAddress
  UINT    cycact  CyclicHandlerActivation
  CYCTIME cyctim  CycleTime
```

5. **Activate Cyclic Handler**

```
ER act_cyc ( HNO cycno, UINT cycact ) ;
```

6. **Reference Cyclic Handler Status**

```
ER ref_cyc ( T_RCYC *pk_rcyc, HNO cycno ) ;
 -(pk_rcyc members)-
  VP      exinf   ExtendedInformation
  CYCTIME lfttim  LeftTime
  UINT    cycact  CyclicHandlerActivation
```

7. **Define Alarm Handler**

```
ER def_alm ( HNO almno, T_DALM *pk_dalm ) ;
 -(pk_dalm members)-
  VP      exinf   ExtendedInformation
  ATR     almatr  AlarmHandlerAttributes
  FP      almhdr  AlarmHandlerAddress
  UINT    tmmode  TimeMode
  ALMTIME almtim  AlarmTime
```

8. **Reference Alarm Handler Status**

```
ER ref_alm ( T_RALM *pk_ralm, HNO almno ) ;
 -(pk_ralm members)-
  VP      exinf   ExtendedInformation
  ALMTIME lfttim  Time Left before Handler Runs
```

9. **Return from Timer Handler**

```
void ret_tmr ( ) ;
```

A.6 μITRON System Manager Module

1. **Get Version Information**

```
ER get_ver ( T_VER *pk_ver );
 -(pk_ver members)-
  UH      maker   OS Maker
  UH      id      Identification Number of the OS
  UH      spver   ITRON or uITRON Specification Version Number
  UH      prver   OS Product Version Number
  UH      prno[4] Product Number(Product Management Information)
  UH      cpu     CPU Information
  UH      var     Variation Descriptor
```

2. **Reference System Status**

```
ER ref_sys ( T_RSYS *pk_rsys ) ;
 -(pk_rsys members)-
  INT     sysstat SystemState
```

3. **Reference Configuration Information**

