OS Abstraction Layer
Application Programming Interface

Document Number: SWRA194

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial ZigBee v1.0 release.	04/08/2005
1.1	Added note on NV initialization to NV Memory API introduction.	07/22/2005
1.2	Modified discussion of .the Task Management API.	08/25/2005
1.3	Changed logo on title page, changed copyright on page footer.	02/27/2006
1.4	Modified power management API.	11/27/2006
1.5	Deprecated osal_self() and osalTaskAdd()	12/18/2007
1.6	Added OSAL_Clock functions.	02/21/2008
1.7	Updated byte to uint8 and added Misc section.	04/01/2009
1.8	Updated NV item ID table and changed ZSUCCESS to SUCCESS.	04/09/2009
1.9	Added a chapter for Simple NV memory system.	08/14/2009
1.10	Added the osal_msg_find() and osal_start_reload_timer() API.	11/09/2009
1.11	Added osal_nv_delete(), osal_nv_len, osal_run_system(), osal_self()	06/04/2011
1.12	Updated osal_start_timerEx()	09/19/2011
1.13	Updated NV ID table for TIMAC.	03/09/2015

TABLE OF CONTENTS

1. Introduction

1.1 Purpose

The purpose of this document is to define the OS Abstraction Layer (OSAL) API. This API allows the software components of a TI stack product, such as Z-Stack™, RemoTI™, and BLE, to be written independently of the specifics of the operating system, kernel or tasking environment (including control loops or connect-to-interrupt systems).

1.2 Scope

This document enumerates all the function calls provided by the OSAL. The function calls are specified in sufficient detail to allow a programmer to implement them.

1.3 Acronyms

API	Application Programming Interface
BLE	Bluetooth Low Energy
NV	Non-Volatile
OSAL	Operating System (OS) Abstraction Layer
RF4CE	RF for Consumer Electronics
RemoTI	Texas Instruments RF4CE protocol stack
TIMAC	Texas Instruments IEEE 802.15.4 MAC layer
Z-Stack	Texas Instruments ZigBee protocol stack

2. API Overview

2.1 Overview

The OS abstraction layer is used to shield the TI stack software components from the specifics of the processing environment. It provides the following functionality in a manner that is independent of the processing environment.

1. Task registration, initialization, starting
2. Message exchange between tasks
3. Task synchronization
4. Interrupt handling
5. Timers
6. Memory allocation

3. Message Management API

3.1 Introduction

The message management API provides a mechanism for exchanging messages between tasks or processing elements with distinct processing environments (for example, interrupt service routines or functions called within a control loop).The functions in this API enable a task to allocate and de-allocate message buffers, send command messages to another task and receive reply messages.

3.2 osal_msg_allocate( )

3.2.1 Description

This function is called by a task to allocate a message buffer, the task/function will then fill in the message and call `osal_msg_send()` to send the message to another task. If the buffer cannot be allocated, msg_ptr will be set to NULL.

NOTE: Do not confuse this function with `osal_mem_alloc()`, this function is used to allocate a buffer to send messages between tasks [using `osal_msg_send()`]. Use `osal_mem_alloc()` to allocate blocks of memory.

3.2.2 Prototype

```
uint8 *osal_msg_allocate( uint16 len )
```

3.2.3 Parameter Details

`len` is the length of the message.

3.2.4 Return

The return value is a pointer to the buffer allocated for the message. A NULL return indicates the message allocation operation failed.

3.3 osal_msg_deallocate( )

3.3.1 Description

This function is used to de-allocate a message buffer. This function is called by a task (or processing element) after it has finished processing a received message.

3.3.2 Prototype

```
uint8 osal_msg_deallocate( uint8 *msg_ptr )
```

3.3.3 Parameter Details

`msg_ptr` is a pointer to the message buffer that needs to be de-allocated.

3.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	De-allocation Successful
INVALID_MSG_POINTER	Invalid message pointer
MSG_BUFFER_NOT_AVAIL	Buffer is queued

3.4 osal_msg_send( )

3.4.1 Description

The osal_msg_send function is called by a task to send a command or data message to another task or processing element. The destination_task identifier field must refer to a valid system task. The osal_msg_send() function will also set the SYS_EVENT_MSG event in the destination tasks event list.

