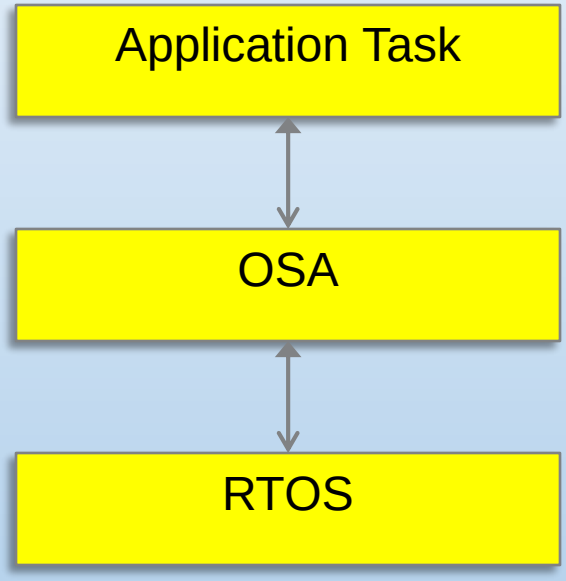
ASR1802 OS API(OSA) 介绍 V0.2

OS Overview

- Kernel
  - ThreadX 5.1
- OSA APIs
  - OSA is the adaption layer between RTOS and applications.
  - OSA provides service to application tasks based on RTOS.
  - OSA API translate the basic operating system services to the native OS implementation
  - The most used APIs include:
    - Initialization
    - Task Management
    - Inter-Task Communication: event flags, semaphores, message queues, mailbox queues
    - Timers

OSA – OS Adapter layer

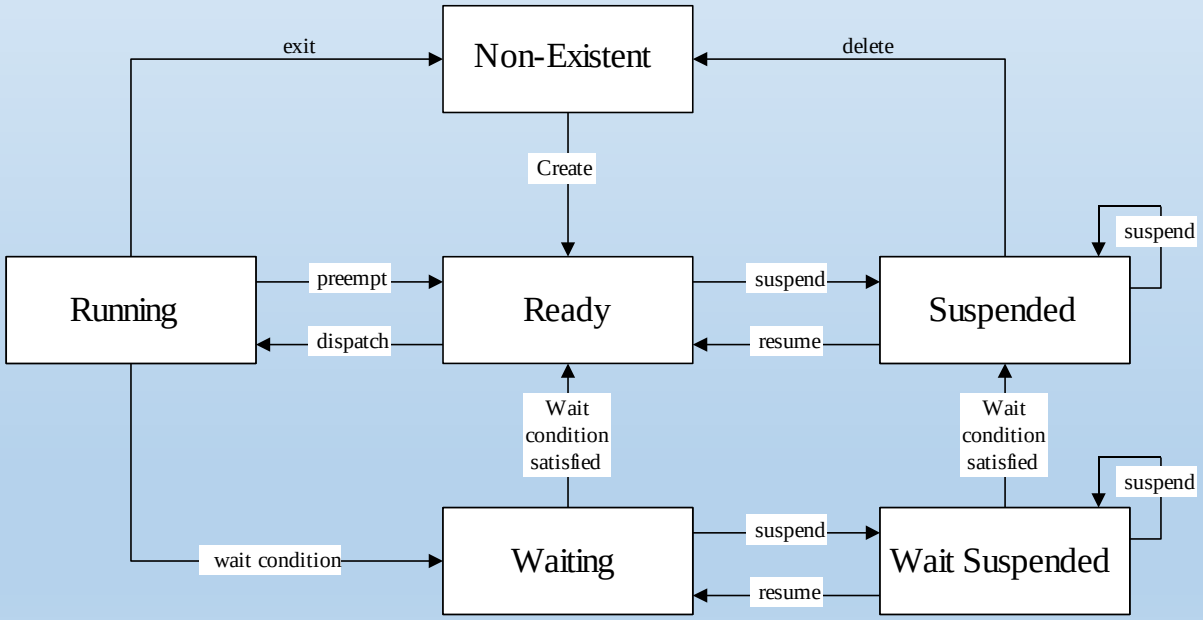


Initialization

- **OSAInitialize()** initializes the native operating system and internal OSA data structures
- The application may create tasks any time after calling **OSAInitialize()**

