CHAPTER 3

RTOS BASICS AND APPLICATIONS IN AUTOMATION ELECTRONICS

3.1 INTRODUCTION

This chapter describes the basics of realtime systems and a behavioral comparison of non-realtime and realtime systems in automation electronics for airbag application. Some of the important characteristics of an RTOS have been compared to those of non realtime systems. A non realtime RTOS is used by average computer users for day-to-day activities like checking e-mail, typing documents, listening to music, watching videos, etc.

3.2 RTOS BASICS

In realtime multitasking system, a lot of events and multiple concurrent tasks run at the same time. Therefore, to meet the system response time requirement, it should be ensured that each task can be executed within the required time frame. Multitasking is a technique used for enabling multiple tasks to share a single processor. Realtime Operating Systems (RTOS) are specially designed to meet rigorous time constraints. In several situations RTOS are present in embedded systems, and most of the time they are not noticed by the users. A good example of this situation may be observed in the automobile industry, where it is estimated that 33% of the semiconductors used in a car are microcontrollers, being common for a car to have dozens of microcontrollers. Seeking for improvements on its products and development time reduction, the car manufacturers have been adopting RTOS to control the software that runs in the vehicles.

Realtime operating system is a subtype of operating system and is mainly responsible for the control and management of variety of hardware resources to enable the hardware system to become available, and provides upper level applications with rich system calls. It schedules execution in a timely manner, manages system resources and provides a consistent foundation for developing application code. The services include:
➢ Basic OS functions
➢ Priority allocation
➢ Task management
➢ Task predictability
➢ Memory management
➢ Memory allocation
➢ Scheduling and interrupt latency control functions
➢ Timer functions and IPC synchronization functions.

To increase the realtime system performance, control algorithm and preemptive scheduling algorithms are implemented efficiently for managing multiple critical tasks. In RTOS’s, the following are important requirements that an OS must meet to be considered as an RTOS in the contemporary sense.

i. The operating system must be multithreaded and preemptive, e.g. handling multiple threads and is able to preempt tasks if necessary.

ii. The OS must support priority of tasks and threads.

iii. A system of priority inheritance must exist. Priority inheritance is a mechanism to ensure that lower priority tasks cannot obstruct the execution of higher priority tasks.

iv. The OS must support various types of thread/task synchronization mechanisms.

v. For predictable response:
   a. The time for every system function call to execute should be predictable and independent of the number of objects in the system.
   b. Non-preemptable portions of kernel functions necessary for inter process synchronization and communication are highly optimized, short and deterministic.
   c. Non-preemptable portions of the interrupt handler routines are kept small and deterministic.
   d. Interrupt handlers are scheduled and executed at appropriate priority.
   e. The maximum time during which interrupts are masked by the OS and by device drivers must be known.
f. The maximum time that device drivers use to process an interrupt, and specific IRQ information relating to those device drivers, must be known.

g. The interrupt latency (the time from interrupt to task run) must be predictable and compatible with application requirements.

vi. For fast response:

a. Run-time overhead is decreased by reducing the unnecessary context switch.

b. Important timings such as context switch time interrupt latency and semaphore get/release latency must be minimum.

3.2.1 FACTORS AFFECTING REALTIME CHARACTERISTICS OF OPERATING SYSTEM

There are varieties of factors impacting a system’s realtime. Among these factors, operating system and its own factors play crucial roles including task management, task scheduling, context switching time, the time of interrupt handle, and so on.

