R8C Family

General RTOS Concepts

Introduction

The advent of microprocessors has opened up several product opportunities that did not exist before. These intelligent processors have embedded themselves into all fields of our lives. As the complexities of the real-time embedded applications increases, benefits of employing a real-time operating system (RTOS) becomes ever more oblivious.

RTOS has become the key to many embedded systems today. There are wide ranges of RTOS/s available to the developers of embedded systems ranging from RTOS for robotics to home appliances.

This document defines RTOS and looks at its basic concepts.

Target Device

Applicable MCU: R8C Family

Contents

1. Guide in using this Document

This document strives to provide users with basic understanding of RTOS including explanation on some basic definitions, fundamental features and basic taxonomy of RTOS. It focuses on explaining the standard RTOS architecture and considerations to be taken in selection of RTOS.

Table 1 Explanation of Document Topics

Topic	Objective	Pre-requisite
Introduction to Real-Time Operating System	An overview of what is an RTOS and its architecture	An understanding of the fundamentals of computer-based system
Selection of RTOS	An explanation of the considerations to be evaluated in the selection of an RTOS	None
Reference Documents	Listing of documents that equip users with knowledge in the pre-requisite requirements	None

2. Introduction to Real-Time Operating System

"Real-Time Operating System (RTOS) is a multitasking operating system intended for real-time applications." – WIKIPEDIA. RTOS is implemented in products all around us, ranging from military, and consumer to scientific applications. Figure 1 depicts an example of RTOS implementation on Renesas automotive dashboard platform.

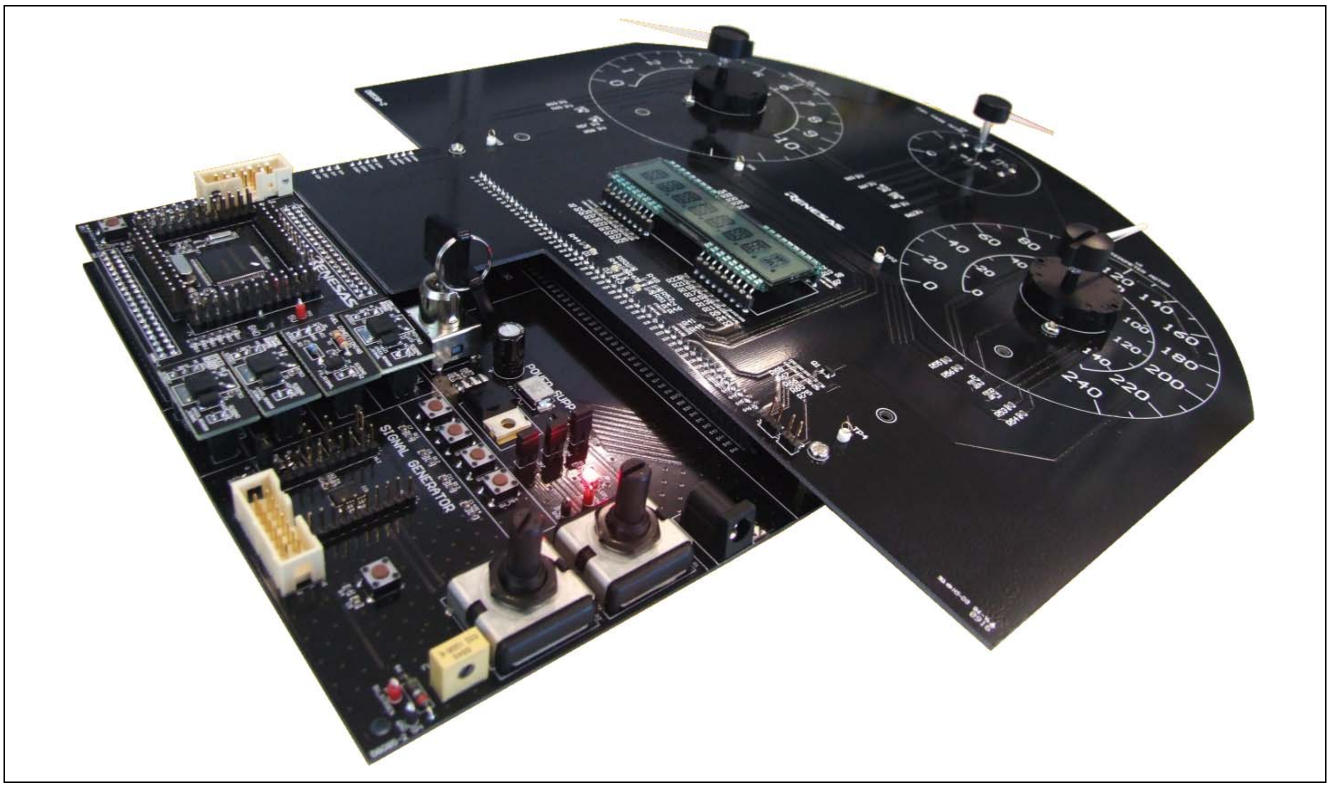

Figure 1 Renesas Automotive Dashboard Platform (with MR8C/4)

2.1 What is RTOS

RTOS comprises of two components, namely, "Real-Time" and "Operating System".

2.1.1 Real-Time

Real-Time indicates an expectant response or reaction to an event on the instant of its evolution. The expectant response depicts the logical correctness of the result produced. The instant of the events' evolution depicts deadline for producing the result.

2.1.2 Operating System

Operating System (OS) is a system program that provides an interface between hardware and application programs. OS is commonly equipped with features like: Multitasking, Synchronization, Interrupt and Event Handling, Input/ Output, Inter-task Communication, Timers and Clocks and Memory Management to fulfill its primary role of managing the hardware resources to meet the demands of application programs.