Task Management

- Allow users to create, delete, suspend, resume, put to sleep, and change the priority of tasks.
- The state transition diagram for tasks:



Task Management--Task States

- Tasks are created by **OSATaskCreate()** and created in the **Ready** state. Created task will be dispatched by the scheduler to running state
- Task is deleted by **OSATaskDelete()**. The task can be deleted only if it is in the **Suspended** state
- **OSATaskSuspend()** put the ready task to the **Suspended** state
- A task in the **Running** state enters the **Waiting** state when waiting for a blocking OSA API call to complete
- **OSATaskSuspend()** puts a task in the **Waiting** state to the **Wait Suspended** state
- A task in the **Wait Suspended** state enters the **Suspended** state until the condition it is waiting for is satisfied
- A task in the **Wait Suspended** state enters the **Waiting state** state until resumed by a call to **OSATaskResume()**

Task Management--Task States

- A task in the **Suspended** state is resumed to the **Ready** state by a call to **OSATaskResume()**
- Only a single call to **OSATaskResume()** is required to change the **Suspended/WaitSuspended** task's state to **Ready/Waiting**, even multiple **OSATaskSuspend()** are called

Task Management--Scheduler

- Priority based preemptive scheduling policy
- Preempted tasks are added to an ordered queue of all ready tasks in the system. Same priority are added to the queue in first-in-first-out order
- No time slicing of tasks with the same priority. A task may, however, allow other ready tasks of the same priority a chance to execute, calling **OSATaskYield()**

Task Management--Context Locking

- Calling **OSAContextLock()** to disable the scheduler
- Does not disable interrupts
- Calling **OSAContextUnlock()** enables the scheduler
- Applications must not make any blocking API calls while the context is locked

Task Management--Sleep

- Calling **OSATaskSleep()** put running task to sleep for a specified number of OS clock ticks by itself

Task Management--Priority Control

- Queried with **OSATaskGetPriority**. It returns the current priority of the task specified
- **OSATaskChangePriority()** changes the priority of a task at any time
- Changing the priority of a running task will cause it to be preempted if the new priority is lower than the priority of the task at the front of the Ready list
- Changing the priority of a ready task will change its position on the Ready list

Inter-task Communication

- OSA provides five mechanisms for inter-task communication
  - Event Flags
  - Semaphores
  - Mutexes
  - Message Queues
  - Mailbox Queues.

Inter-task Communication--Event Flags

- Flag groups provide a mechanism for tasks to wait for asynchronous event notification from multiple sources
- Created by **OSAFlagCreate()**. **OSAFlagCreate()** returns a reference that is used by other event flag functions
- Each flag group has 32 flags. One flag is represented by one bit in a unsigned int.
- Calling **OSAFlagWait()** with a bit mask of flags to wait for event flag
- Four different wait operations specified by **OSAFlagWait()**:
  - **OSA_FLAG_AND -** wait for all of the bits in the input mask to be set, don't clear the event flags
  - **OSA_FLAG_AND_CLEAR -** wait for all of the bits in the input mask to be set, clear all event flags on successful completion
  - **OSA_FLAG_OR** - wait for any of the bits in the input mask to be set, don't clear the event f
  - **OSA_FLAG_OR_CLEAR -** wait for any of the bits in the input mask to be set, clear all event flags of successful completion

Inter-task Communication--Event Flags

- The blocking behavior of **OSAFlagWait()** is specified by the “timeout” parameter
  - **OSA_NO_SUSPEND:** No Block, OS_FLAG_NOT_PRESENT will be returned if the flag condition is not met,
  - **OSA_SUSPEND:** block until the condition specified
  - **Value between 1 and 4,294,967,293:** timeout length in number of OS clock ticks
- **OSAFlagSet()** with a bit mask of event flags to set a flag. All waiting tasks that have their wait condition met will be made ready to run
- **OSAFlagPeek()** checks the status of a flag in a group without blocking by calling