3.4.2 Prototype

```
uint8 osal_msg_send(uint8 destination_task, uint8 *msg_ptr )
```

3.4.3 Parameter Details

`destination_task` is the ID of the task to receive the message.

`msg_ptr` is a pointer to the buffer containing the message. Msg_ptr must be a pointer to a valid message buffer allocated via osal_msg_allocate ().

3.4.4 Return

Return value is a 1-byte field indicating the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Message sent successfully
INVALID_MSG_POINTER	Invalid Message Pointer
INVALID_TASK	Destination_task is not valid

3.5 osal_msg_receive( )

3.5.1 Description

This function is called by a task to retrieve a received command message. The calling task must de-allocate the message buffer after processing the message using the osal_msg_deallocate() call.

3.5.2 Prototype

```
uint8 *osal_msg_receive(uint8 task_id )
```

3.5.3 Parameter Details

`task_id` is the identifier of the calling task (to which the message was destined).

3.5.4 Return

Return value is a pointer to a buffer containing the message or NULL if there is no received message.

3.6 osal_msg_find( )

3.6.1 Description

This function searches for an existing OSAL message matching the task_id and event parameters.

3.6.2 Prototype

```
osal_event_hdr_t *osal_msg_find(uint8 task_id, uint8 event)
```

3.6.3 Parameter Details

`task_id` is the identifier that the enqueued OSAL message must match.

`event` is the OSAL event id that the enqueued OSAL message must match.

3.6.4 Return

Return value is a pointer to the matching OSAL message on success or NULL on failure.

4. Task Synchronization API

4.1 Introduction

This API enables a task to wait for events to happen and return control while waiting. The functions in this API can be used to set events for a task and notify the task once any event is set.

4.2 osal_set_event( )

4.2.1 Description

This function is called to set the event flags for a task.

4.2.2 Prototype

```
uint8 osal_set_event(uint8 task_id, uint16 event_flag )
```

4.2.3 Parameter Details

`task_id` is the identifier of the task for which the event is to be set.

`event_flag` is a 2-byte bitmap with each bit specifying an event. There is only one system event (SYS_EVENT_MSG), the rest of the events/bits are defined by the receiving task.

4.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
INVALID_TASK	Invalid Task

5. Timer Management API

5.1 Introduction

This API enables the use of timers by internal (TI stack) tasks as well as external (Application level) tasks. The API provides functions to start and stop a timer. The timers can be set in increments of 1millisecond.

5.2 osal_start_timerEx( )

5.2.1 Description

This function is called to start a timer. When the timer expires, the given event bit will be set. The event will be set for the task specified by taskID. This timer is a one shot timer, meaning that when the timer expires it isn't reloaded.

5.2.2 Prototype

```
uint8 osal_start_timerEx( uint8 taskID, uint16 event_id,
                          uint32 timeout_value );
```

5.2.3 Parameter Details

`taskID` is the task ID of the task that is to get the event when the timer expires.

`event_id` is a user defined event bit. When the timer expires, the calling task will be notified (event).

`timeout_value` is the amount of time (in milliseconds) before the timer event is set.

5.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Timer Start Successful
NO_TIMER_AVAILABLE	Unable to start the timer

5.3 osal_start_reload_timer( )

5.3.1 Description

Call this function to start a timer that, when it expires, will set an event bit and reload the timeout value automatically. The event will be set for the task specified by taskID.

5.3.2 Prototype

```
uint8 osal_start_reload_timer( uint8 taskID, uint16 event_id,
                               uint32 timeout_value );
```

5.3.3 Parameter Details

`taskID` is the task ID of the task that is to get the event when the timer expires.

`event_id` is a user defined event bit. When the timer expires, the calling task will be notified (event).

`timeout_value` is the amount of time (in milliseconds) before the timer event is set. This value is reloaded into the timer when the timer expires.

5.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Timer Start Successful
NO_TIMER_AVAILABLE	Unable to start the timer

5.4 osal_stop_timerEx( )

5.4.1 Description

This function is called to stop a timer that has already been started. If successful, the function will cancel the timer and prevent the event associated with the timer.

5.4.2 Prototype

```
uint8 osal_stop_timerEx( uint8 task_id, uint16 event_id );
```

5.4.3 Parameter Details

`task_id` is the task for which to stop the timer.

`event_id` is the identifier of the timer that is to be stopped.