RTOS is therefore an operating system that supports real-time applications and embedded systems by providing logically correct result within the deadline required. Such capabilities define its deterministic timing behavior and limited resource utilization nature.

2.2 Why RTOS for Real-Time Application

RTOS is not a required component of all real-time application in embedded systems. An embedded system in a simple electronic rice cooker does not require RTOS. But as the complexity of applications expands beyond simple tasks, benefits of having an RTOS far outweigh the associate costs.

Embedded systems are becoming more complex hardware-wise with every generation. And as more features are put into them in each iteration, application programs running on the embedded system platforms will become increasingly complex to be managed as they strive to meet the system response requirements. An RTOS will be effective to allow the real-time applications to be designed and expanded more easily whilst meeting the performances required.

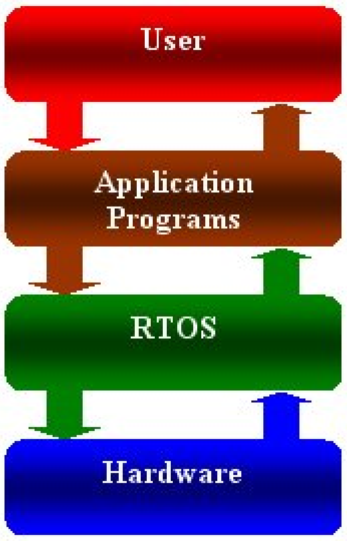

Figure 2 Real-Time Embedded System with RTOS

2.3 Classification of RTOS

RTOS's are broadly classified into three types, namely, Hard Real Time RTOS, Firm Real Time RTOS and Soft Real Time RTOS as described below:

- Hard real-time: degree of tolerance for missed deadlines is extremely small or zero. A missed deadline has catastrophic results for the system
- Firm real-time: missing a deadline might result in an unacceptable quality reduction
- Soft real-time: deadlines may be missed and can be recovered from. Reduction in system quality is acceptable

2.4 Misconception of RTOS

a) RTOS must be fast

The responsiveness of an RTOS depends on its deterministic behavior and not on its processing speed. The ability of RTOS to response to events within a timeline does not imply it is fast.

b) RTOS introduce considerable amount of overhead on CPU

An RTOS typically only require between 1% to 4% of a CPU time.

c) All RTOS are the same

RTOS are generally designed for 3 types of real-time systems (i.e. hard, firm & soft). In addition, they are further classified according to the types of hardware devices (e.g. 8-bit, 16-bit, 32-bit MPU) supported.

2.5 Features of RTOS

The design of an RTOS is essentially a balance between providing a reasonably rich feature set for application development and deployment and, not sacrificing predictability and timeliness. A basic RTOS will be equipped with the following features:

i. Multitasking and Preemptibility

An RTOS must be multi-tasked and preemptible to support multiple tasks in real-time applications. The scheduler should be able to preempt any task in the system and allocate the resource to the task that needs it most even at peak load.

ii. Task Priority

Preemption defines the capability to identify the task that needs a resource the most and allocates it the control to obtain the resource. In RTOS, such capability is achieved by assigning individual task with the appropriate priority level. Thus, it is important for RTOS to be equipped with this feature.

iii. Reliable and Sufficient Inter Task Communication Mechanism

For multiple tasks to communicate in a timely manner and to ensure data integrity among each other, reliable and sufficient inter-task communication and synchronization mechanisms are required.

iv. Priority Inheritance

To allow applications with stringent priority requirements to be implemented, RTOS must have a sufficient number of priority levels when using priority scheduling.

v. Predefined Short Latencies

An RTOS needs to have accurately defined short timing of its system calls. The behavior metrics are:

- Task switching latency: The time needed to save the context of a currently executing task and switching to another task is desirable to be short.
- Interrupt latency: The time elapsed between execution of the last instruction of the interrupted task and the first instruction in the interrupt handler.
- Interrupt dispatch latency. The time from the last instruction in the interrupt handler to the next task scheduled to run.

vi. Control of Memory Management

To ensure predictable response to an interrupt, an RTOS should provide way for task to lock its code and data into real memory.

2.6 RTOS Architecture

The architecture of an RTOS is dependent on the complexity of its deployment. Good RTOSs are scalable to meet different sets of requirements for different applications. For simple applications, an RTOS usually comprises only a kernel. For more complex embedded systems, an RTOS can be a combination of various modules, including the kernel, networking protocol stacks, and other components as illustrated in Figure 3.

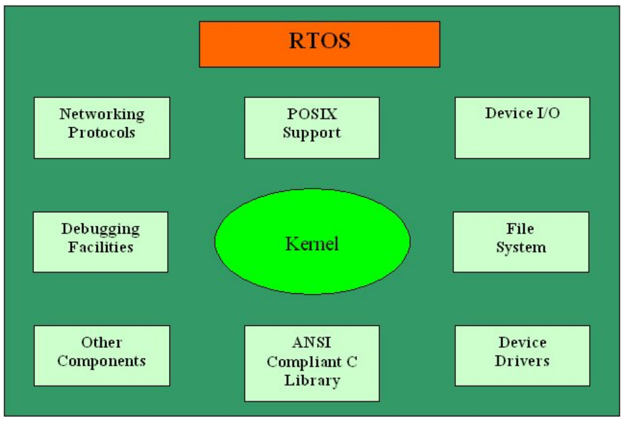

Figure 3 General Architecture of RTOS

2.6.1 Kernel

An operating system generally consists of two parts: kernel space (kernel mode) and user space (user mode). Kernel is the smallest and central component of an operating system. Its services include managing memory and devices and also to provide an interface for software applications to use the resources. Additional services such as managing protection of programs and multitasking may be included depending on architecture of operating system. There are three broad categories of kernel models available, namely:

Monolithic kernel

It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) in kernel space. As such, monolithic kernels provide rich and powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel. Examples are Linux and Windows.

