



NATIONAL AERONAUTICS AND SPACE ADMINISTRATION  
GODDARD SPACE FLIGHT CENTER



## **Flight Software Branch (Code 582)**

OS Abstraction Layer Library, v3.4

Nicholas Yanchik, Code 582

Alan Cudmore, Code 582

Ezra Yeheskeli, Code 582/Raytheon

Date	Change Description:	Affected Pages
9/08/03	APC Merged Semaphore API in	All
9/09/03	APC Changed types and function names to match coding standard	All
9/10/03	APC Merged Memory and Port I/O API	All
9/15/03	APC Filled in details	All
9/16/03	APC Merged interrupt API	All
9/29/03	APC Broke spec into OS and Hardware API documents	All
10/08/03	APC Removed large parts of OS API, using POSIX instead	All
10/20/03	Modified document as a result of 10/19/2003 document review meeting. Combined the HW spec back into this document.	All
10/22/03	Initial release	All
10/23/03	Corrections to some typo	
11/10/03	Added OS_TaskDelay	
11/14/03	Added PCI Bus APIs	
04/14/04	Removed POSIX APIs, Added new Task and Queue APIs	
2/10/05	Updated doc to new format	All
2/11/05-2/14/05	Added Delete, GetIdByName functions, Updated Create functions	
6/15/05 -7/18/05	Updated return codes to match the project	
4/3/07	Update document to include v2.7 changes	All
2/13/08	Updated to include Memory range API, Loader/Symbol table API, Task Delete callback API, and Counting Semaphore API.	
9/5/08	Added Timer API	
3/10/2010	Removed Hardware API	
3/10/2010	Added OS_fsBytesFree API	All
11/15/2010	Added OS_FileOpenCheck API	All
05/24/2011	Added OS_CloseAllFiles, OS_CloseFileByName API. Added a more text on the Volume Table description.	All
12/05/2011	Added OS_rewinddir API	All

## Table of Contents

1 OS Abstraction Layer Introduction .....	6
2 Operating System API.....	7
2.1 Miscellaneous API.....	7
OS_API_Init .....	7
OS_printf .....	8
OS_Tick2Micros .....	9
OS_GetLocalTime .....	10
OS_SetLocalTime .....	11
OS_Milli2Ticks .....	12
2.2 Queue API.....	13
OS_QueueCreate.....	13
OS_QueueDelete.....	15
OS_QueueGet .....	16
OS_QueuePut.....	17
OS_QueueGetIdByName.....	18
OS_QueueGetInfo.....	19
2.3 Semaphore and Mutex API.....	20
OS_BinSemCreate .....	20
OS_BinSemDelete .....	21
OS_BinSemFlush.....	22
OS_BinSemGive.....	23
OS_BinSemTake.....	24
OS_BinSemTimedWait .....	25
OS_BinSemGetIdByName .....	26
OS_CountSemCreate .....	28
OS_CountSemDelete .....	29
OS_CountSemGive.....	30
OS_CountSemTake.....	31
OS_CountSemTimedWait .....	32
OS_CountSemGetIdByName .....	33
OS_MutSemCreate .....	35
OS_MutSemDelete .....	36
OS_MutSemGive.....	37
OS_MutSemTake.....	38
OS_MutSemGetIdByName .....	39
OS_MutSemGetInfo .....	40
2.4 Task Control API.....	41
OS_TaskCreate .....	41
OS_TaskDelete .....	42
OS_TaskInstallDeleteHandler .....	43
OS_TaskExit.....	44
OS_TaskDelay .....	45
OS_TaskSetPriority .....	46
OS_TaskRegister .....	47
OS_TaskGetId.....	48

OS_TaskGetIdByName .....	49
OS_TaskGetInfo .....	50
2.5 Dynamic Loader and Symbol API .....	51
OS_SymbolLookup .....	51
OS_SymbolTableDump .....	52
OS_ModuleLoad .....	53
OS_ModuleUnload .....	54
OS_ModuleInfo .....	55
2.6 Timer API .....	56
OS_TimerCreate .....	56
OS_TimerSet .....	58
OS_TimerDelete .....	59
OS_TimerGetIdByName .....	60
OS_TimerGetInfo .....	61
2.6 Network API .....	62
OS_NetworkGetID .....	62
OS_NetworkGetHostName .....	63
3 File System API .....	64
3.1 Introduction .....	64
3.2 File Descriptors in the OSAL .....	66
3.3 File API .....	68
OS_creat .....	68
OS_open .....	69
OS_close .....	70
OS_read .....	71
OS_write .....	72
OS_chmod .....	73
OS_stat .....	74
OS_lseek .....	75
OS_remove .....	76
OS_rename .....	77
OS_cp .....	78
OS_mv .....	79
OS_ShellOutputToFile .....	80
OS_FDGetInfo .....	81
OS_FileOpenCheck .....	82
OS_CloseAllFiles .....	83
OS_CloseFileByName .....	84
3.4 Directory API .....	85
OS_mkdir .....	85
OS_opendir .....	86
OS_closedir .....	87
OS_readdir .....	88
OS_rewinddir .....	89
OS_rmdir .....	90
3.5 Disk API .....	91

OS_mkfs .....	91
OS_rmfs .....	92
OS_initfs .....	93
OS_mount .....	94
OS_unmount .....	95
OS_GetPhysDriveName .....	96
OS_fsBlocksFree .....	97
OS_fsBytesFree .....	98
OS_chkfs .....	99
4 Interrupt/Exception API .....	100
4.1 System Interrupt API .....	100
OS_IntAttachHandler .....	101
OS_IntEnable .....	102
OS_IntDisable .....	103
OS_IntLock .....	104
OS_IntUnlock .....	105
OS_IntAck .....	106
4.2 System Exception API .....	107
OS_ExcAttachHandler .....	107
OS_ExcEnable .....	108
OS_ExcDisable .....	109
4.3 System FPU Exception API .....	110
OS_FPUExcAttachHandler .....	110
OS_FPUExcEnable .....	111
OS_FPUExcDisable .....	112

# 1 OS Abstraction Layer Introduction

The goal of this library is to promote the creation of portable and reusable real time embedded system software. Given the necessary OS abstraction layer implementations, the same embedded software should compile and run on a number of platforms ranging from spacecraft computer systems to desktop PCs.