```
ER ref_cfg ( T_RCFG *pk_rcfg ) ;
 -(pk_rcfg members)-
  (CPU and/or implementation-dependent information is returned)
```

Appendix B

Existing Modules

The following modules have been developed prior to this thesis. A brief description of their design, pre-requisites, and services offered are given below.

B.1 Interrupt Manager Module[Kri97]

The interrupt manager provides services to let the application and other modules to specify their functions to be added or removed from the service routine of a particular interrupt.

B.1.1 The Design

Interrupts are handled in general, by having an interrupt service routine for each interrupt and having it invoked automatically whenever the interrupt arrives. Conventionally, when this interrupt routine is being executed, some or all the interrupts are disabled or queued, depending on the interrupt's priority.

- *Problems with interrupt level priorities*

In cases where the same interrupt occurs multiple times in succession, fast enough, such that the next interrupt arrives while the previous invocation is being served, then, all subsequent calls are either masked, queued or allowed to be serviced.

If the subsequent calls are masked, we lose some input signals which may be

urgent. If the calls are queued, the system cannot work since the queue keeps on growing while the system is not able to serve the requests and if the interrupts are allowed to be serviced, then the stack grows unbounded since the interrupts keep coming but are not able to return.

The Solution

The approach adapted in the design of this module is to have priorities at the service level rather than at interrupt level. All the interrupts are defined to have equal priority and no interrupt is masked during normal execution. If at any time interrupt masking is required then all interrupts are masked at the same time.

Hence, an application defines services along with their priorities. When an interrupt is raised, the default service routine is invoked which executes service routines in order of their priorities. In case another interrupt arises while the previous one is still being served, the interrupt is allowed to be serviced and the previous one is terminated leaving low priority services unserved.

B.1.2 Pre-requisites

- **Hard Prerequisites :** None.
- **Soft Prerequisites :** None.

The only dependency of this module is a run-time dependency on modules like "virtual memory manager". If it is present, the interrupt manager additionally sets-up the interrupt tables for the protected mode.

B.1.3 Services

The module offers the following services to the application and the other modules.

- **Add Handler -** Add a given service subroutine to the list of an interrupt at the given priority.
- **Remove Handler -** Remove a specified service routine or handler from the list of handlers of the specified interrupt.

B.2 Thread Manager Module[Kri97]

A thread is an independent flow of execution. While nano-kernel supports a single thread of execution, a thread manager provides multiple threads which may be executing in a system concurrently. The module provides the application and other modules, the services related to threads such as thread scheduling, thread creation, thread termination etc. It also provides synchronization services like semaphores and inter thread signaling. Apart from providing these services, the module also keeps the existence of multiple threads transparent to the nano-kernel and other modules.

B.2.1 The Design

Basic thread management involves providing the application with the ability to use multiple threads of execution. This includes services to start a new thread, to schedule threads, to terminate a thread and to modify properties of a thread.

- *Thread Creation*

To start a new thread, the module internally uses the *fork*[1] construct. Externally, the module offers a *create*[2] construct to simplify the application programming.

- *Thread Scheduling*

In embedded systems different applications require different types of scheduling algorithms. Therefore, the design and its data structures make it easy to implement most of the algorithms by just adding a function that determines the next thread to be scheduled.

- *Thread Termination*

A thread is terminated when it exits or is killed by another thread. When a thread terminates the module unlocks all the semaphores locked by the terminating thread

[1] The *fork* construct creates a new thread and returns to the statement following the *fork* call in both threads. Here the two threads are considered to have a parent child relationship.

[2] The *create* construct creates a thread in which control is returned to the specified address. In this case the threads are said to have a peer relationship.

and clears its thread status structure entry. If its parent thread is still alive then the module sends a signal to the parent thread to indicate that a child thread has terminated.

- *Semaphore Management*

The module also provides a simple semaphore scheme to provide mutual exclusion and synchronize access of shared resources. Although the semaphore service provided by this module is very simple and is similar to binary mutex, it is sufficient for most applications. More sophisticated synchronization services can easily be integrated at the application level itself.

- *Signal Management*

The design here presents a bi-priority signal management scheme, in which, signals can have two level of priorities, the default priority and the high priority. A signal at default priority can be received by the target thread only when it gets scheduled at its turn. In case of high priority signal, the target thread is scheduled out of its turn to let it receive the signal. This scheme guarantees that a high priority signal is received by the target thread as soon as the scheduler gets a chance to schedule it. Even better constraints can be achieved by making the sender thread voluntarily relinquish control after sending the signal. This makes the target thread receive the signal almost immediately.

B.2.2 Pre-requisites

- **Hard Prerequisites :** None.
- **Soft Prerequisites :** Interrupt Manager Module.

If the interrupt manager module is present, it attaches the thread scheduler to the 'timer_interrupt' to support pre-emptive scheduling. Otherwise the module supports only non-pre-emptive scheduling.

B.2.3 Services

The module offers the following services to the application and the other modules.

- **Create Thread -** Set up the stack for the new thread, find an empty entry in the thread array, and make proper entries thereby creating a new thread.
- **Thread Exit -** Mark the current thread status as dead, free all allocated resources and send a signal to parent thread thereby killing the calling thread.
- **Kill Thread -** This service is same as thread exit except that this can kill a specified thread rather than the calling thread.
- **Set Thread Priority -** Set the priority of the specified thread to a specified value.
- **Get Thread Priority -** Returns the priority of the specified thread.
- **Get Thread Id -** Get the thread id of current thread.
- **Suspend Thread -** Suspend the execution of given thread.
- **Resume Thread -** Resume execution of a thread.
- **Thread Switch -** Voluntarily release of control.
- **Thread Wait -** Wait for one or more child threads to die.
- **Thread sleep -** Sleep for a given time.
- **Initialize Semaphore -** Initialize a given semaphore.
- **Set Semaphore -** Locks the given semaphore if not already locked or block if it is already locked.
- **Clear Semaphore -** Unlock the given semaphore and unblock a thread blocked on this semaphore.
- **Close Semaphore -** Close a given semaphore.
- **Set Signal Handler -** Sets a given function as the handler of a given signal in the calling thread.

- **Revoke Signal Handler -** Removes the signal handler which was last set and restore the signal handler to the previous one.
- **Send Signal -** Sends a given signal to a specified thread.

Appendix C

Test Application Code

```
#include <libh\u_libtsk.h> /* Task Management functions */
#include <libh\u_libcom.h> /* Task Communication functions */
#include <libh\u_libtim.h> /* Time Management functions */
#include <libh\u_libint.h> /* Interrupt Management functions */
#include <libh\libprint.h> /* Screen-Printing functions */
#include "car_dashboard.h"