5.4.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Timer Stopped Successfully
INVALID_EVENT_ID	Invalid Event

5.5 osal_GetSystemClock( )

5.5.1 Description

This function is called to read the system clock

5.5.2 Prototype

```
uint32 osal_GetSystemClock( void );
```

5.5.3 Parameter Details

None.

5.5.4 Return

The system clock in milliseconds.

6. Interrupt Management API

6.1 Introduction

This API enables a task to interface with external interrupts. The functions in the API allow a task to associate a specific service routine with each interrupt. The interrupts can be enabled or disabled. Inside the service routine, events may be set for other tasks.

6.2 osal_int_enable( )

6.2.1 Description

This function is called to enable an interrupt. Once enabled, occurrence of the interrupt causes the service routine associated with that interrupt to be called.

6.2.2 Prototype

```
uint8 osal_int_enable( uint8 interrupt_id )
```

6.2.3 Parameter Details

`interrupt_id` identifies the interrupt to be enabled.

6.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Interrupt Enabled Successfully
INVALID_INTERRUPT_ID	Invalid Interrupt

6.3 osal_int_disable( )

6.3.1 Description

This function is called to disable an interrupt. When a disabled interrupt occurs, the service routine associated with that interrupt is not called.

6.3.2 Prototype

```
uint8 osal_int_disable( uint8 interrupt_id )
```

6.3.3 Parameter Details

`interrupt_id` identifies the interrupt to be disabled.

6.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Interrupt Disabled Successfully
INVALID_INTERRUPT_ID	Invalid Interrupt

7. Task Management API

7.1 Introduction

This API is used to add and manage tasks in the OSAL system. Each task is made up of an initialization function and an event processing function. OSAL calls `osalInitTasks()` [application supplied] to initialize the tasks and OSAL uses a task table (`const pTaskEventHandlerFn tasksArr[]`) to call the event processor for each task (also application supplied).

Example of a task table implementation:

```
const pTaskEventHandlerFn tasksArr[] =
{
  macEventLoop,
  nwk_event_loop,
  Hal_ProcessEvent,
  MT_ProcessEvent,
  APS_event_loop,
  ZDApp_event_loop,
};

const uint8 tasksCnt = sizeof( tasksArr ) / sizeof( tasksArr[0] );
```

Example of an osalInitTasks() implementation:

```
void osalInitTasks( void )
{
  uint8 taskID = 0;

  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  macTaskInit( taskID++ );
  nwk_init( taskID++ );
  Hal_Init( taskID++ );
  MT_TaskInit( taskID++ );
  APS_Init( taskID++ );
  ZDApp_Init( taskID++ );
}
```

7.2 osal_init_system( )

7.2.1 Description

This function initializes the OSAL system. The function must be called at startup prior to using any other OSAL function.

7.2.2 Prototype

```
uint8 osal_init_system( void )
```


7.2.3 Parameter Details

None.

7.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success

7.3 osal_start_system( )

7.3.1 Description

This function is the main loop function of the task system, repeatedly calling `osal_run_system()`, from an infinite loop. This function never returns. When using a different scheduler, it should call `osal_run_system()` directly and not use this function.

7.3.2 Prototype

```
void osal_start_system( void )
```

7.3.3 Parameter Details

None.

7.3.4 Return

None.

7.4 osal_run_system( )

7.4.1 Description

This function will make one pass through the OSAL taskEvents table and call the task_event_processor function for the first task that is found with at least one event pending. After an event is serviced, any remaining events will be returned back to the main loop for next time around. If there are no pending events (for all tasks), this function puts the processor into a Sleep mode.

7.4.2 Prototype

```
void osal_run_system( void )
```

7.4.3 Parameter Details

None.

7.4.4 Return

None.

7.5 osal_self( )

7.5.1 Description

This function returns the task ID of the currently active OSAL task, corresponding to the index of the task in the OSAL task table. This function can be used by an application to determine the OSAL task ID that it is running under. If called when no OSAL task is active, this function returns the value TASK_NO_TASK.

7.5.2 Prototype

```
uint8 osal_self( void )
```

7.5.3 Parameter Details

None.

7.5.4 Return

OSAL task ID of the currently active task.