The OS Application Program Interfaces (APIs) are broken up into three major sections: Real Time Operating System APIs, File System APIs, and Interrupt/Exception APIs. The Real Time Operating System APIs cover functionality such as Tasks, Queues, Semaphores, Interrupts, etc. The File System API abstracts the file systems that may be present on a system, and has the ability to simulate multiple embedded file systems on a desktop computer for testing. The Interrupt/Exception APIs are for configuring interrupt and exception handlers.

Major changes from the first version of this API include the ability to create objects “on the fly”, meaning they do not require a pre-defined ID in order to create them; instead they return the ID of the created object. Also the corresponding delete functions have been added, allowing the user to create and delete OS objects dynamically. Another change has been the removal of functions that were application specific. This release is aimed at generic embedded systems, not necessarily flight software applications. The addition of the file system API is another major addition, along with a method of simulating embedded file systems on a desktop computer. Finally, the parameters and error return codes have been cleaned up for consistency.

## 2 Operating System API

### 2.1 Miscellaneous API

#### ***OS\_API\_Init***

**Syntax:**

int32 OS\_API\_Init (void);

**Description:**

This function returns initializes the internal data structures of the OS Abstraction Layer. It must be called in the application startup code before calling any other OS routines.

**Parameters:**

none

**Returns:**

OS\_SUCCESS on a successful API init

OS\_ERROR ( or any value less than OS\_SUCCESS ) means the OSAL can not be initialized and therefore, additional OSAL calls should not be made.

**Restrictions:**

SYSTEM: This function should be called by the startup code before any other OS calls.

## ***OS\_printf***

### **Syntax:**

void OS\_printf (const char String, ...);

### **Description:**

This function provides a printing utility similar to printf. There is a #define OS\_UTILITY\_TASK\_ON which, in the VxWorks operating systems, creates a utility task to which all the parameters to OS\_printf are passed. The utility task then prints out the message. This is done so that print statements may be called from tasks that cannot block.

In the other OS's, (and if the #define is not present), OS\_printf provides a pass through to printf.

This function takes all the parameters and formatting options of printf.

### **Parameters:**

String:	The text portion of the print
ellipsis:	The other parameters to print

### **Returns:**

Nothing

### **Restrictions:**

TASK. This function may be called by all application tasks



## ***OS\_Tick2Micros***

**Syntax:**

int32 OS\_Tick2Micros (void);

**Description:**

This function returns the number of microseconds per operating system tick. It is used for computing the delay time in the operating system calls.

**Parameters:**

none

**Returns:**

Microseconds per operating system tick.

**Restrictions:**

TASK. This function may be called by all application tasks

## ***OS\_GetLocalTime***

**Syntax:**

```
int32 OS_GetLocalTime( OS_time_t * time_struct);
```

**Description:**

This function returns the local time of the machine it is on

**Parameters:**

time struct: A pointer to a OS\_time\_t structure that will hold the current time in seconds and milliseconds

**Returns:**

OS\_SUCCESS

**Restrictions:**

TASK. This function may be called by all application tasks

## ***OS\_SetLocalTime***

### **Syntax:**

```
int32 OS_SetLocalTime( OS_time_t * time_struct);
```

### **Description:**

This function allows the user to set the local time of the machine it is on

### **Parameters:**

time struct:     A pointer to a OS\_time\_t structure that holds the current time  
                  in seconds and milliseconds

### **Returns:**

OS\_SUCCESS

### **Restrictions:**

TASK. This function may be called by all application tasks

## ***OS\_Milli2Ticks***

### **Syntax:**

int32 OS\_Milli2Ticks ( uint32 milli\_seconds);

### **Description:**

This function returns the equivalent number of system clock ticks for the give period of time in milliseconds. The number of ticks is rounded up if necessary

### **Parameters:**

milli\_seconds:            Then number of milliseconds to convert to ticks

### **Returns:**

Number of ticks in the given period of milliseconds.

### **Restrictions:**

TASK. This function may be called by all application tasks

## 2.2 Queue API

### ***OS\_QueueCreate***

**Syntax:**

```
int32 OS_QueueCreate ( uint32 *queue_id, const char *queue_name, uint32
                      queue_depth, uint32 data_size, uint32 flags );
```

**Description:**

This is the function used to create a queue in the operating system. Depending on the underlying operating system, the memory for the queue will be allocated automatically or allocated by the code that sets up the queue. Queue names must be unique; if the name already exists this function fails. Names cannot be NULL.

**Parameters:**

- queue\_id :     an id to refer to a specific queue, is passed back to the caller
- queue\_name:    This is a character string to identify the queue. It is used only for display purposes. Example "INPUT\_QUEUE"
- queue\_depth:   This is the maximum number of elements that can be stored in the queue.
- data\_size:     This is the size of each data element on the queue. If the queue is setup to have variable sized items, it is the maximum size.
- flags:         This is for extra queue creation flags. The current flags are  
OS\_FIFO\_QUEUE – use the FIFO queue policy ( default )  
OS\_PRIORITY\_QUEUE – use priority based queue policy  
OS\_FIXED\_SIZE\_QUEUE  
OS\_VARIABLE\_SIZED\_QUEUE

**Returns:**

- OS\_INVALID\_POINTER if a pointer passed in is NULL  
OS\_ERR\_NAME\_TOO\_LONG if the name passed in is too long  
OS\_ERR\_NO\_FREE\_IDS if there are already the max queues created  
OS\_ERR\_NAME\_TAKEN if the name is already being used on another queue  
OS\_ERROR if the OS create call fails  
OS\_SUCCESS if success

**Restrictions:**

SYSTEM ( Software Bus ): This function is normally called by the communication layer software or "middleware" such as the Software Bus task. Application tasks should not

create queues directly unless there is a special requirement (i.e., to buffer data from a device )

## ***OS\_QueueDelete***

### **Syntax:**

```
int32 OS_QueueDelete ( uint32 queue_id );
```

### **Description:**

This is the function used to delete a queue in the operating system. This also frees the respective queue\_id to be used again when another queue is created.

### **Parameters:**

queue\_id : an id to refer to the specific queue to be deleted

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in does not exist  
OS\_ERROR if the OS call to delete the queue fails  
OS\_SUCCESS if success

### **Restrictions:**

SYSTEM ( Software Bus ): This function is normally called by the communication layer software or “middleware” such as the Software Bus task. Application tasks should not delete queues directly unless there is a special requirement (i.e., to buffer data from a device )

## ***OS\_QueueGet***

### **Syntax:**

```
int32 OS_QueueGet ( uint32 queue_id, void *data, uint32 size, uint32 *size_copied,  
                    int32 timeout);
```

### **Description:**

This function is used to retrieve a data item from an existing queue. The queue can be checked, pended on, or pended on with a timeout.