INT current_speed = 0;
INT total_distance = 0, current_distance = 0;
INT current_fuel = INITIAL_FUEL;
INT current_battery_status = INITIAL_BATT;
INT current_temparature = INITIAL_TEMP;
INT current_tyre_pressure[4] = { INITIAL_PRES, INITIAL_PRES,
                                 INITIAL_PRES, INITIAL_PRES };
BOOL current_door_lock[4] = { TRUE, TRUE, TRUE, TRUE };
char char_read = 0;

struct parameter_block_header* application_main (
  struct parameter_block_header* message)
{
  T_CTSK cre_tsk_data;
```

```
T_CMBX cre_mbox_data;
T_DINT def_intr_data;

/* Pass the NUCLEUS HANDLER address to the library */
init_library (message);

/* Create a mailbox to hold the DISPLAY requests */
/* Messages are queued by "message-priority" */
cre_mbox_data.mbxatr = TA_MPRI;
cre_mbox_data.bufcnt = DISPLAY_MBOX_BUF_LEN;
cre_mbx (DISPLAY_MBOX_ID, &cre_mbox_data);

/* Create a mailbox to hold the WARNING requests */
/* Messages are queued by "message-priority" */
cre_mbox_data.mbxatr = TA_MPRI;
cre_mbox_data.bufcnt = WARNING_MBOX_BUF_LEN;
cre_mbx (WARNING_MBOX_ID, &cre_mbox_data);

/* Create a task to update display whenever a request arrives
   This task eventually waits for messages in "DISPLAY_MLBOX" */
cre_tsk_data.tskatr = TA_HLNG;
cre_tsk_data.task = display_current_status;
cre_tsk_data.itskpri = 2;
cre_tsk_data.stksz = DEFAULT_STACK_SIZE;
cre_tsk (DISPLAY_TASK_ID, &cre_tsk_data);
sta_tsk (DISPLAY_TASK_ID, 0);

/* Create a task which WARNs the driver. This task eventually
   waits for messages in the mailbox "WARNING_MBOX" */
cre_tsk_data.tskatr = TA_HLNG;
cre_tsk_data.task = warn_the_driver;
cre_tsk_data.itskpri = 1;
```

```
cre_tsk_data.stksz = DEFAULT_STACK_SIZE;
cre_tsk (WARNING_TASK_ID, &cre_tsk_data);
sta_tsk (WARNING_TASK_ID, 0);

/* Create a task which generates wheel rotation interrupt at random
   intervals - simulates using s/w interrupts and appr. delay */
cre_tsk_data.tskatr = TA_HLNG;
cre_tsk_data.task = generate_wheel_rotation_interrupts;
cre_tsk_data.itskpri = 3;
cre_tsk_data.stksz = DEFAULT_STACK_SIZE;
cre_tsk (GEN_WHEEL_INTR_TASK_ID, &cre_tsk_data);
sta_tsk (GEN_WHEEL_INTR_TASK_ID, 0);

/* Activate the Cyclic Handlers which scan data from time to time */
DEF_CYC (SPEED_HANDLER, TA_HLNG, TCY_ON, speed_handler,
         SPEED_UPDATE_TIME);
DEF_CYC (DISTANCE_HANDLER, TA_HLNG, TCY_ON, distance_handler,
         DISTANCE_UPDATE_TIME);
DEF_CYC (FUEL_HANDLER, TA_HLNG, TCY_ON, fuel_handler,
         FUEL_UPDATE_TIME);
DEF_CYC (BATTERY_HANDLER, TA_HLNG, TCY_ON, battery_handler,
         BATTERY_UPDATE_TIME);
DEF_CYC (TEMPARATURE_HANDLER, TA_HLNG, TCY_ON, temparature_handler,
         TEMPARATURE_UPDATE_TIME);
DEF_CYC (TYRE_PRESSURE_HANDLER, TA_HLNG, TCY_ON,
         tyre_pressure_handler, TYRE_PRESSURE_UPDATE_TIME);
DEF_CYC (DOOR_LOCK_HANDLER, TA_HLNG, TCY_ON, door_lock_handler,
         DOOR_LOCK_UPDATE_TIME);

