A Real-Time Operating System (RTOS) is one that supports real-time applications and embedded systems. Real-time applications have the requirement to meet task deadlines in addition to the logical correctness of the results. A Real-Time Operating System (RTOS) is an operating system that guarantees a certain capability within a specified time constraint. The main difficulties in designing real-time systems are related to time constraints: if an action is performed too late, it is considered as a fault (with different levels of criticism). Designers need to use a solution that fully supports timing constraints and enables them to simulate early on the design process a real-time system.

This paper will explain some Real-Time Operating Systems (RTOS) basics. The material presented here is not intended as a complete coverage of different RTOSes and their features. For the purposes of simplification, I will briefly cover the most important features like categories, components, services, and scheduling algorithm, interrupt handling, inter task communication mechanism of an RTOS. After we explore what constitutes an RTOS and why we might want to use one, I’ll explain each of the basic components of a typical RTOS and show how these building blocks are integrated into the system.

**KEYWORDS**

- kernel: A basic unit of any OS that includes the functions for processes, memory, interrupts, task scheduling, inter-process communication and network subsystems in certain OSs.
- Process: A code that has its independent program counter values and an independent stack.
- Task: A task is an independent process that takes control of the CPU when scheduled by a scheduler. Every task has a TCB.
- Inter Process Communication: An output from one task (or process) passed to another task through the scheduler.
- Priority inversion: A problem in which a low priority task inadvertently does not release the process for a high priority task.

**CATEGORIES OF RTOS**

- **Hard RTOS**: In Hard Real Time System where failure to meet time constraints leads to system failure. The RTOS used in this system is known as Hard RTOS. For example Rocket launching.
- **Soft RTOS**: In Soft Real Time System where no System Failure occurs but the performance is degraded by failure to meet time constraints. The RTOS used in these kinds of systems is called Soft RTOS. For example Production House
Mobile & Hand hold RTOS: The devices such as mobiles, laptops where power saving is a big issue uses the Mobile & hands hold RTOS. For example Smart Phones.

DIFFERENCE BETWEEN RTOS AND OPERATING SYSTEM (OS)

An RTOS differs from common OS, in that the user when using the former has the ability to directly access the microprocessor and peripherals. Such an ability of the RTOS helps to meet deadlines. Other differences are as follows:

- RTOS is deterministic where as OS is not.
- Meeting the deadlines is mandatory in case of RTOS but not in OS.
- Whenever you switch on your system first control goes to the application in case of RTOS but in normal system control goes to the OS.
- RTOS needs more Priority levels.
- It is mandatory to use preemptive scheduling algorithm in case of RTOS but in OS any algorithm can be used.

RTOS SERVICES

Two essential services of RTOS are inter-process communication and scheduling in addition to other OS services. The services of an RTOS can be categorized as:

- **Basic OS Services** such as Memory Management, Process Management, Device Management etc.
- **RTOS Main Services** such as Real time scheduling, Interrupt-latency control and uses of timers.
- **Time Management Services** like time allocation and de-allocation to attain efficiency in given time constraint.
- **Priorities Management** includes priorities allocation and priorities inheritance.
- **Time slicing** for the process execution.
- **Predictability services** which includes predictable timing behaviour and predictable task-synchronization.

RTOS COMPONENTS

Broadly we can divide RTOS in to following components.

1) **Kernel.**: The core of an RTOS is known as the kernel. An API is provided to allow access to the kernel for the creation of tasks, among other things.

2) **Systick**: The heart beat of the kernel is the system tick, often called just systick. Without a system tick in the RTOS, nothing will happen. For every system tick the kernel will check if a task switch needs to be performed. System tick can be implemented with one of the hardware timers in the embedded chip. Using a timer might not be the best way to go for all applications. If you for example have an energy sensitive application you might not want to run the system tick handler each time a timer times out. For example, you might have an application that goes down in sleep mode to save power. The device only has to wake up if an external event occurs. If this external event is an external interrupt, you can call the system tick handler from that external interrupt. This way the application only runs the RTOS when something actually happens. Rest of the time it can be in sleep mode.

3) **Software timers**: Closely associated with the system tick are software timers. Software timers ticks once for each system tick, and is a convenient way to setup for example delays in an RTOS.

4) **Tasks**:

Tasks are like functions, each with its own stack and task control block (TCB). Unlike most functions, however, a task is almost always an infinite loop. That is, once it has been created, it will never exit.

A task is always in one of several **states**. A task can be ready to be executed, that is, in the READY state. Or the task may be suspended (pending), that is, the task is waiting for something to happen before it goes into the READY state. This is called the WAITING state. Here is a short description of the states we will use in our RTOS:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DORMANT</td>
<td>This task is not available for scheduling</td>
</tr>
<tr>
<td>READY</td>
<td>This task is ready to run</td>
</tr>
<tr>
<td>RUNNING</td>
<td>This is the currently-running task</td>
</tr>
<tr>
<td>WAITING</td>
<td>This task is waiting for something. This could be an event, a message or maybe for the RTOS clock to reach a specific value (delayed).</td>
</tr>
</tbody>
</table>

Note: Different RTOSes may have different names for each of these states.