i) SCHEDULING OF TASKS

It is crucial for the realime operating system to adapt preemptive scheduling kernel, which is based on task priority. The μC/OS-II operating system uses this method to implement its scheduling. An operating system with non preemptive scheduling mechanism must have no strict realtime characteristic. Preemptive scheduling provides a good foundation for realtime system. In order to maximize the efficiency of scheduling systems, the operating system should run with certain realtime scheduling algorithm. There are some common realtime scheduling algorithms, such as the Liu and Layland Rate-Monotonic (RM) scheduling algorithm and the earliest deadline priority (EDF) algorithm. The RM scheduling algorithm is a type of static scheduling algorithm, in which the priority of tasks are determined by the length of the cycle of task, and the shorter cycle of task has a higher priority. The EDF algorithm is one of the most popular dynamic priority scheduling algorithms that define priority of tasks according to their deadlines. Clearly, good task scheduling algorithms can improve the operating system’s realtime characteristics. However, it
also consumes a certain degree of system resources. Thus, time complexity of scheduling algorithm, in turn, has an impact on the realtime characteristic.

ii) CONTEXT SWITCHING TIME

In a multi-tasking system, context switch refers to a series of operations that decides the right of using CPU for transferring from one task (which is running) to another (ready for running). In preemptive scheduling systems, there are a lot of events that can cause context switches, such as external interrupt, or releasing of resource which high priority tasks wait for. The linkages of tasks in an operating system are achieved by the process control block (PCB) data structure. When context switches occur, the former task information is saved to the corresponding PCB or stack PCB specified. The new task fetches the original information from corresponding PCB. The time switching consumed depends on the processor architecture, because different processors need to preserve and restore different number of registers; some processors have a single special instruction which is able to achieve all the register’s preserve and restore job; some processors provide a number of registers group, the context switching only need to change the register group pointer. Operating system data structures will also affect the efficiency of context switch.

iii) TIME OF KERNEL PROHIBITING INTERRUPT

To ensure the atomicity of operation to some critical resource, the operating system kernel has to prohibit all of interrupts sometimes. Interrupt will break the sequence of instructions, and may cause damage of data. Prohibiting an interrupt, always delay the response of request and context switching. In order to improve realtime performance of operating system, noncritical operations can be inserted between the critical operations.

iv) TASK MANAGEMENT

- Creating an RT task and getting memory allocation without delay is very difficult since memory has to be allotted and a lot of data structures, code segment must be copied/initialized.
➢ The memory blocks for RT tasks must be locked in main memory to avoid access latencies due to swapping.
➢ Changing runtime priorities is dangerous and it may change the runtime behaviour and predictability of the whole system.

### 3.2.2 TASK BASICS

i. A task is an independent thread of execution that can compete with other tasks for concurrent task execution time.

ii. An application program can also be said to be a program consisting of the tasks and task behaviours in various states that are controlled by OS.

iii. It is like a process or thread in an OS and it is the term used for the process in the RTOSes for the embedded systems. For example, VxWorks and µCOS-II are the RTOSes, which use the term task.

iv. It consists of executable program (codes), state of which is controlled by OS, runs when it is scheduled to run by the OS (kernel) that gives the control of the CPU on a task request (system call) or a message.

v. It is an independent process and no task can call another task. [It is unlike a C or C++function, which can call another function].

vi. It can send signal (s) or message(s) that lets another task run. The OS can only block a running task and let another task gain access of CPU to run the servicing codes.

vii. It runs on a CPU under the state control using a task control block and is processed concurrently.

viii. These are embedded program computational units that run on a CPU under the state control using a task control block.

ix. Tasks = Code + Data + State (context)

Task State is stored in a Task Control Block (TCB) when the task is not running on the processor. The RTOS effectively multiplexes the CPU among the tasks as shown in figure 3.1.
Whenever a task is switching its state, its execution context (represented by the contents of the program counter, stack and registers) is saved by the operating system into a special data structure called a task control block so that the task can be resumed the next time it is scheduled. Similarly the context has to be restored from the task control block when the task state is set to running. The information related to a task stored in the TCB is shown below.

i. Synchronization Information: semaphores, pipes, mailboxes, message queues, file handles etc.

ii. Scheduling Information: priority level, relative deadline, period and state.

iii. Task Parameters: includes task type and event list.

iv. Task Context: includes the task’s program counter (PC), the CPU registers and (optionally) floating-point registers, a stack for dynamic variables and function calls, the stack pointer (SP), I/O device assignments, a delay timer, a time-slice timer and kernel control structures.

v. Address Space: the address ranges of the data and code blocks of the task loaded in memory including statically and dynamically allocated blocks.

vi. Task ID: the unique identifier for a task.