RETURN VALUE	DESCRIPTION
0x00 – 0xFE	ID of active OSAL task
0xFF (TASK_NO_TASK)	No OSAL task is active

8. Memory Management API

8.1 Introduction

This API represents a simple memory allocation system. These functions allow dynamic memory allocation.

8.2 osal_mem_alloc( )

8.2.1 Description

This function is a simple memory allocation function that returns a pointer to a buffer (if successful).

8.2.2 Prototype

```
void *osal_mem_alloc( uint16 size );
```

8.2.3 Parameter Details

`size` – the number of bytes wanted in the buffer.

8.2.4 Return

A void pointer (which should be cast to the intended buffer type) to the newly allocated buffer. A NULL pointer is returned if there is not enough memory to allocate.

8.3 osal_mem_free( )

8.3.1 Description

This function frees the allocated memory to be used again. This only works if the memory had already been allocated with `osal_mem_alloc()`.

8.3.2 Prototype

```
void osal_mem_free( void *ptr );
```

8.3.3 Parameter Details

`ptr` – pointer to the buffer to be “freed”. Buffer must have been previously allocated with `osal_mem_alloc()`.

8.3.4 Return

None.

9. Power Management API

9.1 Introduction

This section describes the OSAL power management system. The system provides a way for the applications/tasks to notify OSAL when it is safe to turn off the receiver and external hardware, and put the processor in to sleep.

There are two functions to control the power management. The first, `osal_pwrmgr_device()`, is called to set the device level mode (power save or no power savings). Then, there is the task power state, each task can hold off the power manager from conserving power by calling `osal_pwrmgr_task_state( PWRMGR_HOLD )`. If a task "Holds" the power manager, it will need to call `osal_pwrmgr_task_state( PWRMGR_CONSERVE )` to allow the power manager to continue in power conserve mode.

By default, when the task is initialized, each task's power state is set to `PWRMGR_CONSERVE`, so if a task does not want to hold off power conservation (no change) it doesn't need to call `osal_pwrmgr_task_state()`.

In addition, by default, a battery-operated device will be in `PWRMGR_ALWAYS_ON` state until it joins the network, then it will change its state to `PWRMGR_BATTERY`. This means that if the device cannot find a device to join it will not enter a power save state. If you want to change this behavior, add `osal_pwrmgr_device( PWRMGR_BATTERY )` in your application's task initialization function or when your application stops/pauses the joining process.

The power manager will look at the device mode and the collective power state of all the tasks before going in to power conserve state.

9.2 osal_pwrmgr_init( )

9.2.1 Description

This function will initialize variables used by the power management system. IMPORTANT: Do not call this function, it is already called by osal_init_system().

9.2.2 Prototype

```
void osal_pwrmgr_init( void );
```

9.2.3 Parameter Details

None.

9.2.4 Return

None.

9.3 osal_pwrmgr_powerconserve( )

9.3.1 Description

This function is called to go into power down mode. IMPORTANT: Do not call this function, it is already called in the OSAL main loop [osal_start_system()].

9.3.2 Prototype

```
void osal_pwrmgr_powerconserve( void );
```


9.3.3 Parameter Details

None.

9.3.4 Return

None.

9.4 osal_pwrmgr_device( )

9.4.1 Description

This function is called on power- up or whenever the power requirements change (ex. Battery backed coordinator). This function sets the overall ON/OFF State of the device's power manager. This function should be called from a central controlling entity (like ZDO).

9.4.2 Prototype

```
void osal_pwrmgr_device( uint8 pwrmgr_device );
```

9.4.3 Parameter Details

`Pwrmgr_device` – changes or sets the power savings mode.

Type	Description
PWRMGR_ALWAYS_ON	With this selection, there is no power savings and the device is most likely on mains power.
PWRMGR_BATTERY	Turns power savings on.

9.4.4 Return

None.

9.5 osal_pwrmgr_task_state( )

9.5.1 Description

This function is called by each task to state whether or not this task wants to conserve power. The task will call this function to vote whether it wants the OSAL to conserve power or it wants to hold off on the power savings. By default, when a task is created, its own power state is set to conserve. If the task always wants to converse power, it does not need to call this function at all.

9.5.2 Prototype

```
uint8 osal_pwrmgr_task_state(uint8 task_id, uint8 state );
```


9.5.3 Parameter Details

`state` – changes a task's power state.