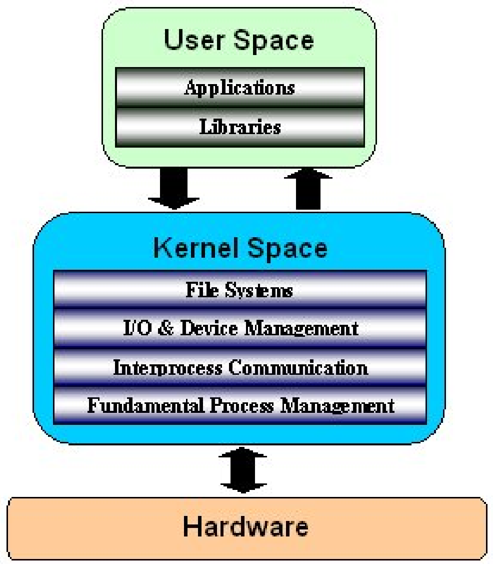

Figure 4 Monolithic Kernel Based Operating System

Microkernel

It runs only basic process communication (messaging) and I/O control. The other system services (file system. networking, etc) reside in user space in the form of daemons/servers. Thus, micro kernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as the kernel is unaffected even if the servers failed (i.e. File System). Examples are AmigaOS and QNX.

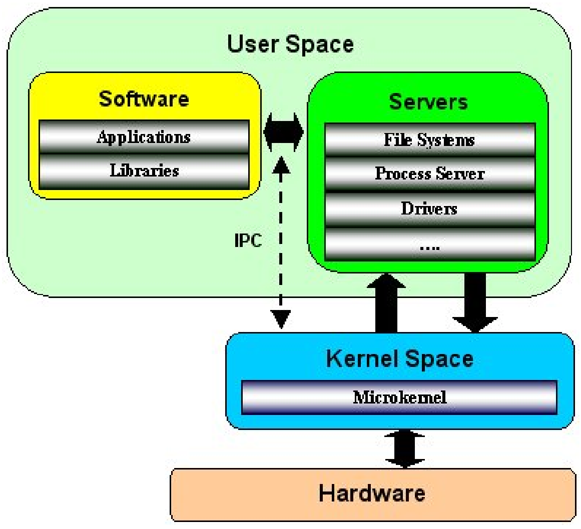

Figure 5 Microkernel Based Operating System

Exokernel

The concept is orthogonal to that of micro- vs. monolithic kernels by giving an application efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems (libOSes) and leaving the management to the application.

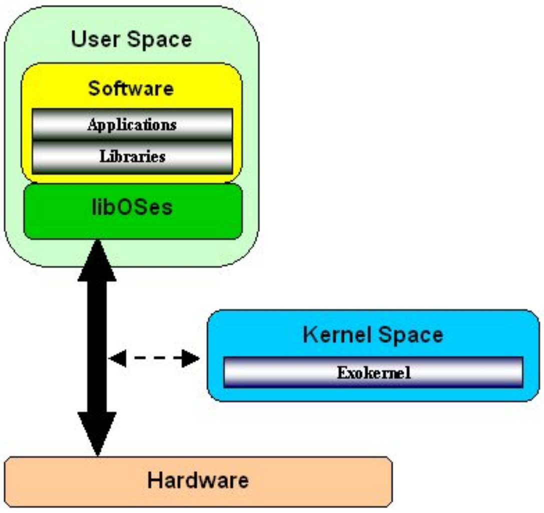

Figure 6 Exokernel Based Operating System

An RTOS generally avoids implementing the kernel as a large monolithic program. The kernel is developed instead as a micro-kernel with added configurable functionalities. This implementation gives resulting benefit in increase system configurability, as each embedded application requires a specific set of system services with respect to its characteristics.

The kernel of an RTOS provides an abstraction layer between the application software and hardware. This abstraction layer comprises of six main types of common services provided by the kernel to the application software. Figure 7 shows the six common services of an RTOS kernel.

