Table of contents

ABSTRACT **1**

ACKNOWLEDGEMENTS **2**

TABLE OF CONTENTS **3**

CHAPTER 1. INTRODUCTION **6**

1.1 REAL-TIME KERNEL FUNCTIONS 7
1.2 REAL-TIME KERNEL ARCHITECTURE 8
1.2.1 Monolithic Kernels *8*
1.2.2 Object-based Kernels *9*
1.3 REAL-TIME KERNEL OPERATION 10
1.3.1 Tick-driven Kernels *10*
1.3.2 Event-driven Kernels *10*
1.4 SURVEY AND ANALYSIS OF EXISTING KERNELS 11
1.4.1 Real-time kernel from Danfoss A/S *11*
1.4.1.1 Data structures *12*
1.4.2 OSEK/VDX Kernel *13*
1.4.3 Asterisk Kernel *14*
1.4.4 HARTEX Kernel *16*
1.5 REQUIREMENTS FOR AND FEATURES OF THE KERNEL BEING DEVELOPED *18*

CHAPTER 2. KERNEL ORGANIZATION **21**

2.1 SUBSYSTEMS – THEIR FUNCTIONS 21
2.1.1 Task Manager *22*
2.1.2 Integrated Event Manager *22*
2.1.3 Resource Manager *23*
2.1.4 Software Bus *23*
2.1.5 Hardware Adaptation Layer (HAL) *23*
2.2 SUBSYSTEM INTERACTIONS FOR KERNEL OPERATION 24

CHAPTER 3. TASK MANAGEMENT **26**

3.1 TYPES OF BASIC TASKS 26
3.1.1 Simple tasks *26*
3.1.2 Compound tasks *26*
3.2 TASK MANAGEMENT 27
3.2.1 Basic task's state transition diagram *27*
3.2.2 Task management data structures *28*
3.4 BASIC TASK SCHEDULING WITHOUT RESOURCE MANAGEMENT 30
3.4.1 Task manager organization and startup *30*
3.4.2 Task management private functions and primitives *34*
3.5 INTEGRATED TASK AND RESOURCE MANAGEMENT – *SBPC PROTOCOL* 36
3.5.1 Stack-Based Priority Ceiling (SBPC) protocol *37*
3.5.2 Integrated task and resource management – implementation *38*
3.5.3 Private functions and primitives *38*
3.6 CASE STUDY 44

CHAPTER 4. TASK INTERACTION 47

4.1 TASK SYNCHRONIZATION 47
4.1.1 Event notification through Boolean vector semaphores – primitives 49
4.1.2 Synchronization primitives 51
4.1.3 Illustration of usage of primitives 52
4.1.4 Different patterns of task synchronizations 53
4.1.4 Different patterns of task synchronizations 54
4.2 TASK COMMUNICATION VIA CONTENT-ORIENTED MESSAGE ADDRESSING 56
4.2.1 Content-oriented message addressing 56
4.2.2 Communication primitives and private functions 59
4.2.3 Illustration of usage of communication primitives to get content oriented message addressing 63
4.2.4 Merits of this communication in Kernel 66

CHAPTER 5. COMPOUND TASKS AND SECONDARY-LEVEL SCHEDULING ALGORITHMS 68

5.1 COMPOUND TASK AND SUBTASKS 68
5.2 SUBTASK SCHEDULING ALGORITHMS 69
5.2.1 FIFO style 69
5.2.2 Static Cyclic Scheduling using the Boolean Vector Semaphores 70
5.2.3 Execution of a arbitrary sequence of subtasks 77
control_memory_table 83

CHAPTER 6. EVENT MANAGEMENT 84

6.1 EVENT DESCRIPTOR 85
6.2 TIME AND EVENT MANAGEMENT IN INTEGRATED EVENT MANAGEMENT 87
6.2.1 Basic Event Processing 87
6.2.2 Essential Event Management 88
6.2.3 Event Descriptor Table (event_descriptor_table) 89
6.3 IMPLEMENTING INTEGRATED TIME AND EXTERNAL EVENT MANAGEMENT 89
6.3.1 Basic event processing implementation 90
6.3.2 Integrated event management primitives 91
6.4 EVENTS OF 'ONE_OFF' TYPE: THEIR SIGNIFICANCE 95

CHAPTER 7. CONCLUSION 97

CHAPTER 8. REFERENCES 98

APPENDIX 99

1. MAIN FUNCTION 99
2. TASK.C 101
3. TASK.H 103
4. EVENTS.C 104
5. EVENTS.H 105

6. RESOURCES.H 106
7. SEMAPHORE.C 107
8. SEMAPHORE.H 108
9. MESSAGE.C 109
10. MESSAGE.H 110
11. TASKMANAGER.C 111
12.TASKMANAGER.H 113
13. RESOURCEMANAGER.C 114
14. RESOURCEMANAGER.H 115
15. RESOURCEMANAGER_.H 116
16. INTEGRATEDEVENTMANAGER.C 117
17. INTEGRATEDEVENTMANAGER.H 119
18. INTEGRATEDEVENTMANAGER_.H 120
19. SYNCHRONIZATIONBUS.C 121
20. SYNCHRONIZATIONBUS.H 122
21. SYNCHRONIZATIONBUS_.H 123
22. COMMUNICATIONBUS.C 124
23. COMMUNICATIONBUS.H 126
24. COMMUNICATIONBUS_.H 127

Chapter 1. Introduction

Embedded systems are increasingly being used in several areas, which include military applications, aerospace systems, automotive systems, industrial automation, medical instrumentation measurement instrumentation etc [1]. Further most of such systems use small microcontrollers. Depending on the context of application where the embedded system is deployed the speed of response and safety requirements vary.

Embedded systems are also known as real time systems since they respond to an input or event and produce the result within a guaranteed time interval (deadline). This time interval can be from few milliseconds or more. Real time systems are further classified as hard real time systems and soft real time systems, based on the strictness to the time period. A hard real time system should complete the specified task within the stipulated time frame. A failure to do so is treated as the failure of the system. Hence hard real-time systems are deterministic. A soft real time system is not very strict. Not completing a task within the time frame is pardonable, provided it will not affect the overall system performance. The consequences for failing to meet timing limits range from mere inconvenience, to the loss of human life. The term real-time has evolved to refer to any application in which a computer is used to control a process.

Just like the open-end computer systems (full fledged systems like personal computer) where there is a separate operating system that handles underlying hardware and provides the wanted support on which the application software resides and works, the embedded systems also have a different operating system called Real-time operating system (RTOS) so that the application and application software developers need not concentrate much on the issues other than application. However it is possible to develop a system without RTOS, which will have no much control over response and priorities.

A real time system may be doing some simple task to complex tasks with many decision loops and interrupts. Often in real time applications the entire job to be done is split into several portions called tasks. Real Time Operating Systems (RTOS) are used to schedule tasks in complex systems, so that the needed end functionality of the system is derived very efficiently.

An RTOS helps to schedule and execute tasks based on priority in a predictable manner. So RTOS has to be multitasking and preemptable. It also should have predictable task synchronization mechanisms. Therefore usage of RTOS will enable you to break the complex system into smaller tasks without worrying about the inter task timing problems. Depending on the complexity and requirements of the application the services to be provided by RTOS vary.

1.1 Real-time Kernel Functions

All the RTOSs have a core called Real-Time Kernel, which provides an operational environment for application tasks via a number of services. It is the Kernel that decides which thread (task) is run at each point in time. It is the kernel on which the rest of the OS is built. The basic system services provided by Kernel are-

- External event management
- Timing event management
- Task management
- Resource management
- Task synchronization
- Task communication

In addition to these basic functions it could provide support for additional system functions, such as-

- Dynamic memory management
- Memory protection
- Peripheral device drivers
- Basic user interface
- Network communication

The previously said functions are a must for any RTOS and are implemented by the Kernel. Small-embedded systems, which are built around a single microcontroller, have very limited resources in all respects, so we cannot afford for a RTOS whose demands exceed the chip resources. In such cases a Real-time Kernel implementing the basic functions could substitute a RTOS. In the current project such a Real-time Kernel has been designed and implemented for AVR microcontroller. This kernel is named as **HARTEX$_\mu$** pronounced as hartex micro.

1.2 Real-time Kernel Architecture

In developing kernels one can follow different architectures deriving the basic functions needed. There are two popular architectures [1] and hence two categories of kernels:

- Monolithic Kernels
- Object-based Kernels

Kernel routines irrespective of the architecture can be broadly subdivided into two classes: internal functions and public system functions (system calls), which may be invoked from within application tasks. These can be structured into a number of layers, e.g. service layer, process management layer, process list management layer etc., and machine-dependent layer. Such a layering provides for kernel portability across different chips, since it is only the lowest layer that is hardware dependent. This last layer is called the *hardware adaptation layer*.

1.2.1 Monolithic Kernels

This is a conventional type of architecture where in the Real-time Kernel is built from specific routines, which share common data structures – tables, queues etc. The Kernel interacts with user-supplied tasks and interrupt service routines (ISRs). The specific routines themselves give the services said earlier. A real-time system implemented by such kernel can be seen in the Fig.1.

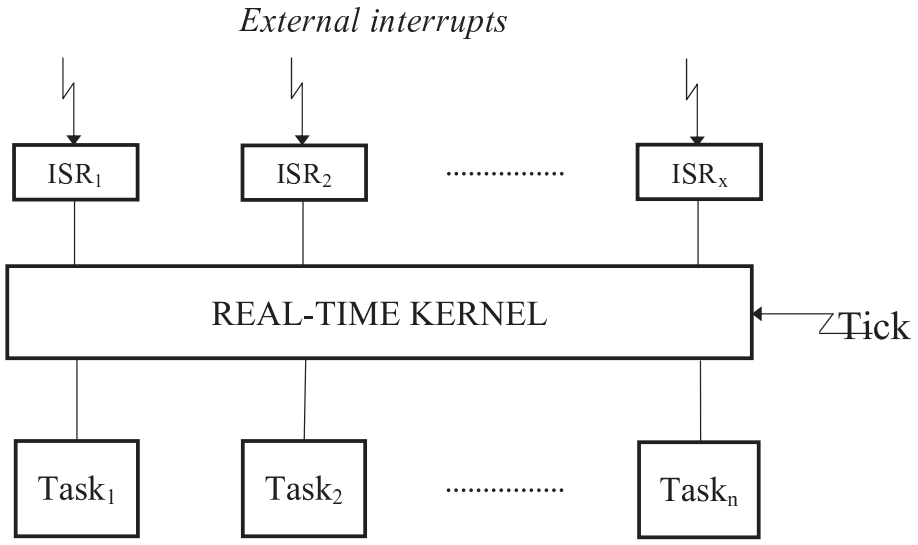


Figure1. Real-time system with a Monolithic Kernel

1.2.2 Object-based Kernels

This architecture emphasizes encapsulation of data and the corresponding system functions into subsystems (objects), whereby subsystems functions are made accessible only through appropriate object interfaces. This division into subsystem is done such that each subsystem contributes towards one service mentioned earlier. Object interfaces consist of internal calls that are invoked by other kernel objects and public (system) calls that may be invoked by application tasks. A real-time system with Object-based Kernel is shown in Fig.2.

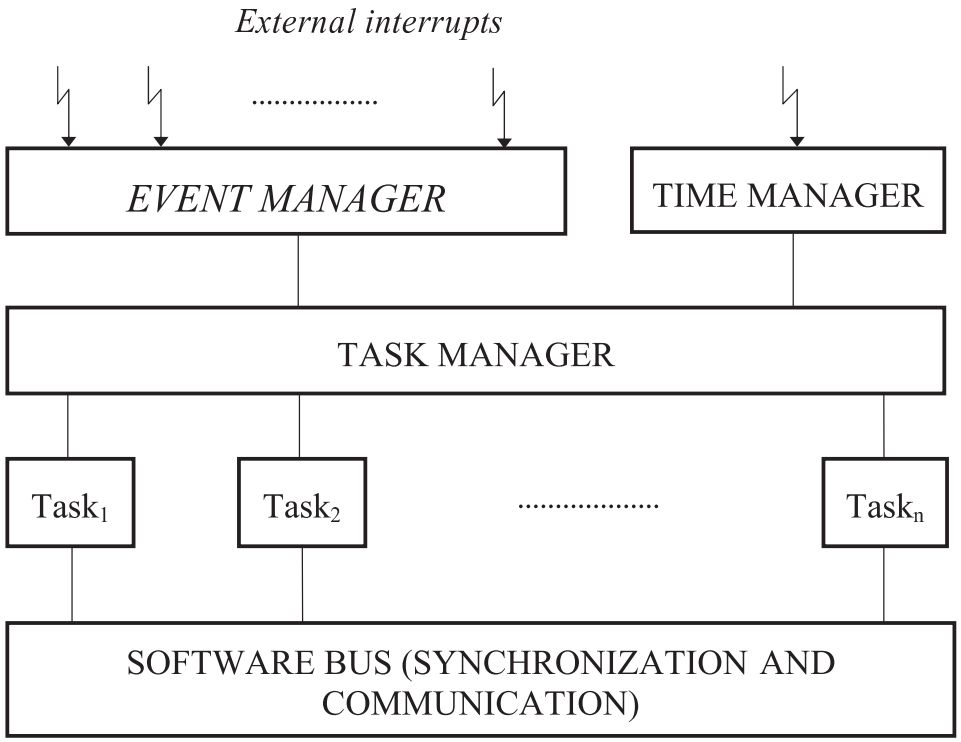


Figure 2 . Real-time system with Object-based Kernel made of various subsystems.

The RTOS consists of a real-time kernel at its core and number of system processes, which provide an extended operational environment for the user processes. The extended operational environment includes support for network communication and associated protocol stacks, file system, graphic user interface, etc. The architecture of RTOS followed a similar trend featuring two types of architectures: *Monolithic RTOS* and *Microkernel Architecture*. But the details of these architectures are of no significance in the current project.

1.3 Real-time Kernel Operation

The kernel has to be called into operation by activating it explicitly. There are two types of kernels with regard to the mechanism of kernel activation:

- Tick-driven Kernels
- Event-driven Kernels

1.3.1 Tick-driven Kernels

These kernels are activated by periodically arriving tick (timer) interrupts resulting in synchronous kernel operation. Further it is possible to adopt two strategies with regard to external event processing:

- External interrupts enabled
- All interrupts except the tick interrupt are disabled

Obviously in the former case the external interrupts are acknowledged and according to their ISR code they may generate requests for the execution of application tasks but these are recognized by the task manager only when the kernel is activated on the next nearest tick instant. This results in a variation of task release times, which is known as *task release jitter*.

1.3.2 Event-driven Kernels

These kernels are activated by sporadically arriving external interrupts resulting in asynchronous kernel operation. Further it is possible to adopt two strategies with regard to generation of timing events:

- without additional timing facilities i.e., timing events are generated exclusively by hardware timers
- with additional timing facilities implemented in software

In practice real-time kernels often use both types of activation mechanisms described above resulting in combined tick-driven and event-driven operation.

1.4 Survey and analysis of existing Kernels

The basic duty of kernel is to do task switching following the defined scheduling policy; and the efficiency in doing this duty is defined by how much overhead is placed in terms of time taken, memory taken etc.

In all real-time application developments whether small or complex it is always advantageous to use a kernel. The reason is using a kernel makes it easier:
to design the structure of application software (firmware)
to service and maintain the system
in further development of the system

Owing to these reasons many software developer groups have developed several kernels following their own requirements. Several of such kernels are discussed in the following sections starting with the classical or conventional type, then recent kernel in use for automotive controls systems and finally discussing a kernel with all latest state-of-the-art in real-time field put into action.

1.4.1 Real-time kernel from Danfoss A/S

This kernel is developed by Danfoss A/S for their real-time systems to be deployed on H8/3002 microcontrollers with 8KB RAM [2]. The requirements from this kernel were:

1. The kernel has to run on a H8/3002 microcontroller with 8KB RAM in the large memory model using development tools from IAR (C-compiler v3.31B, Assembler v 2.11J, Linker v4.48D, Librarian v3.26J).
2. The kernel must support both pre-emptive and non pre-emptive scheduling.
3. It must be possible to create tasks with different stack size (dynamically allocated).
4. A task must be declared as a simple function with two parameters: an integer parameter and a pointer parameter.
5. Semaphores must be available (for monitor-protection, mutual-exclusion and signaling from non tasks).
6. A timer for each task must be available (to let a task sleep an amount of time). The resolution for such timer must be at least 100ms.

Following the requirements stated earlier and their design, the kernel had following data structures that obviously result in lot of jitter due to the processing involved of such data structures.

1.4.1.1 Data structures

The kernel has the following data elements:

1.**Task Pool:** A task pool where the information needed for each task is stored. Each task requires following elements-
A stack pointer to its stack
A timer
A pointer to the first allocated address of the stack
A pointer to the next task in the actual queue

Each task has one such structure, and is called task structure. So kernel has as many task structures as the maximum number of tasks under the kernel.

2. Queues: Three queues one each for: killed tasks, tasks that want to run and for tasks waiting for their timer to run out. Besides these three queues there is one more queue for the semaphores. Further all the queues have the same data structure with three fields: A semaphore value (only for semaphore queue), pointer to first task in the queue, pointer to last task in the queue. The necessary coupling in the queue is obvious from the task structure.

The synchronization and communication is totally busy waiting or blocking style. And tasks are having their own stack areas. Tasks and semaphores are created and killed. It is clear from the data structures that the processing overhead because of kernel is very high both in terms of time and memory demands on the microcontroller. It is obvious that processing of queues is time-consuming and results in considerable and varying overhead (kernel jitter) hence unpredictable response time. No protocol is employed for the purpose of resource sharing which may result in deadlock and may also result in unbounded priority inversion adding more to the problem of unpredictable behaviour.

As the tasks and semaphores are created and killed dynamically, there is every need for memory management to avoid the problems associated with memory fragmentation. A major shortcoming of testing with such a safety critical systems with dynamically allocated memory is that no complete test suite can be created to cover every possible combination of events and inputs that could create a potential error.
Besides all this the communication is not transparent.

1.4.2 OSEK/VDX Kernel

The OSEK/VDX operating system specification was developed by a consortium of automotive companies and suppliers, and is a branded kernel specification for automotive applications. The OSEK operating system is an operating system meant for distributed embedded control units. So this specification doesn't define the implementation details, following this specification several commercially available operating systems and kernels (OSEK/VDX compliant) are developed for a variety of 8, 16 and 32 bit microprocessors.

In order to comment on this specification lets take a look on the goals and the problems the specification is designed to solve.

The specification [3] actually includes specifications for three different components, a kernel, a communications module, and a network management module. A briefing on kernel specifications is given below. The goals in the definition of the OSEK/VDX kernel are:

- Isolation of the developers from the unnecessary details of their target hardware
- Supplying developers a rich set of kernel features and objects to simplify the implementation of embedded applications
- Facilitating the integration of software developed by different entities.

An entity is defined as another developer, team or supplier. The kernel specification defines-

1. A static configuration approach for scalability,
2. A highly efficient scheduling policy, which can be switched between preemption and non-preemption, thus resulting in a mixed policy.
3. Support for portability of application tasks,
4. The ability to be ROMable.

1.4.2.1 OSEK/VDX features

There several advanced features in OSEK/VDX compared to the conventional kernels, they are:

1. It supports basic and extended tasks.
2. Employs Priority ceiling protocol to avoid problems like deadlock and unbounded priority inversion.
3. Employs a synchronization protocol.
4. Advanced communication- involving state and event message communication.
5. Events and event counters are used.

The operating system specification was also designed to address stringent real-time requirements, minimal resource usage, also reliability and cost sensitivity. In all the implementations of such specification, there is always a queue processing involved which are around semaphores or tasks that results in jitter. None of the implementations used advanced and efficient concepts of employing binary vectors for task management, resource management, synchronization and communication.

The specification is successful in its own purpose but due to implementation involving queues the potency of binary vectors is not exploited inmost of the developed kernels of RTOS. So the problem of jitter arising due to kernel is not optimized to the maximum point. Thus such implementations still hold the traits of same problems that were in conventional kernels discussed in earlier section because of lists etc.

1.4.3 Asterisk Kernel

This is the kernel developed by MRTC that makes use of all the latest advancements in the real time system concepts like usage of binary vectors, binary semaphores, non-waiting type of task interaction (synchronization and communication). Also this is meant for small-embedded systems with minimum demands on the microcontroller. The features of Asterisk kernel[4] are:

- A task-model that supports state-of-the art scheduling theory.
- Support for debugging and monitoring.
- Wait- and Lock- free inter-task communication.
- Scalable kernel, meaning that only those subsystems wanted are utilized.
- Predictable performance.

In addition to these there are common features like portable, compilable etc.
Following the requirements the kernel consists of a Task Control Block (TCB) structure that has several relevant fields holding the information about the task it belongs to. Therefore the implementation has a list called TCB list, representing all the tasks in the system. Further each task has its own stack to be used for context storing on context switching because of preemption by higher priority tasks.

The block diagram of Asterisk is shown in Fig.3. The tasks under the kernel are basic tasks, meaning the tasks once started are never blocked, but can be preempted by the other higher priority tasks if they are released.

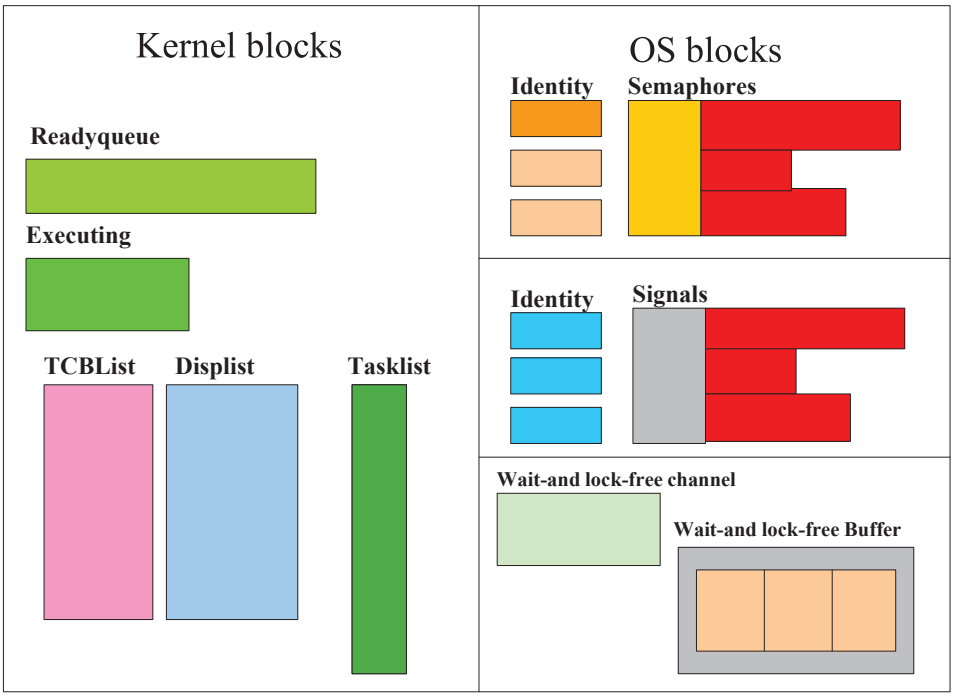


Figure. 3 Asterisk kernel with data structures

In this kernel only the ready task queue (i.e., TCB list) is implemented with the non-traditional data structures where in a binary string or word or vector is used as the queue. The bit position in the word itself is used as the index to identify the associated task. Thus the bit position does the r ole of two things: priority and task identifier in the TCB table, which is implemented as an array. Implicitly this binary vector approach gets the job of priority ordering done. But for other purposes like communication and synchronization still is based on queues and conventional structures. Thus the conventional queues resulting in very big data structures are not completely over come.

The kernel employs Immediate Inheritance Protocol algorithm to avoid problems associated with mutual exclusion on shared resources. The semaphore's structure consist lot of information like list of owners, number of owners etc. Synchronization between tasks is achieved by means of signals. And the communication is a non-blocking type named as Wait- and Lock- free communication, which is implemented with an array of buffers where in the data to be exchanged, is placed.