Type	Description
PWRMGR_CONSERVE	Turns power savings on, all tasks have to agree. This is the default state when a task is initialized.
PWRMGR_HOLD	Turns power savings off.

9.5.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
INVALID_TASK	Invalid Task

10. Non-Volatile Memory API

10.1 Introduction

This section describes the OSAL Non-Volatile (NV) memory system. The system provides a way for applications to store information into the device's memory persistently. It is also used by the stack for persistent storage of certain items required by the ZigBee specification. The NV functions are designed to read and write user-defined items consisting of arbitrary data types such as structures or arrays. The user can read or write an entire item or an element of the item by setting the appropriate offset and length. The API is independent of the NV storage medium and can be implemented for flash or EEPROM.

Each NV item has a unique ID. There is a specific ID value range for applications while some ID values are reserved or used by the stack or platform. If your application creates its own NV item, it must select an ID from the Application value range. See the table below.

VALUE	USER
0x0000	Reserved
0x0001 – 0x0020	OSAL
0x0021 – 0x0040	ZigBeePro: NWK TIMAC: MAC Sample App
0x0041 – 0x0060	APS
0x0061 – 0x0080	Security
0x0081 – 0x00B0	ZDO
0x00B1 – 0x00E0	Commissioning SAS
0x00E1 – 0x0100	Reserved
0x0101 – 0x01FF	Trust Center Link Keys
0x0201 – 0x0300	ZigBee-Pro: APS Links Keys ZigBee-RF4CE: network layer
0x0301 – 0x0400	ZigBee-Pro: Master Keys ZigBee-RF4CE: app framework
0x0401 – 0x0FFF	Application
0x1000 – 0xFFFF	Reserved

There are some important considerations when using this API:

1. These are blocking function calls and an operation may take several milliseconds to complete. This is especially true for NV write operations. In addition, interrupts may be disabled for several milliseconds. It is best to execute these functions at times when they do not conflict with other timing-critical operations. For example, a good time to write NV items would be when the receiver is turned off.
2. Try to perform NV writes infrequently. It takes time and power; also most flash devices have a limited number of erase cycles.
3. If the structure of one or more NV items changes, especially when upgrading from one version of a TI stack software to another, it is necessary to erase and re-initialize the NV memory. Otherwise, read and write operations on NV items that changed will fail or produce erroneous results.

10.2 osal_nv_item_init( )

10.2.1 Description

Initialize an item in NV. This function checks for the presence of an item in NV. If it does not exist, it is created and initialized with the data passed to the function, if any.

This function must be called for each item before calling `osal_nv_read()` or `osal_nv_write()`.

10.2.2 Prototype

```
uint8 osal_nv_item_init( uint16 id, uint16 len, void *buf );
```

10.2.3 Parameter Details

`id` – User-defined item ID.

`len` – Item length in bytes.

`*buf` – Pointer to item initialization data. If no initialization data, set to NULL.

10.2.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_ITEM_UNINIT	Success but item did not exist
NV_OPER_FAILED	Operation failed

10.3 osal_nv_read( )

10.3.1 Description

Read data from NV. This function can be used to read an entire item from NV or an element of an item by indexing into the item with an offset. Read data is copied into *buf.

10.3.2 Prototype

```
uint8 osal_nv_read( uint16 id, uint16 offset, uint16 len, void *buf );
```

10.3.3 Parameter Details

`id` – User-defined item ID.

`offset` – Memory offset into item in bytes.

`len` – Item length in bytes.

`*buf` – Data is read into this buffer.

10.3.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_OPER_FAILED	Operation failed

10.4 osal_nv_write( )

10.4.1 Description

Write data to NV. This function can be used to write an entire item to NV or an element of an item by indexing into the item with an offset.

10.4.2 Prototype

```
uint8 osal_nv_write( uint16 id, uint16 offset, uint16 len, void *buf );
```

10.4.3 Parameter Details

`id` – User-defined item ID.

`offset` – Memory offset into item in bytes.

`len` – Item length in bytes.

`*buf` – Data to write.

10.4.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_ITEM_UNINIT	Item is not initialized
NV_OPER_FAILED	Operation failed

10.5 osal_nv_delete( )

10.5.1 Description

Delete an item from NV. This function checks for the presence of the item in NV. If the item exists and its length matches the length provided in the function call, the item will be removed from NV.