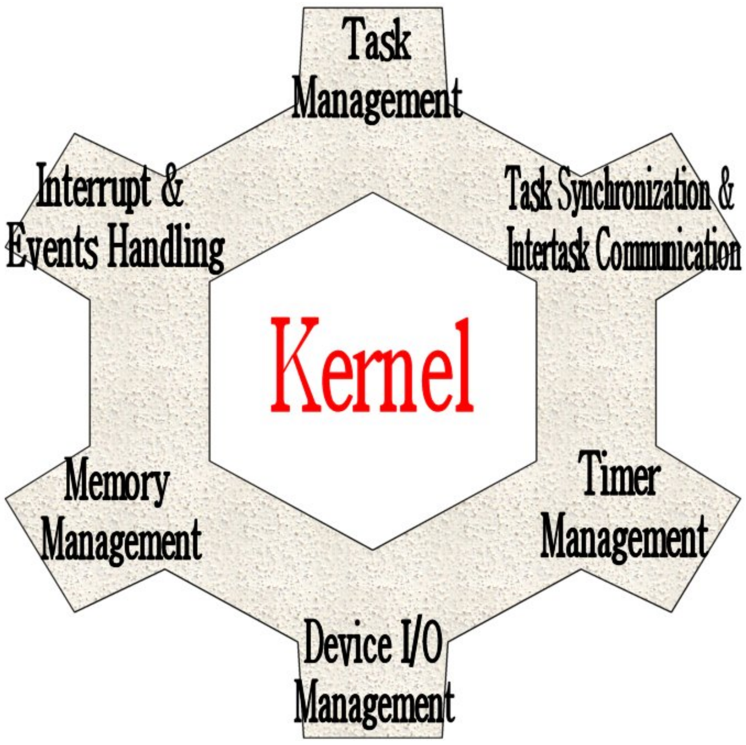

Figure 7 RTOS Kernel Services

2.6.2 Task Management

Task management allows programmers to design their software as a number of separate "chunks" of codes with each handling a distinct goal and deadline. This service encompasses mechanism such as scheduler and dispatcher that creates and maintain task objects.

Task Object

To achieve concurrency in real-time application program, the application is decompose into small, schedulable, and sequential program units known as "Task". In real-time context, task is the basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time. Release time refers to the point in time from which the task can be executed. Deadline is the point in time by which the task must complete. Execution time denotes the time the task takes to execute.

A task object is defined by the following set of components:

- Task Control block (Task data structures residing in RAM and only accessible by RTOS)
- Task Stack (Data defined in program residing in RAM and accessible by stack pointer)
- Task Routine (Program code residing in ROM)

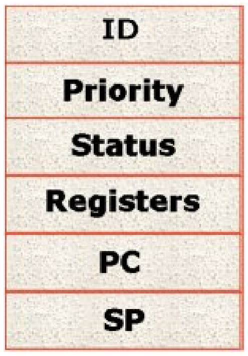

Figure 8 Typical Task Control Block (TCB)

Each task may exist in any of the four states, including running, ready, or blocked and dormant as shown in Figure 9. During the execution of an application program, individual tasks are continuously changing from one state to another. However, only one task is in the running mode (i.e. given CPU control) at any point of the execution. In the process where CPU control is change from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved. This process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

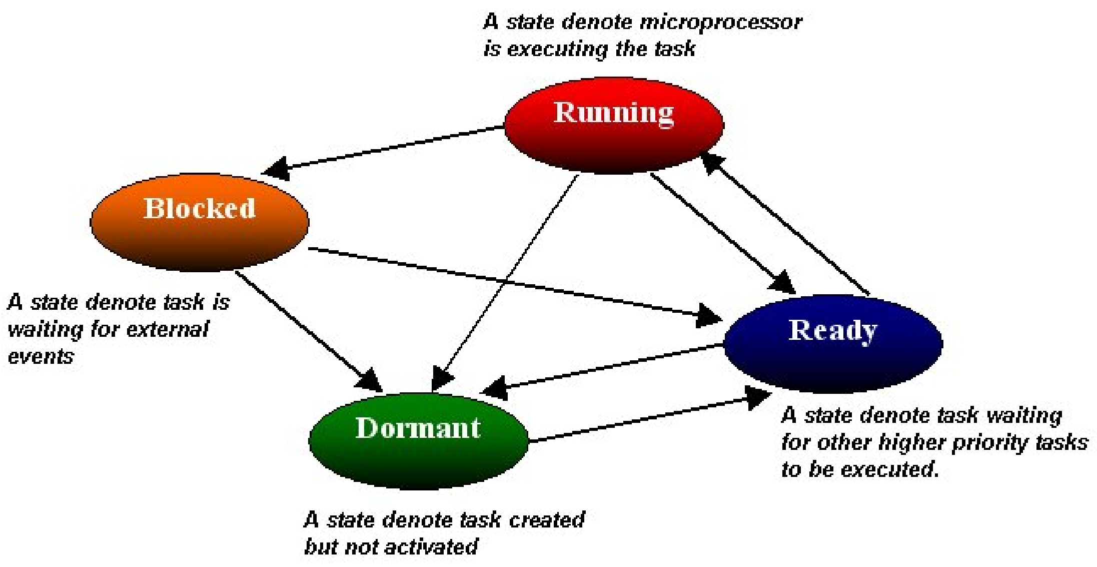
A state denote microprocessor is executing the task
Running
Blocked
A state denote task is waiting for external events
Ready
A state denote task waiting for other higher priority tasks to be executed.
Dormant
A state denote task created but not activated

Figure 9 Possible States Transition of Tasks

Scheduler

The scheduler keeps record of the state of each task and selects from among them that are ready to execute and allocates the CPU to one of them. A scheduler helps to maximize CPU utilization among different tasks in a multi-tasking program and to minimize waiting time. There are generally two types of schedulers: non-preemptive and priority-based preemptive.

Non-preemptive scheduling or cooperative multitasking requires the tasks to cooperate with each other to explicitly give up control of the processor. When a task releases the control of the processor, the next most important task that is ready to run will be executed. A task that is newly assigned with a higher priority will only gain control of the processor when the current executing task voluntarily gives up the control. Figure 10 gives an example of a non-preemptive scheduling