However Asterisk kernel used the binary vector concept very limitedly and hasn't exploited it completely, still the conventional concepts are prevailing as far as communication and synchronization is concerned. The binary vectors are not employed inside the semaphore and signal implementations that make room for the jitter to creep into the communication and synchronization operations of the kernel. In making the kernel to be predictable several dummy paths and dummy coding has been put in. Moreover the lists are traversed until the end even if the element being searched is found before the end of list. It is nothing but wasting of processor cycles which is not advisable.

1.4.4 HARTEX Kernel

This is a kernel developed for distributed hard real-time distributed computer control systems (DCCS). The novel techniques are exploited completely in all the kernel functions, which resulted in a kernel with much less jitter and memory overhead [5]. These novelties are:

- Integrated scheduling and management of tasks and resources via Boolean vector processing.
- Integrated scheduling of soft and hard real-time tasks.
- High performance time management for safe DCCS operation
- Synchronization and Communication among tasks through event notification via binary vector semaphores.
- Communication through implicit (Content-oriented) message addressing.
- No queue or list type data structures because of binary vectors.

Because of all the novel techniques used the kernel resulted in:

- absolutely predictable scheduling of hard real-time tasks
- predictable interaction between real-time tasks
- predictable and deterministic operation of kernel subsystems.

HARTEX is an object-based kernel with its organization as shown in Fig.4. The modules or subsystems are: Event Manager, Task Manager, Time Manager, Software Bus and Hardware Adaptation Layer. All the names are self-explanatory.

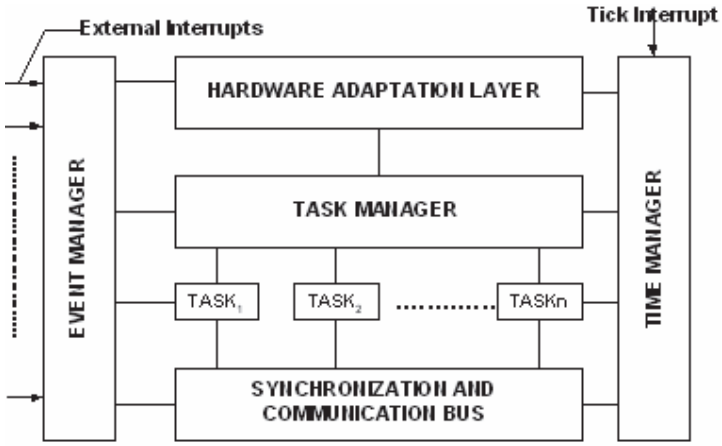


Figure.4 HARTEX organization

The main breakthrough techniques [5] in tick driven HARTEX are:

- Boolean vector processing hence constant execution time of kernel functions independent of the number of tasks involved..
- Instantaneous signaling of multiple tasks about an event and message using Boolean vector semaphores.
- Content-oriented message addressing that eliminates the problem of specifying the source, destination tasks and/or communication objects like mailboxes etc

All these techniques are customized for small embedded systems and new concepts are added resulting in present kernel HARTEX$_\mu$. .

The goals of the new kernel HARTEX$_\mu$ are:

- simple task and time management involving only basic tasks.
- new protocol for integrated task and resource management
- integrated timing and external event management
- non-blocking version of synchronization and communication

1.5 Requirements for and features of the kernel being developed

As part of the thesis "a *scalable real-time kernel for small embedded systems*" has been designed and implemented. The requirements are defined in a manner that the memory and response overhead because of kernel are as minimum as possible, but not compromising on the basic functionality of the kernel. The requirements are:

1. The kernel is meant for a single microcontroller system.
2. The kernel should manage and support basic tasks only following the specific tasking model.
3. It should follow a fixed priority preemptive scheduling policy with a provision for mixing static-cyclic scheduling with event-based scheduling.
4. It should be possible to switch the mode of operation of kernel between preemptive and non-preemptive.
5. The memory footprint of the kernel should be as small as possible.
6. The response should be very fast with minimum jitter because of kernel.
7. It should support non-blocking style of synchronization and communication.

This kernel is a tick-driven one and falls under the category of object-based kernels as described under the section 1.2.2.The features of the kernel are:

- **Usage of binary vectors:** Wherever possible binary vectors are used. Therefore no traditional data structures like queues, lists etc. The position of the bit itself is the identifier of the task and the priority of the task, thus the vector itself gets the behaviour of priority ordered queue. While all the details pertaining to a task are placed in a table implemented as array, thus a lookup operation performed into such array with the identifier gives the task information instantaneously.

- **Integrated event and timing management:** A uniform approach is followed in treating the tick (timer) interrupts and external events, all are treated as events and a event descriptor is defined over such event that manifests the activity to be carried out on the occurrence of such event. Thus the tasks can be released when appropriate time lapses or threshold of external events occurs.

- **Usage of binary vector semaphores:** Semaphores are used as synchronization and communication objects, which make it possible to instantaneously notify a number of tasks about the event occurrence or message ready/arrival.

- **One common stack shared by all tasks:** All the basic tasks under the kernel share a common stack contributing to lesser memory overhead.

- **Integrated resource and task management:** An elegant protocol called SBPC protocol [1] is employed to achieve this integrated management. This is explained under chapter 3(section 3.5)

- **Content-oriented message addressing:** Very advanced communication technique called content-oriented message addressing is implemented. By this approach the messages are addressed amongst the tasks just by the name of the variable being communicated. Thus freeing the user tasks and application developers from all the associated details of senders, message size, source and destination etc., thus providing a transparent communication.

- **Scalability and compilability:** Kernel is scalable and compilable according to the application requirements by configuring the subsystems. By compilable it means that the data structures in the kernel are allocated depending on the number of tasks in the application, meaning that if the application has only 5 tasks then only that many resources needed are allocated else if there are 15 tasks then the resources for those many tasks are allocated. This is made possible because of the static configuration made possible in the kernel.

- **Layered structure:** Layered organization of kernel provides for easy portability across different chips (as of now it is for AVR 8-bit microcontrollers). The kernel is designed and implemented in the form of layers, which interact with their adjacent layers by appropriate interfaces provided. Thus only the bottom most layer next to hardware (called hardware adaptation layer) is only hardware dependent. By replacing this layer as per the deployment hardware the kernel is made completely portable.

- **Secondary-level scheduling:** By exploiting the integrated event management and semaphores different types of scheduling of hard real-time tasks is possible. Thus the kernel provides First In First Out (FIFO), Static cyclic and Arbitrary sequence (evaluated at run time) scheduling. This is explained under chapter 5.

Chapter 2. Kernel Organization

The kernel is organized into various modules or sub-systems as shown in Fig 5. Each subsystem is encapsulated into system objects, which interact with each other through the appropriate interfaces provided. These interfaces are called internal calls. Each subsystem provides certain public calls that can be invoked by other subsystems and/or application tasks (which form the firmware). All such subroutines forming the interface are called as primitives.

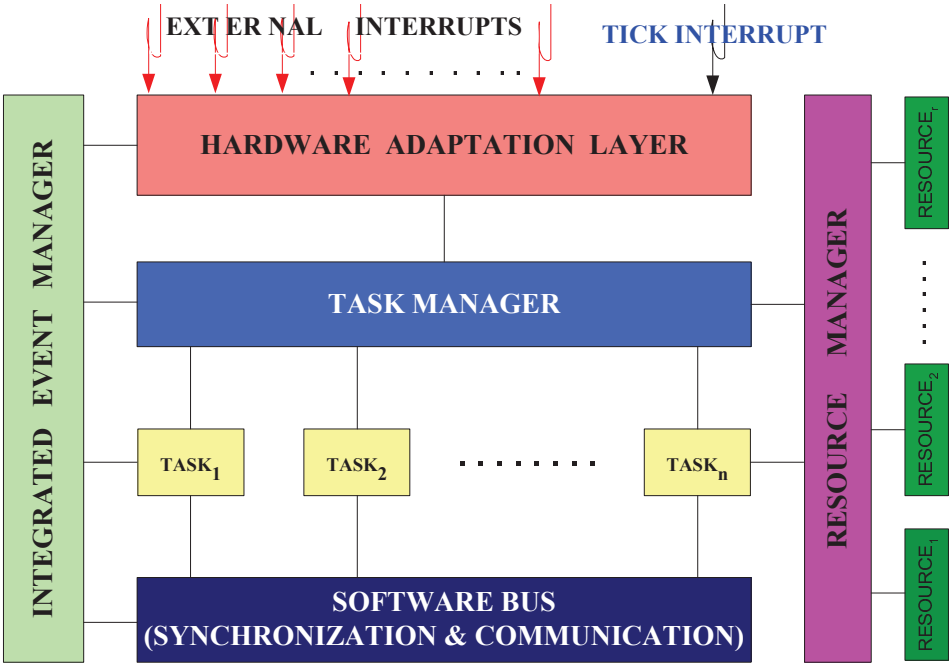


Figure 5. The structure of the small kernel developed

Analyzing the kernel functionality and partitioning the total system into subsystems with standard interfacing results in greater flexibility and scalability, because the subsystems can be customized according to the individual applications requirements without influencing the internal structure and implementation of other objects (subsystems).

2.1 Subsystems – their functions

The kernel subsystems are:

- Task Manager
- Integrated Event Manager
- Resource Manager
- Software Bus
- Hardware Adaptation Layer

2.1.1 Task Manager

Task manager is responsible for scheduling the tasks following the scheduling policy and managing the tasks state transitions following the state-transition diagram (Fig 8). It encapsulates the necessary data structures for this operation and provides the interface that can be used by other subsystems of the kernel. It is responsible for context switching (context saving and restoring) on task preemption and resumption.

2.1.1.1 Requirement specifications and associated implementation guidelines

1. The kernel must support basic tasks only. Basic tasks have a liner structure and are invoked as subroutines, and all such basic tasks use the same common stack.
2. The kernel must support both preemptive and non-preemptive priority scheduling. To this end there are two primitives ***enable_preemption( )*** and ***disable_preemption( ).***
3. Task management should be done according to the basic task transition diagram (Fig. 8) of the task model in the kernel. Thus the task scheduler is made up of two primitives or scheduling routines ***preempt( )*** and ***schedule( )***. The latter is used when a task comes to an end and a new task has to be started, while *preempt( )* is used when the current task has to be preempted by a higher priority task. This *preempt( )* primitive encapsulates the context-switching with the scheduler.
4. The kernel should support two types of tasks: one is simple basic tasks (with linear structure) other is a compound basic task consisting of several subtasks, which should able to be scheduled following a secondary-level schedule policy local for this task only. To decide the execution sequence and subtasks to be executed semaphores are used.(see Chapter 5).

2.1.2 Integrated Event Manager

This subsystem manages timing and external events in an integrated approach, and invokes appropriate primitives in task manager and the software bus that inturn take relevant action. More about this subsystem is explained in chapter 6.

2.1.2.1 Requirement specifications and associated implementation guidelines

1. The external events and internal timing events are to be treated uniformly. So there is a common data structure for all event descriptors.
2. The events are to be obviously updated by HAL. There two primitives needed that are invoked on appropriate events from HAL. (See chapter 6).

2.1.3 Resource Manager

This module makes sure that the shared resources are accessed in a mutually exclusive manner. It uses a protocol that results in integrated task and resource management.

2.1.3.1 Requirement specifications and associated implementation guidelines

1. All the shared resources must be accessed in an atomic manner. For this there are two services through which the application tasks can lock or free the resource, which are provided by *lock( )* and *unlock( )*.
2. Further it should support the integrated task and resource management following Stack-based Priority Ceiling protocol. For this purpose a framework is setup as explained in section 3.5.

2.1.4 Software Bus

This module takes care of providing the duties necessary for inter task synchronization and communication and is explained in greater detail under chapter 4.

2.1.4.1Requirement specifications

1. It should implement the interaction following the single node local interaction version of HARTEX Communication protocol consisting of Event notification layer and Content-oriented messaging layer.
2. Execution time for messaging or event notification between tasks must be independent of number of tasks involved in the communication. This is made possible by Boolean vector semaphores.

2.1.5 Hardware Adaptation Layer (HAL)

Event counters are linked with the basic event processing mechanism. Event counter is updated following the basic event processing. HAL invokes the Integrated Event Manager following the basic event processing.

2.1.4.1 Requirement

The interrupt service routines (that are meant for kernel services) should be very short so that the jitter is minimized.

2.2 Subsystem interactions for kernel operation

All the subsystems interact with each other to meet the requirements and to derive the needed functionalities of the kernel. For this purpose each subsystem has certain data structures of which some are global and some are private accessible by means of appropriate interfaces (primitives and calls) provided. Thus subsystems interact by calling the relevant primitives under each other.

As soon as the kernel is put into operation first HAL takes control and initializes all the needed hardware, then it operates on event descriptors eventually invoking the Integrated Event Manager on various events and Task Manager directly by the appropriate primitives. Further the Integrated event manager according to the event descriptors invokes the Task manager and software bus via ***release***, ***signal and broadcast*** primitives. Task manager following the calls from other subsystems schedules the tasks under it. The tasks and task manager interact with resource manager whenever a shared resource is to be used with mutual exclusion.

While the tasks on reaching the synchronization points of signaling and checking whether an event is signaled or not, call primitives (***signal_and_release*** and ***test_and_reset***) with synchronization bus and finish their execution sequence accordingly. In communication case the tasks call primitives (***broadcast*** and ***receive***) with the communication bus.

To start the explanation of interaction of subsystem we start with the integrated event manager module. In this subsystem, there is a data entity that contains details about which tasks are to be released, which tasks are to be signaled on occurrence of a certain threshold of associated specific events. Further this data entity also contains the particulars of semaphores that are to be employed in the case of signaling to achieve the synchronization associated with event occurrence. This entity is called as *event descriptor*. The operations release, signal, and send message operate over the binary vectors concerned with appropriate modules.

Now move to task manager, this module operates totally on and around one Boolean vector called Active Task Vector, which can be modified by calling the primitives provided. The ATV is updated by the integrated event manager according to the descriptors of all the events (that are notified by interrupt service routines by HAL subsystem) as per the application.

Software bus module provides data structures and primitives to access them so that the inter-task synchronization and communication is done. There are separate semaphores for synchronization and communication. When a task has to be signaled about an event, the bit under the flags vector of the semaphore is set. One should remember that an event is mapped to one semaphore. The task waiting on such event on reaching its synchronization point checks the value of the bit corresponding to it, if set (i.e., event has occurred) the task continues its operation, if not set just exits following the non-blocking style of communication. When coming to communication, the arrival or readiness of message is signaled by means of semaphores meant for communication and the receiver task on reaching message reading point checks the semaphore mapped to the message in which it is interested, if the message is ready the receiver task starts copying from source buffer to its local buffer.

In the Hardware adaptation layer (HAL) the basic event processing is done invoking integrated event manager where the important activity of event counter updating is carried out. In this module (HAL) a timer interrupt updates some flags as per the lapsed time and invokes event manager on various events, while in the external interrupt service routines the external event basic processing is done. Integrated event manager updates the event counters. When the counter expires necessary action is taken. This action is defined during initialization of *event descriptors*, and is called as *op-code*. Based on the op-code one or more different primitives under various modules are called. . Thus HAL interacts with Task manager and Integrate event manager modules.

To achieve mutual exclusion of shared resources the resource manager primitives- **lock (***resource***)** and **unlock (***resource***)** are called in, thus the resource manager enters into the interaction ground. All these interactions are clearly explained under subsequent chapters.

Chapter 3. Task Management

The application software (called as firmware) in real-time systems is decomposed into several discrete, significant and appropriate smaller jobs. These portions of work are implemented as subroutines and are referred to as *tasks*. All the tasks have a fixed statically defined priority. Based on the task's execution characteristics there are two types of tasks: *basic tasks* and *extended tasks*. Basic tasks once start executing can never be blocked, while extended tasks are those which can be blocked in between their execution sequence waiting for some signal or event etc.

3.1 Types of basic tasks

All the tasks are implemented as subroutines with no parameters. Further to implement the hard real-time tasks and ordinary tasks, the tasks under the task manager of current kernel are categorized into two types: simple tasks and compound tasks.

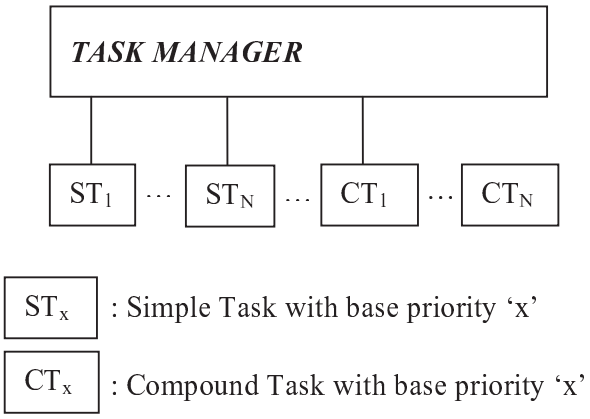


Figure 6. Task Manager and application tasks

```
simpleTask_X (void)
{
  //local declaration;
  statement1;
  statement2;
  //etc
  taskExit( );
}
```

Figure 7. A simple task

3.1.1 Simple tasks

These are the simple basic tasks with a fixed sequence of code and no subtasks involved. Such a task is implemented in code as shown in Fig 7.

3.1.2 Compound tasks

To facilitate for the scheduling of real-time tasks following a different scheduling policy from that of the primary task manager, several tasks are made as subtasks and put together under one task, which is called as *compound task*. The scheduling policies can be- FIFO (first in first out), static-cyclic scheduling or arbitrary sequence.

3.2 Task management

In a single processor system the kernel will be carrying out multitasking by allocating microprocessor to various tasks but to only one task at a time. Therefore all the tasks including the running task and other tasks should be in some meaningful conditions referred to as ***states***. The task model in the kernel consists of following states:

- ***Running state*** - This corresponds to the situation when the task is being executed, it is to be noted that in a single processor systems only one task can be in this state.
- ***Ready state*** - This corresponds to the situation when the tasks are ready to be executed but are not currently being executed. One or more tasks can be in this state.
- ***Preempted state*** - This is a status when a task has been preempted by a higher priority task. Again one or more tasks can be in this state.
- ***Inactive state*** - This is the case when the task has finished its execution and is no more in any of the other three states.

The task manager manages the tasks following the state transition diagram explained in the next section.

3.2.1 Basic task's state transition diagram

Task state transition diagram (Fig 8) shows the possible states for a task and the valid state transitions for a task among these available states in the model.

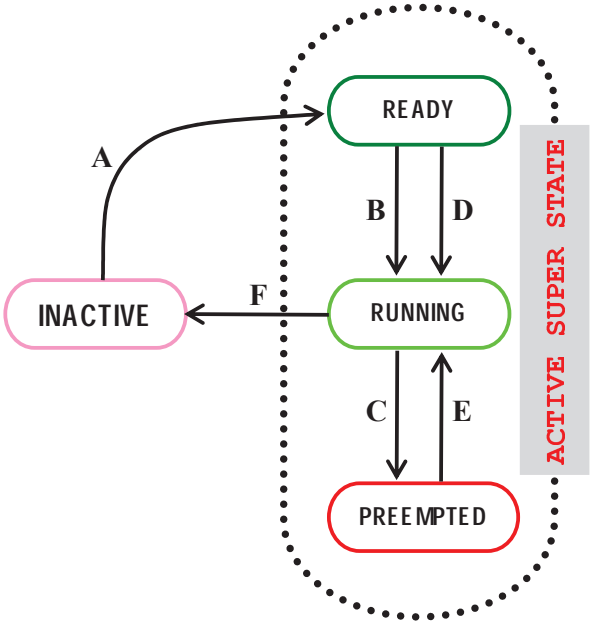


Fig. 8 Task state transition diagram in the kernel

The transitions are described in the following table:

TABLE I: Valid transitions

Transition	*Former state*	*New state*	*Description*
A: release	*INACTIVE*	*READY*	A new task is set into the ready state by a call to primitive **release (*tasks*)**. The task(s) is/are now visible to the task scheduler, making task to transit from **inactive** to **ready** state**.**
D: schedule	*READY*	*RUNNING*	When there is no running task in the system scheduler is invoked by a call to ***schedule( ),*** which determines the highest priority ready task, thus this task transits **ready** to **running** state.
C: preempt	*RUNNING*	*PREEMPTED*	While one task is running and a higher priority task is released following **A**, current task is preempted and is moved from **running** to **preempted** state. Actually scheduler is invoked every time after each **transition A** by a call to preempt( ) from ***release( )***.
B: preempt	*READY*	*RUNNING*	After transition **C,** the new ready task is put into execution. The task transits from **ready** to **running** state. Always following transition **C,** transition **B** is done**. But it should be noted that two different tasks are involved in these two different transitions.**
E: preempt	*PREEMPTED*	*RUNNING*	When the highest priority task exits running, and if the preempted task is the higher priority task, its context is restored and starts its execution from where it stopped before being preempted (transition **C**). Thus task transits from **preempted** to **running** state. This is carried out in continuation of **F.**
F: exit	*RUNNING*	*INACTIVE*	A task on completion of its execution leaves the processor, thus transiting from **running** to **inactive** state.

However every transition described in the above table is made possible available by the primitives in task manager subsystem. These primitives manipulate the data structures, which directly influence the states of the tasks.

3.2.2 Task management data structures

The state of each task is determined by the value of certain variables meant for task management. These are special data structures called Boolean vectors, which are string or an array of Boolean variables (whose value can be either 0-reset or 1-set). The only attribute of such a data structure is length or size, which is expressed in bits. Mentioning the position of a bit in the vector, it can be identified and manipulated discretely. The

advantage in using such Boolean vectors is, the logical operations on them are very fast and are accomplished in constant time, irrespective of the number of bits being set or reset. Consider an n-bit length Boolean vector, which has 'n' Boolean variables as illustrated in the Fig 9.

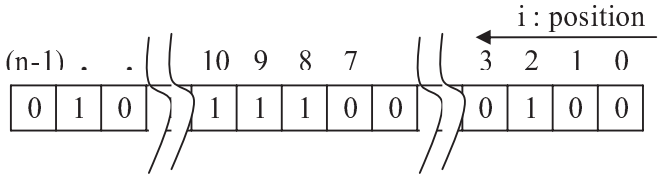


Figure.9 Boolean vector

Each bit can be uniquely accessible and manipulate-able by identifying the bit through its position. Task manager has two variables of this Boolean vector data-type, they are:

- Active Task Vector
- preemptionFlag
- Blocked Task Vector

3.2.2.1 Active Task Vector

As there are sixteen tasks under the kernel, the task manager employs a 16-bit Boolean vector called **Active Task Vector**. In this vector each of the bit is uniquely mapped to one task and the bit-position itself indicates that task's priority. This means if bit position is 'i', it is mapped to **task_i** with priority also equal to '**i**'. Further this position '**i**' is used as an index to task address table, which has the pointers to all the sixteen tasks under the kernel. This is illustrated in the following figure.

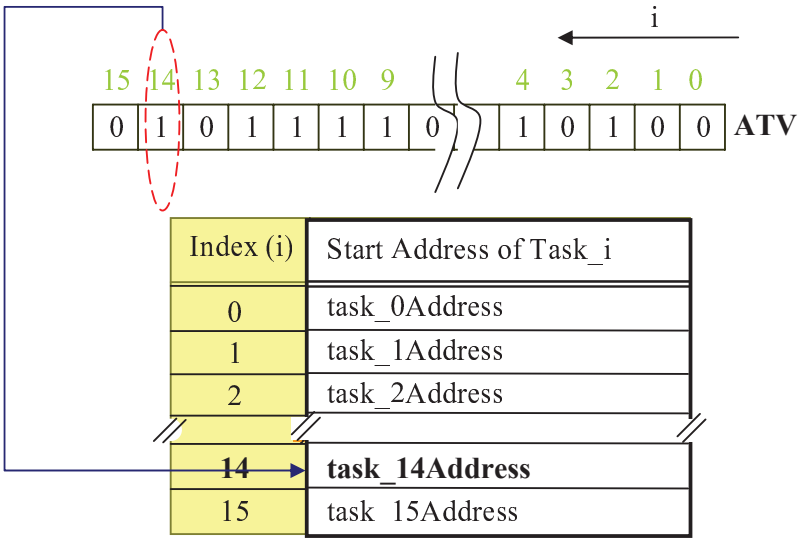


Figure.10 Illustration showing how ATV and start address table are related

3.2.2.2 Blocked Task Vector (BTV)

This Boolean vector is used in the context of resource management and is used to mask the tasks that are not eligible to be executed because of the shared resource (which is used during the task's execution) needed by the task is not available. This vector is employed following the stack-based ceiling priority protocol [6].