/* Define the Distance handler */
def_intr_data.intatr = TA_HLNG;
def_intr_data.inthdr = update_distance_per_one_rotation_of_wheel;
```

```
  def_int (WHEEL_INTERRUPT_NO, &def_intr_data);

  /* Define the keyboard handler */
  def_intr_data.intatr = TA_HLNG;
  def_intr_data.inthdr = keyboard_handler;
  def_int (0x09, &def_intr_data);

  /* Process User inputs */
  for (;;) {
    switch (get_key()) {
      case 'f': current_fuel ++; break;
      case 'v': current_fuel --; break;
      case 'b': current_battery_status++; break;
      case 'n': current_battery_status--; break;
      case 't': current_temparature ++; break;
      case 'g': current_temparature --; break;
      case 'p': current_tyre_pressure[get_key()-'1'] ++; break;
      case 'l': current_tyre_pressure[get_key()-'1'] --; break;
      case 'd': current_door_lock[get_key()-'1'] = TRUE;
                door_lock_handler(); break;
      case 'c': current_door_lock[get_key()-'1'] = FALSE;
                door_lock_handler(); break;
      case 'r': current_distance = 0; break;
    }
  }
}

void display_current_status (void)
{
  T_MSG buffer_msg;
  struct display_message *actual_msg;
  INT row,col;
```

```
  init_display ();

  for (;;) {
    /* Receive a msg OR Wait for a message indefinitely */
    rcv_msg (&buffer_msg, DISPLAY_MBOX_ID);
    actual_msg = (struct display_message *) buffer_msg.msgaddr;
    switch (actual_msg->sender_id) {
      case SPEED_HANDLER: PRINT (3,12,actual_msg->value); break;
      case TOTAL_DISTANCE: PRINT (4,12,actual_msg->value/10);
        print_hex (actual_msg->value%10); break;
      case CURRENT_DISTANCE: PRINT (5,12,actual_msg->value/10);
        print_hex (actual_msg->value%10); break;
      case FUEL_HANDLER: PRINT (3,41,actual_msg->value); break;
      case BATTERY_HANDLER: PRINT (4,41,actual_msg->value); break;
      case TEMPARATURE_HANDLER: PRINT (5,41,actual_msg->value); break;
      case TYRE_FRNT_L: PRINT (4,69,actual_msg->value); break;
      case TYRE_FRNT_R: PRINT (4,74,actual_msg->value); break;
      case TYRE_REAR_L: PRINT (5,69,actual_msg->value); break;
      case TYRE_REAR_R: PRINT (5,74,actual_msg->value); break;
      case DOOR_FRNT_L: gotoxy (8,70);
        print_char ((actual_msg->value)? 'X':'-'); break;
      case DOOR_FRNT_R: gotoxy (8,75);
        print_char ((actual_msg->value)? 'X':'-'); break;
      case DOOR_REAR_L: gotoxy (9,70);
        print_char ((actual_msg->value)? 'X':'-'); break;
      case DOOR_REAR_R: gotoxy (9,75);
        print_char ((actual_msg->value)? 'X':'-'); break;
    }
  }
}
void warn_the_driver (void)
```

```
{
  T_MSG buffer_msg;
  char *warning;
  int row;

  for (;;) {
    /* Receive a msg OR Wait for a message indefinitely */
    rcv_msg (&buffer_msg, WARNING_MBOX_ID);
    /* Pring Warning */
    gotoxy (15, 25);
    for (warning = (char *) buffer_msg.msgaddr, row = 15;
      *warning; warning ++) {
      if (*warning == '\n') {
        row ++;
        gotoxy (row, 25);
      }
      else print_char (*warning);
    }
    dly_tsk (200);
    /* Clear the warning */
    for (row=15; row<=20; row++) {
      gotoxy (row, 25);
      print_string ("                                                ");
    }
  }
}
static INT td = 0, cd = 0; /* `td' and `cd' in Meters */
void update_distance_per_one_rotation_of_wheel ()
{
  td += CAR_WHEEL_2_PI_R;
  if (td >= 100) {
    td -= 100;
```