### 3.2.3 TASK STATES

A task has the following states:

i) Idle state [Not attached or not registered]

(ii) Ready State [Attached or registered]

(iii) Running state

(iv) Blocked (waiting) state

(v) Delayed for a preset period
Figure 3.2 shows different states of a task j for example. Most tasks are blocked or ready most of the time because generally only one task can run at a time per CPU. The number of items in the ready queue can vary greatly, depending on the number of tasks the system needs to perform and the type of scheduler that the system uses. On simpler non-preemptive but still multitasking systems, a task has to give up its time on the CPU to other tasks, which can cause the ready queue to have a greater number of overall tasks from the ready state to the executed state. When a task is “spawned”, either by the operating system, or another task, it is to be created, which involves loading it into the memory, creating and updating certain OS data structures such as the task Control Block, necessary for running the task within the multi-tasking environment. During such times the task is in the new state.

Once these are over, it enters the ready state where it waits. At this time it is within the view of the scheduler and is considered for execution according to the scheduling policy. A task is made to enter the running state from the ready state by the operating system dispatcher when the scheduler determines the task to be the one to be run according to its scheduling policy. While the task is running, it may execute a normal or abnormal exit according to the program logic, in which case it enters the terminated state and then removed from the view of the OS. Software or hardware interrupts may also occur while the task is running. In such a case, depending on the priority of the interrupt, the current task may be transferred to the ready state and wait for its next time allocation by the scheduler.
Figure 3.2 Task states

i. **Idle (created) state**

The task has been created and memory is allotted to its structure. However, it is not ready and is not schedulable by kernel.

ii. **Ready (Active) State**

The created task is ready and is schedulable by the kernel but not running at present as another higher priority task is scheduled to run and gets the system resources at this instance.

iii. **Running state**

In this state, execution of the codes and accessing the system resources will take place. It will run till it needs some IPC (input) or wait for an event or till it gets preempted by another higher priority task than this one.

iv. **Blocked (waiting) state**

Execution of task codes suspends after saving the needed parameters into its context. It needs some IPC (input) or it needs to wait for an event or wait for higher priority task to block for enabling running after blocking.

3.2.4 **INTER PROCESS COMMUNICATION AMONG TASKS**

A key feature of any kind of operating system is to be able to pass data between processes (tasks, threads, and interrupt service routines). Inter process communication (IPC) is used for synchronisation, mutual exclusion and data exchange of multiple tasks in various applications.
There is a great variety of applications where IPC is used, e.g. telecom-, robotic- and control systems.

IPC is used in many applications and is aimed to solve mutual exclusion, synchronisation and data exchange among co-operating processes. Due to different application needs, different IPC mechanisms have evolved. For instance, applications that often perform synchronisation need a mechanism that is optimised for that. If the mechanism also supports data exchange, it is more likely that it doesn’t provide an optimal solution to synchronisation. For applications that both need synchronisation and data exchange yet another mechanism could be the best choice. Accordingly, there are IPC mechanisms that are optimal for solving one or two, or all of the tasks an IPC mechanism is supposed to solve i.e. mutual exclusion, synchronisation and data exchange. But there are some implementation dependent attributes, which can be associated with the primitives. For example, consider an application that needs to exchange data between a group of processes in a synchronous way. The attributes here are collective and block, which are listed below together with some other attributes.