3.2.2.3 Task start address table (SAT)

In addition to this non-traditional data structure there is one more data structure, which is a table (array) of start addresses of all the application tasks. This is implemented as an array of pointers to the tasks that are implemented as subroutines with no parameters. The task start address table is shown in Fig 10.

3.2.2.4 Running Task (RT)

This is the variable that holds the identity of the current running task in the system. The task is identified by the task number which is also the tasks priority.

3.4 Basic task scheduling without resource management

Based on the value of bit corresponding to the task, and the value of the variable ***RT*** in the system, a task's state can be determined. The primary duty of task manager is to schedule the tasks following the scheduling policy and task model of the kernel. See the task state transition diagram, when the bit (in **ATV**) corresponding to a task is set (bit value is 1) the task is in ***active super state***, if it is clear (bit value is 0) the task is in ***inactive state.*** The task that is running is identified by the value of **RT.** Thus all the other tasks whose bits are set will be either in the ***ready state*** or ***preempted state***.

3.4.1 Task manager organization and startup

Task manager contains the most important **scheduler** responsible for scheduling the tasks. This scheduler consists of two system functions implementing the fixed priority scheduling policy. Task manager is made up of task management data structures, private functions and primitives. ***Primitives*** are the interface functions through which the user tasks interact with or invoke the kernel modules thus manipulating the kernel data structures. **Private functions** mean the system subroutines that can only be called from within the same module or other modules but not by the user tasks directly. Actually these private functions are invoked in the primitives.

The following functions and primitives along with task management data structures make the task manager:

- **Private functions**
    1. **schedule( )**
    2. **preempt ( )**

    Private functions operate on task management data structures ATV, BTV and RT. They just read but never modify ATV or BTV during their course, while RT is updated or modified.
- **Primitives**
    1. ***release(tasks)***
    2. ***taskExit( )***
    3. ***enablePreemption( )***
    4. ***disablePreemption( )***

The primitives ***enablePreemption( )*** and ***disablePreemption( )*** are subroutines that manipulate the preemptionFlag of the system.

The scheduler is called into action in two situations:

- ***SITUATION A:***

When **schedule( )** is invoked by the system. This is the case when there is no Running Task in the system. This happens when:

a. When the kernel is started for the first time i.e. in the startup when there is no running task.

OR

b. When a task ***exits*** and no running task is there i.e. when task leaves from the running state and enters the inactive state.

- ***SITUATION B:***

When **preempt( )** is invoked inside the call to ***release( )***. This is the case when a task is ***released***, thus When a task is ***released*** i.e., a task when leaves inactive state and enters ready state.

The operation cycles of scheduler in both situations are explained below in steps:

SITUATION A:

a. When the kernel is started for the first time and there is no running task.

Scheduling is initiated immediately after all the necessary hardware and kernel data structures are initialized. It is explained below how it is achieved:

1. When the system is started, **main( )** routine takes control in which all initialization (hardware and kernel) subroutines are invoked. The pseudo code of main-

```
main ()
{
initialize_hardware();
start_kernel();
}
```

2. The subroutine **initialize_hardware( )** does the hardware initialization necessary for the kernel and the application software. Its pseudo code:

```
initialize_hardware();
{
  /* initialize ports, interrupts etc */
  /* initialize timers */
}
```

3. The start_kernel subroutine initializes the kernel and starts the task manager (scheduler) invoking **schedule( )**. Its pseudo code:

```
start_kernel( )
{
  ….;  //initialize the needed kernel data structures etc
  while  ( TRUE)
    {
      while(ATV)      // if there is any active task i.e. ready task
      {
      schedule( );  // scheduler is always active in the back ground
      }
    }
}
```

Very important:

Notice the infinite loop in the **start_kernel( )** subroutine, where in the **schedule ( )** is invoked whenever there is nothing in the foreground. System function **schedule( )** is invoked only at this point by the system itself (i.e. **main** routine) and nowhere else.

SITUATION A: (continued)

b. When a task *exits* no running task is there i.e. immediately after every transition F.

1. A task about to exit calls ***taskExit( )*** making transition F.
2. Microcontroller becomes free and returns to the infinite loop of the system.
3. The **schedule( )** is invoked if there is any active task(ready task).

In both cases under **SITUATION A, schedule ( )** does:

1. Finds the set most significant bit 'HP' in ATV.
2. Starts this highest priority task **task_HP( )**.

SITUATION B:

When a task is released by calling primitive *release( )* i.e. task on leaving inactive state and entering ready state.

This release of task can be in the startup or when a task is running.

1. When a new task is released by calling primitive ***release(***tasks***)***, the bit corresponding to the task(s) released is set. The Boolean vector argument specifies the tasks to be released. Transition **A** is affected.

1) The primitive ***release( )*** invokes the scheduler by calling ***preempt( ),*** where the ***set*** most significant bit position in **ATV *'HP'*** (highest priority task) is found, but there can be two cases-
   - Case 1 'No running task in the system': Then the task *task_HP( )* is called and starts execution.
   - Case 2 'There is running task in the system': Then the value of RT is compared with the (***HP***) in the ***ATV.*** Accordingly-
     - If the **RT** is less than ***HP*****:**
       1. The current running task *task_RT( ) is preempted* following the transition **C.**
       2. The context of running task is pushed (saved) on stack.
       3. The task number of running task is pushed (saved) on stack.
       4. The new task *task_HP( )* is called and starts executing following transition **B**.
       5. After task task_HP ( ) finishes execution calling taskExit( )the recently preempted task is restored and starts execution

following the transition . If any preemption arises the step (2) is repeated.

- If the **RT** is not less than ***HP-*** the current task task_RT continues execution, thus no transitions occur.

2) A task after completion of its execution sequence calls a primitive taskExit( ) and exits the scheduler or system following transition **F**. This taskExit( ) clears the bit in ATV that corresponds to the task exiting.
3) When there is no running task scheduler is invoked by call to ***schedule( )***.
4) The scheduler finds the set most significant bit position '***HP'*** *in* ***ATV***, loads the task 'task_HP( )' and starts executing the task.

3.4.2 Task management private functions and primitives

• private functions

These private functions implement the defined fixed priority preemptive scheduling policy for the task scheduling. The task manager can only call these functions.

1. schedule( ) : On being invoked finds the most significant set bit in the ATV and starts the task corresponding to that bit. This is invoked by the system when there is no running task and there are active (ready) tasks.

```
schedule( )
{
    if (ATV)
    {
          RT = find_msb(ATV);
         Start_the_new_HP_task();// call the new task
     }
}
```

2. preempt( ) – When invoked finds the highest priority task, if there is any active task and if is greater than the running task priority running task is preempted and new task is started. This function does context switching associated with preemption and resumption. The pseudo code is given here.

```
preempt()
{
   if (preemptionFlag == TRUE)
   {
     HP = find_most_significant_bit(ATV);
```

```
        if (RT != NO_TASK)    // if there is any running task
          {
            if (HP > RT)
             {
              STORE_RUNNINGTASK_CONTEXT;
              STORE_RT;           // store running task number on stack
               RT = HP;
              Start_the_new_HP_task();// call the new task
              LOAD_RT;  // restore the task number that is preempted from stack
              RESTORE_PREEMPTED_TASK_CONTEXT;
             }
          }
        else
           {
            RT = HP;
            Start_the_new_HP_task();// call the new task
            }
    }
}
```

• primitives

These are invoked by the user tasks and get the services of task manager.

1. ***release(****Boolean vector* ***tasks)*** – This primitive is called whenever a task(s) has to be released. This primitive is invoked by passing Boolean vector argument that indicates the tasks to be released. It just sets the bits corresponding to the tasks released and calls preempt( ).

```
release(Boolean vector tasks)
{
ATV = ATV | tasks;
preempt( );
}
```

2. ***taskExit( )*** - This is called from the task that is exiting the scheduler. When a task has reached its finishing point, before leaving the running state it calls this primitive. It just clears the bit in ATV that corresponds to the exiting task.

```
taskExit()
{
   clear_the_bit_corresponding_to_exited_task();
   RT = NO_TASK;// there is no task running in the system
}
```

3. ***enablePreemption( ) and disablePreemption( )*** – The first primitive is called to change the mode of operation of the system from preemption disabled mode to preemption enabled mode. While second one is for disabling the preemption mode. The pseudocodes are

```
enablePreemption( )
{
preemptionFlag = TRUE;
}
```

```
disablePreemption( )
{
preemptionFlag = FALSE;
}
```

3.5 Integrated task and resource management – *SBPC Protocol*

In multi-tasking kernels to avoid inconsistency in data and resources that are shared, the shared resources need to be used in an atomic fashion. The general atomic access can be atomic access can be viewed in two different styles:

- **Use of object or resource reservation:** If one task reserves a resource then competing tasks when reach the point of using the locked resource are told: *"This resource is locked, go and do something else, or give up your time slot"*. The concept requires use of a scheduler in the process of resource management. This traditional kind of atomic operations are implemented by using concepts like mutex or semaphores to give a thread mutually exclusive access to a resource. Primary design issues are avoidance of deadlocks and perhaps minimizing the amount of time a resource is locked.

- **Use of execution locking:** If one thread needs exclusive access to a shared resource then it locks the resource and all or just competing threads, are prevented from even getting executing time for as long as the locking last. The concept used is: *"I do not want anybody to interrupt me for a while"*. The concept does not necessarily require use of a scheduler. Thread deadlocks cannot take place. A primary design issue is the amount of time a resource is locked.

So by considering the second approach we are sure that deadlock can never happen in the system. A protocol as per the second approach proposed by Liu [6] called ***stack-based ceiling priority protocol*** is followed to achieve integrated resource and task management in the kernel.

For this purpose one should know the resource management framework under the kernel, it goes as following:

- **Resource Control Block(RCB):** As the kernel is a static the shared resource are known in advance, so each of such shared resources are accessed through a

Resource Control Block (**RCB**) which have the ceiling priority for the resources which is equal to the highest priority task's priority that can access the resource.

Illustration- say resource1 can be shared by tasks task_1( ), task_4( ) and task_8 with priorities 1, 4 and 8. Then the RCB defined as-

RCB [*resource_identifier***] = {ceiling priority}** for this resource1 looks like:

RCB [*resource1***] = { 8 },** since among all the tasks that have access to resource1 the highest priority task has a priority of 8.

- **System Ceiling:** Further there is one parameter called system ceiling that at any instant equals to the highest value of ceiling priorities of all the resources locked in the system at that instant.

3.5.1 Stack-Based Priority Ceiling (SBPC) protocol

There is a variable **∏(t),** which represents the system-ceiling at time 't'. This variable is initialized to a value **Ω,** which is less than the least priority of all the tasks in the kernel. Defining Rules of the Protocol [6]:

0. Updating of the System Ceiling: When all resources are free the system ceiling **∏(t)** will be equal to **Ω**. This **∏(t)** is updated each time a resource is locked or unlocked.

1. Scheduling Rule: After a task is released it is stopped from execution until its assigned priority is higher than the current system-ceiling **∏(t)** of the system. At all times tasks that are executable are scheduled in a priority-driven preemptive manner according to their assigned (base) priorities.

2. Allocation Rule: Whenever a task requests a resource, it is allocated the resource.

It is clear from the scheduling rule that a task is schedule only when all the resources it needs during its execution are free, because this happens only when the base priority of task is greater than **∏(t)**. Following this no dead lock is possible. *Base priority* is the task priority that is allocated to it statically; it can be used as a synonym to *task priority*.

3.5.2 Integrated task and resource management – implementation

Data structures:

For this resource management following SBPC protocol four data items are introduced, they are:

- Resource Control Block - **RCB**
- System-ceiling - **∏(t)**
- System-ceiling stack - **∏-stack**
- Blocked Tasks Vector - **BTV**

System-ceiling stack (∏-stack): Following the **rule 0** of the protocol the value of system-ceiling is updated whenever a resource is occupied or freed. But to retain the history we need a data entity, this **∏-stack** is exactly for that purpose.

Blocked Tasks Vector (BTV): To accommodate the rule 1 of the protocol the tasks that are not eligible to be executed owing to resource unavailability need to be marked. This masking of non-eligible tasks is done by **BTV,** which is also a 16-bit Boolean vector. Again the bit position corresponds to task number. So by setting a bit in the BTV, that particular task is blocked meaning no more available fro scheduling. The use is further illustrated under the primitives' section **3.5.3**.

3.5.3 Private functions and primitives

Resource manager realizes the protocol's ***rule 0*** through its primitives – ***lock( )*** and ***unlock( )***. While the ***rule 1*** is achieved in task management via primitive- ***preempt( ).*** And **rule 2** is obvious because of ***rule 0*** and ***rule 1***. All the other primitives remain unchanged as stated under **section 3.4.2.** All the primitives' pseudo codes are given below.

The resource manager module is an individual entity with the said data structures and primitives (see section 3.5.3.1) , but its operation through its primitives implementing the SBPC protocol results in integrated resource and task management.

3.5.3.1 Resource management primitives

Resource manager provide its service to user tasks with two primitives- lock(resource) and unlock(resource). ***Lock( )*** primitive is used by task to lock the resource on which it is called. While ***unlock( )*** frees the resource. The region in the task execution which uses a shared resource is called critical region. At the beginning of the region lock( ) is used while at the termination unlock( ) is used. When a task uses more that one shared resource during its execution, special care must be taken while locking multiple resources. The locking scenarios should confirm to the following sequence of locking and unlocking:

1. **Resources are locked sequentially:** In this case the resources are locked and unlocked in series, i.e. every lock of certain resource is followed by unlock of the same resource. An example sequence of this type-
   …lock(R1)…..unlock(R1) …lock(R2)…..unlock(R2) …lock(R3)…..unlock(R3)

2. **Nested locking of resources:** In some cases a task needs to use multiple shared resources simultaneously. The syntactical rule that has to be followed is shown by the following example.
   …lock(R1) …lock(R2)…lock(R3)…unlock(R3) …unlock(R2) …unlock(R1)
   This rule exactly resembles the rules for bracket opening and closing around mathematical terms in an expression. Like **(R1(R2(R3) R2)R1)**, opening brace is analogous to locking and closing is analogous to unlocking. The rule of thumb in this case is the sequence of locking should always be followed by the opposite sequence of unlocking.

3.5.3.1.1 lock(resource_name)

The pseudo code for the primitive is-

```
lock(Resource resource)
{
 if_ceiling_priority_of_resource > = system_ceiling)
   {
    push_On_Pi_Stack(current_system_ceiling);
    PI = RCB[resource].ceiling;// update system ceiling
    BTV = (TVEC)~(BTV | (TVEC)PI_Table[PI+1]);
                        //clear the bits corresponding to non-
                        //schedulable tasks under BTV
   }
}
```

The execution of lock( ) proceeds as following:

1. If the priority of the resource being locked is greater than or equal to the current system ceiling priority goto2 else exit
2. The current system ceiling is pushed on PI-stack.
3. The system ceiling is updated with the ceiling priority of the resource locked.
4. All the bits corresponding to tasks whose base priority is not greater than the updated system ceiling priority are reset, while other bits are set.

The main job done here is updating BTV to mask the non-executable tasks.

3.5.3.1.2 unlock(Resource resource_name)

The pseudocode for primitive:

```
unlock(Resource resource)
{
  if (system_ceiling == resource_ceiling)
    {
      pop_the_PiStack( &PI);

      BTV = (TVEC)~(BTV & (TVEC)PI_Table[PI+1]);
         //unmask tasks based on past&current system ceilings

      preempt();
    }

}
```

The execution of unlock proceeds as per following sequence:

1. If the system ceiling equals to the ceiling of resource being unlocked goto2 else exit
2. Get the next highest system ceiling priority by popping the PI_stack and assign this value to the current system ceiling.
3. Unmask all the tasks whose priorities are greater than current system ceiling.
4. Invoke the scheduler by calling preempt( ).

3.5.3.2 Augmented task management functions and primitives

The private function preempt( ) implementation results in the needed integration between task manager and resource manager resulting in integrated task and resource management. All the other functions and primitives remain same.

3.5.3.2.1 preempt( ) private function

After every ***release(tasks)*** or **unlock (resource)** **`preempt()`** is invoked. The sequence of execution of this function is as following:

1. Evaluates the most significant bit '**HP**' in (ATV & BTV).
2. If there is any task running then goto3 else goto11.
3. If **HP** is greater than **RT** i.e., if the running task is not the highest priority task among the schedulable tasks then goto4 else do nothing.
4. Store the context of the running task by pushing all the registers on stack.
5. Store the running task number **RT** by pushing it on stack**.**
6. Start the **HP** task by the index **HP** to Start Address Table (SAT) of tasks.
7. After the task **task_HP** finishes execution the control returns to preempt then stack is popped restoring preempted task number into **RT**. So **RT** now holds the task number.
8. Again the stack is popped thus restoring the context of task **task_RT**
9. Now the control returns to the point in preempted task where it left last time.
10. The task_RT is executed return.
11. Start the new task task_HP( ) and let it run.

The pseudo code is-

```
preempt()
{
  if(preemptionFlag = = TRUE )
  {
  HP = find_most_significant_bit(ATV & BTV);

  If(RT != NO_TASK)    //if there is any running task
    {
      if (HP > RT)
        {
          STORE_RUNNINGTASK_CONTEXT;  // store the context on stack

          STORE_RT;        // store running task number on stack

          RT = HP;

          let_the_task_RT_to_execute;// call the new task

          LOAD_RT;         // restore  the task number from stack

          RESTORE_PREEMPTED_TASK_CONTEXT;  // restore the context from stack
        }
    }
  else
    {
    RT = HP;
      let_the_task_RT_to_execute;      // just let it execute
    }

}
```

The stack management becomes very simple because of this ***preempt( )*** function. Observe the following situation of the system, where task_1 is running, stack is empty:

1. An interrupt comes and task_2 is released and eventually preempt( ) is called.
2. Task_1 is preempted and stack stores first the context and then the task number.
3. Task_2 starts running
4. One interrupt occurs releasing task_4 and preempt is called from task_2, context and task number are pushed on stack.
5. Task_4 starts running and finishes execution, now the control returns to preempt( ) call in task_2, task_2 number and context are restored was.
6. Task_2 runs and on finishing execution returns to preempt( ) in task_1.
7. Task_1 number and context are restored and task_1 continues execution.

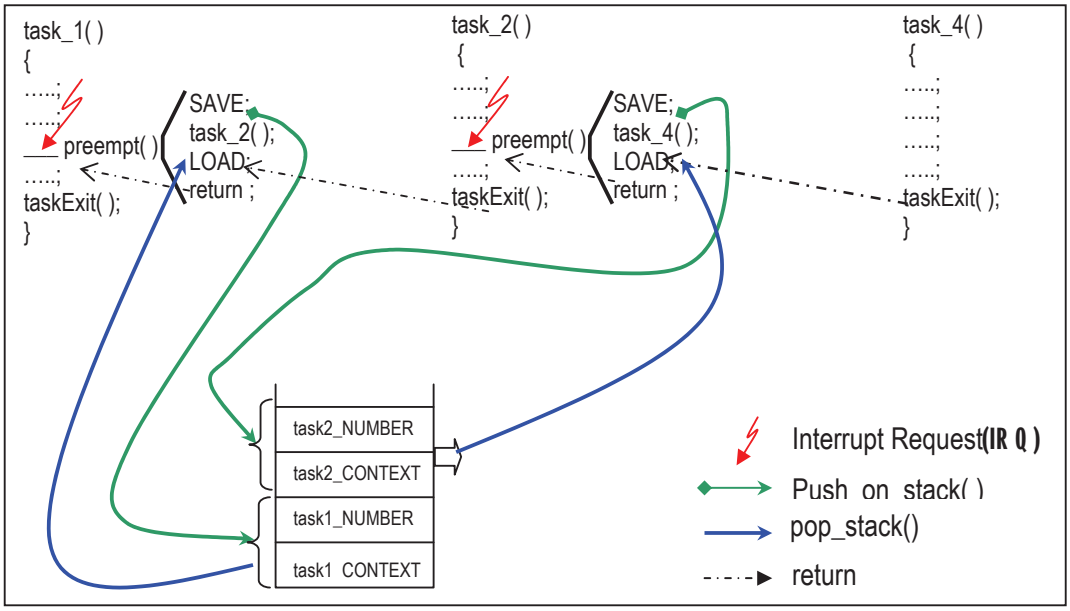


Figure.11 Illustration of Preempt( ) and associated context switching etc.

An important thing is to be noticed here, the preempt( ) indicates*(see step 8 in the execution of preempt)* which preempted task to be run, after finishing the higher priority task it loads that preempted task's context . Now it starts executing the preempted task. As the context is already restored the task starts from the point where it was preempted during preemption.

3.5.3.2.2 schedule( ) private function

This is invoked whenever there is no task to run. So the pseudo code for schedule is very simple its execution is as following:

1. If there is any active task it finds the Highest Priority task else exits.
2. Then just lets the task to execute.

See the pseudo-code for ***schedule( )***.

```
schedule()
{
  if_there_is_any_active_task
  {
     RT = find_most_significant_bit(ATV);
     let_the_task_RT_to_execute;     // just let it execute
  }
}
```

3.5.3.2.3 release(Boolean vector tasks) primitive

Integrated Event Manager or tasks call this primitive, by passing a vector argument. This vector is **OR**-ed with **ATV.**

```
release(Boolean vector tasks)
{
  ATV = ATV | tasks;
  preempt( );
}
```

3.5.3.2.4 taskExit( ) primitive

When a task has finished its execution sequence, before leaving the running state it calls this primitive. This call just clears the bit in ATV that corresponds to the task that is exiting.

```
taskExit()
{
   clear_the_bit_corresponding_to_exited_task();
   RT = NO_TASK; // there is no task running in the system
}
```

3.5.3.2.5 enablePreemption( )

The first primitive is called to change the mode of operation of the system from preemption disabled mode to preemption enabled mode.

3.5.3.2.4 disablePreemption( ) – This one is for disabling the preemption mode.

The pseudocodes are:

```
enablePreemption( )
{
preemptionFlag = TRUE;
}
```

```
disablePreemption( )
{
preemptionFlag = FALSE;
}
```

3.6 Case Study

Illustration of how integrated task and resource management is achieved. Consider a case when all the shared resources are free and task task_2 is running. The figure describes the execution pattern of tasks in the sequence of their occurrence as indicated by the step number.