5) **Scheduler**:

The real key is designing the scheduler. Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But, the choice of data structure depends also on the maximum number of tasks that can be on the ready list. There are two major types of scheduler from which you can choose:

a) **Event-driven**:

It is also known as Priority-Controlled Scheduling Algorithm. Usually different tasks have differing response requirement. For example, in an application that controls a motor, a keyboard and a display, the motor usually requires faster reaction time than the keyboard and display. This makes an event-driven scheduler must in event-driven systems, every task is assigned a priority and the task with the highest priority is executed. The order of execution depends on this priority.
The rule is very simple: **The scheduler activates the task that has the highest priority of all tasks that are ready to run.**

| Task 1 | | | | | Task 1 | | | | | Task 1 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State: Waiting | | | | | State: Running | | | | | State: Ready |
| Priority: 100 | | | | | Priority: 50 | | | | | Priority: 20 |

**b) Time-sharing:**
The most common time-sharing algorithm is called *Round-Robin*. With round-robin scheduling, the scheduler has a list of the tasks that make up the system and it uses this list to check for the next task that is ready to execute. If a task is READY, that task will execute. Associated with each task is its ‘time-slice’. This time-slice is the maximum time a task can execute for each round the scheduler makes.

**INTERTASK COMMUNICATION AND RESOURCE SHARING**
Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.) There are three common approaches to resolve this problem:

- Temporarily masking/disabling interrupts
- Binary semaphores
- Message passing

**a) Temporarily masking/disabling interrupts:**
General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wished. Modern CPUs make the interrupt disable control bit (or instruction) inaccessible in user mode to allow operating systems to prevent user tasks from doing this. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single-processor systems, if the application runs in kernel mode and can mask interrupts, often that is the best (lowest overhead) solution to preventing simultaneous access to a shared resource. While interrupts are masked, the current task has exclusive use of the CPU; no other task or interrupt can take control, so the critical section is effectively protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency, or else this method will increase the system’s maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few source code lines long and contains no loops. This method is ideal for protecting hardware bitmapped registers when the bits are controlled by different tasks.

When the critical section is longer than a few source code lines or involves lengthy looping, an embedded/real-time programmer must resort to using mechanisms identical or similar to those available on general-purpose operating systems, such as semaphores and OS-supervised intercrosses messaging.

**b) Binary semaphores:**
A binary semaphore is either locked or unlocked. When it is locked, a queue of tasks can wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. Problems with semaphore based designs are well known: priority inversion and deadlocks.

In **priority inversion**, a high priority task waits because a low priority task has a semaphore. A typical solution is to have the task that has a semaphore run at (inherit) the priority of the highest waiting task. But this simplistic approach fails when there are multiple levels of waiting (A waits for a binary semaphore locked by B, which waits for a binary semaphore locked by C). Handling multiple levels of inheritance without introducing instability in cycles is not straightforward.

In a **deadlock**, two or more tasks lock a number of binary semaphores and then wait forever (no timeout) for other binary semaphores, creating a cyclic dependency graph. The simplest deadlock scenario occurs when two tasks lock two semaphores in lockstep, but in the opposite order. Deadlock is usually prevented by careful design, or by having floored semaphores (which pass control of a semaphore to the higher priority task on defined conditions).

**C) Message passing:**
The other approach to resource sharing is for tasks to send **messages**. In this paradigm, the resource is managed directly
by only one task; when another task wants to interrogate or manipulate the resource, it sends a message to the managing task. This paradigm suffers from similar problems as binary semaphores that is priority inversion occurs when a task is working on a low-priority message, and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its in-box. Protocol deadlocks occur when two or more tasks wait for each other to send response messages.

**INTERRUPT HANDLING IN RTOS**
Since an interrupt handler blocks the highest priority task from running, and real-time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware as long as possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns). The interrupt handler then queues work to be done at a lower priority level, often by unblocking a driver task (through releasing a semaphore or sending a message). The scheduler often provides the ability to unblock a task from interrupt handler context.

An RTOS maintains an internal bookkeeping for the objects it manages like threads, mutexes and so on. Updates to this bookkeeping must be made atomic. For this reason it can be problematic when an interrupt handler calls RTOS functions while the application is also doing so. The RTOS function called from an interrupt handler can find the bookkeeping to be in an inconsistent state because of the application updates. In general there are two major architectures dealing with this problem, the Unified and the segmented architecture. RTOSes implementing the Unified architecture solve the problem by just disabling interrupts while the internal bookkeeping is updated. The downside of this however is that interrupt latency increases and interrupts might get lost.

The Segmented architecture follows a different approach. With this architecture the interrupt handler does not make direct RTOS calls but delegates the RTOS related work to a separate handler. This handler runs at a priority higher than any thread but lower than the interrupt handlers. The advantage of this architecture is that the RTOS adds very little or even zero cycles to the interrupt latency. As a result, RTOS implementing the segmented architecture are more predictable and can deal with much higher interrupt rates compared to RTOSes implementing the Unified architecture.