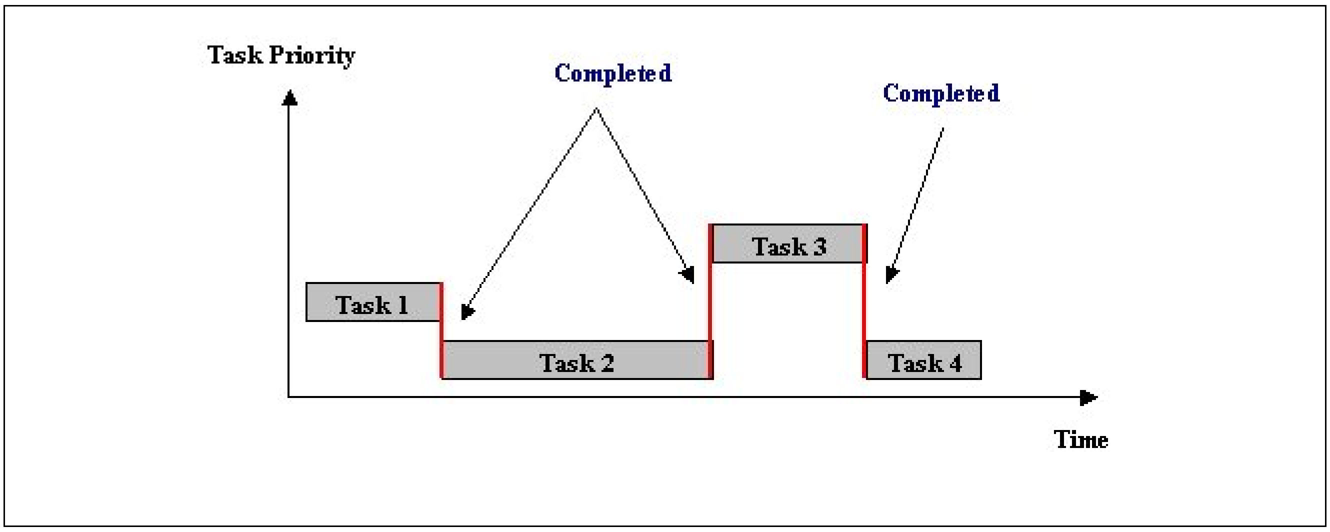

Figure 10 Non-preemptive Scheduling

Priority-based preemptive scheduling requires control of the processor be given to the task of the highest priority at all time. In the event that makes a higher priority task ready to run, the current task is immediately suspended and the control of the processor is given to the higher priority task. Figure 11 shows an example of a preemptive scheduling.

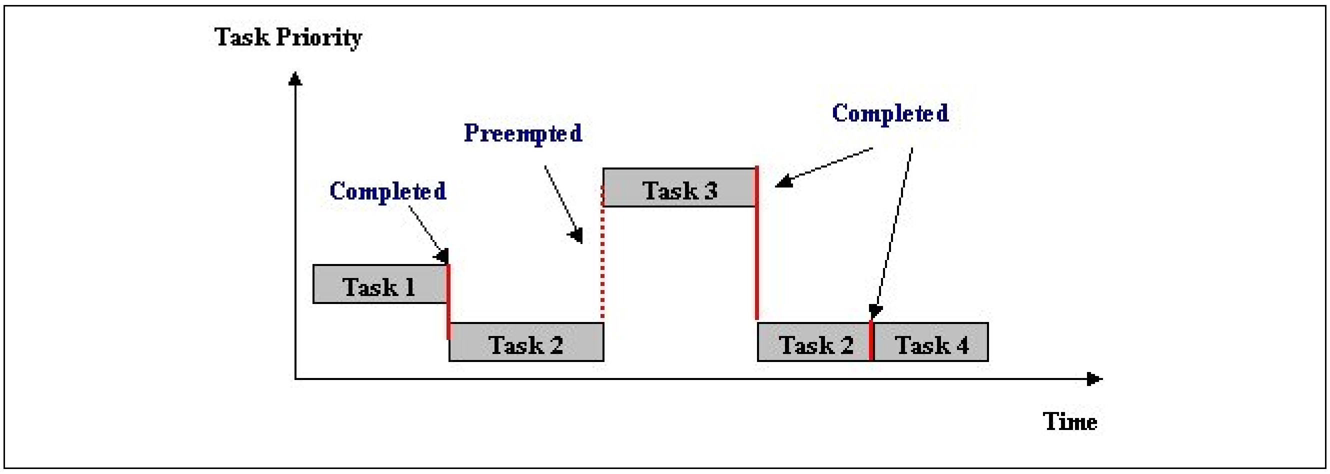

Figure 11 Preemptive Scheduling

Dispatcher

The dispatcher gives control of the CPU to the task selected by the scheduler by performing context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution passes through one of three areas: through the task program code, through an interrupt service routine, or through the kernel.

2.6.3 Task Synchronization & Intertask Communication

Task synchronization and intertask communications serves to enable information to be transmitted safely from one task to another. The service also makes it possible for tasks to coordinate and cooperate with one another.

Task Synchronization

Synchronization is essential for tasks to share mutually exclusive resources (devices, buffers, etc) and/or allow multiple concurrent tasks to be executed (e.g. Task A needs a result from task B, so task A can only run till task B produces it). Task synchronization is achieved using two types of mechanisms; 1) Event Objects and 2) Semaphores.

Event objects are used when task synchronization is required without resource sharing. They allow one or more tasks to keep waiting for a specified event to occur. An event object can exist in either of two states: triggered and non-triggered. An event object in a triggered state indicates that a waiting task may resume. In contrast, if the event object is in a non-triggered state, a waiting task will need to stay suspended.