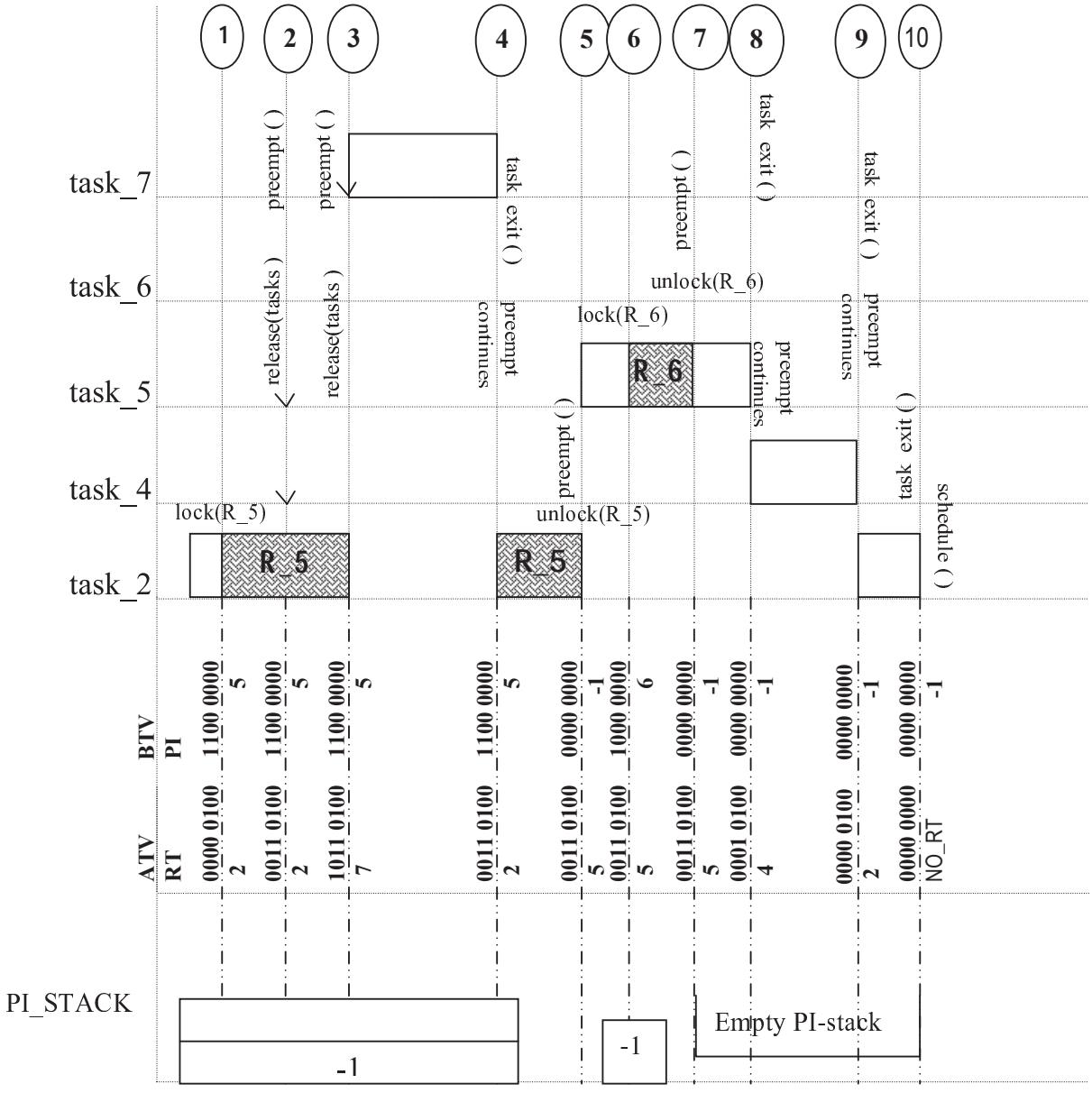


Figure. 12 Illustration of integrated task and resource management.

STEP 0: In the beginning **initial values** of all relevant variables:

ATV = 0000 0100 RT = 2 PI = -1 (Ω) //as said in SBPC Protocol

BTV = 0000 0000

STEP 1: Task_2 locks Resource R_5(Ceiling Priority is 5). Following the call of lock(R_5) , the present PI (-1) is pushed on stack, the updated values are:

BTV = 1100 0000 PI = 5

STEP 2:

a) New tasks task_4 and task_5 are released, by release (0001 1000). ATV and BTV are updated accordingly.

ATV= 0011 0100 BTV= 0011 0000

b) preempt( ) is invoked inside the release (tasks_vector) and accordingly nothing is changed. **RT = 2 PI = 5**

STEP 3:

a) Task_7 is released by calling release(1000 0000). ATV is updated accordingly.

ATV=1011 0100

b) preempt( ) is invoked inside the release (tasks_vector) and accordingly task_2 is preempted its context is stored. New task_7 is executed. Thus the values of variables are changed. **RT = 7 PI = 5**

STEP 4:

a) Task_7 completes its execution and calls task_exit( ). ATV and RT are updated.

ATV=0011 0100 RT=NO_TASK

b) Preempted task task_2 is resumed by restoring its context.

RT = 2 PI = 5

STEP 5:

a) Task_2 unlocks the resource R_5. The values of PI, BTV are updated accordingly. Also the stack is cleared.

BTV=0000 0000 PI = -1

b) preempt( ) is invoked and accordingly the tasks are scheduled. The task_2 is preempted, its context is stored, and the new task_5 starts execution.

RT = 5

STEP 6: Task_5 locks Resource R_6, whose Ceiling Priority is 6. Following the call to lock(R_6), the present PI (-1) is pushed on stack, the value of PI is updated accordingly. **PI = 6**

STEP 7:

a)Task_5 unlocks the resource R_6 by calling unlock(R_6) primitive. The values of PI, BTV are updated accordingly. Also the stack is updated or cleared.

BTV=0000 0000 **PI = -1**

b) preempt( ) is invoked and accordingly nothing happens.

RT = 5

STEP 8:

a)Task_5 completes its execution and calls taskExit( ). ATV and RT are updated accordingly.

ATV=0001 0100 **RT= NO_TASK**

BTV = 0000 0000

b) Cotrol returns to the previous preempt call and as there is no running **preempt**( ) finds the HP = 4 and starts the new task_4.

RT = 4

STEP 9:

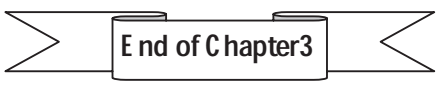
a)Task_4 completes its execution and calls task_exit( ). ATV is updated accordingly.

ATV=0000 0100

b) Task_2 is restored and continues. **RT = 2**

STEP 10: Task_2 completes its execution and calls task_exit( ). ATV is updated accordingly.

ATV=0000 0000



Chapter 4. Task Interaction

The requirements for the kernel to be simple (small memory foot print), fast and powerful, are met by the novel techniques used to achieve task interaction. The tasks are basic tasks which cannot be blocked, so the techniques should achieve non-blocking type of interaction among the tasks. Tasks interact in two ways:

- Synchronizing each other
- Communicating with messages.

4.1 Task synchronization

Synchronization in existing kernels

Following the types of tasks under the kernel, several synchronization paradigms are implemented in different kernels. Synchronization (defined under next section) is basically of two types:

- **Synchronous:** The sender invokes the request (signals) and blocks waiting for the response. The task is in busy waiting or is blocked.
- **Asynchronous:** The sender issues the request (signal) and continues processing; there is no waiting for response.

In terms of complexity synchronous type is more difficult than the asynchronous type. Synchronous techniques are not recommended for embedded real-time systems because of increased complexity, limited functionality and unpredictable behaviour. Under the HARTEX$_{\mu}$ kernel developed all the tasks are basic tasks that can never be blocked, so it employs asynchronous producer consumer interaction which is of non-blocking type.

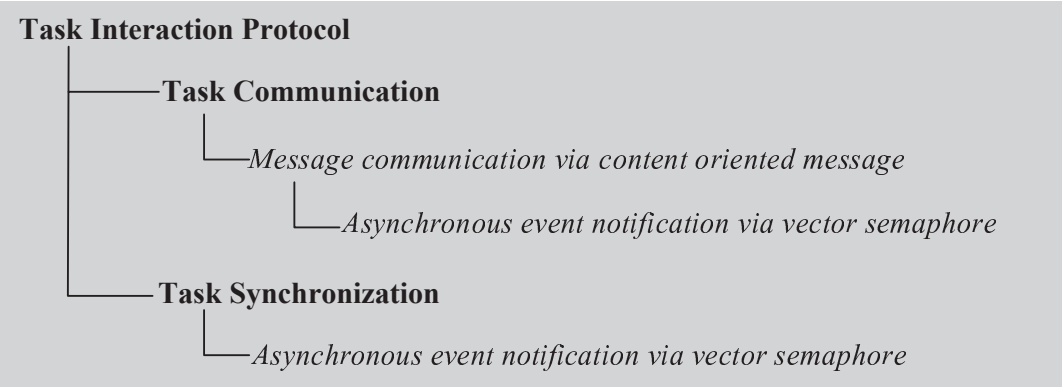


Figure.13 HARTEX$_{\mu}$ task interaction protocol

The tasks in HARTEX$_{\mu}$ kernel interact following the scaled down HARTEX communication protocol [9]. The original HARTEX protocol is meant for both basic and extended tasks and achieves distributed interaction. This protocol is scaled down to basic tasks only and for interaction in single node. The organization of scaled down version of HARTEX communication and synchronization protocol is as shown in Fig 13.

What is synchronization between tasks?

Some task(s) need to start or continue its execution beyond a certain point in its execution path, depending on whether the other task has completed some execution. This is called *task synchronization*. Such points in each task that are significant in task synchronization are called *synchronization points*. The synchronization and synchronization points in two tasks (interacting by synchronizing) are illustrated with the following figure (Fig 14).

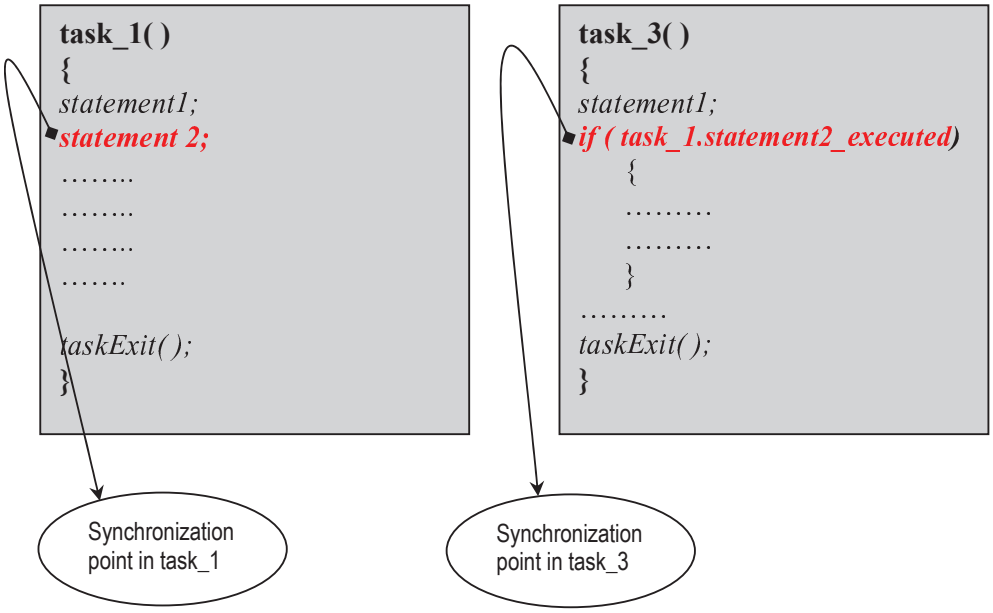


Figure.14 Task synchronization basics

The reason for such synchronization among the tasks is very much obvious in several real-time applications. Consider an application where in one task (task_1) is doing the data acquisition from the sensor and carrying out some preprocessing, while the other task (task_3) is doing some control using this condition variable reported by *task_1*. Say if *task_1* hasn't finished its statement_2 and *task_3* is released, there is no need for task_3 to proceed for computing control action. But if *task_1* has produced the plant condition variable by executing *statement_2,* task_3 will continue its execution.

4.1.1 Event notification through Boolean vector semaphores – primitives

In the kernel, synchronization among the tasks is done by means of event notification. In event notification a synchronizer task raises an event on reaching its synchronization point, while the synchronizing task on reaching its synchronizing point just check for this event, if occurred it will continue execution else it just exits.

Event notification is realized by means of semaphores called Boolean vector semaphores. Semaphore is an object or data entity that each task can check and then change the value. Depending on the value found the task takes a different course of execution accordingly. Boolean vector semaphore is a special semaphore, which consists of Boolean vector of 16-bit length, each bit corresponding to one task under the kernel. So each such semaphore is mapped to one unique event, which makes it possible to implement event notification in a predictable manner.

4.1.1.1 Semaphore Control Block (SCB)

The semaphore is implemented as a Semaphore Control Block **(SCB)**. Therefore for each event there is one semaphore on one-to-one unique mapping, and each semaphore is having a **SCB**. The **SCB** is a data structure with two fields as following:

```
Semaphore_Control_Block
{
Boolean_vector flags;
Boolean_vector tasks;
}
```

Illustration of SCB:

SCB	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
released **tasks**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**

In the above figure of the SCB (initialized so) notice the values of **flags** and **tasks** vectors. The meaning of **released tasks** is explained later. If task(s) has to be signaled about an event that is mapped to this semaphore or SCB, the **flags** vector has to be modified such that all the bits corresponding to the tasks signaled are set. Say for example tasks task_15, task_14, and task_1 are signaled. Then the **flags** field in SCB is as following:

SCB	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
released **tasks**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

The synchronizing tasks on reaching the synchronizing points just check the bit under flags field corresponding to their task number if set they continue beyond the synchronization point.

The second field ***released tasks*** indicate that tasks (for example in this SCB: tasks 14,10 and 3) are to be released on occurrence of that event. It is for the obvious reasoning associated with non-blocking event notification, the released tasks (TASKS) is the subset of tasks that are signaled.

4.1.1.2 Organization of SCB into SCB_Table

There are sixteen (16) semaphores for the purpose of task synchronization. All these SCBs are put in a table (implemented as array), where each semaphore is accessed by the index into that table (array). Each indexed row corresponds to one SCB. The table is as shown in Fig 14. The first column in the table is not an actual column in the table; it is just shown here to indicate the semaphore or SCB ID. If we need to access semaphore_i or SCB_i, we just say SCB[i].

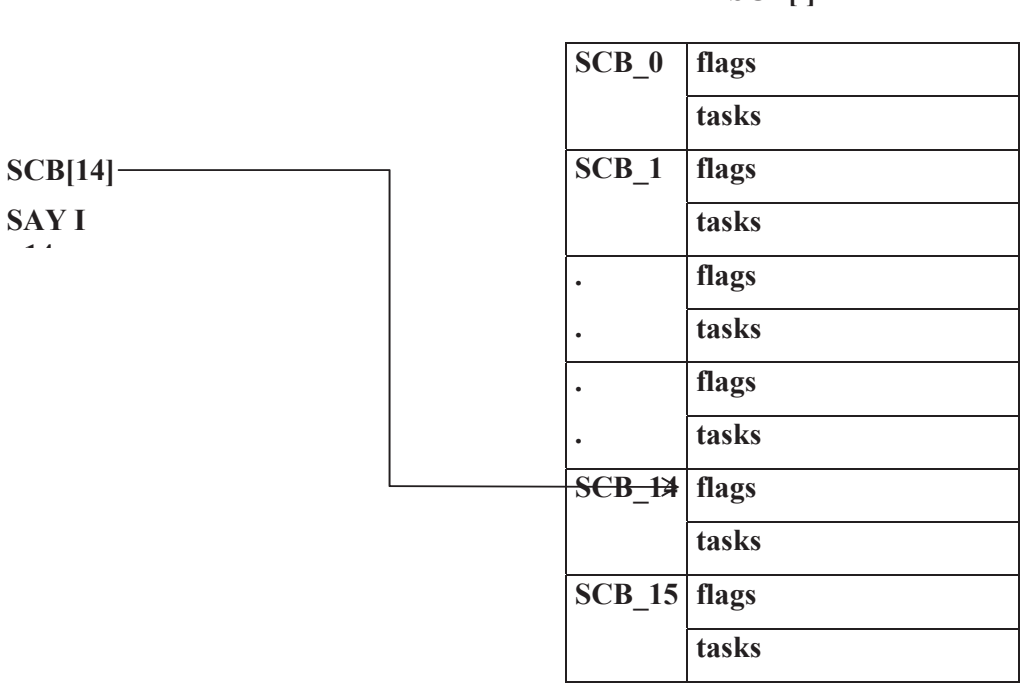


Figure 15. SCB_Table structure and accessing the table

4.1.2 Synchronization primitives

The semaphores i.e., SCBs which are the kernel data structures are accessible by all the tasks but only through the appropriate interface provided in synchronization module under ***software bus***. The primitives for synchronization through event notification are:

1. signal_and_release (semaphore, tasks)
2. test_and_reset (semaphore)

It is by calling these primitives on reaching the synchronization points the user tasks will use SCBs in SCB table for event notification.

4.1.2.1 signal_and_ release (semaphore, tasks) primitive

On calling this primitive on a particular `semaphore` and specifying synchronizing `tasks` (a Boolean vector), the tasks are notified about the event with which the semaphore is mapped. The pseudo code for such primitive is given below.

```
signal_and_release (char semaphore, BooleanVector tasks)
{
  SCB[semaphore].FLAGS = SCB[ semaphore ].FLAGS  |  tasks ;
  ATV  =  ATV |  SCB[ semaphore ].TASKS ;
}
```

The arguments passed to this primitive are semaphore and tasks. Semaphore is of numeric type that is used as index to the SCB table, while argument 'tasks' is a Boolean vector. The `semaphore` argument identifies the SCB, while the `tasks`(Boolean vector) mentions the tasks that are to be signaled of this event. This is done just by *OR-ing* the `tasks` with ***flags*** field of the corresponding SCB. The other job done by this primitive call is releasing of the tasks mentioned by ***tasks*** field under that SCB. This primitive is called on reaching synchronization point at the sending end i.e., synchronizer task, from where we need to notify/signal an event.

4.1.2.2 test_and_reset (semaphore) primitive

This primitive is called by the synchronizing task(s) on reaching the synchronizing point in their execution sequence. The synchronizing task when running ***(running state)*** on reaching the synchronizing point in their execution sequence, checks whether the

semaphore (to which event is mapped) has signaled it or not, if signaled it starts it continues executing else the task just exits. The pseudo code for primitive is given below.

```
char test_and_reset (char semaphore)
{
 if   ( SCB[semaphore].FLAGS[RT] = = 1 )
                          // check whether the semaphore has signaled the  running task or not
   {
   SCB[semaphore].FLAGS[RT] = 0 ;  // clear the bit in FLAGS vector
   return(true) ;
   }
 else
   {
   return(false) ;
   }
}
```

4.1.3 Illustration of usage of primitives

As said previously, the **signal_and_release( )** is called within the synchronizer task raising the event, and the event is mentioned by the semaphore. While the **test_and_reset ( )** is called on a semaphore by the synchronizing tasks interested in that event. Thus whenever an event occurs corresponding semaphore is signaled and receiving task checks the corresponding semaphore. Consider a synchronization example between two tasks- task_1, task_2 and task_5: task_2 is the synchronizer task signaling an event1 mapped with semaphore1 on which the synchronizing tasks task_1 and task_5 are to be synchronized, while its needed to release task_1.

The sequence goes like following:

1. Initially the semaphore1 i.e., SCB[semaphore1] looks like this:

SCB[1]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
released **tasks**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

2. Task_2 on reaching the synchronization point signals the event1 by calling **signal_and_release (*semaphore1, tasks*).** The argument ***tasks*** holds the value 0000 0000 0010 0010 (0x0022), that is the bits corresponding to task_1 and task_5 are set in the ***tasks*** argument.
3. The call to signal_and_release( ):
   i. updates ***flags*** field under the SCB[*semaphore1*] whose value become:

SCB[1]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
tasks	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

   ii. Task task_1 is released thus the bit ATV [1] under ATV is set.

4. Tasks task_5 or task_1 when running, on reaching its synchronization point checks the semaphore1 by calling **test_and_reset (*semaphore1*).** This call:
   i. Checks **flags[5]** / **flags[1]** i.e., fifth or first bit under ***flags*.**
   ii. If that bit -
      a. is set:
         1. clears the bit i.e., **flags[5]** / **flags[1]** is reset.

This bit is cleared by task_1

SCB[1]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tasks	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

         2. returns **TRUE**.
      b. if not set – returns **FALSE.**
5. If the call returns true it continues to execute beyond the synch point, else it just exits.

SYNCHRONIZER TASK

```
task_2
{
.....;
.....:
signal_and_release(semaphore1,0x0022);
....;
.....:
taskExit( );
}
```

SYNCHRONIZING TASK

```
task_5
{
.....;
.....:
if (test_and_reset(semaphore1))
  { ....,
    .....:
  }
taskExit( );
}
```

4.1.4 Different patterns of task synchronizations

The tasks synchronization (non-blocking style) can be between two or more, depending on how many synchronized tasks and synchronizing tasks are involved there are different event notification styles, the ones supported by present kernel are:

- One-to-one event notification
- One-to-many event notification
- Many-to-one event notification
- Many-to-many event notification
- Broadcast event notification

All the styles of synchronization or event notification are achieved by the value of ***tasks*** argument passed to the signal_and_release (*semaphore*, *tasks*) primitive call.

4.1.4.1 One-to-one event notification

In this style one task signals an event (one) to one task. This is achieved by invoking the signal_and_release (*semaphore, tasks*) primitive with ***tasks*** argument having a value with only one set bit.

4.1.4.2 One-to-many event notification

When the argument ***tasks*** in the signal_and_release invocation has two or more set bits, the event notification achieved is of one-to-many style. Thus by this type of event notification one task (synchronizer) can synchronize two or more tasks (synchronizing tasks). This is as illustrated under 4.1.3.

4.1.4.3 Many-to-one event notification

In this case many tasks (two or more) notify individually one separate event to one single task via the primitive **signal_and_release (semaphore, tasks)**, but this primitive is called individually in each of the two or more separate synchronizer (notifying) tasks at their synchronizing points. Further each such call to the primitive is invoked on different semaphores (different events), while the values of 'tasks' vector in all these calls have one common bit position set that corresponds to the notified or signaled task(synchronizing task). This results in many-to-one event notification.

The pseudo code of notified task looks like following.

```
task_N
{
  if ( test_and_reset ( 0 )  )
           {
             //do something0
           }
  if ( test_and_reset ( 1 )  )
            {
              //do something1
            }
  …
  …
  if ( test_and_reset ( n )  )
           {
           …..;  //do somethingN
           }
  task_exit( );
}
```

This is one kind of notified task pseudocode, but it can also be like this-

```
task_N
{
  if (test_and_reset(0) & test_and_reset(1) & test_and_reset(2)  )
           {
             //do something
           }

  task_exit( );
}
```

The synchronizer task remains the same as illustrated in section 4.1.3.

4.1.4.4 Many-to-many event notification

Several one-to-many and/or many-to-one and/or one-to-one event notifications when brought together will result in many-to-many style of event notification. This is mentioned for the same of completeness.

4.1.4.5 Broadcast

This style of event notification is a special case of one-to-many type event notification; where one task notifies about an event to all the tasks under the kernel. When the synchronizer task wants to broadcast about an event it just invokes the signal_and_release(semaphore, tasks) with ***tasks*** argument having all the bits set. Thus the broadcast is implemented in the kernel.

4.2 Task communication via content-oriented message addressing

The kernels provide means for tasks to communicate among themselves to achieve the needed system functionality. This kernel under discussion is meant only for real-time systems with single microcontroller; hence all the communication is local between task in one chip. Further as said earlier the communication is also of non-blocking type.

4.2.1 Content-oriented message addressing

An advanced communication technique called **Content Oriented Messaging** is employed. As the name implies, the particulars of senders, receivers and corresponding message buffers are not specified explicitly but are implied by the name of the variable or message being sent or received. The task sending a message is referred to as *sender task,* while the task receiving the message as *receiver task*.

The communication is implemented on top of the non-blocking synchronization concept explained under previous section, where the message arrival or readiness is notified by means of an event (which is mapped to a Boolean semaphore). These semaphores are called Messaging Semaphores, which have the semaphore control blocks with same structure as of synchronization semaphores. So the sender task on reaching its communication point sends message, actually speaking it doesn't send the message but it just notifies the receivers that message is ready by signaling them (and eventually releasing all or some of them). While the receiver tasks on reaching their communication points just check whether the message is ready, if ready they just copy the message from source to their local buffers and continue their execution sequence.

Thus the communication is happening in two steps:

1. **Sender task** signals the message readiness or availability to the receiver tasks.
2. **Receiver tasks** on reaching their communication points copy the message to their local buffers.

Following this every task will have one message buffer whose size will be equal to the size of the largest message which that task will receive. Similarly every message will be having a source buffer.