**DATA EXCHANGE ATTRIBUTES:**

i. Blocked: Synchronous communication.


iii. Partly blocked: Synchronous communication that timeouts after a specified time.

iv. Buffered: Holds data until completion.

v. Non-buffered: No buffering of data.

vi. Reliable: Reliable communication over network i.e. data will not be lost.

vii. Unreliable: Unreliable communication over network i.e. data may be lost.

viii. Collective: Communication between a group of processes i.e. broadcast or multicast Communication.

ix. Point-to-point: Communication between two processes.
SYNCHRONIZATION AND MUTUAL EXCLUSION ATTRIBUTES:

i. Blocked.
ii. Non-blocked.
iii. Partly blocked.
iv. Collective- Synchronisation between a group of processes.
v. Point-to-point- Synchronisation between two processes.
vi. Deadlock free.

CRITICAL REGIONS

i. Simultaneous use of a shared resource can lead to disastrous consequences.
ii. Resources that can only be used by one at a time are called serially reusable and the use of the resource cannot be interrupted.
iii. The code that accesses a serially reusable resource is called a critical section or critical region that cannot be interrupted by another task.
iv. Thus there must be a mechanism that prevents a collision, i.e. the simultaneous use of such a resource).

Examples:

Writing shared memory.

Accessing I/O device.

Example program showing writing into shared memory by task1 and task 2 that result into the following output.

```c
task1 ()
{
    Possible Output: I am I am Task 2 Task1
    printf ("I am Task 1");
}

Task2 ()
{
    Printf ("I am Task 2");
}
```
3.2.5 INTER PROCESS COMMUNICATION MECHANISMS

Depending on the RTOS, different mechanisms are supported for protecting critical region of codes and the important one is, semaphore. It is divided into:

i. Binary

ii. Counting

iii. Mutex

SEMAPHORES

i. It is an OS primitive for controlling access to critical regions.

ii. It is a kernel object that one or more tasks of execution can acquire or release for the purpose of synchronization or mutual exclusion.

A semaphore is like a key that allows a task to carry out some operation or to access a resource.

iii. There may be several keys for each semaphore.

iv. If the task can acquire the semaphore, it can carry out the intended operation or access the resource. Otherwise the task is blocked, if it chooses to wait for the release of the semaphore.

i) BINARY SEMAPHORE

A binary semaphore has either a value of 0 (unavailable) or 1 (available) and is shown in figure 3.3. A binary semaphore is a shared resource and any task may release it.

![Figure 3.3 Binary semaphore](image-url)
In the following program, the critical region printf (...) is protected by a binary semaphore (only values 0 and 1 are used).

```c
void task1 ()
{
    p(s);
    printf("I am Task 1");
    v(s);
}

void task2 ()
{
    p(s);
    printf("I am Task 2");
    v(s);
}
```

The output of the above program will be “I am Task 1” and “I am Task 2” respectively.

ii) COUNTING SEMAPHORE

A counting semaphore uses a count to be able to give more than one task access, 0 (unavailable) or > 0 (available) as shown in figure 3.4. The count can be bounded (initial value gives maximum number) unbounded (no maximum value). A counting semaphore is a shared resource and any task can release it.

![Figure 3.4 counting semaphore](image)

**Figure 3.4 counting semaphore**
iii) MUTUAL EXCLUSION SEMAPHORE (Mutex)

A mutex is a special binary semaphore that supports ownership and often recursive access (recursive mutex) and only the task that has locked a mutex can release it. Mutex is created in unlocked state whose state diagram is shown in figure 3.5.

![State Diagram of Mutex](image)

**Figure 3.5 Mutex**

### 3.2.6 TASK MANAGEMENT

The responsibility of task management includes the scheduling of the multiple tasks. Since scheduling is pure overhead, the RTOS implements scheduling algorithms that are less complex. The simpler the algorithm, the faster, smaller (less memory) and more predictable the task will be executed. The simplest approach to the scheduling problem is to assign (static/dynamic) priorities to all tasks. It's quite obvious as an example that the creation and deletion of tasks shouldn't be done during RT-tasks. It is actually the system designer who has to decide which tasks get which priorities, because many programmers have the urge to prioritize their tasks. In this scenario, it's important to see there's only one task which should obtain the highest priority.