### **Parameters:**

- queue\_id : This is the queue ID from the queue that was created.
- data: This is a pointer to the buffer where the item gets copied.
- size: This is the maximum size of the data element that is being read. If it is a fixed size queue, then only the number of bytes corresponding to the initial queue size will be copied.
- size\_copied: This is the actual size of the data ( in bytes ) that was copied.
- timeout: This is the timeout value, in ticks for the queue get call. A value of OS\_PEND (or -1) will cause the call to block until a message arrives. A value of OS\_CHECK (or 0) will cause the call to return immediately if there is nothing on the queue.

### **Returns:**

OS\_ERR\_INVALID\_ID if the given ID does not exist  
OS\_INVALID\_POINTER if a pointer passed in is NULL  
OS\_QUEUE\_EMPTY if the Queue has no messages on it to be recieved  
OS\_QUEUE\_TIMEOUT if the timeout was OS\_PEND and the time expired  
OS\_QUEUE\_INVALID\_SIZE if the size copied from the queue was not correct  
OS\_SUCCESS if success

### **Restrictions:**

SYSTEM ( Software Bus ): This function is normally called by the communication layer software or “middleware” such as the Software Bus task.



## ***OS\_QueuePut***

### **Syntax:**

```
int32 OS_QueuePut ( uint32 queue_id, void *data, uint32 size, uint32 flags);
```

### **Description:**

This function is used to send data on an existing queue. The flags can be used to specify the behavior of the queue if it is full.

### **Parameters:**

- queue\_id:      This is the queue ID from the queue that was created.
- data:            This is a pointer to the data to be sent.
- size:            This is the size of the data element that is being sent.
- flags:           OS\_QUEUE\_BLOCK – specify that the task should block on a full queue during the send.
- OS\_QUEUE\_NONBLOCK – this is the default behavior where the call will return an error on a full queue.
- OS\_QUEUE\_URGENT – In the systems that support this feature, the message will be marked as high priority.

### **Returns:**

- OS\_ERR\_INVALID\_ID if the queue id passed in is not a valid queue
- OS\_INVALID\_POINTER if the data pointer is NULL
- OS\_QUEUE\_FULL if the queue cannot accept another message
- OS\_ERROR if the OS call returns an error
- OS\_SUCCESS if success

### **Restrictions:**

SYSTEM (Software Bus): This function is normally called by the communication layer software or “middleware” such as the Software Bus task.

## ***OS\_QueueGetIdByName***

### **Syntax:**

```
int32 OS_QueueGetIdByName (uint32 *queue_id, const char *queue_name);
```

### **Description:**

This function takes a queue name and looks for a valid queue with this name and returns the id of that queue.

### **Parameters:**

queue\_id:      The id of the queue, passed back to the caller.

queue\_name:    The name of the queue for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if the name or id pointers are NULL  
OS\_ERR\_NAME\_TOO\_LONG the name passed in is too long  
OS\_ERR\_NAME\_NOT\_FOUND the name was not found in the table  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_QueueGetInfo***

### **Syntax:**

```
int32 OS_QueueGetInfo (uint32 queue_id, OS_queue_prop_t *queue_prop);
```

### **Description:**

This function takes queue\_id, and looks it up in the OS table. It puts all of the information known about that queue into a structure pointer to by queue\_prop.

### **Parameters:**

queue\_id:      The id of the queue to look up.

queue\_prop:    A pointer to a structure to hold a queue's information  
That information includes:  
free: whether or not it's in use  
id: the queue's OS id  
creator: the task that created this queue  
name: the string name of the queue

### **Returns:**

OS\_INVALID\_POINTER if queue\_prop is NULL  
OS\_ERR\_INVALID\_ID if the ID given is not a valid queue  
OS\_SUCCESS if the info was copied over correctly

### **Restrictions:**

## 2.3 Semaphore and Mutex API

### ***OS\_BinSemCreate***

**Syntax:**

```
int32 OS_BinSemCreate(uint32 *sem_id, const char *sem_name,  
                      uint32 sem_initial_value, uint32 options);
```

**Description:**

This function creates a binary semaphore. Semaphore names must be unique; if the name already exists this function fails. Names cannot be NULL.

**Parameters:**

sem\_id: a unique semaphore identifier passed back to the caller

sem\_name: An arbitrary semaphore name.

sem\_initial\_value: the initial state of the semaphore.

options: optional flags to pass in. This is OS dependant

**Returns:**

OS\_INVALID\_POINTER if sen name or sem\_id are NULL  
OS\_ERR\_NAME\_TOO\_LONG if the name given is too long  
OS\_ERR\_NO\_FREE\_IDS if all of the semaphore ids are taken  
OS\_ERR\_NAME\_TAKEN if this is already the name of a binary semaphore  
OS\_SEM\_FAILURE if the OS call failed  
OS\_SUCCESS if success

**Restrictions:**

## ***OS\_BinSemDelete***

### **Syntax:**

```
int32 OS_BinSemDelete ( uint32 sem_id );
```

### **Description:**

This is the function used to delete a binary semaphore in the operating system. This also frees the respective sem\_id to be used again when another semaphore is created.

### **Parameters:**

sem\_id : an id to refer to the specific semaphore to be deleted

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid binary semaphore  
OS\_SEM\_FAILURE the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_BinSemFlush***

### **Syntax:**

int32 OS\_BinSemFlush(uint32 sem\_id);

### **Description:**

This function releases all the tasks waiting on the given semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system.

### **Returns:**

OS\_SEM\_FAILURE the semaphore was not previously initialized or is not  
in the array of semaphores defined by the system  
OS\_ERR\_INVALID\_ID if the id passed in is not a binary semaphore  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_BinSemGive***

**Syntax:**

```
int32 OS_BinSemGive(uint32 sem_id);
```

**Description:**

This function gives back a binary semaphore

**Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system.

**Returns:**

OS\_SEM\_FAILURE the semaphore was not previously initialized or is not  
in the array of semaphores defined by the system  
OS\_ERR\_INVALID\_ID if the id passed in is not a binary semaphore  
OS\_SUCCESS if success

**Restrictions:**

## ***OS\_BinSemTake***

### **Syntax:**

```
int32 OS_BinSemTake(uint32 sem_id);
```

### **Description:**

This function reserves a binary semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system.