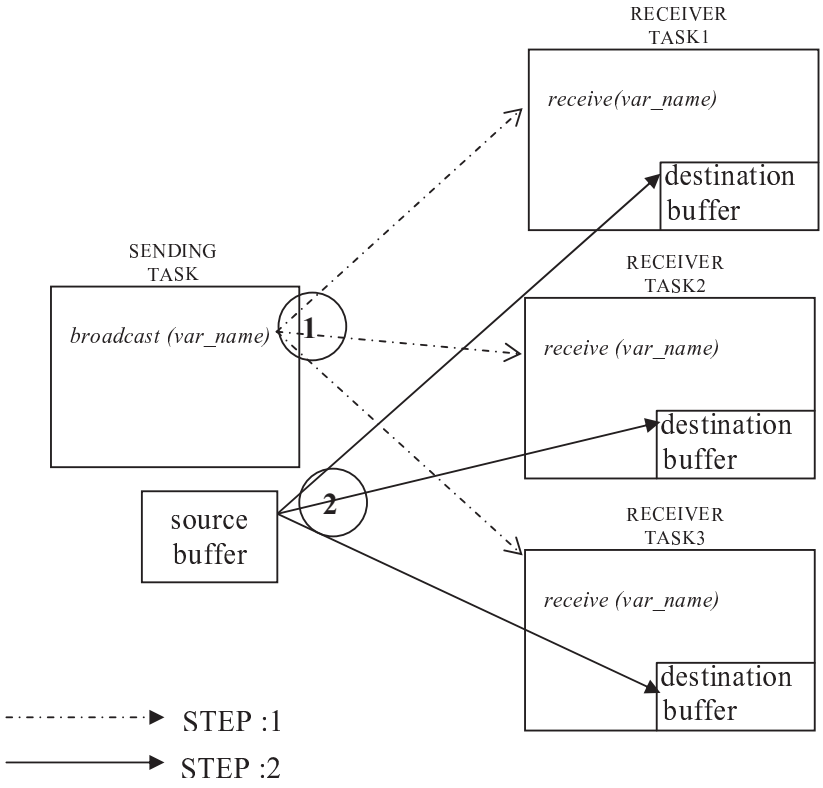


Figure.16 Task Communication in steps

4.2.1.1 Task Control Block (TCB) and TCB_Table

Therefore every user task under the Kernel has one **Task Control Block (TCB),** which has a field that contains the address of the buffer for that task. Its structure is like this:

```
TCB[task_number_X] = { &messageDestinationBufer_X }
```

There is a table of such TCBs implemented as an array of type TCB, and the task number itself is used to access this TCB for that task. It is named as control block but practically it is no way concerned for control of the task to which it belongs.

4.2.1.2 Message Control Block (MCB) and MCB_Table

To achieve the content oriented messaging a data structure for messages called **Message Control Block (MCB)** is made. Each message has one **MCB**. It has three fields specifying: i. What are all the receiver tasks.

ii. Address of the message source, and iii. Length of the message.

```
MCB[message_ID] =   { Boolean vector receivers_for_this_message;
                      Address        &message_source_buffer;
                      Char           message_length;
                    }
```

All such MCBs for all the messages in the kernel are put together as an MCB_Table, which is implemented as an array of MCB type.

4.2.1.3 Mapping the messages to the indexes

To access the MCB for a message named *variable_X* we just say MCB[*variable_X*]. This is made possible by mapping all the message names or identifiers to unique numbers. After such mapping, wherever *variable_X* appears it is replaced with the number to which it is mapped. Illustration of how this mapping helps in accessing is shown below.

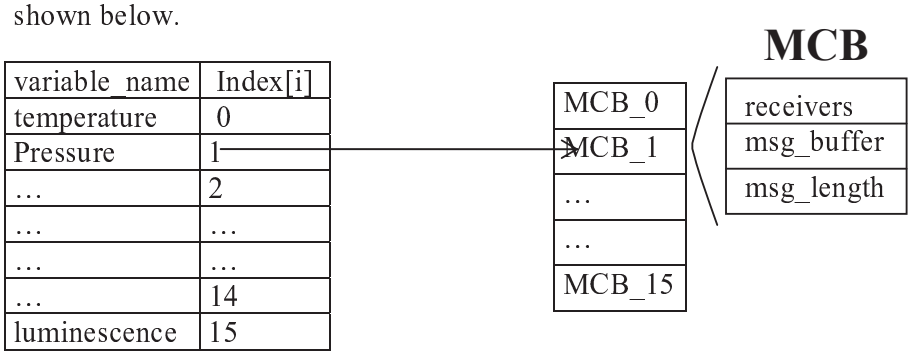


Figure.17 Accessing the MCB with variable names.

After such mapping (done by preprocessor macro definitions in 'C' implementation), wherever we say like send (pressure), it is equal to send(1) .

4.2.1.4 Messaging semaphores and MsgSCBs

Each and every message is mapped to one event which is implemented as a *message semaphore* or *Messaging Semaphore Control Block* **(MsgSCB).** All these MsgSCBs are put under a table **MsgSCB_Table** implemented as an array of MsgSCB type. Again the index equivalent of variable is used to access these semaphores. Thus there is a one-to-one correspondence between the MCB and MsgSCB for the same index.

All these data structures are kernel data structures under communication and tasks modules, and can be accessed and through primitives provided to achieve the communication.

4.2.2 Communication primitives and private functions

There are five primitives under communication part of software bus:

- **broadcast(*variable_name*)**
- **receive(*variable_name*)**
- **get_ message_source_buffer (*variable_name*)**
- **get_message_destination_buffer ( )**
- **get_ message_length (*variable_name*)**

Now a task sending a message, say it sends temperature will just say **broadcast (***temperature***)**. While at the receiver end it just calls **receive (***temperature***)**. This is as simple as said and the duty of kernel is finished as far as communication is concerned. This calling of primitives is illustrated below.

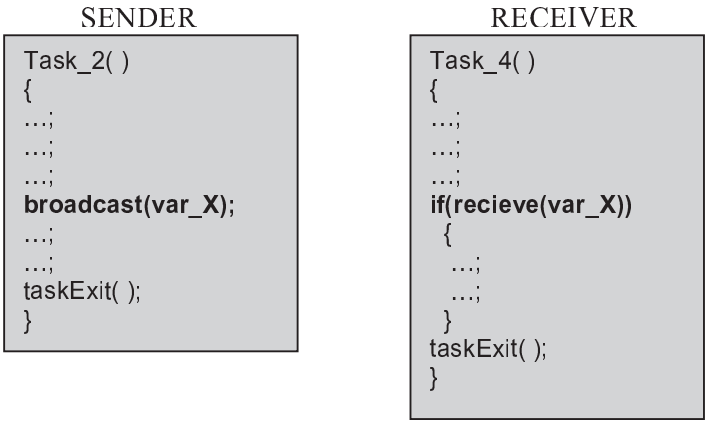


Figure.18 Illustrating primitive calls from tasks

4.2.2.1 broadcast (variable_name)

The pseudo code for the primitive is-

```
broadcast (var_name)
{
  tasks = MCB[var_name].receivers ;                    // get the receiver task IDs
  msg_semaphore_number = var_name ;                    // get the associated semaphore
  msg_signal_release(msg_semaphore_number,tasks) ;     // notify that message is
                                                       //ready or arrived
}
```

When the **sender** invokes this primitive on appropriate message, it does:

1. Fetch the receiver tasks identities from the MCB_Table into ***tasks*** in the form of Boolean vector.
2. Then corresponding semaphore is also fetched.
3. The associated MsgSCB is signaled and also the tasks in that SCB are released.

Thus all the registered receivers are notified that the message has arrived or ready.

4.2.2.2 receive (variable_name)

The pseudo code for this primitive:

```
receive (var_name)
{
 msg_semaphore_number = var_name;
 if ( msg_test_and_reset( msg_semaphore_number ) )
   {
    copy_message( MCB[var_name].source_message_buffer,
                  TCB[RT].destination_message_buffer,
                  MCB[var_name].message_length ) ;
                                          // copy to local buffer

    return(true) ;
   }
 else
   {
    return(false) ;
   }
}
```

Receiver on invoking this primitive call on the message with which it is registered:

1. The associated message semaphore is checked whether this task is signaled
2. If signaled copies the message from source to the destination
3. And returns true when the copying finishes.
4. If the semaphore for the task is not signaled i.e., message not ready it returns false to the receiver task.

The calls ***msg_signal_and_release( )*** and ***msg_test_and_reset( )*** are the exact equivalents to signal_and_release( ) and test_and_reset( ). But the messaging versions of primitives are private functions acting on messaging semaphores or MsgSCB, while the latter ones act on synchronization semaphores. The pseudocodes of these primitives are:

- ***msg_signal_and_release( )***

```
msg_signal_and_release (msg_semaphore_number, tasks)
{
 MsgSCB[msg_semaphore_nuber].FLAGS = MsgSCB[ semaphore ].FLAGS  |  tasks ;
 ATV  =  ATV |  MsgSCB[ msg_semaphore_number].TASKS ;
}
```

- ***msg_test_and_reset( )***

```
msg_test_and_reset (msg_semaphore_number)
{
 if  ( MsgSCB[msg_semaphore_number].FLAGS[RT] = = 1 )
                          // check whether the semaphore has signaled the  running task or not
   {
   MsgSCB[semaphore].FLAGS[RT] = 0 ;  // clear the bit in FLAGS vector
   return(true) ;
   }
 else
   {
   return(false) ;
   }
}
```

It is possible to realize all patterns of asynchronous communication just by configuring the message control block. The number of tasks under the ***receiver tasks*** field defines the pattern. If only one registered receiver task is there it results in one-to-one communication, if two or more receivers are there it results in one-to-may communication, and if all the tasks are registered for a message it results in broadcast style. Several one-to-one communication to one single task as receiver results in many-to-one style. Such one/many-to-one combinations results in many-to-many style.

4.2.2.3 get_ message_source_buffer (variable_name)

This primitive is called to get the source address of the corresponding message address. It returns the pointer to the buffer. The pseudocode-

```
get_msg_source_buffer(message)
{
   return(MCB[message].sourceBuffer);
}
```

4.2.2.4 get_message_destination_buffer ( )

This is called by the user tasks after being notified of the message arrival. It gives the address of the local buffer for the task that is calling this primitive. It returns a pointer. Its pseudocode-

```
get_msg_destination_buffer(void)
{
   return (TCB[RT].msgLocalBuffer);
}
```

4.2.2.5 get_ message_length (variable_name)

This is called inside the ***receive(message)*** when **copy_message( )** is executed after the message has been notified. It returns the unsigned character value equal to the size of the message.

```
get_msg_length(message)
{
   return (MCB[message].length);
}
```

4.2.2.6 copy_message(source_address,destination_address,length,)

This called in the receive( ) primitive to copy message from the source buffer to the destination buffer. The arguments passed can be seen in the pseudocode-

```
copy_message(source_address, destination_address,length)
{
   while(length--) {
      destination[length] = source[length];
   }
}
```

4.2.3 Illustration of usage of communication primitives to get content oriented message addressing

The kernel developed is an integrated system. The difference between integrated and non-integrated systems is that transparent communication (can be over a network or locally in one node) is supported by integrated systems. Transparent network imply that all the communication is built into the kernel. This building starts from configuring the messages, messaging semaphores (MsgSCB), message control block (MCB), task control block (TCB) and other structures that are related to communication.

4.2.3.1 Configuration details:

1. Mapping messages to serial numbers uniquely so that the messages can be accessed by names itself. Suppose there are 'n' messages in the system. Then the mapping is as shown in following table (TABLE II).

TABLE II

Identifying Index [i]	message name or variable name
0	variable0_name
1	variable1_name
2	variable2_name
…	……
…	……
15	variable15_name

2. Configuring MCBs individually corresponding to each message or variable that will be communicated. The MCBs of all messages are put together as a table (array). As said earlier, each MCB has three fields- ***receiver tasks(****vector****)***, ***messagebufferAddress(****pointer****)***, ***message length(****unsigned char****)***. MCB corresponding to a message is accessed by saying **MCB[*variable_name*]**. Variable_name will be substituted by the index with which it is mapped (as in step1).

3. Now each message or variable is mapped to a messaging semaphore MsgSCB. All MsgSCBs are put in a table (array). So the mapping is just achieved by the same index of message. Meaning for variable0_name the semaphore will have semaphore with MsgSCB [variable0_name].

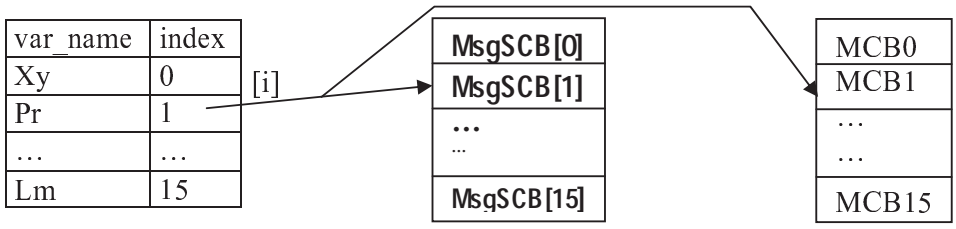


Figure.19 mapping and connection between variables and MCB, message SCBs.
All configurations are done statically.

4.2.3.2 Using the primitives:

There are two situations for communication:

A. Communication initiated by the Integrated event manager (IEM)
B. Communication between the tasks initiate by the involved tasks.

Case A:

In this case associated event's (event_X) descriptor is indexed by X into the event descriptor table. This event descriptor is-

Event_descriptor_table[X]:

Mode	Type	threshold	Event Counter	opcode	Semaphore	tasks	Message Index	Next Event
*	*	X_{th}	X_{th}	OPC_SEND_MSG	semaphore_X	0x8201	variableX_name	*

1	0	0	0	0	0	2	0	0	0	0	0	0	0	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The fields marked with * means its value can be any of the possible value. Threshold number is $X_{th,}$ a number between 0 and 256. Message named variableX_name is to be broadcasted to tasks mentioned by ***tasks*** field in the event_descriptor field . Here tasks 15, 9 and 1 are to be signaled about the message.

The semaphore_X i.e. MsgSCB[sem_X] is used to signal the tasks. Such MsgSCB[semaphore_X] has its fields initialized as:

MsgSCB[variableX_name]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
released tasks	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0

With message index 'varX_name ' MCB is accessed.The MCB[varX_name] is-

```
MCB[variableX] = { 0x8201,
                   *bufAddressX,
                   lengthX
                 }
```

On the expiry of such event counter, the integrated event manager calls the primitive broadcast with appropriate arguments mentioned in the event descriptor:

```
broadcast(variableX_name);
```

Following this call MsgSCB[variableX_name] is signaled to tasks 15,9 and 1 and eventually tasks(15,9 and 1) mentioned by ***tasks*** field are released . Thus message arrival or ready notification is done. Now step 2 in communication is taken care by the receiver task. See the pseudocode for one of the receiver tasks, task_9( ).

```
Task_9( )
{
if(recieve(var_X))
  {
  union msg_destination_buffer_9 *tempPtr =
                      get_message_destination_buffer();

  local_val = tempPtr->expected_variable_type;//the stream is got
            //as per the particulars of variable datatype or struct type
}
taskExit( );
}
```

The receiver task makes a check whether message has arrived or not by calling **receive()** if arrived copies to local buffer. Message is now in the local buffer for the task as defined in the task control block(TCB). This message is purely in unformatted form as a string of bytes. By calling get_message_destination_buffer(), the local buffer address is got, now the byte stream is to be resolved according to the data type particulars. This is deeper implementation details.

Case B: Tasks communicating directly

The entire configuration process for MCB, MsgSCB, message mapping remains the same and are initialized according to the message. Notice the pseudocode for sender task-

```
void task_7(void)
{
.....;
.....:
broadcast(variableX_name);
taskExit();
}
```

The sender just calls broadcast (variable_name) and the message attached to this index is notified to the receiver tasks as mentioned in the MCB for the message. Say that task_7 sends message to tasks 15,9 and 1. The step 1 in communication is finished by this call. Now the receiver tasks are released by some other means, but can also be done by the sender task explicitly. The receiver tasks 15,9 and 1 on reaching their communication point does the same action as shown in the pseudocode for task_9 previously.

4.2.4 Merits of this communication in Kernel

4.2.4.1 Very fast, instantaneous and fixed in time

One should remember the two steps in communication *(section 4.2.1)*. The kernel spends very few machine cycles in the associated notification as it involves just Boolean operations like OR, AND etc. involving the Boolean vectors of one semaphore. This takes constant and fixed time irrespective of the bits being set or reset. In the second step a considerable amount of time (directly proportional to the size of the message) is

spent in the second step. However it is to be understood that this time is spent at individual points in time when the corresponding receiver task is running. Thus time for communication is very small or instantaneous irrespective of how many tasks are communicating. In several existing RT-kernels the message is copied to all the task's message queues at the instant of sending, this results in consuming lot of time and is varies depending on number of tasks to which the message has to be sent. Further sending a message to task, which is not the running one, makes the running task to be delayed very much.

4.2.4.2 No message queues

Further the concept of allocating one local message buffer to each task involved in communication and one buffer each for source messages and the design implementation resulted in no need for message queues. This resulted in reducing the memory overhead and time overhead associated with message queue processing.

4.2.4.3 No risk of message loss

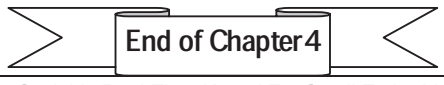
The tasks are basic tasks and at any instance it can consume only one message and once the message is consumed the buffer can be rewritten and will be rewritten only if the task needs one more message, but this will happen after previous message is consumed. Further the source message has its own buffer and will be updated by the appropriate tasks as and when necessary.

4.2.4.4 Highly compilable

A task is allocated a message buffer if and only if it requires at the static configuration stage of the application tasks with the kernel. This results in allocation of memory to tasks as per the individual tasks requirement.

4.2.4.5 Transparent communication

Due to the implicit addressing application programmers are freed from the trouble of specifying the receiver tasks, buffer addresses, message length. This results in transparent communication, because all the communication is built into the kernel by configuring the communication part in the software bus module.



Chapter 5. Compound tasks and Secondary-level scheduling algorithms

Task management is done following the fixed priority scheduling policy, where each task is having a unique base priority. In the kernel a task can take priority from 0 till 15. The task priority is a whole number not a floating-point number. But in several control applications there is a need for several tasks having same priority. Such tasks need to be executed in a fixed sequence, for example in FIFO basis. In some cases the tasks need to be scheduled following static cyclic scheduling i.e., say there are a set of hard real-time tasks that are to be released at a certain frequency periodically.

5.1 Compound task and subtasks

To meet this kind of special requirement, under the kernel there is a task type called **compound task**. See the figure (Fig.6- Chapter 3) there are two types of user tasks under the task manager. A simple task is a basic task with linear structure as shown in figure (Fig.7- chapter 3). By saying linear structure it means there is no change in execution sequence and all significant jobs are done by the statements or subroutines but not calling other tasks.

Compound task is a segregation of several simple basic tasks, which are executed as subroutines (with no parameters) following a rule. Further there is a scheduling algorithm according to which these simple tasks execution and execution sequence is carried out. The contained simple tasks under the compound task are called **subtasks.** In this context compound task is called as ***parent or mother task*** of subtasks. Again compound task is also a basic task i.e., it can never be blocked.

All the subtasks are executed at the priority of the corresponding mother task. It should be kept in mind that the priorities (unique) are allocated to the tasks under task manager with no discrimination of compound or simple. Therefore there are totally 16 tasks under the task manager of which some are simple tasks while the remaining are compound tasks. Further the task manager treats both tasks uniformly. It is the body of compound task that takes care of scheduling subtasks.

5.2 Subtask scheduling algorithms

Out of all the subtasks under a compound task, one or more (can also be all or none) are run each time the compound task is invoked. The subtasks and sequence of subtask execution may also change at each invocation of compound task. These are all done according to the local scheduling algorithm followed within the compound task. This algorithm is referred to as ***secondary-level scheduling algorithm***, thus the compound tasks act as ***secondary level schedulers*** as illustrated in the Fig 20.

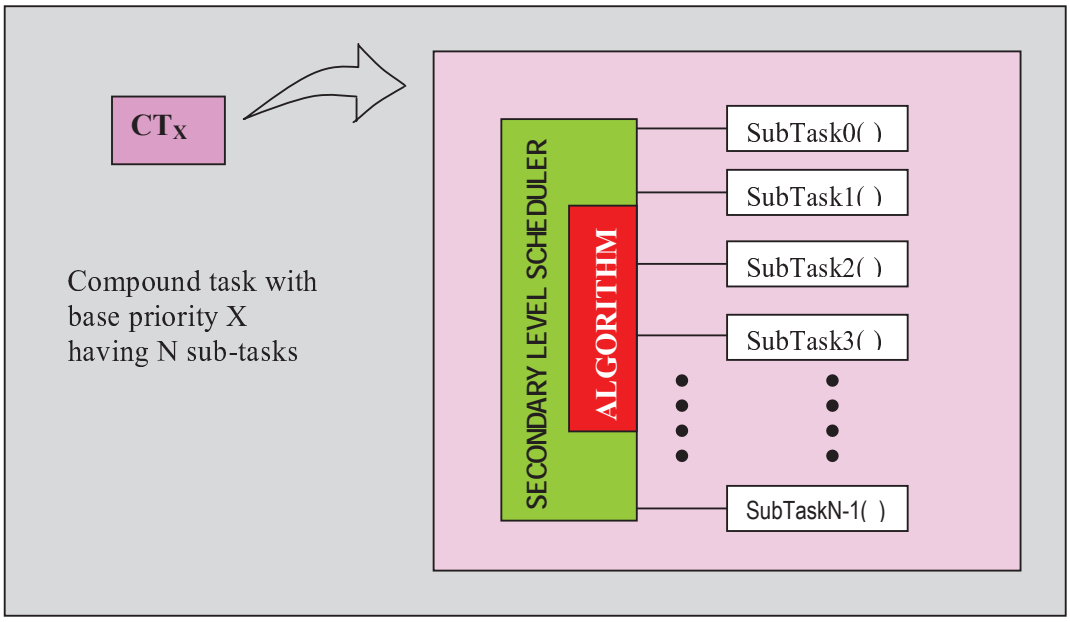


Figure.20 Compound task as secondary-level scheduler with subtasks

The three types of secondary-level scheduling algorithms supported under this kernel are:

- FIFO style
- Static-cyclic scheduling
- Arbitrary invocation sequence following state-logic controller algorithm

5.2.1 FIFO style

In this type of compound task, all the subtasks are invoked and the sequence of execution is fixed, this sequence is built in a precedence-constrained manner. Meaning that subtasks are placed in the sequence according to their precedence and significance. If a subtask subTask_X( ) is of high importance it is positioned at the top in sequence. Thus subtasks are invoked in the order of their precedence. A compound task following FIFO style subtask execution has following pseudocode:

```
Compound_task_X ( )
{
  subtask_1 ( ); // call subtask_1 as subroutine with no parameters
  subtask_2 ( ); // call subtask_2 as subroutine with no parameters
  subtask_3 ( ); // call subtask_3 as subroutine with no parameters
  subtask_4 ( ); // call subtask_4 as subroutine with no parameters
  …
  …
  subtask_N( ); // call subtask_N as subroutine with no parameters
  taskExit( );
}
```

5.2.2 Static Cyclic Scheduling using the Boolean Vector Semaphores

This compound task employs static cyclic scheduling algorithm to execute the subtask. In such algorithm the subtasks are periodic having a fixed period for release and they are to be released at their respective frequency. When two or more tasks are released at the same time the subtasks are executed in a precedence-constrained manner.

5.2.2.1 Static-cyclic scheduling and methods to develop such schedule

In many hard real-time systems (can also be in soft real-time systems) there are periodic tasks that are to be run and hence released at a predefined individual period or frequency. Based on this knowledge a schedule is developed statically which will schedule all the cyclic tasks.

5.2.2.1.1 Developing static-cyclic schedule

The method to develop a static-cyclic schedule is very simple. There are two approaches:

- offline: table-driven implementation
- online: tick-driven implementation

A) Off-line approach:

As all the tasks are released periodically with certain frequency there are many chances that the pattern of release will be repeated after certain time. The length of such cycle can be calculated by mathematical formulae. Such cycle which is repeated in the schedule is called ***major cycle***. While the tasks are released only at points that correspond to the common unit or resolution of all tasks periods. This point also has some time period of repetition, this cycle is called ***minor cycle***.