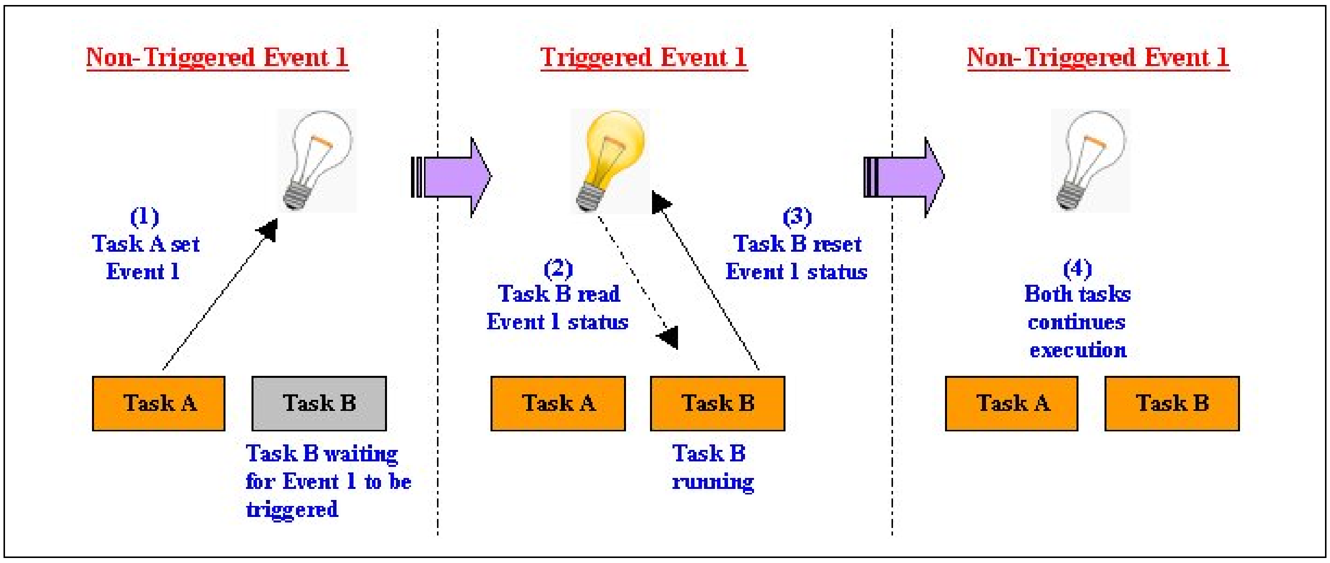
Non-Triggered Event 1
(1) Task A set Event 1
Task A
Task B
Task B waiting for Event 1 to be triggered
Triggered Event 1
(2) Task B read Event 1 status
(3) Task B reset Event 1 status
Task A
Task B
Task B running
Non-Triggered Event 1
(4) Both tasks continues execution
Task A
Task B

Figure 12 Working Principle of Event Objects

Sharing of resource among tasks can be a problem when the coordination is not well done. For instance, if task A begins reading a set of data currently being updated by task B, task A might received corrupted data – mixture of new and existing data. To resolve this problem, RTOS kernels provide a semaphore object and associated semaphore services to ensure the integrity of data.

A semaphore has an associated resource count and a wait queue. The resource count indicates availability of resource. The wait queue manages the tasks waiting for resources from the semaphore. A semaphore functions like a key that define whether a task has the access to the resource. A task gets an access to the resource when it acquires the semaphore. The resource count of a semaphore determines the number of times the semaphore can be acquired. When a task acquires the semaphore, its count decrement by one. Likewise, its count increment by one when a task releases the semaphore. Generally. There are three types of semaphore:

- Binary Semaphores (semaphore value of either 0 or 1 to indicate unavailability and availability respectively)
- Counting Semaphores (semaphore value of 0 or greater indicating it can be acquired/released multiple times)
- Mutually Exclusion Semaphores (semaphore value of 0 or 1 but lock count can be 0 or greater for recursive locking)

Figure 13 illustrates the types of semaphore.