One of the fundamental functions of an operating system is scheduling. There are 2 types of uni-processor operating systems in general.

- Uni-programming
- Multi-programming.
Uni-programming operating system executes only single job at a time while multiprogramming operating system is capable of executing multiple jobs concurrently. Multiprogramming is used to improve the efficiency of the micro-processor i.e., when the currently running task is blocked due to waiting for an I/O event, another task may be scheduled to run. At this moment, the issue of scheduling is involved, deciding which task should be executed by the processor, as usually more than one task may be ready for being scheduled.

Resource utilization is the basic aim of multiprogramming operating system and there are many scheduling algorithms available for multi-programming operating system that focuses on the design and development aspect of novel scheduling. For managing multiple critical tasks, µc/os-II OS is used in this work in the view of optimum utilization of resources.

3.2.7 PRIORITY LEVELS IN A TYPICAL REALTIME OPERATING SYSTEM

To be able to ensure that response to every event is generated by executing tasks within specific deadlines, it is essential to allocate the CPU and other computational resources to various tasks in accordance with their priorities. The priority of a task may be fixed or static. It may be calculated based on their computing time requirements, frequency of invocations or deadlines. However, it has been established that policies that allow task priorities to be adjusted dynamically are more efficient. The priority will depend on how quickly a task has to respond to a particular event. An event may be some activity of the process or may be the elapsing of a specified amount of time.

Advanced multi-tasking RTOSs mostly use preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with policies such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled for up to a maximum time, set by the time-slice parameter, in a Round Robin manner. Task priorities can also be set. Hundreds of priority levels are commonly available for scheduling. Specific tasks can also be indicated to be non preemptive.
3.2.8 TOTAL PRIORITY LEVELS IN µC/OS-II AND µC/OS-III

Task priorities in µc/os RTOSs may range from 1 to 63, where 63 is the lowest priority level and 1 is highest priority level. The recommended user priority levels for the particular application are in the range of 46 to 62. This avoids any conflicts with network communications.

µc/OS-II

- Preemptible priority-driven realtime scheduling.
- 64 priority levels (max 64 tasks).
- 8 reserved for µC/OS-II.
- Each task is an infinite loop.
- Deterministic execution times for most µC/OS-II functions and services.
- Nested interrupts could go up to 256.

µc/os-III

i. µc/os-III manages an unlimited number of application tasks, constrained only by a processor’s access to memory. µc/os-III also supports an unlimited number of priority levels (typically configured for between 8 and 256 different priority levels).

ii. µc/os-III allows multiple tasks to run at the same priority level. When equal priority tasks are ready-to-run, µC/OS-III runs each for a user-specified amount of time. Each task can define its own time quanta and give up its time slice if it does not require the full time quanta.

Table 3.1 Shows comparative analysis with respect to features of µc/os RTOS family.