In making an off-line static schedule there are certain ***assumptions***, which are:

1. For each static-cyclic schedule there is a *periodic task* set. All tasks are periodic.

   $\{\tau_1, \tau_2, \tau_3 \ldots\ldots \tau_N\}$

2. Each periodic task has a fixed period(T), execution time(C) and deadline(D).

   $\tau_1 : T_1, C_1, D_1 = T_1$

   $\tau_2 : T_2, C_2, D_2 = T_2$

   .....

   $\tau_N : T_N, C_N, D_N = T_N$

A static cyclic schedule is constructed for these tasks, and its major cycle schedule only will be shown here. The major cycle has a period T_{maj} and minor cycle T_{min}. Each major cycle has ***m*** minor cycles i.e., $\boldsymbol{m} = T_{maj} / T_{min}$. While T_{maj} and T_{min} calculated as:

T_{maj} = greatest common divisor ($T_1, T_2 \ldots\ldots\ldots\ldots T_N$)

T_{min} = least common multiple ($T_1, T_2 \ldots\ldots\ldots\ldots T_N$)

Assumption 3: Sum of all the execution periods for all the tasks is less than the minor cycle period i.e., $\Sigma C_1 <= T_{min}$. This implies non-preemptive scheduling of periodic tasks

Table-driven implementation of off-line static scheduling:

Now see the execution patter of all the tasks in Fig 21.In the figure only three tasks are shown. This is pattern pertaining to the major cycle, which when repeated results in a complete schedule.

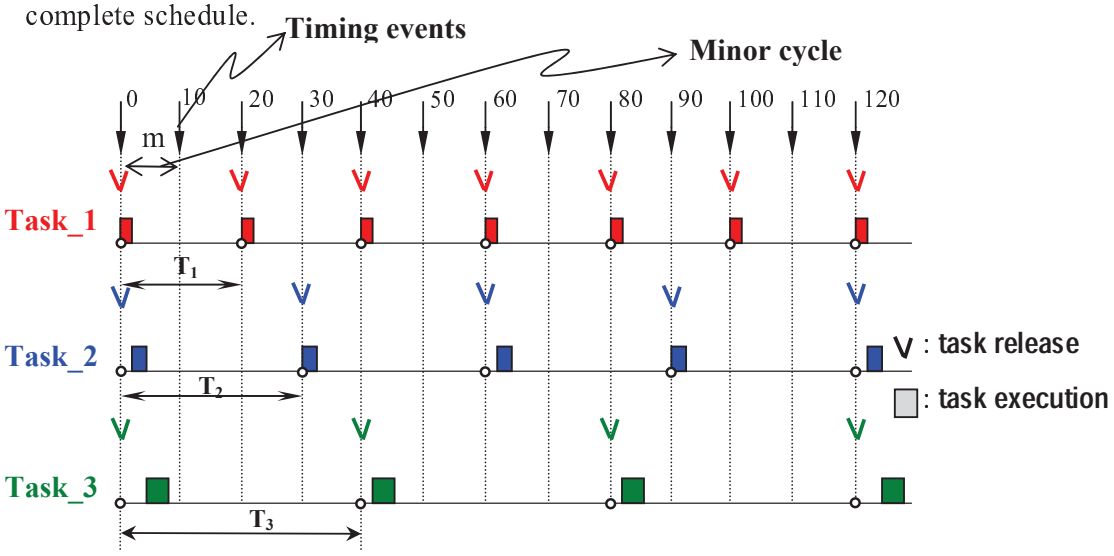


Figure.21 Static cyclic scheduling pattern complete major cycle

Let us take a static schedule for only three cyclic tasks being task_1, task_2 and task_3 with periods 20, 30 and 40 time units respectively. While priorities: p(task_1) > p(task_2) > p(task_3).Accordingly we get the period for major cycle T_{maj} = 120 and minor cycle T_{min} = 10 units. Thus we get m = T_{maj}/T_{min} = 12.

From the Fig20, it is easy to see that the schedule can be represented by a 12 X 3 matrix called '**SCHEDULE'** having Boolean flags f^i_j as elements, where **i** represents the row number (minor cycle number) and **j** represents the column number (task number). If the **task_X** has to be released during a minor cycle **Y,** the flag f^Y_X should be set. So such a schedule can be represented as a table (implemented as matrix) as shown in Table II.

TABLE III

task / minor cycle	Task_1	Task_2	Task_3
0	1	1	1
1	0	0	0
2	1	0	0
3	0	1	0
4	1	0	1
5	0	0	0
6	1	1	0
7	0	0	0
8	1	0	1
9	0	1	0
10	1	0	0
11	0	0	0
12	1	1	1

This can further be generalized to develop any schedule involving 'n' tasks and 'm' minor cycles for one major cycle. Where we get a ***m* x *n*** SCHEDULE matrix i.e., a matrix of dimensions ***(T_{maj} / T_{min})* X *number of tasks*** . Row **i** of this matrix contains flags f^i_1, f^i_2, ……. f^i_n which are initialized to 1 or 0, if 1 means the task to which the flag belongs is executable while 0 means not executable.

The schedule is run by the scheduler implemented in an interrupt service routine (ISR), which is periodically activated by timer interrupts arriving at interval equal to that of minor cycle T_{min} period. This offline schedule development is called as table-driven implementation.

Such a scheduler looks like-

```
Static_cyclic_scheduler is:
{
  restart timer(Tmin);

  // initially i =0

  get SCHEDULE[i];    // get the row from schedule matrix corresponding to i - minor cycle

  for(j =1, j<= n; j++ )
    {
    if (f i j = 1) execute_task(τj) ;
    }

  i = (i+1) mod m;    // i on reaching vale m should start from zero again
  exit;               // RTI
}
```

B) On-line schedule development:

The table driven implementation has a very big shortcoming, i.e. whenever the task with periods happen to be prime numbers it results in a very big table. Also if there are many tasks it also results in increase of table size. This is over come by developing the schedule on-line at run-time i.e. generating flags f^i_j by employing timers (ST). Each task is allocated a scaling timer (ST_x) whose period of overflow is equal to the task period. That is-

$T_1 = k_1 T_{min}$ *// this interval measured by* ST_1

$T_2 = k_2 T_{min}$ *// this interval measured by* ST_2

……

$T_n = k_n T_{min}$ *// this interval measured by* ST_n

The algorithm which develops the schedule and drives the schedule is as following:

```
Static cyclic scheduler is:
{
  restart timer(Tmin)

  for (j =1; j<= n; j++)
    {
     decrement STj
```

```
    if (STj= 0)
      {
        STj = kj  ;  // reload the scaling timer

        fj = 1;      // set the flag indicating τj is to be run
      }
    else  fj = 0;
   }

  for (j = 1;j <= n; j++)  // this is the driver routine
  {
    if(fj = 1) execute_task(τj);
  }

  exit;         //RTI
}
```

In the first for loop all the scaling timers are updated and if the timers have expired they are reloaded and the flag corresponding to that task in that minor cycle is set. If the timers haven't expired the flags are reset. When all the flags in a minor cycle are updated, the schedule of row is ready. This for loop is finished, then the driver for loop executes the schedule made in the previous step. Thus the schedule is made just before being consumed. This method is called interval-driven implementation.

5.2.2.2 Implementing Static-cyclic scheduler by Boolean vector semaphores

The algorithm is not implemented explicitly in the compound task but is achieved using the concept of synchronization using Boolean vector semaphores. All the subtasks under a compound task are mapped on one-to-one basis to different semaphores, whose Semaphore Control Blocks have the ***tasks*** initialized with a Boolean vector with bit corresponding to the mother task is set. The compound task pseudocode is-

```
task_X ( )
{
  if(test_and_reset(semaphore1))   subTask1( );
                        // subtaskN is called as subroutine - subTaskN has high precedence
  if(test_and_reset(semaphore2))   subTask2( );
                        // subtask2 is called as subroutine
  ……
  ……

  if(test_and_reset(semaphoreN))   subTaskN( );

                        // subtask1 is called as subroutine- subTask1 has less precedence
  taskExit( );
}
```

1. The mapping of semaphores with subtasks is implemented by the if conditions.
2. Then corresponding to each subtask an event is defined whose threshold occurrence results in the same time period as the cyclic subtask period.
3. These events are mapped to the semaphores with which the corresponding subtasks interact.
4. Setting the semaphore field under the event descriptor with semaphore number this mapping is achieved.
5. The ***tasks*** Boolean vector field under each event descriptor is initialized with a value that corresponds to the mother task under which are all the subtasks.
6. Under the semaphore control block of each semaphore the **tasks** vector field is given a value which has one set bit corresponding to the mother task.
7. In all these event descriptors the opcode field is defined as OPC_SIGNAL.
8. Thus on the expiry of such event counters the primitive ***signal_and_release( )*** is invoked separately, by passing *semaphore* and *tasks* as arguments.
9. Following this call, the tasks mentioned by ***tasks*** (compound task) is signaled and tasks (compound task) mentioned in the ***tasks*** of semaphores is released.
10. Correspondingly the subTasks are signaled through the semaphores.
11. Based on the priority given to the mother task, when it is scheduled to run carries its execution as shown in its pseudo code. It checks whether a semaphore is signaled and executes corresponding subtask.

Thus the static cyclic scheduling is realized by means of Boolean vector semaphores.

Consider the same example under 5.2.2.1.1 A, but the tasks are now subtasks put under a compound task whose ID number is say 7. subTask1( ), subTask2( ) and subTask3( ) with precedence of: subTask1( ) > subTask2( ) > subTask3( ).The cyclic periods are 20,30 and 40 seconds. We define three events whose threshold occurrence results in the period of the cyclic subTasks. Let the events be event_1, event_2 and event_3. All the events have the same opcode and it is OPC_SIGNAL. The semaphore field under each event holds three different semaphores whose indices are semaphore1, semaphore2 and semaphore3. All the events are registered under the corresponding resolution tables, in this example all are of same resolution i.e., seconds. The semaphore control blocks SCBs of the mentioned semaphores are initialized as following.

SCB[semaphore1] : It is initialized such that when its signaled and released the task task_7( ) is released. The SCB [semaphore3] has the fields as following:

SCB[semaphore1]	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flags	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
released **tasks**	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

The remaining two semaphores are also initialized similarly with the **tasks[7]** bit set.

The three events in its event descriptor have tasks field. This field for all the events has same Boolean vector value with bit 7 set. Now when such events expire i.e., at periods 20,30 and 40 seconds, the primitive signal_and_release( ) is invoked with semaphores 1,2,3 and ***tasks*** argument with bit 7 set. Following this primitive execution the task_7 (compound task) is signaled and released. The mother task when scheduled to run checks the semaphores then executes the subtasks by invoking them as subroutines. The pseudo code for such compound task is-

```
task_7 ( )
{
  if(test_and_reset(semaphore1))   subTask1( );
                        // subtask3 is called as subroutine
  if(test_and_reset(semaphore2))   subTask2( );
                        // subtask2 is called as subroutine
  if(test_and_reset(semaphore3))   subTask3( );
                        // subtask1 is called as subroutine
  taskExit( );
}
```

Notice the order of positioning the calls to the subtasks this ordering is done according to the precedence. As subTask1 has high precedence it is the first in the sequence, and the sequence continues with decreasing precedence subTasks.

Irrespective of the number of subtasks the concept remains the same.

5.2.3 Execution of a arbitrary sequence of subtasks

There are applications where there is a need for some extended tasks. Extended tasks are the tasks that can be blocked waiting on some event. In the absence of extended tasks such type of behaviour can be emulated by the compound tasks, which use the State Logic controller type algorithm to schedule the subtasks [1]. But to use this, the task should be split into appropriate subtasks and a state machine specifying the execution sequence for these subtasks should be developed. Consider a state machine whose state transition graph is given below, which specifies the execution sequence of subtasks influenced by the events. This is a typical example of event driven state machine (Moore machine).

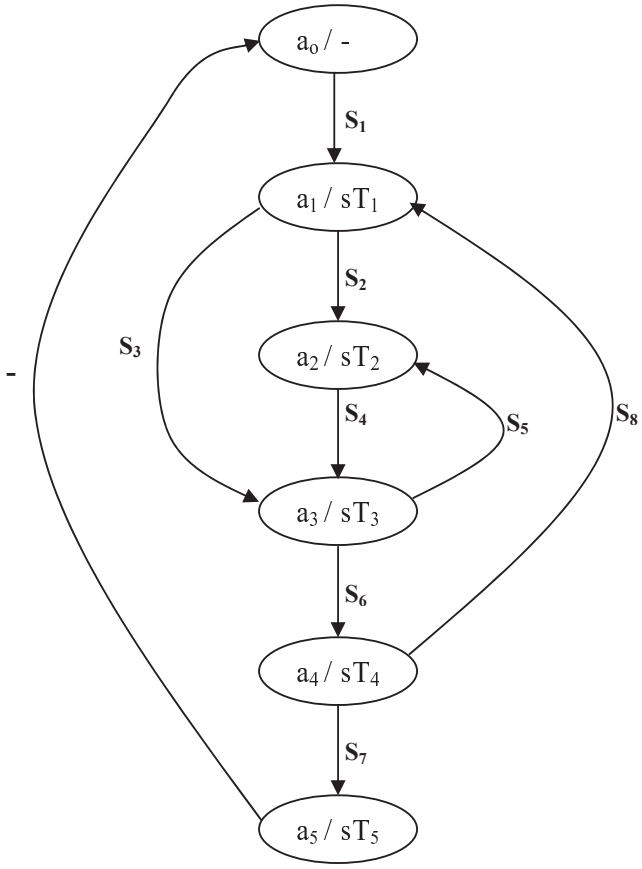


Figure.22 State transition graph of a Moore machine.

Here sT_1,sT_2,sT_3 … sT_5 stand for the subtasks 1,2,3… 5. While S_1,S_2…S_7 stand for the semaphores.Now following the steps[1] of developing the BDD for a MooreMachine represented by the above graph, we get the BDD_Table.While the control memory table consists of the subtasks that are mapped with the state correspondingly.We thus develop

the next state mappings for a_0, a_1… a_5 and make BDD for each next state mapping, number the nodes and develop the needed BDD table. The structure of BDD Table and Control Memory Table are shown on the next page.

All the transitions are effected by the events that are notified by means of semaphores in the kernel. Every event is mapped one-to-one with the semaphores. Initially the machine will be in state a_0 . When activated for the first time if event1 i.e. semaphore1 (S1) is signaled the machine exits the state a_0 and enters a_1. Thus executing the subTask1( ) i.e. sT_1 . After the execution of sT_1( ) it waits for event3 or event2, which are mutually exclusive events notified by the semaphores S3 and S2. This execution sequence continues only if the appropriate events occur. Meaning if the machine is in state a_2 and if the events i.e. S1, S2, S3, S5, S6, S7, S8 are signaled the machine will not respond and remains in the same state. But if event4 i.e. S4 is signaled it moves to next state i.e. a_3. It can be clearly seen that the machine is waiting at the states for the enabling event(s) to be signaled. This concept when imported into a compound task we achieve the extended task behaviour implicitly. The method will be described with an example but before going into details it is important to know the requirements in developing such task.

The requirements are:

1. A task should to be divided into portions and such portions should be encapsulated in separate subtasks.
2. An event-driven state machine should be developed that defines the subtask execution sequence, where each state is mapped to one subtask.
3. The events that enable different transitions from the same state must be mutually exclusive thus meeting the consistency properties. This means if $S_1, S_2 \ldots S_n$ are enabling events for different transitions from one same state, at any instant only one of these should be signaled i.e. if $\mathbf{S_i = 1}$ then $\mathbf{S_{j \neq i} = 0}$

Based on the state machine designed the state transition graph is developed following the standard procedures [1]. Then the graph is decomposed into binary decision diagrams as per rules proceeding in steps as following:

STEP 1: Identify the next state mappings of each state in the state transition graph.

Illustration: For the state transition graph shown in Fig 22 we get next state mappings:

$F(a_0) = a_1(S_1)$

$F(a_1) = a_2(S_2)\ ,\ a_3(S_3)$

$F(a_2) = a_3(S_4)$

$F(a_3) = a_2(S_5)\ ,\ a_4(S_6)$

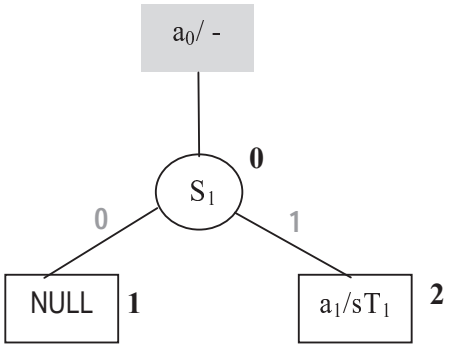
$F(a_4) = a_1(S_8),\ a_5(S_7)$

$F(a_5) = a_0(-)$

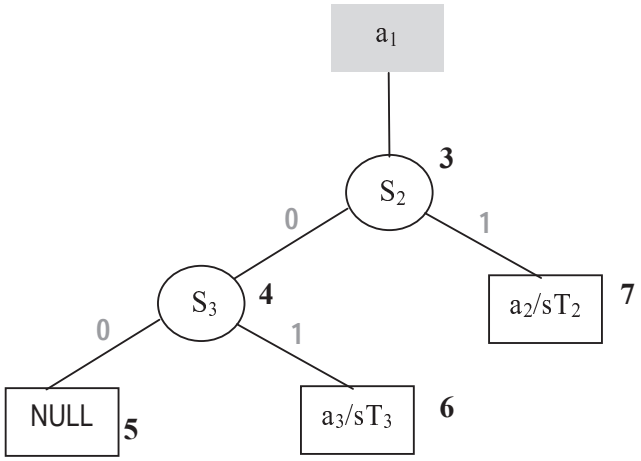
STEP 2: Develop the binary decision diagram for each state, following the next state mapping for that state.

Illustration:

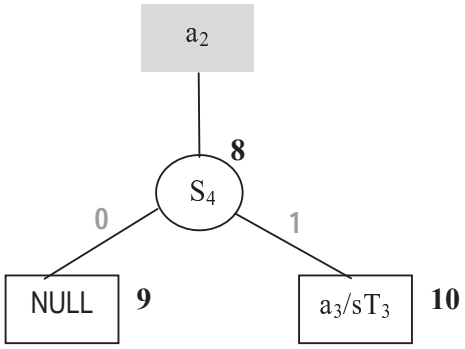
i. $F(a_0) = a_1(S_1)$



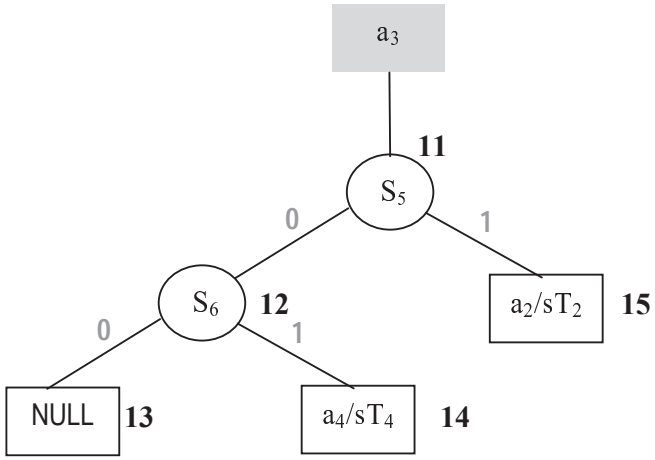
ii. $F(a_1) = a_2(S_2)\ ,\ a_3(S_3)$



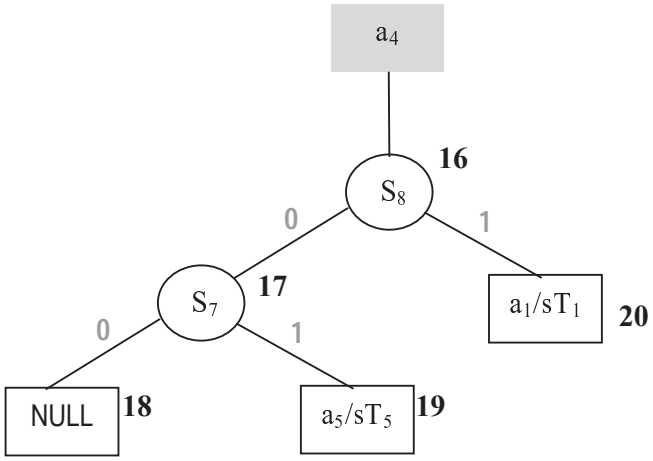
iii. $F(a_2) = a_3(S_4)$



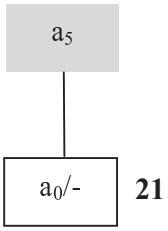
iv. $F(a_3) = a_2(S_5)$, $a_4(S_6)$



v. $F(a_4) = a_1(S_8)$, $a_5(S_7)$



vi. $F(a_5) = a_0(-)$



STEP 3: Develop a Binary decision diagram table for the entire state machine.

1. Number continuously all the nodes each next state mapping's binary decision diagrams (BDD) starting with zero. See the colored numbers against each node in the BDDs under step 2.
2. Now construct a table as shown in table (TABLE IV).
3. First fill the Node number (Node no.) column starting from zero till the last node's number.
4. Fill the Node type column with either S or a if the node is semaphore or a state respectively.
5. See the successors for each semaphore node fill them accordingly. When semaphore is not signaled the resulting node should be written under successor zero, if signaled under successor1.

TABLE IV: BDD_Table

Node No.	N_type	N_index	successor_0	successor_1	BLOCK
0	S	1	NULL	2	
1	-	-	-	-	$F(a_0)$
2	a	1			
3	S	2	4	7	
4	S	3	NULL	6	
5	-	-	-	-	$F(a_1)$
6	a	3	-	-	
7	a	2	-	-	
8	S	4	NULL	10	
9	-	-	-	-	$F(a_2)$
10	a	3	-	-	
11	S	5	12	15	
12	S	6	NULL	14	
13	-	-	-	-	$F(a_3)$
14	a	4	-	-	
15	a	2	-	-	
16	S	8	17	20	
17	S	7	NULL	19	
18	-	-	-	-	$F(a_4)$
19	a	5	-	-	
20	a	1	-	-	
21	a	0	-	-	$F(a_5)$

STEP 4: Use the standard algorithm meant for event driven state logic controllers as the compound task body. Such a compound task pseudocode is as following-

```
task_X( )
{
 do
 {
  state = o;
  index = BDD_Table[i].N_index  //initially i =0
  //determine current state

  switch (BDD_Table[i].N_type)
  {
   case 'S' :
    if (test_and_reset( arbSCB [index] ) )
      {
      i = BDD_Table[i]. Successor1;
      }
    else if(BDD_Table[i]. Successor0 = = NULL)
       {
       break_do_while loop( );
       }
          else
          {
          i = BDD_Table[i]. Successor0;
          }
    break;
   case 'a' :
    if(control_memory[index].subtask != NULL)
      {
       start(control_memory[index].subtask);
       i = BDD_Table[i].successor1;
      }
     break; // switch end
  }
 } while(not( state = =1 &&
 control_memory[index].imm_transition!=1)
 taskExit( )
}
```

This alogirthm operates on the BDD_Table developed and invokes the subtasks which are placed under the control_memory table. This control_memory table is an array of structure which has two fields.

control_memory [] = { address_of_the_subtask;
immediate_transition;
}

Each element in control_memory_table is uniquely mapped to one state, thus there are same number of elements in the control_memory table as the number of states.

Depending on the signaled semaphore if it signals a relevant event corresponding to the state in which the machine is, the transition is done and the mapped control_memory is executed by the compound task. If the subtask address field is not empty, the subtask pointed to by the address field is invoked. The control_memory_table looks like shown in Fig.23(a). Control_memory_table for the discussed example is in Fig 23(b). Following this the compound task exits. In the next invocation the task remembers its previous state and when receives relevant event then when its running the corresponding subtask is invoked through its address in the control_memory table.

control_memory_table

a)

Subtask	imm_transition
subtask1Address	..
subtask2Address	..
subtask3Address	..
subtask4Address	..
……..	..
……..	..
……..	..
subtaskNAddress	..

b)

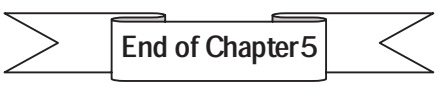
subtask	imm_transition
NULL	..
subtask1Address	
subtask2Address	..
subtask3Address	..
subtask4Address	..
Subtask5Address	1..