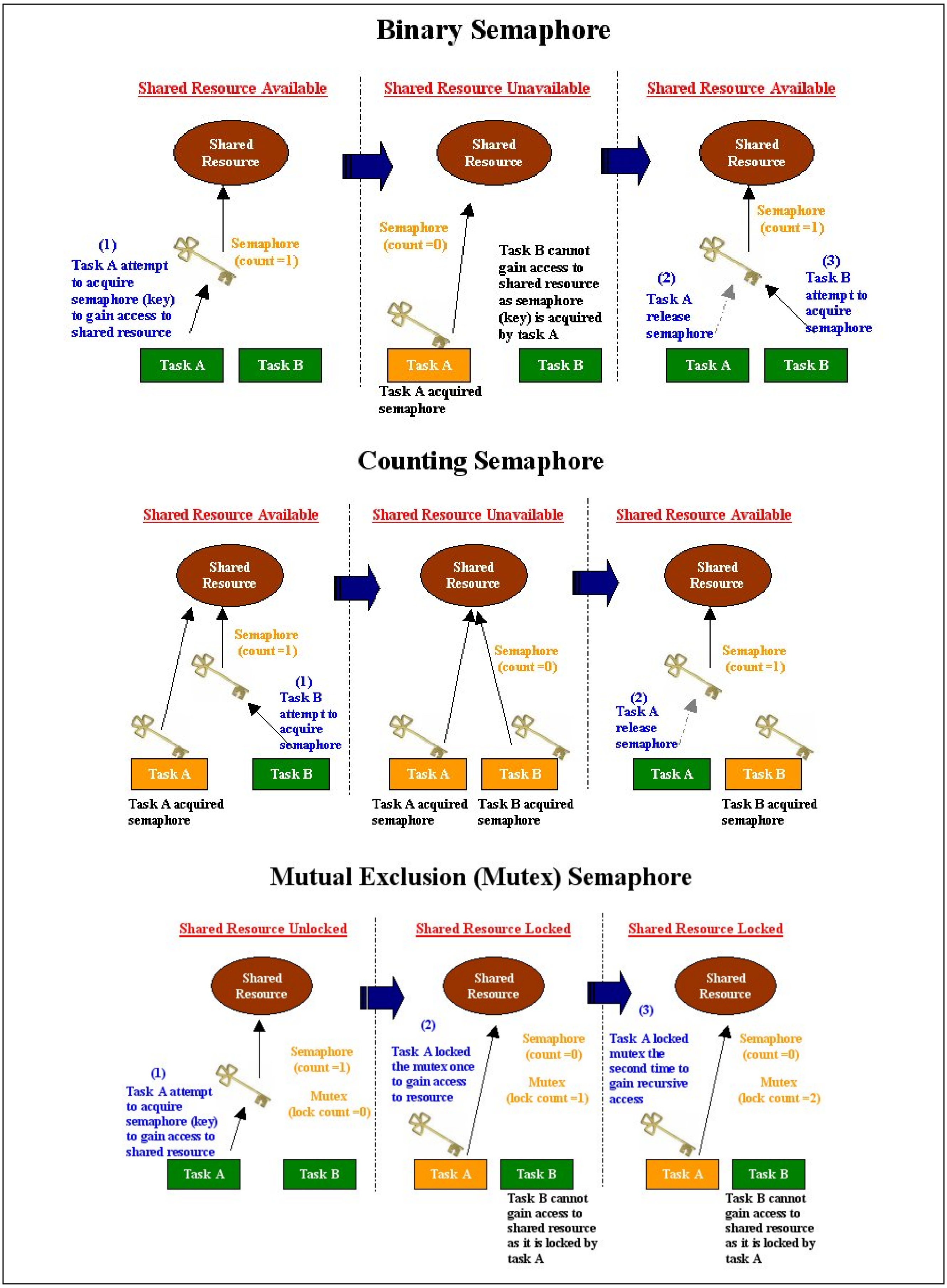
Binary Semaphore
Shared Resource Available
Shared Resource
(1)
Task A attempt to acquire semaphore (key) to gain access to shared resource
Semaphore (count =1)
Task A
Task B
Shared Resource Unavailable
Shared Resource
Semaphore (count =0)
Task B cannot gain access to shared resource as semaphore (key) is acquired by task A
Task A
Task B
Task A acquired semaphore
Shared Resource Available
Shared Resource
Semaphore (count =1)
(2)
Task A release semaphore
(3)
Task B attempt to acquire semaphore
Task A
Task B
Counting Semaphore
Shared Resource Available
Shared Resource
Semaphore (count =1)
(1)
Task B attempt to acquire semaphore
Task A
Task B
Task A acquired semaphore
Shared Resource Unavailable
Shared Resource
Semaphore (count =0)
Task A
Task B
Task A acquired semaphore
Task B acquired semaphore
Shared Resource Available
Shared Resource
Semaphore (count =1)
(2)
Task A release semaphore
Task A
Task B
Task B acquired semaphore
Mutual Exclusion (Mutex) Semaphore
Shared Resource Unlocked
Shared Resource
(1)
Task A attempt to acquire semaphore (key) to gain access to shared resource
Semaphore (count =1)
Mutex (lock count =0)
Task A
Task B
Shared Resource Locked
Shared Resource
(2)
Task A locked the mutex once to gain access to resource
Semaphore (count =0)
Mutex (lock count =1)
Task A
Task B
Task B cannot gain access to shared resource as it is locked by task A
Shared Resource Locked
Shared Resource
(3)
Task A locked mutex the second time to gain recursive access
Semaphore (count =0)
Mutex (lock count =2)
Task A
Task B
Task B cannot gain access to shared resource as it is locked by task A

Figure 13 Types of Semaphore

Intertask Communication

Intertask communication involves sharing of data among tasks through sharing of memory space, transmission of data and etc. Few of mechanisms available for executing intertask communications includes:

- Message queues
- Pipes
- Remote procedural calls (RPC)

A message queue is an object used for intertask communication through which task send or receive messages placed in a shared memory. Tasks and ISRs send and receive messages to the queue through services provided by the kernel. A task seeking for a message from an empty queue is blocked either for a duration or until a message is received. The sending and receiving of messages to and from the queue may follow 1) First In First Out (FIFO), 2) Last in First Out (LIFO) or 3) Priority (PRI) sequence. Usually, a message queue comprises of an associated queue control block (QCB), name, unique ID, memory buffers, queue length, maximum message length and one or more task waiting lists. A message queue with a length of 1 is commonly known as a mailbox.

A pipe is an object that provide simple communication channel used for unstructured data exchange among tasks. A pipe can be opened, closed, written to and read from. Traditionally, a pipe is a unidirectional data exchange facility. There are two descriptors respectively at each end of the pipe for reading and writing. Data is written into the pipe as an unstructured byte stream via one descriptor and read from the pipe in the FIFO order from the other. Unlike message queue, a pipe does not store multiple messages but stream of bytes. In addition, data flow from a pipe cannot be prioritized.

Remote procedure call (RPC) component permits distributed computing where task can invoke the execution of a another task on a remote computer, as if the task ran on the same computer.

2.6.4 Memory Management

An embedded RTOS usually strive to achieve small footprint by including only the functionality needed for the user's applications. There are two types of memory management in RTOSs. They are Stack and Heap managements.

In a multi-tasking RTOS, each task needs to be allocated with an amount of memory for storing their contexts (i.e. volatile information such as registers contents, program counter, etc) for context switching. This allocation of memory is done using task-control block model (as mentioned in section 2.6.2). This set of memory is commonly known as kernel stack and the management process termed Stack Management.