### **Returns:**

OS\_SEM\_FAILURE the semaphore was not previously initialized  
or is not in the array of semaphores defined by the system  
OS\_ERR\_INVALID\_ID the Id passed in is not a valid binar semaphore  
OS\_SEM\_FAILURE if the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**



## ***OS\_BinSemTimedWait***

### **Syntax:**

int32 OS\_BinSemTimeWait(uint32 sem\_id , uint32 msec);

### **Description:**

This function reserves a binary semaphore with a timeout.

### **Parameters:**

sem_id:	an index identifying the semaphore in the an array of semaphores that where defined in the system
msec:	the timeout in milliseconds to wait

### **Returns:**

- OS\_SEM\_TIMEOUT if semaphore was not relinquished in time
- OS\_SUCCESS if success
- OS\_SEM\_FAILURE is the semaphore call returned an error
- OS\_ERR\_INVALID\_ID if the ID passed in is not a valid semaphore ID

### **Restrictions:**

## ***OS\_BinSemGetIdByName***

### **Syntax:**

```
int32 OS_BinSemGetIdByName (uint32 *sem_id, const char *sem_name);
```

### **Description:**

This function takes a binary semaphore name and looks for a valid binary semaphore with this name and returns the id of that semaphore.

### **Parameters:**

sem\_id:           The id of the semaphore, passed back to the caller.

sem\_name:        The name of the semaphore for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if semid or sem\_name are NULL pointers  
OS\_ERR\_NAME\_TOO\_LONG if the name given is too long to have been stored  
OS\_ERR\_NAME\_NOT\_FOUND if the name was not found in the table  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_BinSemGetInfo***

### **Syntax:**

```
int32 OS_BinSemGetInfo (uint32 sem_id, OS_mut_sem_prop_t *sem_prop);
```

### **Description:**

This function takes `sem_id`, and looks it up in the OS table. It puts all of the information known about that semaphore into a structure pointer to by `sem_prop`

### **Parameters:**

<code>sem_id</code> :	The id of the semaphore to look up.
<code>sem_prop</code> :	A pointer to a structure to hold a mutex's information That information includes: free: whether or not it's in use id: the mutex's OS id creator: the task that created this mutex name: the string name of the mutex

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid semaphore  
OS\_INVALID\_POINTER if the `sem_prop` pointer is null  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemCreate***

### **Syntax:**

```
int32 OS_CountSemCreate(uint32 *sem_id, const char *sem_name,  
                        uint32 sem_initial_value, uint32 options);
```

### **Description:**

This function creates a counting semaphore. Semaphore names must be unique; if the name already exists this function fails. Names cannot be NULL.

### **Parameters:**

**sem\_id:** a unique semaphore identifier passed back to the caller

**sem\_name:** An arbitrary semaphore name.

**sem\_initial\_value:** the initial state of the semaphore.

**options:** optional flags to pass in. This is OS dependant

### **Returns:**

OS\_INVALID\_POINTER if sem\_name or sem\_id are NULL  
OS\_ERR\_NAME\_TOO\_LONG if the name given is too long  
OS\_ERR\_NO\_FREE\_IDS if all of the semaphore ids are taken  
OS\_ERR\_NAME\_TAKEN if this is already the name of a counting semaphore  
OS\_SEM\_FAILURE if the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemDelete***

### **Syntax:**

```
int32 OS_CountSemDelete ( uint32 sem_id );
```

### **Description:**

This is the function used to delete a counting semaphore in the operating system. This also frees the respective sem\_id to be used again when another semaphore is created.

### **Parameters:**

sem\_id : an id to refer to the specific semaphore to be deleted

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid counting semaphore  
OS\_SEM\_FAILURE the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemGive***

### **Syntax:**

```
int32 OS_CountSemGive(uint32 sem_id);
```

### **Description:**

This function gives back a counting semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system.

### **Returns:**

OS\_SEM\_FAILURE the semaphore was not previously initialized or is not  
in the array of semaphores defined by the system  
OS\_ERR\_INVALID\_ID if the id passed in is not a counting semaphore  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemTake***

### **Syntax:**

```
int32 OS_CountSemTake(uint32 sem_id);
```

### **Description:**

This function reserves a counting semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system.

### **Returns:**

OS\_SEM\_FAILURE the semaphore was not previously initialized  
or is not in the array of semaphores defined by the system  
OS\_ERR\_INVALID\_ID the Id passed in is not a valid counting semaphore  
OS\_SEM\_FAILURE if the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemTimedWait***

### **Syntax:**

```
int32 OS_CountSemTimeWait(uint32 sem_id , uint32 msec);
```

### **Description:**

This function reserves a counting semaphore with a timeout.

### **Parameters:**

sem_id:	an index identifying the semaphore in the an array of semaphores that where defined in the system
msec:	the timeout in milliseconds to wait

### **Returns:**

- OS\_SEM\_TIMEOUT if semaphore was not relinquished in time
- OS\_SUCCESS if success
- OS\_ERR\_INVALID\_ID if the ID passed in is not a valid semaphore ID
- OS\_SEM\_FAILURE if the semaphore call returned an error//

### **Restrictions:**



## ***OS\_CountSemGetIdByName***

### **Syntax:**

```
int32 OS_CountSemGetIdByName (uint32 *sem_id, const char *sem_name);
```

### **Description:**

This function takes a counting semaphore name and looks for a valid counting semaphore with this name and returns the id of that semaphore.

### **Parameters:**

sem\_id:           The id of the semaphore, passed back to the caller.

sem\_name:        The name of the semaphore for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if semid or sem\_name are NULL pointers  
OS\_ERR\_NAME\_TOO\_LONG if the name given is too long to have been stored  
OS\_ERR\_NAME\_NOT\_FOUND if the name was not found in the table  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_CountSemGetInfo***

### **Syntax:**

```
int32 OS_CountSemGetInfo (uint32 sem_id, OS_mut_sem_prop_t *sem_prop);
```

### **Description:**

This function takes `sem_id`, and looks it up in the OS table. It puts all of the information known about that semaphore into a structure pointer to by `sem_prop`

### **Parameters:**

<code>sem_id</code> :	The id of the semaphore to look up.
<code>sem_prop</code> :	A pointer to a structure to hold a mutex's information That information includes: free: whether or not it's in use id: the mutex's OS id creator: the task that created this mutex name: the string name of the mutex

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid semaphore  
OS\_INVALID\_POINTER if the `sem_prop` pointer is null  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_MutSemCreate***

### **Syntax:**

int32 OS\_MutSemCreate(uint32 \*sem\_id, const char \*sem\_name, uint32 options);

### **Description:**

This function creates a mutex semaphore. Semaphore names must be unique; if the name already exists this function fails. Names cannot be NULL.