Figure.23 (a) Structure of Control memory table (b) control_memory_table for the example

If the transition from a state is immediate then the field imm_transiton of control_memory_table for that state is set to 1.

Who will signal the semaphores?

Semaphores are signaled by the application tasks or IEM and needs to be configured in the context of this arbitrary subtask execution carried out by the compound task. The implementation framework is incorporated into HARTEX$_\mu$. This is provided to support for such compound tasks, which may be used in future.



Chapter 6. Event Management

The purpose of the real-time system is to recognize various timing, external and internal events and to generate relevant reactions by executing the corresponding application software (firmware) subroutines (tasks) in a timely and predictable manner. It is the duty of event manager to identify the occurring events and accordingly indicate task manager or/and software what kind of operation is to be taken when the event counter for one or more event(s) expire.

In the microcontroller all the necessary timing is provided by the timer interrupts provided by the timer hardware module (real time clock). In several existing kernels tasks are given a time stamp (the instant in time at which the task should be released). A software clock is setup in the appropriate tick interrupt for this purpose. When the time in this clock matches the time stamp(s), corresponding task(s) are released. This is how the static scheduling associated with timing is achieved. While the dynamic scheduling is made possible by setting up counters around the related interrupts, when this threshold is reached action is taken. This type of kernels has different modules handling the tick interrupts and external events called time manager and event manager respectively.

In present kernel, design and implementation of Event Manager is made simple and unified, by treating all the interrupts (tick interrupts and all hardware interrupt etc.) in the system as events. Thus we have one subsystem called Integrated Event Manager that provides the integrated time and event management. Further each individual significant event is given a structure called Event Descriptor which contains all the particulars about the action to be taken when the event counter expires. Event counter is set equal to the associated threshold.

What is threshold value?

In embedded systems all events are conveyed by the interrupt requests (IRQs). But an operation is to be done when this event has occurred for some number of times. Sometimes this number can be one or some other value. This value of repetition number is called threshold. Now we see how time management is achieved by events-Say we need to release a task task_X( ) every one second. In the microcontroller we initialize

the hardware so that a tick interrupt comes every 10 mill-seconds, say it as an event. To get one second, this event must occur 100 times, so by defining a threshold value for this event as 100 and defining the action as **release of *task_X,*** the needed time management is achieved by the event management concept.

6.1 Event Descriptor

There is an event descriptor for each significant event. By significant event we mean that it is having some significant operation associated with its happening. This operation can be:

- Synchronize tasks by invoking signal_and_release( ) under synchronization part of software bus
- Send message to tasks by invoking broadcast( ) under communication part of software bus.
- Release tasks by invoking release( ) under task manager
- Enable the next event by invoking enableNext( ) under the Integrated event manager.

And to each primitive call we need to feed appropriate arguments, and these are also specified with the event descriptor. The event descriptor has 9 fields as shown below-

Event Descriptor:

mode	type	threshold	Event counter	opcode	semaphore	tasks	messageindex	nextEvent

Each field meaning and other details are:

- **mode :** This specifies the status of event either enable or disabled.
  possible values: ENABLE / DISABLE
  where ENABLE = 1, DISABLE = 0

- **type :** This specifies the type of event whether it's a free running or on-off type.
  possible values: FREE_RUNNING / ONE_OFF
  where FREE_RUNNING = 2, ONE_OFF = 1

- **threshold :** This indicates the threshold for the event
  possible values: 0 to 255

- **event counter:** This is the decrement counter for the event that decrements on each occurrence of the event. On every counter expiry (counter reaching 0) it is reloaded with threshold.

  possible values: 0 to 255

- **opcode :** This specifies what are the actions to be done on expiry of event counter. Based on this value corresponding subsystem primitives are invoked.

  possible values: OPC_SIGNAL, OPC_SEND_MSG,
  OPC_RELEASE, OPC_ENABLE_EVENT.

  Where ***OPC_SIGNAL = 0x01***
  OPC_SEND_MSG = 0x02
  OPC_RELEASE = 0x04
  OPC_ENABLE_EVENT= 0x08

  Further:

  OPC_SIGNAL- indicates the **signal_and_release( )** invocation
  OPC_SEND_MSG- indicates the **broadcast( )** invocation
  OPC_RELEASE – indicates the **release( )** invocation
  OPC_ENABLE_EVENT- indicates the **enableNext( )** invocation.

  NOTE: The opcode can take one or more values simultaneously because notice the possible values and their actual values. By just OR-ing between two different values you get one new value, which can be fed to the opcode field for an event.

- **semaphore:** This specifies the semaphores associated with the synchronization operation if the opcode has ***OPC_SIGNAL*** with it.

- **tasks:** This specifies the tasks to be released if opcode contains ***OPC_RELEASE.***

- **message index:** this specifies the source message index that is to be broadcasted if opcode contains ***OPC_SEND_MSG.***

- **next event:** this specifies the next event index when the event is of ONE_OFF type and opcode for the event contains ***OPC_ENABLE_NEXT.***

6.2 Time and event management in Integrated Event Management

6.2.1 Basic Event Processing

6.2.1.1 Basic Timing Event Processing

A specific amount of time can be measured in multiples of tick interrupt (events) that occurs with a fixed period. This tick interrupt corresponds to the fine granularity of the time in the system. In the present kernel basic tick interrupt has a time period of 10 milliseconds. Consider bigger periods which are in hours, minutes etc then measuring time in multiples of 10 milliseconds is very difficult and takes huge memory for the threshold and event counter fields in the event descriptor. To overcome this problem, in the *tick interrupt* there are four flags corresponding to resolutions:

- **Basic time unit i.e., 10 milliseconds** in present case.
- **1 second**
- **1 minute**
- **1 hour**

These flags are set whenever the corresponding time has elapsed. Four different types activities are carried out whenever these flags are set, that is the essence of event management. After finishing these activities the flags are cleared. Again the flags are set as the time elapses the corresponding activity is executed and then the flag is cleared. This is repeated continuously which accounts for time management.

This portion of timing event management is called Basic Timing Event Processing and is done in the interrupt service routine of the tick interrupt in Hardware Adaptation Layer subsystem. Because of this BTEP, resolutions of second, minute, and hour are made available.

6.2.1.2 Basic external event processing

The external events are conveyed by the external interrupt requests (IRQ), while the corresponding interrupt service routine (ISR) carries out the basic interrupt processing specific to the type of interrupt then it invokes the Integrated Event Manager (IEM) through the primitives on the associated event descriptor.

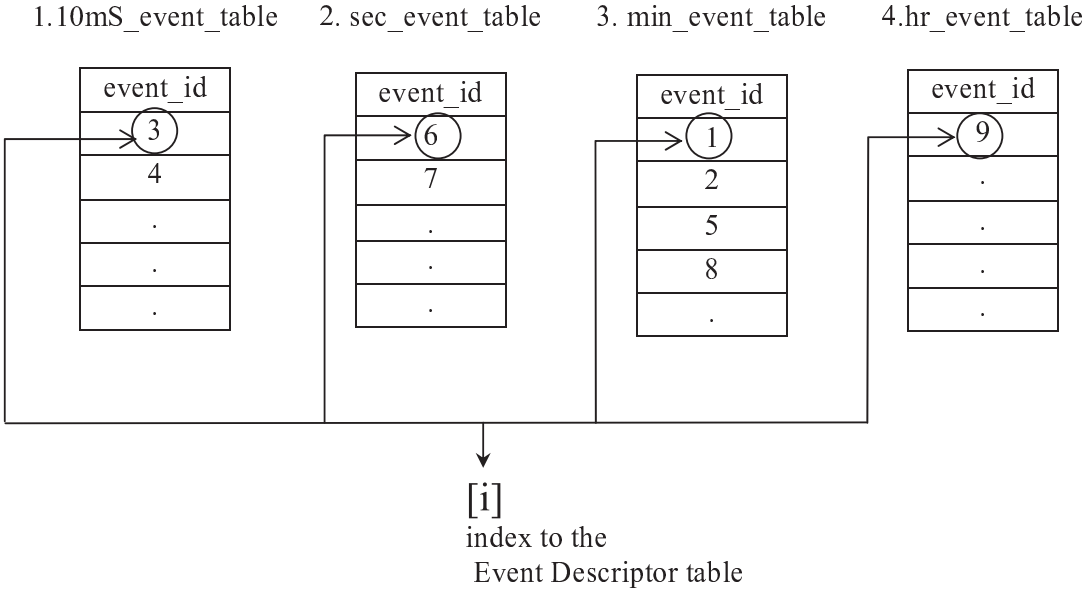
6.2.2 Essential Event Management

The actual event management is performed by Integrated Event Manager (IEM), which is invoked by HAL (from the tick ISR) through the primitives provided in IEM. Whatever be the type of event, the action to be taken should be same (from the operational perspective of the Integrated Event Manager) and to make such homogeneity in the implementation of the Integrated Event Manager, the following solution is proposed. It consists of four tables of events, where each table contains events of one resolution

Following this solution we have four tables for the events that occur periodically:

1. 10mS_event_table
2. sec_event_table
3. min_event_table
4. hr_event_table

Each table holds the indexes of all those events whose granularity is same i.e., 10ms_event_table has the indexes of the events under the Event Descriptor table whose granularity is 10 milliseconds. Sec_event_table holds indexes of events whose granularity is in seconds. These index tables are implemented as an array of type **unsigned char**.



6.2.3 Event Descriptor Table (event_descriptor_table)

All the events are initialized and put together under a table called Event Descriptor Table. This table is implemented as an array of Event Descriptor type(mentioned under section 6.1) . The kernel is a static one that is all the tasks are known and the associated activities are all configured appropriately and similarly all the events are initialized filling in the appropriate fields with appropriate relevant values as per the application requirements. The Event Descriptor table looks as following-

Table V

[i] index fetched from individual resolution tables

	Mode	Type	Threshold	Event counter	Opcode	Semaphores	Tasks	MessageIndex	Next Event
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
..	…	..	…	…	…	…	…	…	…
..	MAX	…	…	…	…	…	…	…	…

All the fields are filled relevantly, and the events corresponding to each granularity are placed under the corresponding Index Tables. *The event descriptor for event 'X' is accessed by saying* ***event_descriptor_table[X].***

6.3 Implementing integrated time and external event management

TIME MANAGEMENT: In the tick interrupt (occurring at the least resolution wanted) the basic timing event processing is done. The Basic Timing Event Processing involves

the raising of four flags(mSec,sec,min,hr) , which correspond to the time unit of duration 10 milliseconds , 1 second, 1 minute and 1 hour respectively.

The flags are raised only when that much amount of time corresponding to the flag has lapsed. After updating the flags accordingly in each tick interrupt, the event manager is invoked through the primitives with the events in the tables corresponding to the resolution of the flags that are set.

6.3.1 Basic event processing implementation

6.3.1.1 Basic timing event processing implementation

The hardware is initialized so that we get a tick interrupt every 10 milliseconds. The pseudo code is for the tick_ISR:

```
Tick_ISR ( )   // occurring every 10 milliseconds
{
  raise ( msec_flag ) ;
  msec++;   // initially msec= 0

  if   (msec = = 100)
    {
     raise sec_flag ;
     sec++;
     msec = 0 ;   // reset the msec counter to zero
    }

  if (sec = = 60)
    {
     raise min_flag ;
     min++;
     sec = 0 ;  // reset the sec counter to zero
    }

  if (min = = 60)
    {
    raise hour_flag ;
    min = 0 ;  // reset the min counter to zero;
    }

  // event manager invocation on different tables

  if(msec_flag is true)  sweep_table ( 10ms_event_table) ;

  if(sec_flag is true)  sweep_table (sec_event_table) ;

  if(min_flag is true)  sweep_table (min_event_table) ;

  if(hour_flag is true)  sweep_table (hr_event_table) ;

  clear_all_timing_flags;
}
```

6.3.1.2 Basic external event processing implementation

All the events are mapped on one to one basis with the external interrupts ***(IRQs)***. The correspondent basic event processing is achieved with the interrupt service routines***(ISRs)***. In the ISR:

1. The interrupt and hardware specific servicing is done .
2. Then the Integrated Event Manager **(IEM)** is invoked by calling ***eventManager(event_descriptor)*** **primitive**.

The pseudocode of an ISR for an external event 'X' is:

```
ISR_External_Event_X ( )
{
 ..............; // hardware and interrupt specific interrupt processing
 ..............;
 ..............;
 eventManager( event_descriptor_table[X] ) ;
                     //invoke IEM by passing the event descriptor
                     //registered for this interrupt
}
```

6.3.2 Integrated event management primitives

To access the Event Descriptors and interpret them the IEM provides following primitives:

- sweepTable(IndexTable, tableLength)
- eventManager(event_descriptor)
- enable(event_descriptor)
- disable(event_descriptor)

To be specific all the parameters are passed as pointers or by address. This avoids the time and memory overhead involved in copying the big data structures.

6.3.2.1 sweepTable(IndexTable, tableLength) primitive

The pseudo code for this call is:

```
sweepTable(IndexTable , tableLength)
{
  for( i = 0 ; i < tableLength; i++)
    {
       eventManager(event_descriptor_table[IndexTable[i] ] ) ; // invoke eventManager on each event
                                                               // whose index is in the IndexTable
    }
}
```

This primitive is invoked by BTEP in HAL by passing table and length of table as arguments. It inturn invokes the eventManager on each event descriptor whose index is in the IndexTable.

Illustration:

If seconds event table is passed to this sweepTable( ), it calls eventManager ( ) primitive on each Event Descriptor that is registered under the IndexTable: seconds event table. These indexes under the index table correspond to the respective Event Descriptor in the Event Descriptor table mentioned. What the eventManager does and its operation are explained under next section.

6.3.2.2 eventManager (event_descriptor) primitive

This primitive brings the operation of Integrated Event Manager. Its here the events are processed individually by processing their Event Descriptors in Event Descriptor table, and all the other modules- task manager, synchronization, communication bus and tasks are put into action as per the opcode mentioned for that event description. The pseudocode for the primitive:

```
eventManager (event_descriptor)
{
 if (event_descriptor.mode = = ENABLE)     // enabled events only
   {
     (event_descriptor.event_counter)-- ;  // event counter is decremented

    if (event_descriptor.event_counter = = 0)
      {
       if(event_descriptor.type = = FREE_RUNNING)
                                  // if event is free  running type
         {
           event_descriptor.event_counter =
                      event_descriptor.threshold ;
                                  // reload the event counter with the threshold value
         }

       else if (event_descriptor.type = = ONE_OFF )
                                   // if event is one-off type
         {
           disable(event_descriptor) ;  // disable the current event
         }

       executeOpcode(event_descriptor.opcode)
      }

   }
}
```

Execution sequence in the primitive:

1. The event is checked for its mode. If mode is enabled then goto2 else exit.
2. The event counter is decremented by one.
3. If the event counter for that event has expired (become zero) goto4 else exit
4. The event type is checked, if FREE_RUNNING type goto5 else goto6
5. Event counter is reloaded with the threshold for that event.
6. If event is of ONE_OFF type goto7 else exit.
7. Disable the current event by calling ***disable(****event descriptor****)*** primitive.
8. Exit

The private subroutine that is accessed from this primitive is:

- **executeOpcode(***event_descriptor***)**

6.3.2.2.1 executeOpcode (*event_descriptor*) – private subroutine

This subroutine decodes the opcode with each event descriptor on which it is called. And according to the opcode value the related primitives under different modules are invoked by passing appropriate arguments.

The execution sequence is very simple:

1. If opcode on the event is OPC_SIGNAL it calls signal_and_release( ) under synchronization part of software bus module. The values under the fields Semaphore and Tasks are passed as arguments to the primitive call.

2. If opcode contains OPC_SEND_MSG it calls the broadcast( ) primitive invoking the communication part of software bus. The value under the Message Index field of the event descriptor is passed as argument to the call.

3. If opcode has OPC_ENABLE_EVENT value, enableNext( ) is called.

4. If opcode has OPC_RELEASE value the release( ) primitive is called which takes the value under the **tasks** field as argument thus releasing the tasks mentioned by this tasks field.

One should keep in mind that the opcode may take a Boolean vector value with one or more bits set (as said under section). So on the event counter expiry one or more actions can be taken simultaneously.

Its pseudocode is:

```
executeOpcode(event_descriptor)
{
  if ( event_descriptor.OPC & OPC_SIGNAL)      // synchronization
    {
     signal_and_release(event_descriptor.semaphore,
                        event_descriptor.tasks);
    }

  if ( event_descriptor.OPC & OPC_SEND_MSG)    // communication
    {
     broadcast(event_descriptor.message);
    }

  if ( event_descriptor.OPC & OPC_ENABLE_EVENT)
                                    // enable next event if the event is ONE_OFF type
    {
     enableNext(event_descriptor);
    }

  if ( event_descriptor.OPC & OPC_RELEASE)     // task management
    {
     release(event_descriptor.tasks);
    }

}
```

The subroutine executeOpcode is invoked by passing the event descriptor as pointer, so all the accessing of fields also follows the pointer type access. But in primitive this is not mentioned. Thus it is as simple as this to implement the integrated Event and Timing Management.

There is one more private subroutine invoked in the executeOpcode( ) routine:

- ***enableNext (event_descriptor)***

6.3.2.2.2 enableNext(*event_descriptor*) private subroutine

When the event is of ONE_OFF type on the expiry of the event counter, that event which is next to be started has to be enabled. This is the subroutine that enables the event whose index is mentioned under the field of 'nextEvent' in the event descriptor for that event. It just switches the mode of the event mentioned under next event from DISABLE to ENABLE. Its pseudocode is:

```
enableNext(event_descriptor)
{
  event_descriptor_table[event_descriptor.nextEvent].mode = ENABLE;
}
```

6.3.2.3 enable(EventDescriptor) primitive

This primitive is used to disable an event. It just sets the mode of the passed EventDescriptor to ENABLE. Its pseudocode is:

```
enable(event_descriptor)
{
  event_descriptor.mode = ENABLE;
}
```

This primitive can be called by user tasks also to enable an event.

6.3.2.4 disable(EventDescriptor) primitive

This primitive disables an event whose event descriptor is passed to it. It just changes the value under mode field in the event descriptor to DISABLE. The pseudocode:

```
disable(event_descriptor)
{
  event_descriptor.mode = DISABLE;
}
```

If an event is of type ONE_OFF, on the expiry of its event counter the associated next event has to be enabled while the current event has to be disabled. Calling this primitive disables the current event. *This can be called from the application tasks also*. In some situations a task needs to disable an event, it is made possible by this primitive.

6.4 Events of 'ONE_OFF' type: their significance

As explained under section 6.2.2, by establishing flags for corresponding to 10 milliseconds, 1 second, 1 minute and 1 hour we achieve event counters with same resolution. But say if we need a time that is not completely of one resolution say 20 minutes 30 seconds and 10 milliseconds. To achieve this kind of time intervals with the events, we define three events with different granularities- one with minutes, one with seconds and one with 10 milliseconds. All these events are defined as ON-OFF type. Further only one of them is enabled. When this enabled event's event counter expires, this current event is disabled while the one (of the remaining two events) is enabled. When this event expires, it is disabled while the last is enabled. On the expiry of this event's counter the necessary action(s) are invoked by the IEM. And this event is disabled while the first is enabled.

This entire thing is made possible by defining the three chain events event descriptors as explained. Say these three events are having their indices 3,5,9 in Event Descriptor Table. Therefore event_3 is having event descriptor event_descriptor_table[3], event_5 has event_descriptor_table[5] and event_9 is controlled by event_descriptor_table[9].

DETAILS:

1.Say event_3 has a 10 millisecond, event_5 has 1 second and event_9 has 1minute resolution. So thresholds for each of the event are 10, 30 and 20 respectively.

2. While event_9 is enabled and remaining two are disabled.

3. Event_9 on its counter expiry enables event_5, event_5 on its counter expiring enables event_3 and finally on event_3's counter expiration event_9 is to be enabled.

4. At the same time of enabling next event the current events are disabled

5. So on the expiry of event_3 counter action is to be taken, say task_2( ) has to be released. While on the expiry of event_5 and event_9 we need to enable next event mentioned under the next event field.

The above-specified details are achieved by the following event descriptor configuration. The most important thing is how the nextEvent field links the related ONE_OFF events. Notice this field carefully.

TABLE VI

	Mode	Type	Thres	Ev cntr	Opcode	Semaphores	Tasks	MessageIndex	Next Event
0									
1									
2									
3	DISABLED	ONE_OFF	10	10	OPC_RELEASE | OPC_ENABLE_NEXT	-	0x02	-	9
4									
5	DISABLED	ONE_OFF	30	30	OPC_ENABLE_NEXT	-	-	-	3
6									
7									
8									
9	ENABLED	ONE_OFF	20	20	OPC_ENABLE_NEXT	-	-	-	5
..	…	..	…	…	…	…	…	…	…
..	MAX	…	…	…	…	…	…	…	…

End of Chapter6

Chapter 7. Conclusion

The kernel implementation was done in five phases and resulted in five versions of the **HARTEX$_\mu$** starting with version 0.0 to 0.5. The development was carried out in a cycle that coincided with spiral software development cycle. During each phase a new module is added into the kernel.

First task manager module is implemented and the most important context switching and preemption is tested in detail. Then development continued with other modules in the order: resource manager (ver 0.1), Integrated event manager (ver 0.2), synchronization part of software bus module (ver 0.4) and finally communication part of software bus module (ver 0.5). After each version the modules are tested on an individual basis and integrated basis for their respective functionalities. All the modules met the individual requirements which resulted in **HARTEX$_\mu$** .

Both periodic task and aperiodic tasks were released by the integrated event management and tested. The main concerns behind this kernel development were fast, powerful and less memory overhead. **HARTEX$_\mu$** is developed in 'C' employing GNU GCC compiler which is a free ware. HARTEX$_\mu$ when compiled with GNU GCC reported that total code size was less than 3 KB size while the memory used both by user tasks and kernel was less than 200 bytes. There was no apparent jitter when tested on the oscilloscope, which is as expected following the novel techniques employing Boolean vectors.

Future work

Kernels have a great potency but it needs to be configured as per the application software. The future work on this kernel might be to develop some monitoring support, a configuration tool using which the kernel can be configured easily and exactly as per user tasks.

Chapter 8. References

[1]. **Christo Angelov**
Updated lecture notes of SIS courses, Semester2 MSc Mechatronics, University of Southern Denmark – Mads Clausen Institute for product Innovation.

[2]. **Software group**
Design requirements and specifications of a Real-time kernel for H8/3002 microcontrollers, Danfoss A/S, Nordborg.

[3] **OSEK/VDX steering committee**
OSEK/VDX RTOS specifications for automotive applications, Version 2.2.1, January 16th, 2003.

[4] **Andreas Engberg, Anders Pettersson**
Asterix: A prototype of a small-sized real-time kernel, Malardalen University, MRTC, SWEDEN.

[5] **Christo Angelov**
Design specification of HARTEX $_{AVR}$ - A distributed hard real-time kernel for AVR microcontroller-based embedded applications, University of Southern Denmark – Mads Clausen Institute for product Innovation, Software Engineering Group
.

[6] **Jane W.S. Liu**
Real-Time Systems, section 8.6, p300-301, Prentice Hall, 2000.

[7] **Krzysztof Sierszecki**
Stack-Based Ceiling Priority Protocol Implementations, rev 0.2 , 2003, University of Southern Denmark – Mads Clausen Institute for product Innovation, Software Engineering Group

[8] **Krzysztof Sierszecki**
Stack management algorithm for AVR microcontrollers and implementation with GNUGCC compiler

Appendix

C- source code of HARTEX$_\mu$ kernel, specific to GNU GCC compiler.