```
    total_distance ++;
  }
  cd += CAR_WHEEL_2_PI_R;
  if (cd >= 100) {
    cd -= 100;
    current_distance ++;
  }
}
void speed_handler (void)
{
  static char *msg_to_warn = "Watch Out!!\nYou are going just too fast";
  static char *door_lock_msg = "Check Out!!\nDoors unlocked";
  T_MSG buffer_msg;
  static struct display_message msg_to_display;
  INT count;
  static INT last_known_td = 0, last_known_total_distance = 0;

  /* Update Speed */
  current_speed = (((total_distance - last_known_total_distance)*100 +
                (td - last_known_td)) *(1000/SPEED_UPDATE_TIME)*18)/5;
  last_known_td = td;
  last_known_total_distance = total_distance;
  /* Display Speed */
  SEND_DISPLAY_MSG (SPEED_HANDLER, current_speed, 1, msg_to_display);
  /* Warnings */
  if (current_speed > 0)
    for (count = 0; count < 4; count ++)
      if (!current_door_lock[count])
        SEND_WARNING_MSG (&door_lock_msg[0], 1);
  if (current_speed > SPEED_LIMIT)
    SEND_WARNING_MSG (&msg_to_warn[0], 1);
}
```

```
void distance_handler (void)
{
  T_MSG buffer_msg;
  static struct display_message msg_disp[2];

  SEND_DISPLAY_MSG (TOTAL_DISTANCE, total_distance, 1, msg_disp[0]);
  SEND_DISPLAY_MSG (CURRENT_DISTANCE, current_distance, 1, msg_disp[1]);
}
void fuel_handler (void)
{
  static char *msg_to_warn1 = "Check Out! \nFuel is going down";
  static char *msg_to_warn2 = "Attention!! \nFuel too low";
  T_MSG buffer_msg;
  static struct display_message msg_to_display;
  static INT last_known_distance = 0;

  if ((total_distance - last_known_distance) >= CAR_MILAGE) {
    if (current_fuel) current_fuel --;
    last_known_distance = total_distance/CAR_MILAGE*CAR_MILAGE;
  }
  if (current_fuel < FUEL_LIMIT2)
    SEND_WARNING_MSG (&msg_to_warn2[0], 1)
  else if (current_fuel < FUEL_LIMIT1)
    SEND_WARNING_MSG (&msg_to_warn1[0], 1);
  SEND_DISPLAY_MSG (FUEL_HANDLER, current_fuel, 2, msg_to_display);
}
void battery_handler (void)
{
  T_MSG buffer_msg;
  static struct display_message msg_to_display;
  static char *msg_to_warn = "Battery is going down \nTake care";
```

```
  if (current_battery_status < MIN_BATTERY)
    SEND_WARNING_MSG (&msg_to_warn[0], 1);
  SEND_DISPLAY_MSG (BATTERY_HANDLER, current_battery_status, 2,
    msg_to_display);
}
void temparature_handler (void)
{
  T_MSG buffer_msg;
  static struct display_message msg_to_display;
  SEND_DISPLAY_MSG (TEMPARATURE_HANDLER, current_temparature, 3,
    msg_to_display);
}
void tyre_pressure_handler (void)
{
  INT count;
  T_MSG buffer_msg;
  static struct display_message msg_to_display[4];
  static char *msg_to_warn = "Check Out! \nTyre pressure low";

  for (count=0; count < 4; count++) {
    if (current_tyre_pressure[count] < TYRE_PRESSURE_LIMIT)
      SEND_WARNING_MSG (&msg_to_warn[0], 1);
    SEND_DISPLAY_MSG (TYRE_FRNT_L+count, current_tyre_pressure[count],
                  2, msg_to_display[count]);
  }
}
void door_lock_handler (void)
{
  INT count;
  T_MSG buffer_msg;
  static struct display_message msg_to_display[4];
```

```
  for (count=0; count < 4; count++)
    SEND_DISPLAY_MSG (DOOR_FRNT_L+count, current_door_lock[count], 2,
      msg_to_display[count]);
}
void generate_wheel_rotation_interrupts ()
{
  static INT random_speed = 0, i = 1;
  INT time_to_wait_in_msec;
#define MIN_SPEED 30
#define AVG_SPEED 70
#define MAX_SPEED 120

  init_rand (5);
  for (;;)
  {
      i --;
      if (!i) {
        if (random_speed <= MIN_SPEED) {
          random_speed += get_random(2);
          i = 1;
        }
        else if (random_speed <= AVG_SPEED) {
          random_speed +=  get_random(2);
          i = 10;
        }
        else if (random_speed <= MAX_SPEED) {
          random_speed += (get_random (3) - 1);
          i = 20;
        }
        else {
          random_speed -= get_random(3) ? 0:1;
          i = 30;
```