### **Parameters:**

sem\_id: a unique semaphore identifier passed back to the caller

sem\_name: An arbitrary semaphore name.

options: optional flags to pass in. This is OS dependant

### **Returns:**

OS\_INVALID\_POINTER if sem\_id or sem\_name are NULL

OS\_ERR\_NAME\_TOO\_LONG if the sem\_name is too long to be stored

OS\_ERR\_NO\_FREE\_IDS if there are no more free mutex Ids

OS\_ERR\_NAME\_TAKEN if there is already a mutex with the same name

OS\_SEM\_FAILURE if the OS call failed

OS\_INVALID\_SEM\_VALUE if initial value of semaphore is 0

OS\_SUCCESS if success

### **Restrictions:**

SYSTEM: This function should only be called by system code. This is the method that the Software Bus used to protect its global data. It might be better to rely on a mutex rather than turning off scheduling.

## ***OS\_MutSemDelete***

### **Syntax:**

```
int32 OS_MutSemDelete ( uint32 sem_id );
```

### **Description:**

This is the function used to delete a binary semaphore in the operating system. This also frees the respective sem\_id to be used again when another mutex is created.

### **Parameters:**

sem\_id : an id to refer to the specific semaphore to be deleted

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid mutex  
OS\_ERR\_SEM\_NOT\_FULL if the mutex is empty  
OS\_SEM\_FAILURE if the OS call failed  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_MutSemGive***

### **Syntax:**

int32 OS\_MutSemGive (uint32 sem\_id );

### **Description:**

This function releases a mutex semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system

### **Returns:**

OS\_SUCCESS if success  
OS\_SEM\_FAILURE if the semaphore was not previously initialized  
OS\_ERR\_INVALID\_ID if the id passed in is not a valid mutex

### **Restrictions:**

SYSTEM: This function should only be called by system code.

## ***OS\_MutSemTake***

### **Syntax:**

int32 OS\_MutSemTake (uint32 sem\_id );

### **Description:**

This function allocates a mutex semaphore

### **Parameters:**

sem\_id:            an index identifying the semaphore in the an array of semaphores  
                    that where defined in the system

### **Returns:**

OS\_SUCCESS if success

OS\_SEM\_FAILURE if the semaphore was not previously initialized or is  
not in the array of semaphores defined by the system

OS\_ERR\_INVALID\_ID the id passed in is not a valid mutex

### **Restrictions:**

SYSTEM: This function should only be called by system code.

## ***OS\_MutSemGetIdByName***

### **Syntax:**

```
int32 OS_MutSemGetIdByName (uint32 *sem_id, const char *sem_name);
```

### **Description:**

This function takes a mutex name and looks for a valid mutex semaphore with this name and returns the id of that semaphore.

### **Parameters:**

sem\_id:           The id of the semaphore, passed back to the caller.

sem\_name:        The name of the semaphore for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if semid or sem\_name are NULL pointers  
OS\_ERR\_NAME\_TOO\_LONG if the name given is too long to have been stored  
OS\_ERR\_NAME\_NOT\_FOUND if the name was not found in the table  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_MutSemGetInfo***

### **Syntax:**

```
int32 OS_MutSemGetInfo (uint32 sem_id, OS_mut_sem_prop_t *sem_prop);
```

### **Description:**

This function takes sem\_id, and looks it up in the OS table. It puts all of the information known about that mutex into a structure pointer to by sem\_prop

### **Parameters:**

sem_id:	The id of the mutex to look up.
sem_prop:	A pointer to a structure to hold a mutex's information That information includes: free: whether or not it's in use id: the mutex's OS id creator: the task that created this mutex name: the string name of the mutex

### **Returns:**

OS\_ERR\_INVALID\_ID if the id passed in is not a valid semaphore  
OS\_INVALID\_POINTER if the sem\_prop pointer is null  
OS\_SUCCESS if success

### **Restrictions:**



## 2.4 Task Control API

### ***OS\_TaskCreate***

**Syntax:**

```
int32 OS_TaskCreate(uint32 *task_id, const char *task_name,  
                    const void *function_pointer, const uint32 *stack_pointer,  
                    uint32 stack_size, uint32 priority, uint32 flags);
```

**Description:**

Creates a task and passes back the id of the task created. Task names must be unique; if the name already exists this function fails. Names cannot be NULL.

**Parameters:**

task_id:	a reference to the task just created, is passed back to the caller
task_name:	an arbitrary character string to identify the task by.
function_pointer:	an entry point to the task ( task Main routine )
stack_size:	The size of the stack to be allocated for the task
priority:	An integer between 1 and 255 specifying the new task's priority. 1 = highest, 255 = lowest.
flags:	optional flags to pass. Use the OS_FP_ENABLED flag to use floating point operations in tasks.

**Returns:**

OS\_INVALID\_POINTER if any of the necessary pointers are NULL  
OS\_ERR\_NAME\_TOO\_LONG if the name of the task is too long to be copied  
OS\_ERR\_INVALID\_PRIORITY if the priority is bad  
OS\_ERR\_NO\_FREE\_IDS if there can be no more tasks created  
OS\_ERR\_NAME\_TAKEN if the name specified is already used by a task  
OS\_ERROR if the operating system calls fail  
OS\_SUCCESS if success

**Restrictions:**

## ***OS\_TaskDelete***

### **Syntax:**

```
int32 OS_TaskDelete ( uint32 task_id );
```

### **Description:**

This function is used to delete a task in the operating system. This also frees the respective task\_id to be used again when another task is created.

### **Parameters:**

task\_id : an id to refer to the specific task to be deleted

### **Returns:**

OS\_ERR\_INVALID\_ID if the ID given to it is invalid

OS\_ERROR if the OS delete call fails

OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_TaskInstallDeleteHandler***

### **Syntax:**

```
int32 OS_TaskInstallDeleteHandler(void *function_pointer);
```

### **Description:**

This function is used to install a callback that is called when the task is deleted. The callback is called when OS\_TaskDelete is called with the task ID. A task delete handler is useful for cleaning up resources that a task creates, before the task is removed from the system.