Inter-task Communication--Semaphores

- Created by **OSASemaphoreCreate()**. **OSASemaphoreCreate()** returns a reference that is used by other semaphore functions.
- **OSASemaporeAcquire()** acquires the semaphore. It will return OS_SUCCESS when semaphore count is greater than zero, or block
- The blocking behavior of **OSASemaphoreAcquire()** is specified by the “timeout” parameter like event flags
- **OSASemaphoreRelease()** releases a semaphore. The semaphore count is incremented by one and the blocked waiting task is called
- **OSASemaphoreDelete**() deletes a semaphore. It should never be deleted when there are tasks waiting for

Inter-task Communication--Mutexes

- Created with **OSAMutexCreate(). OSAMutexCreate()** returns a reference that is used by other mutex functions .
- Using **OSAMutexLock()** to lock a mutex. The blocking behavior is specified by the “timeout” parameter, like event flags.
- Released by **OSAMutexUnlock()**. If there is a task waiting for a mutex when **OSAMutexUnlock()** is called, the task at the front of the mutex wait queue will be next to acquire the mutex.
- If a mutex owner calls **OSAMutexLock()** again before calling **OSAMutexUnlock(),** an **OS_FAIL** status is returned
- A mutex should never be deleted when there are tasks waiting for a **OSAMutexLock()** operation to complete

Inter-task Communication--Message Queues

- Message queues hold messages of variable size
- Created with **OSAMsgQCreate(). OSAMsgQCreate()** returns a reference that is used by other message queue functions .
- Messages are sent with **OSAMsgQSend()**. The inputs to **OSAMsgQSend()** are a pointer to the message data, the size of the message in bytes, and a “timeout” value. The timeout definition likes event flags
- Messages are removed from a message queue by **OSAMsgQRecv()**. The inputs to **OSAMsgQRecv()** are a pointer to a user data buffer, the size of the user data buffer in bytes, and a timeout value
- The number of messages on a message queue can be read with **OSAMsgQPoll()**
- A message queue should never be deleted when there are tasks waiting for

Inter-task Communication--Mailbox Queues

- Mailbox queues hold items that are pointers to void
- Created by **OSAMaiboxCreate(). OSAMailboxQCreate()** returns a reference that is used by other mailbox queue functions .
- An item is put on a mailbox queue with **OSAMailboxQSend()** . The inputs to **OSAMailboxQSend()** are a pointer to the item to put on the queue (i.e. a pointer to a 'void*' item) and a "timeout" value. The timeout definition likes event flags
- An item is removed from a mailbox queue by **OSAMailboxQRecv().** The inputs to **OSAMailboxQRecv()** are a pointer to a pointer to void and a timeout value
- The number of messages on a mailbox queue can be read with **OSAMailboxQPoll()**
- A mailbox queue should never be deleted when there are tasks waiting for

Timer Management

- The timer management services provide primitives to create, start, stop, delete, and get the status of timers
- **OSATimerCreate()** is used to create timers. It returns a reference that is used to identify the created timer.
- An inactive timer may be activated by **OSATimerStart()**. Its inputs are a pointer to a handler function that will be called each time the timer expires, a pointer to void that is passed as an argument to the handler function, the initial timer period in OS clock ticks and the timer reload value in OS clock ticks
- The **timer** expiry counter may be examined with **OSATimerGetStatus()**
- An active timer may be deactivated by **OSATimerStop()**
- A timer must be inactive before calling either **OSATimerStart()** or **OSATimerDelete().**

OS Clock

- The tick counter can be read by **OSAGetTicks()**

Memory management

- The memory API is malloc() and free()
- malloc() calls OsaMemAlloc()
- free() calls OsaMemFree()
- OSA memory API will invoke Nucleus memory service

Data Types

- The following are data types used by OSA:

Type	Definition
UINT8	Unsigned char
UINT16	Unsigned short
UINT32	Unsigned long
OSA_STATUS	Implementation dependant
OSATaskRef	Void *
OSAPoolRef	Void *
OSASemaRef	Void *
OSAMutexRef	Void *
OSATimerRef	Void *
OSAMsgQRef	Void *
OSAMaliboxQRef	Void *
OSAFlagRef	Void *

API Document

Thank You!