Table 3.1 Comparative analysis on µc/os RTOSs

<table>
<thead>
<tr>
<th>Feature</th>
<th>µC/OS</th>
<th>µC/OS-II</th>
<th>µC/OS-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preemptive Multitasking</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Maximum number of tasks</td>
<td>64</td>
<td>255</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of tasks at each priority level</td>
<td>1</td>
<td>1</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Feature</td>
<td>Option 1</td>
<td>Option 2</td>
<td>Option 3</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Round robin scheduling</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Semaphores</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mutual exclusion semaphores</td>
<td>No</td>
<td>Yes</td>
<td>Yes (Nestable)</td>
</tr>
<tr>
<td>Event flags</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Message mailboxes</td>
<td>Yes</td>
<td>Yes</td>
<td>No (not needed)</td>
</tr>
<tr>
<td>Message queues</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fixed sized memory management</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal a task without requiring a semaphore</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Option to post without scheduling</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Send messages to a task without requiring a message queue</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Software timers</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task suspend/resume</td>
<td>No</td>
<td>Yes</td>
<td>Yes (Nestable)</td>
</tr>
<tr>
<td>Deadlock prevention</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Code footprint</td>
<td>3K to 8K</td>
<td>6K to 26K</td>
<td>6K to 24K</td>
</tr>
<tr>
<td>Data footprint</td>
<td>1K+</td>
<td>1K+</td>
<td>1K+</td>
</tr>
<tr>
<td>ROMable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Run-time configurable</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Compile-time configurable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pend on multiple objects</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task registers</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Built-in performance measurements</td>
<td>No</td>
<td>Limited</td>
<td>Extensive</td>
</tr>
<tr>
<td>User definable hook functions</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time stamps on posts</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Built-in kernel awareness support</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimizable scheduler in assembly language</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Catch a task that returns</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Tick handling at task level</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Source code available</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Deadlock prevention</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of services</td>
<td>~20</td>
<td>~90</td>
<td>~70</td>
</tr>
</tbody>
</table>
3.3 BASICS OF SCHEDULING

Scheduling is the method by which tasks are given access to system resources and the need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one task at a time). Scheduling is the heart of any computer system since it contains decision of giving resources between possible tasks. Sharing of computer resources between multiple tasks is also called as scheduling. The CPU is one of the primary computer resources and its scheduling is essential to an operating system’s design. Efficient resource utilization is achieved by sharing system resources amongst multiple users and system tasks. Optimum resource sharing depends on the efficient scheduling of competing users and system tasks for the processor, which renders task scheduling, an important aspect of a multiprogramming operating system. As the processor is the most important resource, task scheduling, which is called CPU scheduling, becomes all the more important in achieving the above mentioned objectives. Part of the reason for using multiprogramming is that the operating system itself is implemented as one or more tasks, so there must be a way for the operating system and application tasks to share the CPU. In choosing which algorithm to use, the properties of the various algorithms should be considered.

3.3.1 CRITERIA FOR COMPARING CPU SCHEDULING ALGORITHMS

- CPU utilization – percent of time that the CPU is busy executing a task.
- Throughput – number of tasks that are completed per time unit.
- Response time – amount of time it takes from when a request was submitted until the first response occurs (but not the time it takes to output the entire response).
- Waiting time – the amount of time before a task starts after first entering the ready queue (or the sum of the amount of time a process has spent waiting in the ready queue).
- Turnaround time – amount of time to execute a particular task from the time of submission through the time of completion.
- Fairness: make sure each process gets a fair share of the CPU.

It is desirable to
i. Maximize CPU utilization

ii. Maximize throughput

iii. Minimize turnaround time

iv. Minimize waiting time

v. Minimize response time

3.3.2 TYPES OF SCHEDULING

In realtime systems scheduling algorithms are classified into two categories: Static algorithm and Dynamic algorithm. Based on execution attributes of tasks, dynamic algorithm assigns priorities at runtime. This algorithm allows switching of priorities between tasks. In contrast with dynamic algorithm, a static algorithm assigns priorities at design time. All assigned priorities remain fixed throughout the execution of task. Figure 3.6 gives the classification of available scheduling algorithms for realtime systems.

![Figure 3.6 Types of scheduling algorithms](image)

i. **Rate Monotonic (RM)**: In RM algorithm, tasks have to be periodic in nature and deadline must be equal to its period. Tasks are scheduled according to their period. The rate of task is the inverse of its period. This algorithm implemented by assigning fixed priority to tasks, the higher its rate, higher the priority.
ii. **Deadline Monotonic (DM):** The other algorithm for scheduling all the realtime tasks based on their deadline is known as deadline monotonic. In this algorithm, priorities are decided by considering relative deadline. Tasks with shortest deadline get the highest priority. If one task will get higher priority than the other then its relative deadline must be shorter as compared to other tasks. RM and DM are identical except priorities are automatically computed from rate of task or deadline.