### **Parameters:**

function\_pointer : The address of the callback function. The function should have the prototype : void functionname(void);

### **Returns:**

OS\_ERR\_INVALID\_ID if the ID given to it is invalid  
OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_TaskExit***

### **Syntax:**

void OS\_TaskDelete ( void );

### **Description:**

This function allows a task to delete itself (exit). It frees its task Id to be used again by another task. This function doesn't delete any resources used by the task.

### **Parameters:**

None

### **Returns:**

None

### **Restrictions:**

## ***OS\_TaskDelay***

### **Syntax:**

Int32 OS\_TaskDelay(uint32 millisecond );

### **Description:**

Causes the current thread to be suspended from execution for the period of millisecond.

### **Parameters:**

millisecond: time interval to delay.

### **Returns:**

OS\_ERROR if sleep fails

OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_TaskSetPriority***

### **Syntax:**

```
int OS_TaskSetPriority(uint32 task_id, uint32 new_priority);
```

### **Description:**

Sets the priority for the specified task.

### **Parameters:**

task\_id:        The predefined task ID. The task must be created.

new\_priority:   The new priority, between 1 and 255.

### **Returns:**

OS\_ERR\_INVALID\_ID if the ID passed to it is invalid  
OS\_\_ERR\_INVALID\_PRIORITY if the priority is greater than the max allowed  
OS\_ERROR if the OS call to change the priority fails  
OS\_SUCCESS if success

### **Restrictions:**

This function should be used in system software and special situations such as the software bus initialization code.

## ***OS\_TaskRegister***

**Syntax:**

```
int OS_TaskRegister(void);
```

**Description:**

Registers the task, performing application and OS specific initialization.  
This function should be called at the start of each task.

**Parameters:**

none

**Returns:**

OS\_ERR\_INVALID\_ID if there the specified ID could not be found  
OS\_ERROR if the OS call fails  
OS\_SUCCESS if success

**Restrictions:**

This function should be called at the start of each application task.

## ***OS\_TaskGetId***

**Syntax:**

Int32 OS\_TaskGetId (void);

**Description:**

This function returns a unique identification number for task/thread where this routine was called.

**Parameters:**

none

**Returns:**

Mission specific.

Task Id of the calling task

**Restrictions:**

TASK. This function may be called by all application tasks



## ***OS\_TaskGetIdByName***

### **Syntax:**

```
int32 OS_TaskGetIdByName (uint32 *task_id, const char *task_name);
```

### **Description:**

This function takes a task name and looks for a valid task with this name and returns the id of that task.

### **Parameters:**

task\_id:        The id of the task, passed back to the caller.

task\_name:     The name of the task for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if the pointers passed in are NULL  
OS\_ERR\_NAME\_TOO\_LONG if the name to be found is too long to begin with  
OS\_ERR\_NAME\_NOT\_FOUND if the name wasn't found in the table  
OS\_SUCCESS if SUCCESS

### **Restrictions:**

## ***OS\_TaskGetInfo***

### **Syntax:**

```
int32 OS_TaskGetInfo (uint32 task_id, OS_task_prop_t *task_prop);
```

### **Description:**

This function takes task\_id, and looks it up in the OS table. It puts all of the information known about that task into a structure pointer to by task\_prop

### **Parameters:**

task_id:	The id of the task to look up.
task_prop:	A pointer to a structure to hold a task's information That information includes: creator: the task that created this task stack_size: the size of the stack for this task priority: this task's current priority name: the string name of the task

### **Returns:**

OS\_ERR\_INVALID\_ID if the ID passed to it is invalid  
OS\_INVALID\_POINTER if the task\_prop pointer is NULL  
OS\_SUCCESS if it copied all of the relevant info over

### **Restrictions:**

## 2.5 Dynamic Loader and Symbol API

The Dynamic Loader and Symbol API are defined in OSAL 2.11, but not implemented. This API is intended to work with the vxWorks dynamic loader, the CEXP dynamic loader for RTEMS, and the dlopen/dlsym API on Linux, OSX, and Cygwin.

### ***OS\_SymbolLookup***

**Syntax:**

```
int32 OS_SymbolLookup (uint32 *SymbolAddress, char *SymbolName);
```

**Description:**

This function will lookup the address of a symbol.

**Parameters:**

\*SymbolAddress:     A pointer to the variable where the address of the symbol will be stored.

\*SymbolName:        The name of the symbol to look up.

**Returns:**

OS\_ERROR if the symbol is not found.

OS\_INVALID\_POINTER if one of the parameters are NULL.

OS\_SUCCESS if the symbol is found.

**Restrictions:**

**Some operating systems do not support symbol table lookup.**

## ***OS\_SymbolTableDump***

### **Syntax:**

int32 OS\_SymbolTableDump ( char \*filename, uint32 SizeLimit );

### **Description:**

This function dumps the system symbol table to the specified filename.

### **Parameters:**

- filename:        The full path/filename to save the symbol table.
- SizeLimit:       The maximum size in bytes to write to the file. This parameter is used to limit the amount of data that can be written to a filesystem. The symbol table file could be quite large, and could fill a RAM disk or other embedded storage device.

### **Returns:**

- OS\_SUCCESS if the symbol table was written to the file.  
OS\_INVALID\_POINTER if the filename is NULL  
OS\_FS\_ERR\_PATH\_INVALID if the filename is invalid  
OS\_ERROR if there was a problem writing the symbol table to the file.

### **Restrictions:**

## ***OS\_ModuleLoad***

### **Syntax:**

```
int32 OS_ModuleLoad ( uint32 *ModuleId, char *FileName,  
                      OS_ModuleInfo_t *OptLoadAddress );
```

### **Description:**

This function loads a new ELF object module into the operating system. This is intended for the loader in Traditional Real Time Operating Systems ( RTOSs ) such as vxWorks. The ELF object loader will load an “unlinked” object module into the system, resolve the external references, and enter it’s global symbols into the system symbol table for use. The dynamic loaders available include the vxWorks object loader and the RTEMS/Linux based CEXP. The desktop operating systems have a way of implementing this as well ( dlopen, dlsym on linux, OS X, cygwin )

### **Parameters:**

- \*ModuleId: A pointer to where the module ID will be stored.
- \*FileName: The path/filename of the module to load.
- \*OptLoadAddress: This is a pointer to an optional structure to specify the load information for the module. The structure contains address information for the Code Segment ( Text ), Initialized Data ( data ), and Uninitialized Data ( BSS ). This structure can be used to locate a module at these specific addresses ( if the underlying operating system supports it )

### **Returns:**

- OS\_SUCCESS if the load was successful.
- OS\_ERROR if there was a problem with the load.
- OS\_INVALID\_POINTER if one of the parameters is a NULL pointer
- OS\_ERR\_NO\_FREE\_IDS if the module table is full
- OS\_ERR\_NAME\_TAKEN if the module name is in use

### **Restrictions:**

## ***OS\_ModuleUnload***

### **Syntax:**

int32 OS\_ModuleUnload ( uint32 ModuleId );

### **Description:**

This function unloads the specified module from the system. Not all operating system module loaders support the unload function.