1. Main function

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#include "../kernel_0_5/TaskManager.h"
#include "../kernel_0_5/ResourceManager.h"
#include "../kernel_0_5/IntegratedEventManager.h"
#include "../kernel_0_5/SynchronizationBus.h"
#include "../kernel_0_5/CommunicationBus.h"

#include "../kernel_0_5/include/typedef.h"
#include "../kernel_0_5/include/global.h"

#include "task.h"
#include "events.h"


SIGNAL(SIG_INTERRUPT7)
{
   eventManager(EXT_EVENT_7);
}

SIGNAL(SIG_INTERRUPT6)
{
   release(TASK6V);
}

/*also write here the external interrupts that are linked with the */
/*events in the event descriptor table */

/*this interrupt should occurs every 10 milli seconds*/

SIGNAL (SIG_OUTPUT_COMPARE1A)
{
   static FLAG msec_FLAG = CLEAR;
   static FLAG sec FLAG  = CLEAR;
   static FLAG min_FLAG  = CLEAR;
   static FLAG hr FLAG   = CLEAR;
   static UBYTE msec = 0;
   static UBYTE sec  = 0;
   static UBYTE min  = 0;

   msec FLAG = RAISE;                  /*raise the 10msec FLAG*/
   msec++;
   if(msec == 100){
      sec FLAG = RAISE;              /*raise the sec FLAG*/
      sec++;
      msec = 0 ;          /*reset the milliSecond counter*/
   }

   if(sec == 60){
      min_FLAG = RAISE;                /*raise the min_FLAG*/
```

```
      min++;
      sec = 0 ;                          /*reset the second counter*/
   }

   if(min == 60){
      hr FLAG = RAISE;                   /*raise the min FLAG*/
      min = 0 ;                          /*reset the minute counter*/
   }

  /*  EVENT MANGER INVOCATION*/
   if (msec FLAG == RAISE)
         sweepTable( 10 mS EVENT Table, MAX 10mS EVENT);

   if (sec FLAG  == RAISE)
         sweepTable(SEC_EVENT_Table, MAX_SEC_EVENT);

   if (min_FLAG  == RAISE)
         sweepTable(MIN_EVENT_Table, MAX_MIN_EVENT);

   if (hr_FLAG   == RAISE)
         sweepTable(HR EVENT Table, MAX HR EVENT);

   /*clearFlags*/

   msec_FLAG = CLEAR;
   sec FLAG  = CLEAR;
   min_FLAG  = CLEAR;
   hr FLAG   = CLEAR;
}


void hd_init(void)
{
   UWORD ocr = 40000;
   DDRA  = 0xFF;                         /* 0-7 output */

   DDRB  = 0xFF;                         /* 0-7 output */
   PORTB = 0xFF;
   DDRD  = 0x00;                         /* 0-7 input */

   EIMSK |= (1<<INT7) | (1<<INT6);/*enable 6, 7 external interrupts */
   EICR  |= (1<<ISC71) | (1<<ISC61);
               /* interrupt request for INT7 ,INT6 on falling edge */

   OCR1AH = (UBYTE)((ocr >> 8)& 0xFF);
   OCR1AL = (UBYTE)(ocr & 0x00FF);

   TCCR1B = (1<<CTC1)|(0<<CS12)|(0<<CS11)|(1<<CS10);
                            /* prescale factor 1 and counter cleared*/

   TIMSK  |= (1<<OCIE1A);
           /* Enable Timer1 Output CompareMatch Interrupt*/
}

int main(void)
{
   hd init();
   kernelStartUp();
   return 0;
}
```

2. task.c

```
#include <avr/io.h>
#include <avr/interrupt.h>

#include "../kernel_0_5/TaskManager.h"
#include "../kernel_0_4/ResourceManager.h"

#include "../kernel_0_5/include/typedef.h"
#include "../kernel_0_5/include/global.h"
#include "../kernel_0_5/SynchronizationBus.h"
#include "../kernel_0_5/CommunicationBus.h"

#include "task.h"
#include "message.h"
#include "resource.h"
#include "semaphore.h"

TCB tdef TCB[MAX TASK] = {
   {
      (void*)0,
   },
   {
      (void*)0,
   },
   {
      (void*)&msgDestBuf2,
   },
   {
      (void*)0,
   },
   {
      (void*)0,
   },
   {
      (void*)&msgDestBuf5,
   },
   {
      (void*)&msgDestBuf6,
   },
   {
      (void*)0,
   }
};

void (*start_task[MAX_TASK])(void) =
{ task0, task1,task2,task3,task4,task5,task6,task7};

void delay(ULONG delay)
{
   while( delay--) {
      asm volatile ( "nop"::);
   }
}

void task0(void)
{
   sei();
   PORTB ^= 0x01;
   taskExit();
```

```
}

void task1(void)
{
   sei();
   PORTB ^= 0x02;
   taskExit();
}

void task2(void)
{
   sei();
   PORTB ^= 0x04;
   taskExit();
}

void task3(void)
{
   sei();
   PORTB ^= 0x08;
   taskExit();
}

void task4(void)
{
   sei();
   PORTB ^= 0x10;
   taskExit();
}

void task5(void)
{
   sei();
   PORTB ^= 0x20;
   struct pressure s *tempPtr = getMsgSourceBuffer(PRESSURE);
   tempPtr->fieldA = 0xA5;
   broadcast(PRESSURE);
   taskExit();
}

void task6(void)
{
   sei();
   PORTB ^= 0x40;
   if ( receive(PRESSURE)) {
      union msgDestBuf6_s *tempPtr = getMsgDestinationBuffer();
      PORTB = tempPtr->pressure.fieldA;
      delay(100000);
   }
   taskExit();
}

void task7(void)
{
   sei();
   PORTB ^= 0x80;
   taskExit();
}
```

3. task.h

```
#ifndef __TASK_H__
#define   TASK H

#include "message.h"

#define TASK0 0
#define TASK1 1
#define TASK2 2
#define TASK3 3
#define TASK4 4
#define TASK5 5
#define TASK6 6
#define TASK7 7

#define TASK0V (TVEC)bit_vect[TASK0]
#define TASK1V (TVEC)bit vect[TASK1]
#define TASK2V (TVEC)bit vect[TASK2]
#define TASK3V (TVEC)bit_vect[TASK3]
#define TASK4V (TVEC)bit vect[TASK4]
#define TASK5V (TVEC)bit_vect[TASK5]
#define TASK6V (TVEC)bit_vect[TASK6]
#define TASK7V (TVEC)bit vect[TASK7]

#define MAX TASK 8

void task0(void);
void task1(void);
void task2(void);
void task3(void);
void task4(void);
void task5(void);
void task6(void);
void task7(void);

union msgDestBuf6_s {
   struct temperature s temperature;
   struct pressure_s    pressure;
} msgDestBuf6;

union msgDestBuf5_s {
   struct temperature s temperature;
} msgDestBuf5;

union msgDestBuf2_s {
   struct pressure_s pressure;
} msgDestBuf2;

#endif
```

4. Events.c

```
#include "events.h"
#include "message.h"

/* typedef struct {
      UBYTE mode;
      UBYTE type;
      ULONG threshold;
      ULONG event_counter;
      UBYTE OPC;
      Semaphore semaphore;
      TVEC tasks;
      MsgIndex message;
      UBYTE nextEvent;
} ED tdef;
*/


/* initialize all fields*/
ED tdef ED Table[MAX EVENTS] = {
   {
   ENABLE, FREE_RUNNING, 1, 1,OPC_RELEASE,  NA, 0x01,           NA, NA
   },
   {
   ENABLE, FREE RUNNING, 1, 1,OPC SEND MSG, NA,   NA,  TEMPERATURE, NA
   },
};

/*fill the approproate indeces in the 10 milli-Second Events Table*/
IndexTable  10 mS EVENT Table[MAX 10mS EVENT] = {
};
/*fill the approproate indeces in the 1 Second Events Table*/
IndexTable SEC_EVENT_Table[MAX_SEC_EVENT]    = { 0
};
/*fill the approproate indeces in the 1 Minute Events Table*/
IndexTable MIN_EVENT_Table[MAX_MIN_EVENT]    = {
};
/*fill the approproate indeces in the 1 Hour Events Table*/
IndexTable HR EVENT Table[MAX HR EVENT]      = {
};

IndexTable EXT EVENT Table[MAX EXT EVENT]    = { 1
};
```

5. Events.h

```
#ifndef __EVENT_DESCRIPTORS_H__
#define   EVENT DESCRIPTORS H

#include "../kernel_0_5/IntegratedEventManager.h"

#include "../kernel_0_5/include/typedef.h"

#include "task.h"
#include "semaphore.h"

#define NA    0

#define MAX_10mS_EVENT 0
#define MAX SEC EVENT  1
#define MAX_MIN_EVENT  0
#define MAX HR EVENT   0

#define MAX_EXT_EVENT  1

#define MAX_EVENTS (MAX_10mS_EVENT + MAX_SEC_EVENT + MAX_MIN_EVENT +
MAX_HR_EVENT + MAX_EXT_EVENT)

#define EXT_EVENT_7  &ED_Table[1]

extern ED_tdef ED_Table[];

extern IndexTable _10_mS_EVENT_Table[];
extern IndexTable SEC_EVENT_Table   [];
extern IndexTable MIN EVENT Table   [];
extern IndexTable HR_EVENT_Table    [];
extern IndexTable EXT EVENT Table   [];

#endif
```

6. resources.h

```
#ifndef __RESOURCE_H__
#define   RESOURCE H

#include "../kernel_0_5/include/typedef.h"

#define RESOURCE0 (Resource)0
#define RESOURCE1 (Resource)1

#define MAX RESOURCE 2

RCB tdef RCB[MAX RESOURCE] = {
   {
      0
   },
   {
      1
   }
};

UBYTE PI_Stack[MAX_RESOURCE];

#endif
```

7. semaphore.c

```
#include "semaphore.h"

SCB_tdef SCB[MAX_SYNCH_SEMAPHORE] = {
    {
      0x00,0x00
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00  /*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    },
    {
      0x00,0x00/*as of now the TVEC is 8 bit one */
    }
};
```

8. semaphore.h

```
#ifndef __SEMAPHORES_H__
#define   SEMAPHORES_H

#include "../kernel_0_5/include/typedef.h"

/*number of semaphores for event notification in sychronizing */

#define MAX_SYNCH_SEMAPHORE 16

#define SEM0  (Semaphore)0
#define SEM1  (Semaphore)1
#define SEM2  (Semaphore)2
#define SEM3  (Semaphore)3
#define SEM4  (Semaphore)4
#define SEM5  (Semaphore)5
#define SEM6  (Semaphore)6
#define SEM7  (Semaphore)7
#define SEM8  (Semaphore)8
#define SEM9  (Semaphore)9
#define SEM10 (Semaphore)10
#define SEM11 (Semaphore)11
#define SEM12 (Semaphore)12
#define SEM13 (Semaphore)13
#define SEM14 (Semaphore)14
#define SEM15 (Semaphore)15


#endif
```

9. message.c

```
#include "message.h"

MCB_tdef MCB[MAX_MESSAGE] = {
   {
      0x60,
      (void*)&sourceBuf0,
      sizeof(sourceBuf0)
   },
   {
      0x42,
      (void*)&sourceBuf1,
      sizeof(sourceBuf1)
   },
};

MsgSCB tdef MsgSCB[MAX MESSAGE] = {
   {
      0,
      0x60
   },
   {
      0,
      0x42
   },
};
```

10. message.h

```
#ifndef   MESSAGE H
#define __MESSAGE_H__

#include "../kernel_0_5/include/typedef.h"

#define TEMPERATURE (MsgIndex)0
#define PRESSURE    (MsgIndex)1

#define MAX_MESSAGE 2

struct temperature_s {
   UBYTE fieldA;
   UBYTE fieldB;
} sourceBuf0;

struct pressure_s {
   ULONG fieldA;
} sourceBuf1;

#endif
```

11. TaskManager.c

```
#include <avr/interrupt.h>

#include "TaskManager.h"
#include "TaskManager .h"

#include "include/global.h"

void preempt(void)
{
   TVEC HP;

   ENTER_CRITICAL;
  if (ATV) {             // Assumption: ATV is non zero value
   HP = find_msb(ATV & BTV);
   if (RT != NO TASK) {          // there is running task
      if (HP > RT) {
         STORE_CONTEXT;
         STORE RT;
         RT = HP;

         (*start task[RT])();    // just let it run

                 LOAD RT;
         LOAD_CONTEXT;
      }
   }
   else {
      RT = HP;
      (*start_task[RT])();       // just let HP run
   }
  }
 }
   EXIT_CRITICAL;
}

void schedule(void)
{
   ENTER_CRITICAL;
   if (ATV) {
   RT = find msb(ATV);

   (*start task[RT])();                    // just let it run

   }
   EXIT CRITICAL;
}

void release(TVEC tasks)
{
   ENTER_CRITICAL;
   ATV |= (TVEC)tasks;
   preempt();
   EXIT_CRITICAL;
}
```

```
void taskExit(void)
{
   cli();  // disable interrupts - necessary !
   ATV &= (TVEC)~bit vect[RT];
   RT = NO_TASK;      // there is no task running in the system
}

void kernelStartUp(void)
{
   sei();                                /* enable interrupts */

   ATV = 0;
   BTV = (TVEC)~0;
   RT  = NO TASK;

   while (TRUE) {
      while (ATV) {   // schedule all active tasks
         schedule();
      }
   }
}

UBYTE find_msb( TVEC vector)     // fast, time constant routine
{                     // Assumption: vector is non zero value !!!
   if ( vector > 0x0F) {         // TVEC is UBYTE - unsigned 8 bit
      if ( vector > 0x3F) {
              if ( vector > 0x7F) {
            return 7;
         }
         else {
            return 6;
         }
      }
      else {
              if ( vector > 0x1F) {
            return 5;
         }
         else {
            return 4;
         }
      }
   }
   else {
      if ( vector > 0x03) {
         if ( vector > 0x07) {
            return 3;
         }
         else {
            return 2;
         }
      }
      else {
              if ( vector > 0x01) {
            return 1;
         }
         else {
            return 0;
         }
      }
   }
}
```

12.TaskManager.h

```
#ifndef __TASK_MANAGER_H__
#define __TASK_MANAGER_H__

#include "include/typedef.h"


void kernelStartUp(void);

void release(TVEC tasks);
void taskExit(void);
void preempt(void);

#endif
```

13. ResourceManager.c

```
#include <avr/interrupt.h>

#include "TaskManager.h"
#include "ResourceManager.h"
#include "ResourceManager_.h"

#include "include/typedef.h"
#include "include/global.h"

void lock(RESOURCE resource)
{
   ENTER_CRITICAL;
   if (RCB[resource].ceiling >= PI) {
      pushPiStack(PI);
      PI = RCB[resource].ceiling;
      BTV = (TVEC)~(BTV | (TVEC)PI Table[PI+1]);
   }
   EXIT CRITICAL;
}

void unlock(RESOURCE resource)
{
   ENTER CRITICAL;
   if (RCB[resource].ceiling == PI) {
      popPiStack( &PI);
      BTV = (TVEC)~(BTV & (TVEC)PI_Table[PI+1]);
      preempt();
   }
   EXIT_CRITICAL;
}

void pushPiStack(SBYTE pi)
{
   ENTER_CRITICAL;
   PI Stack[top++]= pi;
   EXIT_CRITICAL;
}

void popPiStack(SBYTE *pi)
{
   ENTER_CRITICAL;
   *pi = PI Stack[--top];
   EXIT_CRITICAL;
}
```

14. ResourceManager.h

```
#ifndef __RESOURCE_MANAGER_H__
#define   RESOURCE_MANAGER_H

#include "include/typedef.h"

void lock  (RESOURCE resource);
void unlock(RESOURCE resource);

#endif
```

15. ResourceManager_.h

```
#ifndef __RESOURCE_MANAGER__H__
#define   RESOURCE MANAGER  H

#include "include/typedef.h"

extern TVEC BTV;

SBYTE  PI = -1;
UBYTE  top = 0;                          /* top of stack index*/

extern RCB tdef RCB[];

extern UBYTE PI_Stack[];

UWORD PI_Table[17] = { 0x0000, 0x0001, 0x0003, 0x0007,
                       0x000F, 0x001F, 0x003F, 0x007F,
                       0x00FF, 0x01FF, 0x03FF, 0x07FF,
                       0x0FFF, 0x1FFF, 0x3FFF, 0x7FFF,0xFFFF };

void pushPiStack(SBYTE pi);
void popPiStack (SBYTE *pi);

#endif
```

16. IntegratedEventManager.c

```
#include <avr/interrupt.h>

#include "IntegratedEventManager.h"
#include "IntegratedEventManager_.h"
#include "TaskManager.h"
#include "SynchronizationBus.h"
#include "CommunicationBus.h"
#include "include/global.h"

void sweepTable(IndexTable *index_table, UBYTE length)
{
   int i;

   for (i = 0; i < length; i++) {
      eventManager( &ED_Table[index_table[i]]);
   }
}

void eventManager(ED_tdef *EventDescriptor)
{
   if(EventDescriptor->mode == ENABLE){/* If the event is enabled*/

      (EventDescriptor->event_counter)--;
                                    /* decrement the event counter*/

      if(EventDescriptor->event_counter == 0){
         executeOPC(EventDescriptor);
         EventDescriptor->event_counter = EventDescriptor->threshold;
         if (EventDescriptor->type == ONE OFF) {
                           /*if event is one-off type*/
            disable(EventDescriptor);
         }
      }
   }
}

void executeOPC(ED_tdef *EventDescriptor)
{
   if ( EventDescriptor->OPC & OPC_SIGNAL) {
      signal_and_release(EventDescriptor->semaphore,
                         EventDescriptor->tasks);
   }
   if ( EventDescriptor->OPC & OPC SEND MSG) {
      broadcast(EventDescriptor->message);
   }
   if ( EventDescriptor->OPC & OPC ENABLE EVENT) {
      enableNext(EventDescriptor);
   }
   if ( EventDescriptor->OPC & OPC_RELEASE) {
      release(EventDescriptor->tasks);
   }
}


/* disables the event*/
void disable(ED_tdef *EventDescriptor)
{
   EventDescriptor->mode = DISABLE;
```

```
}

/* enables that event to be started after the current event*/
void enableNext(ED_tdef *EventDescriptor)
{
   ED Table[EventDescriptor->nextEvent].mode = ENABLE;
}

/*enables the event*/
void enable(ED_tdef *EventDescriptor)
{
   EventDescriptor->mode = ENABLE;
}
```

17. IntegratedEventManager.h

```
#ifndef __INTEGRATED_EVENT_MANAGER_H__
#define   INTEGRATED_EVENT_MANAGER_H

#include "include/typedef.h"

#define OPC_SIGNAL            0x01
#define OPC_RELEASE           0x02
#define OPC_SEND_MSG          0x04
#define OPC_ENABLE_EVENT      0x08

#define DISABLE      0
#define ENABLE       1

#define ONE_OFF      1
#define FREE_RUNNING 2

void sweepTable  (IndexTable *index_table, UBYTE length);
void eventManager(ED_tdef *EventDescriptor);

void enable(ED_tdef *EventDescriptor);

#endif
```

18. IntegratedEventManager_.h

```
#ifndef __INTEGRATED_EVENT_MANAGER__H__
#define   INTEGRATED_EVENT_MANAGER_ H

#include "include/typedef.h"

#define  DISABLE 0
#define  ENABLE  1

extern ED_tdef    ED_Table         [];
extern IndexTable _10_mS_EVENT_Table[];
extern IndexTable SEC_EVENT_Table  [];
extern IndexTable MIN_EVENT_Table  [];
extern IndexTable HR_EVENT_Table   [];

extern IndexTable EXT_EVENT_Table  [];

void executeOPC(ED_tdef *EventDescriptor);
void disable   (ED_tdef *EventDescriptor);
void enableNext(ED_tdef *EventDescriptor);


#endif
```

19. SynchronizationBus.c

```
#include <avr/interrupt.h>

#include "SynchronizationBus.h"
#include "SynchronizationBus .h"

#include "TaskManager .h"
#include "TaskManager.h"

#include "include/global.h"

void signal_and_release(Semaphore semaphore, TVEC tasks)
{
   ENTER CRITICAL;
   SCB[semaphore].flags |= tasks;
   release(SCB[semaphore].tasks);
   EXIT CRITICAL;
}

UBYTE test_and_reset(Semaphore semaphore)
{
   ENTER CRITICAL;
   if (SCB[semaphore].flags & bit_vect[RT]) {
       SCB[semaphore].flags &= ~bit vect[RT];
                        /*clear the flag corresponding to the RT*/
      EXIT_CRITICAL;
      return TRUE;
   }
   else {
      EXIT_CRITICAL;
      return FALSE;
   }
}
```

20. SynchronizationBus.h

```
#ifndef   SYNCHRONIZATION BUS H
#define __SYNCHRONIZATION_BUS_H__

#include "include/typedef.h"

void signal_and_release(Semaphore semaphore, TVEC tasks);
UBYTE    test_and_reset(Semaphore semaphore);

#endif
```

21. SynchronizationBus_.h

```
#ifndef __SYNCHRONIZATION_BUS__H__
#define   SYNCHRONIZATION_BUS__H

#include "include/typedef.h"

extern SCB_tdef SCB[];
extern volatile UBYTE RT;

#endif
```

22. CommunicationBus.c

```
#include <avr/interrupt.h>

#include "CommunicationBus.h"
#include "CommunicationBus .h"

#include "TaskManager.h"

#include "include/typedef.h"
#include "include/global.h"

void msg_signal_and_release(Semaphore semaphore, TVEC tasks)
{
   ENTER CRITICAL;
   MsgSCB[semaphore].flags |= tasks;
   release(MsgSCB[semaphore].tasks);
   EXIT CRITICAL;
}

UBYTE msg_test_and_reset(Semaphore semaphore)
{
   ENTER CRITICAL;
   if (MsgSCB[semaphore].flags & bit_vect[RT]) {
       MsgSCB[semaphore].flags &= ~bit vect[RT];
                                        /*clear the flag corresponding
to the RT*/
      EXIT_CRITICAL;
      return TRUE;
   }
   else {
      EXIT CRITICAL;
      return FALSE;
   }
}


void broadcast(MsgIndex message)
{
   ENTER_CRITICAL;
   msg_signal_and_release(message,MCB[message].receivers);
   EXIT CRITICAL;
}

UBYTE receive(MsgIndex message)
{
   ENTER CRITICAL;
   if (msg_test_and_reset(message)){
      copy message(MCB[message].sourceBuffer,
                   TCB[RT].msgLocalBuffer, MCB[message].length);
      EXIT_CRITICAL;
      return TRUE;
   }
   else{
      EXIT_CRITICAL;
      return FALSE;
   }

}
```

```
void* getMsgSourceBuffer(MsgIndex message)
{
   return(MCB[message].sourceBuffer);
}

void* getMsgDestinationBuffer(void)
{
   return (TCB[RT].msgLocalBuffer);
}

UBYTE getMsgLength(MsgIndex message)
{
   return (MCB[message].length);
}

void copy_message( UBYTE *source, UBYTE *destination, UBYTE length)
{
   while(length--) {
      destination[length] = source[length];
   }
}
```

23. CommunicationBus.h

```
#ifndef __COMMUNICATION_BUS_H__
#define   COMMUNICATION_BUS_H

#include "include/typedef.h"

void  broadcast(MsgIndex message);
UBYTE receive  (MsgIndex message);

void* getMsgSourceBuffer     (MsgIndex message);
void* getMsgDestinationBuffer(void);
UBYTE getMsgLength           (MsgIndex message);

#endif
```

24. CommunicationBus_.h

```
#ifndef   COMMUNICATION BUS  H
#define __COMMUNICATION_BUS__H__

#include "include/typedef.h"

extern TCB tdef       TCB[];

extern MCB tdef       MCB[];
extern MsgSCB_tdef    MsgSCB[];
extern volatile UBYTE RT;

void  msg_signal_and_release(Semaphore semaphore, TVEC tasks);
UBYTE msg_test_and_reset    (Semaphore semaphore);

void copy_message( UBYTE *source, UBYTE *destination, UBYTE length);

#endif
```