iii. **Least Slack Time First (LST):** It is type of dynamic algorithm which assigns priority dynamically. Tasks are scheduled according to their slack: the smaller the slack, the higher the priority. Slack is computed by using the difference between the deadline, the ready time and the run time.

iv. **Earliest Deadline First (EDF):** The most common dynamic priority scheduling algorithm for realtime systems is the EDF. Here priorities are dynamically reassigned at runtime based on the time still available for each task to reach its next deadline. Both static and dynamic systems are scheduled by EDF algorithm. A queue of tasks is maintained in ascending order of deadline and whenever a processor gets free, the head of the queue will be assigned to the processor according to EDF algorithm. When new task arrives, its deadline will be compared with the deadline of currently executing task, and in case if deadline of newly arrived task is closer to the current time, it will receive the processor and the old task will be preempted and placed back in the queue. As compared to other algorithms EDF is optimal and simple to implement, giving much better utilization of processor. This research work proposes an EDF scheduling algorithm which helps to schedule tasks dynamically, enhance the throughput, reduce the overload and also helps to decrease energy consumed in data transmission. EDF improves the system’s performance and allows a better exploitation of resources.

### 3.4 COMPARISON OF NON RTOS AND RTOS SYSTEMS

In non-RTOS implementation, task scheduling is not based on “priority” always. It is programmed to handle scheduling in such a way that it manages to achieve high throughput. Here throughput means – the total number of tasks that complete their execution per unit time. In such a case, sometimes execution of a high priority tasks will get delayed in order to serve 5 or 6 low
priority tasks. High throughput is achieved by serving 5 low priority tasks than by serving a single high priority one.

In an RTOS implementation, scheduling is always priority based. In this work, μc/os-II RTOS uses preemptive task scheduling method which is based on priority levels. Preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such a change is known as a context switch. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system. Here a high priority process gets executed over the low priority ones. All low priority process execution will get paused. A high priority process execution will get override only if a request comes from an even high priority process.

In this work, analog to digital conversion is implemented under non-RTOS and RTOS scenarios. In the former, the actual bandwidth of an ADC is characterized by its sampling rate and to a lesser extent by how it handles errors such as aliasing. In the later, sampling rate is characterized by task execution time.

3.5 BEHAVIORAL COMPARISON OF RTOS AND NON-RTOS IN AUTOMATION ELECTRONICS

In this work, an automobile airbag system, one of the most critical systems in a modern car, is considered under behavioral comparison of non realtime and realtime scenarios. An airbag should be deployed within a particular time interval (for example, within 1 second) after the sensors in the car detect a collision. A car equipped with a realtime computer system will always guarantee that the airbag will be deployed within 1 second, no matter what happens. The realtime system will put all the non-critical tasks, such as powering the stereo, switching on the air-conditioning, activating cruise control, etc. on hold and will immediately deploy the airbag as soon as it receives the signal from the sensors. In realtime systems, a difference of even 0.000001 second may be important.

If a car is equipped with a non realtime system, however, there is no guarantee that the airbag will be deployed exactly within 1 second after the sensor detects a collision. Such a non realtime system might deploy the airbag after finishing the request to activate the cruise control, which happened to come before the request to deploy the airbag. An airbag system that deploys
even 0.1 second later than the expected time is as bad as not deploying at all. Because it might already be too late to save the life of the passenger.