### **Parameters:**

ModuleId:     The ID of the module to unload.

### **Returns:**

OS\_SUCCESS if the unload was successful.

OS\_ERROR if there was a problem with unloading the module  
( will be expanded as the function is implemented )

### **Restrictions:**

## ***OS\_ModuleInfo***

### **Syntax:**

```
int32 OS_ModuleInfo ( uint32 ModuleId, OS_ModuleInfo_t *ModuleInfo);
```

### **Description:**

This function fills out the OS\_ModuleInfo\_t structure with data about the module identified by ModuleId. The OS\_ModuleInfo\_t structure contains the following fields: CodeAddress, CodeSize, DataAddress, DataSize, BSSAddress, BSSSize, and Flags. The primary use is to obtain the location of the dynamically loaded module.

### **Parameters:**

ModuleId:     The ID of the module to unload.

\*ModuleInfo: A pointer to the structure where the module information will be stored.

### **Returns:**

OS\_SUCCESS if the module information was retrieved.

OS\_ERROR if there was a problem with getting the module information, or the module is invalid.

( will be expanded as the function is implemented )

### **Restrictions:**

## 2.6 Timer API

The timer API is a generic interface to the OS timer facilities. It is implemented using the POSIX timers on Linux and vxWorks and the native timer API on RTEMS. The OS X version is not complete, and will have to be simulated, since the POSIX timer API is not supported on OS X. Cygwin support is TBD. The number of timers supported is controlled by the configuration parameter `OS_MAX_TIMERS`.

### ***OS\_TimerCreate***

#### **Syntax:**

```
int32 OS_TimerCreate (uint32 *timer_id, const char *timer_name, uint32
*clock_accuracy, OS_TimerCallback_t callback_ptr)
```

#### **Description:**

This function creates a new timer and associates a callback routine.

#### **Parameters:**

`*timer_id`      A pointer to the variable where the OSAL ID of the new timer will be stored.

`*timer_name`: The name of the timer to be created.

`*clock_accuracy`: A pointer to the variable where the accuracy of the timer is stored. The accuracy is in microseconds. This parameter will give an indication of the minimum clock resolution of the timer.

`callback_ptr`: The function pointer of the timer callback or ISR that will be called by the timer. The user's function is declared as follows:  
`void timer_callback(uint32 timer_id)`  
Where the `timer_id` is passed in to the function by the OSAL.

#### **Returns:**

`OS_INVALID_POINTER` if one of the pointer parameters is zero.  
`OS_ERR_NAME_TOO_LONG` if the name parameter is too long.  
`OS_ERR_NAME_TAKEN` if the name is already in use by another timer.  
`OS_ERR_NO_FREE_IDS` if all of the timers are already allocated.  
`OS_TIMER_ERR_INVALID_ARGS` if the callback pointer is zero.  
`OS_TIMER_ERR_UNAVAILABLE` if the timer cannot be created.  
`OS_SUCCESS` if the timer has been created successfully.

#### **Restrictions:**



Depending on the OS, the `timer_callback` function may be similar to an interrupt service routine. System calls the cause the code to block are generally not supported.

## ***OS\_TimerSet***

### **Syntax:**

```
int32 OS_TimerSet ( uint32 timer_id, uint32 start_msec, uint32 interval_msec );
```

### **Description:**

This function programs the timer with a start time and an optional interval time. The start time is the time in microseconds when the user callback function will be called. If the interval time is non-zero, the timer will be reprogrammed with that interval in microseconds to call the user callback function periodically. If the start time is zero, the timer will be disabled.

### **Parameters:**

- timer\_id:        The ID of the timer to program.
- start\_msec:      The start time in microseconds of when to first call the user callback function. If this parameter is zero, the timer will be disabled.
- interval\_msec:   The interval time in microseconds of what the periodic timer will be programmed for. The user callback function will be called every “interval\_msec” seconds after the initial start time. If this parameter is zero, then the timer will only call the user callback function once after the start\_msec time.

### **Returns:**

- OS\_ERR\_INVALID\_ID if the timer\_id is not valid.
- OS\_TIMER\_ERR\_INTERNAL if there was an error programming the OS timer.
- OS\_SUCCESS if the timer was programmed successfully.

### **Restrictions:**

The resolution of the times specified is limited to the clock accuracy returned in the OS\_TimerCreate call. If the times specified in the start\_msec or interval\_msec parameters are less than the accuracy, they will be rounded up to the accuracy of the timer.

## ***OS\_TimerDelete***

### **Syntax:**

int32 OS\_TimerDelete ( uint32 timer\_id )

### **Description:**

This function deletes the specified timer.

### **Parameters:**

ModuleId:     The ID of the timer to delete.

### **Returns:**

OS\_SUCCESS if the deletion of the timer was successful.  
OS\_ERR\_INVALID\_ID if the timer\_id is invalid.  
OS\_TIMER\_ERR\_INTERNAL if there was a problem deleting the timer in the host OS.

### **Restrictions:**

## ***OS\_TimerGetIdByName***

### **Syntax:**

int32 OS\_TimerGetIdByName (uint32 \*timer\_id, const char \*timer\_name);

### **Description:**

This function takes a timer name and looks for a valid timer with this name and returns the id of that timer.

### **Parameters:**

\*timer\_id:     The id of the timer, passed back to the caller.

timer\_name:    The name of the timer for which the id is being sought

### **Returns:**

OS\_INVALID\_POINTER if the name or id pointers are NULL

OS\_ERR\_NAME\_TOO\_LONG the name passed in is too long

OS\_ERR\_NAME\_NOT\_FOUND the name was not found in the table

OS\_SUCCESS if success

### **Restrictions:**

## ***OS\_TimerGetInfo***

### **Syntax:**

```
int32 OS_TimerGetInfo (uint32 timer_id, OS_timer_prop_t *timer_prop);
```

### **Description:**

This function takes timer\_id, and looks it up in the OS table. It puts all of the information known about that timer into a structure pointer to by timer\_prop.