10.5.2 Prototype

```
uint8 osal_nv_delete( uint16 id, uint16 len );
```


10.5.3 Parameter Details

`id` – User-defined item ID.

`len` – Item length in bytes.

10.5.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_ITEM_UNINIT	Item is not initialized
NV_BAD_ITEM_LEN	Incorrect length parameter
NV_OPER_FAILED	Operation failed

10.6 osal_nv_item_len( )

10.6.1 Description

Get length of an item in NV. This function returns the length of an NV item, if found, otherwise zero.

10.6.2 Prototype

```
uint16 osal_nv_item_len( uint16 id );
```

10.6.3 Parameter Details

`id` – User-defined item ID.

10.6.4 Return

Return value indicates the result of the operation.

RETURN VALUE	DESCRIPTION
0	NV item not found
1 - N	Length of NV item

10.7 osal_offsetof( )

10.7.1 Description

This macro calculates the memory offset in bytes of an element within a structure. It is useful for calculating the offset parameter used by NV API functions.

10.7.2 Prototype

```
osal_offsetof(type, member)
```

10.7.3 Parameter Details

`type` – Structure type.

`member` – Structure member.

11. Simple Non-Volatile Memory API

11.1 Introduction

This section describes the OSAL Simple Non-Volatile memory system. Like the OSAL NV memory system, the Simple NV memory system provides a way for applications to store information into the device's memory persistently. On the other hand, unlike the OSAL NV memory system, the Simple NV memory system provides much simpler API to drive the application code size and the stack code size down as well as the code size of the OSAL Simple NV system implementation. The user can read or write an entire item, but it cannot partially read or write the item.

Just like NV memory system, each NV item has a unique ID. There is a specific ID value range for applications while some ID values are reserved or used by the stack or platform. If your application creates its own NV items, it must select an ID from the Application value range. See the table below.

Note that the table might not apply to a certain custom stack build.

VALUE	USER
0x00	Reserved
0x01 – 0x6F	Reserved for ZigBee RF4CE network layer
0x70 – 0x7F	Reserved for ZigBee RF4CE application framework (RTI)
0x80 – 0xFE	Application
0xFF	Reserved

There are some important considerations when using this API:

1. These are blocking function calls and an operation may take several hundred milliseconds to complete. This is especially true for NV write operations. In addition, interrupts may be disabled for several milliseconds. It is best to execute these functions at times when they do not conflict with other timing-critical operations. For example, a good time to write NV items would be when the receiver is turned off.
2. Furthermore, the functions must not be called from an interrupt service routine, unless otherwise specified by a separate application note or release note of an implementation.
3. Try to perform NV writes infrequently. It takes time and power; also most flash devices have a limited number of erase cycles.
4. If the structure of one or more NV items changes, especially when upgrading from one version of a TI stack software to another, it is necessary to erase and re-initialize the NV memory. Otherwise, read and write operations on NV items that changed will fail or produce erroneous results.

11.2 osal_snv_read( )

11.2.1 Description

Read data from NV. This function can be used to read an entire item from NV. Read data is copied into *pBuf.

11.2.2 Prototype

```
uint8 osal_snv_read( osalSnvId_t id, osalSnvLen_t len, void *pBuf );
```


11.2.3 Parameter Details

`id` – User-defined item ID.

`len` – Item length in bytes.

`*pBuf` – Data is read into this buffer.

11.2.4 Return

Return value indicates the result of the operation. Note that an attempt to read an item, which was never written before, would result into the NV_OPER_FAILED return code.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_OPER_FAILED	Operation failed

11.3 osal_snv_write( )

11.3.1 Description

Write data to NV. This function can be used to write an entire item to NV.

11.3.2 Prototype

```
uint8 osal_snv_write( osalSnvId_t id, osalSnvLen_t len, void *pBuf );
```

11.3.3 Parameter Details

`id` – User-defined item ID.

`len` – Item length in bytes.

`*pBuf` – Data to write.

11.3.4 Return

Return value indicates the result of the operation. Note that it is allowed to write an item that was never before initialized into the NV system by other means.