Upon the completion of a program initialization, physical memory of the MCU or MPU will usually be occupied with program code, program data and system stack. The remaining physical memory is called heap. This heap memory is typically used by the kernel for dynamic memory allocation of data space for tasks. The memory is divided into fixed size memory blocks, which can be requested by tasks. When a task finishes using a memory block it must return it to the pool. This process of managing the heap memory is known as Heap management.

2.6.5 Timer Management

In embedded systems, system and user tasks are often scheduled to perform after a specified duration. To provide such scheduling, there is a need for a periodical interrupt to keep track of time delays and timeout. Most RTOSs today offer both "relative timers" that work in units of ticks, and "absolute timers" that work with calendar date and time. For each kind of timer, RTOSs provide a "task delay" service, and also a "task alert" service based on the signaling mechanism (e.g. event flags). Another timer service provided is in meeting task deadline by cooperating with task schedulers to determine whether tasks have met or missed their real-time deadlines.

2.6.6 Interrupt and Event Handling

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. A fundamental challenge in RTOS design is supporting interrupts and thereby allowing asynchronous access to internal RTOS data structures. The interrupt and event handling mechanism of an RTOS provides the following functions:

- Defining interrupt handler
- Creation and deletion of ISR
- Referencing the state of an ISR
- Enabling and disabling of an interrupt
- Changing and referencing of an interrupt mask

and help to ensure:

- Data integrity by restricting interrupts from occurring when modifying a data structure
- Minimum interrupt latencies due to disabling of interrupts when RTOS is performing critical operations
- Fastest possible interrupt responses that marked the preemptive performance of an RTOS
- Shortest possible interrupt completion time with minimum overheads

2.6.7 Device I/O Management

An RTOS kernel is often equipped with a device I/O management service to provide a uniform framework (application programmer's interface-"API") and supervision facility for an embedded system to organize and access large numbers of diverse hardware device drivers. However, most device driver APIs and supervisors are "standard" only within a specific RTOS.

3. Selection of RTOS

RTOS tends to be a selection for many embedded projects. But is an RTOS always necessary? The answer lies on careful analysis in understanding what an application needs to deliver to determine whether implementing RTOS is a requirement or an extravagance.

Most programmers are not familiar with RTOS constraints and requirements. An RTOS is usually chosen based on its performance or one's comfort and familiarity with the product. However, such a selection criteria is insufficient. To make matter worse, there is a wide variety of RTOS ranging from commercial RTOS, open-source RTOS to internally developed RTOS to choose from. Therefore, it is incumbent upon the programmers to exercise extra caution in the selection process.

The selection criteria of RTOS can be broadly classified into two main areas; technical features of RTOS and commercial aspect of the implementation.

3.1 Technical Considerations

3.1.1 Scalability

Size or memory footprint is an important consideration. Most RTOS are scalable in which only the code required is included in the final memory footprint. Looking for granular scalability in an RTOS is a worthwhile endeavor, as it minimizes memory usage.

3.1.2 Portability

Often, a current application may outgrow the hardware it was originally designed for as the requirements of the product increases. An RTOS with such a capability can therefore be ported between processor architectures and between specific target systems.

3.1.3 Run-time facilities

Run-time facilities refer to the services of the kernel (i.e. intertask communication, task synchronization, interrupts and events handling, etc). Different application systems have different sets of requirements. Comparison of RTOSs is frequently between the kernel-level facilities they provided.

3.1.4 Run-time performance

Run-time performance of an RTOS is generally governed by the interrupt latency, context switching time and few other metric of kernel performance. This consideration is useful if the performance assessment of the application on a given RTOS is to prototype its performance-critical aspects on standard hardware.

3.1.5 Development tools

A sufficient set of development tools including debugger; compiler and performance profiler might help in shortening the development and debugging time, and improve the reliability of the coding. Commercial RTOSs usually have a complete set of tools for analyzing and optimizing the RTOSs' behavior whereas Open-Source RTOSs will not have.

3.2 Commercial Considerations

3.2.1 Costs

Costs are a major consideration in selection of RTOS. There are currently more than 80 RTOS vendors. Some of the RTOS packages are complete operating systems including not only the real-time kernel but also an input/output manager, windowing systems, a file system, networking, language interface libraries, debuggers, and cross platform compilers. And the cost of an RTOS ranges from US$70 to over US$30,000. The RTOS vendor may also require royalties on a per-target-system basis, which may varies between USS5 to more than US$250 per unit. In addition, there will be maintenance required and that can easily cost between US$100 to US$5,000 per year.

3.2.2 License

An RTOS vendor usually has a few license models for customers to choose from. A perpetual license enables customers to purchase the development set and pay an annual maintenance fee, which entitles he/her to upgrades and bug fixes. An alternative model known as subscription model allow customers to "rent" the development set whilst paying an annual fee to renew the access. This model provides customers with lower technology acquisition fees but cost from annual renewal fee can escalate after many years.

3.2.3 Supplier stability/ longevity

Development with RTOS is not a problem free process. Reliable and consistent support from supplier is a critical factor in ensuring the prompt completion of a project. Supplier longevity thus helps to determine the availability of support.

4. Reference Documents

Publication

- An Embedded Software Primer (David E. Simon)
- Real-Time Concepts for Embedded Systems (Qing Li with Caroline Yeo)

Website

- EMBEDDED.COM, http://www.embedded.com.

Website and Support

Renesas Technology Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry
csc@renesas.com

Revision Record

		Description	
Rev.	Date	Page	Summary
1.00	Jan.01.10	—	First edition issued