```
        }
    }
    if (random_speed != 0) {
      time_to_wait_in_msec = 1000/5*18*CAR_WHEEL_2_PI_R/random_speed;
      tslp_tsk (time_to_wait_in_msec/10);
      asm INT WHEEL_INTERRUPT_NO
    }
  }
}
static unsigned int rsl[55];
void init_rand(int seed)
{
  int i;
  for (i=0; i<55; i++) rsl[i] = i^seed;
}
int get_random (int max)
{
  int i;
  for (i=0; i<55; ++i) rsl[i] = (rsl[i] + rsl[(i+24) % 55])%max;
  return rsl[0];
}
void init_display()
{
  initscr ();
  print_fast_from_now_on();
  gotoxy (3, 5); print_string ("Speed: ");
  gotoxy (4, 5); print_string ("TDist: ");
  gotoxy (5, 5); print_string ("CDist: ");
  gotoxy (3, 35); print_string ("Fuel: ");
  gotoxy (4, 35); print_string ("Batt: ");
  gotoxy (5, 35); print_string ("Temp: ");
  gotoxy (3, 67); print_string ("Tyre Pressure");
```

```
  gotoxy (7, 68); print_string ("Door Locks");
  gotoxy (23, 0);
  print_string ("‘a z’ ‘f v’ ‘b n’ ‘t g’ ‘p1-4 l1-4’ ‘d1-4 c1-4’");
  door_lock_handler();
}
void init_library (struct parameter_block_header* message)
{
  init_u_int_h (message);
  init_u_task_m (message);
  init_u_comm_m (message);
  init_u_time_m (message);
}
void keyboard_handler (void)
{
  char scancode,temp;

  scancode = in_port_1b (0x60);
  temp = in_port_1b (0x61);
  outport_1b (0x61, temp | 0x80);
  outport_1b (0x61, temp);
  if (scancode > 0) char_read = scancode;
}
char get_key ()
{
  char c;
  char *scan_to_ascii =
    "?1234567890-=\b\t"  /* Scan codes 00 to 0f */
    "qwertyuiop[]\n?as"  /* Scan codes 10 to 1f */
    "dfghjkl;’‘?\\zxcv"  /* Scan codes 20 to 2f */
    "bnm,./"             /* Scan codes 30 to 35 */
    ;
```

```
  while (char_read <= 0)
  ;
  c = char_read;
  char_read = 0;
  if (c == 0x39) return ' ';
  if (c == 1 || c == 0x1d || c == 0x2a || c > 0x35)
    return c;
  return scan_to_ascii[c-1];
}
```

Bibliography

[CRT] CRTX. URL. *http://www.n2.net/starcom/crtx.html.*

[ES] AI Lab Zurich : Links : Embedded and Real-Time Systems. URL. *http://www.ifi.unizh.ch/groups/ailab/links/embedded.html.*

[Kri97] Kshitiz Krishna. A modular kernel architecture for embedded systems. Master's thesis, Indian Institute of Technology, Kanpur, India, 1997.

[oCH] National Research Council of Canada Harmony. URL. *http://wwwsel.iit.nrc.ca/projects/harmony.*

[OS/] Microware Systems Corporation OS/9. URL. *http://www.microware.com/ProductsServices/Technologies/os-91.html.*

[POS92] POSIX. System application program interface - amendmment 1: Real time extension. ieee project p1003.4, draft 13. In *Portable Operating System Interface Part 1*, September 1992.

[RTE] Redstone Military Arsenal RTEMS. URL. *http://lancelot.gcs.redstone.army.mil/rtems.html.*

[Sak93] Ken Sakamura. Industrial specifications 3.0 - for micro kernels. In *ITRON 3.0 An Open and Portable Real-Time Operating System for Embedded Systems.* http://tron.um.u-tokyo.ac.jp/ TRON/ITRON/spec-e.html, June 1993.

[VxW] Wind River Systems VxWorks. URL. *http://www.wrs.com/html/vxwks52.html.*