Moving objects have a momentum and unless an outside force acts on an object, the object will continue to move at its present speed and direction. Cars consist of several objects, including the vehicle itself, loose objects in the car and, of course, passengers. If these objects are not restrained, they will continue moving at whatever speed the car is traveling at, even if the car is stopped by a collision. Stopping an object's momentum requires force acting over a period of time. When a car crashes, the force required to stop an object is very great because the car's momentum has changed and the goal of any supplemental restraint system is to help stop the passenger while doing as little damage to him or her as possible. What an airbag wants to do is to slow the passenger's speed to zero with little or no damage. The airbag has the space between the passenger and the steering or dashboard and a fraction of a second to work with. Even that tiny amount of space and time is valuable, however, if the system can slow the passenger evenly rather than forcing an abrupt halt to his or her motion.

3.5.1 METRICS OF REAL-TIME

A number of parameters are required to quantify realtime systems. To understand these parameters, it is important to define the term “latency.”

**Latency:**

Latency is the time that elapses between a stimulus and the response to it. Using this definition in airbag example, latency in realtime systems is defined as the time elapsed between the triggering of an event (signal sent by sensors), requesting a particular task (deploying the airbag), and the actual task being executed (airbag deployed). In other words, it is the delay between an action and a response to that particular action. Following are some important metrics helping to quantify a realtime system:

a) **Interrupt Latency:**

This is the time elapsed between the generation of an interrupt and the start of the execution of the corresponding interrupt handler. Example: When a hardware device performs a task, it
generates an interrupt (an IRQ). This interrupt has the information about the task to be performed and about the interrupt handler (ISR) to be executed. The interrupt handler then performs the particular task.

b) **Scheduling Latency:**

It is the time between a wakeup (the stimulus) signaling that an event has occurred and the kernel scheduler getting an opportunity to schedule the task that is waiting for the wakeup to occur (the response). Wakeups can be caused by hardware interrupts, or by other tasks. Scheduling latency is also known as task-response latency or dispatch latency.

c) **Worst-case Latency:**

This is defined as the maximum time that can lapse before the desired event occurs. Worst-case refers to the condition when the system is under heavy load - more CPU load and I/O operations occurring that are typical for the particular system. As far as realtime is concerned, it is very important to define results not only on an average basis, but in a worst-case scenario as well. Example: The 1 second (maximum) limit discussed in the airbag example is the worst-case scenario. It means that, on average, the airbag will be deployed in, say, 0.3 seconds, but in a worst-case scenario (system under heavy load), the airbag will be deployed within 1 second.

### 3.5.2 AIRBAG CONTROL TRIGGERING

Airbag control unit (ACU) monitors various sensors including accelerometers and detects a collision. The ACU has to decide whether or not to deploy the airbag once the sensors located throughout the car report a collision. As shown in figure 3.8, a crash sensor is used to trigger a rapid expansion to trigger to protect human life from the impact of accident. If a collision is detected by sensors at time $t=t_1$, the ignition of a gas generator propellant is triggered at time $t=t_2$ to inflate a bag as shown in figure 3.7. Gas inflates the airbag with velocities of up to 320km/h. As a realtime constraint, the trigger must be within 10-20msec after the collision, i.e., $\Delta t=t_2-t_1$. Otherwise, it
will be a non real-time event wherein there is no guarantee that the airbag will be deployed exactly within 1 second after the sensor detects a collision.

**Figure 3.7 Trigger system for airbag control**

**Figure 3.8 Collision detection for airbag control**

**Airbag Function during a Collision**

- Automobile makes initial contact with metal as an example in this work at time, 0.
- Microprocessor decides to deploy airbags and signal is sent to deployment mechanism at time, 30 ms.
• Pyrotechnic device ignites, and gas enters airbag at time, 32 ms.
• Airbag is fully inflated at time, 60ms.

3.6 SUMMARY

In recent years, research on enhancement of the safety of vehicle occupant receives great attention and has become an important research area in automobile engineering. Related safety issues being studied among others include occupant classification, detection and crash severity analysis. Nowadays, most of the manufactured automobiles are equipped with standard airbag systems to improve the safety of the occupant. When being a realtime system, airbag unit in a car is saving millions of human lives. Under non realtime, it is meaningless to use this application as far as human safety is concerned.