RETURN VALUE	DESCRIPTION
SUCCESS	Success
NV_OPER_FAILED	Operation failed

12. OSAL Clock System

12.1 Introduction

This section describes the OSAL Clock system. The system provides a way to keep date and time for a device. This system will keep the number of seconds since 0 hrs, 0 minutes, 0 seconds on 1 January 2000 UTC. The following two data types/structures are used in this system (defined in OSAL_Clock.h):

```
// number of seconds since 0 hrs, 0 minutes, 0 seconds, on the
// 1st of January 2000 UTC
typedef uint32 UTCTime;

// To be used with
typedef struct
{
  uint8 seconds;  // 0-59
  uint8 minutes;  // 0-59
  uint8 hour;     // 0-23
  uint8 day;      // 0-30
  uint8 month;    // 0-11
  uint16 year;    // 2000+
} UTCTimeStruct;
```

You must enable the OSAL_CLOCK compiler flag to use this feature. **In addition**, this feature **does not** maintain time for sleeping devices.

12.2 osalTimeUpdate( )

12.2.1 Description

Called from osal_run_system() to update the time. This function reads the number of MAC 320usec ticks to maintain the OSAL Clock time. Do not call this function anywhere else.

12.2.2 Prototype

```
void osalTimeUpdate( void );
```

12.2.3 Parameter Details

None.

12.2.4 Return

None.

12.3 osal_setClock( )

12.3.1 Description

Call this function to initialize the device's time.

12.3.2 Prototype

```
void osal_setClock( UTCTime newTime );
```

12.3.3 Parameter Details

`newTime` – new time in seconds since 0 hrs, 0 minutes, 0 seconds, on 1 January 2000 UTC.

12.3.4 Return

None.

12.4 osal_getClock( )

12.4.1 Description

Call this function to retrieve the device's current time.

12.4.2 Prototype

```
UTCTime osal_getClock( void );
```

12.4.3 Parameter Details

None.

12.4.4 Return

Current time, in seconds since zero hrs, 0 minutes, 0 seconds, on 1 January 2000 UTC.

12.5 osal_ConvertUTCTime( )

12.5.1 Description

Call this function to convert `UTCTime` to `UTCTimeStruct`.

12.5.2 Prototype

```
void osal_ConvertUTCTime( UTCTimeStruct * tm, UTCTime secTime );
```

12.5.3 Parameter Details

secTime - time in seconds since 0 hrs, 0 minutes, 0 seconds, on 1 January 2000 UTC.

tm – pointer to time structure.

12.5.4 Return

None.

12.6 osal_ConvertUTCSecs( )

12.6.1 Description

Call this function to convert `UTCTimeStruct` to `UTCTime`.

12.6.2 Prototype

```
UTCTime osal_ConvertUTCSecs( UTCTimeStruct * tm );
```

12.6.3 Parameter Details

`tm` – pointer to time structure.

12.6.4 Return

Converted time, in seconds since 0 hrs, 0 minutes, 0 seconds, on 1 January 2000 UTC.

13. OSAL Misc

13.1 Introduction

This section describes miscellaneous OSAL functions that do not fit into the previous OSAL categories.

13.2 osal_rand( )

13.2.1 Description

This function returns a 16-bit random number.

13.2.2 Prototype

```
uint16 osal_rand( void );
```

13.2.3 Parameter Details

None.

13.2.4 Return

Returns a random number.

13.3 osal_memcmp( )

13.3.1 Description

Compare to memory sections.

13.3.2 Prototype

```
uint8 osal_memcmp( const void GENERIC *src1, const void GENERIC *src2,
unsigned int len );
```

13.3.3 Parameter Details

src1 – memory compare location 1.

src2 – memory compare location 2.

len – length of compare.

13.3.4 Return

TRUE - same, FALSE - different.

13.4 osal_memset( )

13.4.1 Description

Sets a buffer to a specific value.

13.4.2 Prototype

```
void *osal_memset( void *dest, uint8 value, int len );
```

13.4.3 Parameter Details

`dest` – memory buffer to set "value" to.

`value` – what to set each byte of "dest".

`len` – length to set "value" in "dest".

13.4.4 Return

Pointer to where in the buffer this function stopped.

13.5 osal_memcpy( )

13.5.1 Description

Copies one buffer to another buffer.

13.5.2 Prototype

```
void *osal_memcpy( void *dst, const void GENERIC *src, unsigned int len );
```

13.5.3 Parameter Details

`dst` – destination buffer.

`src` – source buffer.

`len` – length of copy.

13.5.4 Return

Pointer to end of destination buffer.