**MEMORY MANAGEMENT IN RTOS**
Memory Management is even more critical in an RTOS than in other operating systems. Because here speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block; however, this is unacceptable as memory allocation has to occur in a fixed time in an RTOS. The simple fixed-size-blocks algorithm works astonishingly well for simple embedded systems.

**EXAMPLES OF RTOS**
There are many widely deployed real-time operating systems. Some of them are:

**QNX:** As a microkernel-based OS, QNX[2] is based on the idea of running most of the OS in the form of a number of small tasks, known as servers. The system is quite small, with earlier versions fitting on a single floppy disk.

QNX Neutrino (2001) has been ported to a number of platforms and now runs on practically any modern CPU that is used in the embedded market. This includes the x 86 families, MIPS, PowerPC, SH-4 and the closely related family of ARM, StrongARM and XScale CPUs.

As of September 12, 2007, QNX offers a license for non-commercial users.

**RTLinux:** RT-Linux[3] is an operating system in which a small real-time kernel coexists with the Posix-like Linux kernel. The intention is to make use of the sophisticated services and highly optimized average case behavior of a standard time-shared computer system while still permitting real-time functions to operate in a predictable and low-latency environment. It was developed by Victor Yodaiken (Yodaiken 1999), Michael Barabanov (Barabanov 1996), Cort Dougan and others at the New Mexico Institute of Mining and Technology and then as a commercial product at FSM Labs. Wind River Systems acquired FSM Labs embedded technology in February 2007 and now makes a version available as Wind River Real-Time Core for Wind River Linux.

RTLinux was based on a lightweight virtual machine where the Linux "guest" was given a virtualized interrupt controller and timer - and all other hardware access was direct. From the point of view of the real-time "host", the Linux kernel is a thread. Interrupts needed for deterministic processing are processed by the real-time core, while other interrupts are forwarded to Linux, which runs at a lower priority than real-time threads. Linux drivers handle almost all I/O. First-In-First-Out pipes (FIFOs) or shared memory can be used to share data between the operating system and RTCore.

**Windows CE:** Windows CE[4] is optimized for devices that have minimal storage—a Windows CE kernel may run in under a megabyte of memory. Devices are often configured without disk storage, and may be configured as a “closed” system that does not allow for end-user extension (for instance, it can be burned into ROM). Windows CE conforms to the definition of a real-time operating system, with deterministic interrupt latency. From version 3 and onward, the system supports 256 priority levels and uses priority inheritance for dealing with priority inversion. The fundamental unit of execution is the thread. This helps to simplify the interface and improve execution time.

Since then, Windows CE has evolved into a component-based, embedded, real-time operating system. It is no longer targeted solely at hand-held computers many platforms have been based on the core Windows CE operating system, including...
Real Time Operating System

Microsoft's Auto PC, Pocket PC 2000, Pocket PC 2002, Windows Mobile 2003, Windows Mobile 2003 SE, Windows Mobile 5.0, Windows Mobile 6, Smart phone 2002, Smart phone 2003, Portable Media Center and many industrial devices and embedded systems. Windows CE even powered select games for the Dreamcast, was the operating system of the Gizmondo handheld, and can partially run on modified Xbox game consoles.

CONCLUSION
An RTOS must respond in a timely manner to changes, but that does not necessarily mean that an RTOS can handle a large throughput of data. In fact in an RTOS, small response times are valued much higher than power, or data speed. Sometimes an RTOS will even need to drop data to ensure that it meets its strict deadlines. In essence, that provides us with a perfect definition: an RTOS is an operating system designed to meet strict deadlines. Beyond that definition, there are few requirements as to what an RTOS must be, or what features it must have. Some RTOS implementations are very powerful and very robust, while other implementations are very simple, and suited for only one particular purpose.

According to me what I can conclude about RTOS is that an RTOS may not be necessary in a small scale embedded system. It is necessary when scheduling of multiple processes, ISRs and devices is important. An RTOS is must to monitor the processes that are response time controlled and event controlled processes.

FUTURE SCOPE
There are many new RTOS which are evolving day by day. These RTOS helps in different kind of application like embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control, and scientific research equipment). The TPF[^5] is one of the latest RTOS which is helping in many applications and hopefully will increase its usage in coming days.

TPF is an IBM real-time operating system for mainframes descended from the IBM System/360 family, including zSeries and System z9. The name is initialism for Transaction Processing Facility. TPF evolved from the Airlines Control Program (ACP), a free package developed in the mid-1960s by IBM in association with major North American and European airlines. In 1979, IBM introduced TPF as a replacement for ACP — and as a priced software product. The new name suggests its greater scope and evolution into non-airline related entities.

Current users include Sabre (reservations), Amadeus (reservations), VISA Inc (authorizations), American Express (authorizations), EDS SHARES (reservations), Holiday Inn (central reservations), CBOE (order routing), Singapore Airlines, KLM, Qantas, Amtrak, Marriott International, Worldspan and the NYPD (911 system).

TPF delivers fast, high-volume, high-throughput transaction processing, handling large, continuous loads of essentially simple transactions across large, geographically dispersed networks. The world's largest TPF-based systems are easily capable of processing tens of thousands of transactions per second. TPF is also designed for highly reliable, continuous (24x7) operation.

REFERENCES