### **Parameters:**

timer\_id:        The id of the timer to look up.

timer\_prop:     A pointer to a structure to hold a timer's information  
                 That information includes:  
                 creator: the OS task ID of the task that created this timer  
                 name: the string name of the timer  
                 start\_time: the start time in microseconds, if any  
                 interval\_time: the interval time in microseconds, if any  
                 accuracy: the accuracy of the timer in microseconds

### **Returns:**

OS\_INVALID\_POINTER if timer\_prop pointer is NULL  
OS\_ERR\_INVALID\_ID if the ID given is not a valid timer  
OS\_SUCCESS if the info was copied over correctly

### **Restrictions:**

## 2.6 Network API

### ***OS\_NetworkGetID***

**Syntax:**

```
int32 OS_NetworkGetID(void);
```

**Description:**

Returns the network ID similar to the unix call “gethostid”.

**Parameters:**

none.

**Returns:**

OS\_ERROR if the operating system calls fail  
OS\_SUCCESS if success

**Restrictions:**

## ***OS\_NetworkGetHostName***

**Syntax:**

```
int32 OS_NetworkGetHostName(char *host_name,  
                             uint32 name_len);
```

**Description:**

Returns the network name of the system.

**Parameters:**

none.

**Returns:**

OS\_ERROR if the operating system calls fail  
OS\_INVALID\_POINTER if the host\_name pointer is NULL  
OS\_SUCCESS if success

**Restrictions:**

## 3 File System API

### 3.1 Introduction

The File System API is a thin wrapper around a selection of POSIX file APIs. In addition the File System API presents a common directory structure and volume view regardless of the underlying system type. For example, vxWorks uses MS-DOS style volume names and directories. For example, a vxWorks RAM disk might have the volume “RAM:0”. With this File System API, volumes are represented as Unix-style paths where each volume is mounted on the root file system:

- RAM:0/file1.dat becomes /mnt/ram/file1.dat
- FL:0/file2.dat becomes /mnt/fl/file2.dat

This abstraction allows the applications to use the same paths regardless of the implementation and it also allows file systems to be simulated on a desktop system for testing. On a desktop Linux system, the file system abstraction can be set up to map virtual devices to a regular directory. This is accomplished through the **OS\_mkfs** call, **OS\_mount** call, and a BSP specific volume table that maps the virtual devices to real devices or underlying file systems.

In order to make this file system volume abstraction work, a “Volume Table” needs to be provided in the Board Support Package of the application. The table has the following fields:

- **Device Name:** This is the name of the virtual device that the Application uses. Common names are “ramdisk1”, “flash1”, or “volatile1” etc. But the name can be any unique string.
- **Physical Device Name:** This is an implementation specific field. For vxWorks it is not needed and can be left blank. For a File system based implementation, it is the “mount point” on the root file system where all of the volume will be mounted. A common place for this on Linux could be a user’s home directory, “/tmp”, or even the current working directory “.”. In the example of “/tmp” all of the directories created for the volumes would be under “/tmp” on the Linux file system. For a real disk device in Linux, such as a RAM disk, this field is the device name “/dev/ram0”.
- **Volume Type:** This field defines the type of volume. The types are: **FS\_BASED** which uses the existing file system, **RAM\_DISK** which uses a **RAM\_DISK** device in vxWorks, RTEMS, or Linux, **FLASH\_DISK\_FORMAT** which uses a flash disk that is to be formatted before use, **FLASH\_DISK\_INIT** which uses a flash disk with an existing format that is just to be initialized before it’s use, **EEPROM** which is for an EEPROM or PROM based system.
- **Volatile Flag:** This flag indicates that the volume or disk is a volatile disk (RAM disk ) or a non-volatile disk, that retains its contents when the system is rebooted. This should be set to **TRUE** or **FALSE**.



- **Free Flag:** This is an internal flag that should be set to FALSE or zero.
- **Is Mounted Flag:** This is an internal flag that should be set to FALSE or zero. Note that a “pre-mounted” FS\_BASED path can be set up by setting this flag to one.
- **Volume Name:** This is an internal field and should be set to a space character “ ”.
- **Mount Point Field:** This is an internal field and should be set to a space character “ ”.
- **Block Size Field:** This is used to record the block size of the device and does not need to be set by the user.

### Example Volume Tables:

1. A volume table for vxWorks with a RAM disk and a FLASH disk:

```
OS_VolumeInfo_t OS_VolumeTable [NUM_TABLE_ENTRIES] =
{
{" /ramdev0", " ", RAM_DISK, TRUE, TRUE, FALSE, " ", " ", 0 },
{" /eedev0", " ", ATA_DISK, FALSE, TRUE, FALSE, " ", " ", 0 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 }
};
```

2. A volume table for Linux using the host disk to simulate the file systems:

```
OS_VolumeInfo_t OS_VolumeTable [NUM_TABLE_ENTRIES] =
{
{" /ramdev0", " /tmp", FS_BASED, TRUE, FALSE, FALSE, " ", " ", 0 },
{" /eedev0", " /tmp", FS_BASED, FALSE, TRUE, FALSE, " ", " ", 0 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 }
};
```

3. A volume table for RTEMS, which has a RAM disk and a pre-mounted path to an existing directory:

```
OS_VolumeInfo_t OS_VolumeTable [NUM_TABLE_ENTRIES] =
{
{" /ramdev0", " /dev/nvda", RAM_DISK, TRUE, TRUE, FALSE, " ", " ", 512 },
{" /eedev0", " /eeprom", FS_BASED, FALSE, FALSE, TRUE, "CF", " /eeprom", 512 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 },
{"unused", "unused", FS_BASED, TRUE, TRUE, FALSE, " ", " ", 0 }
};
```

Note that in the RTEMS case, the RAM disk is created using the RTEMS Non Volatile disk device ( /dev/nvda ). The OSAL currently uses the RTEMS Non Volatile disk as a RAM disk type. In the future the OSAL may support the regular RAM disk or both. The “pre-mounted” path in the second table entry is for an EEPROM drive. Since RTEMS has a Unix style root file system, the OSAL currently does not use virtual paths. The OSAL path is equivalent to the RTEMS path. In other words, the OSAL virtual path of “/eeprom/my-dir/file.dat” maps to “/eeprom/my-dir/file.dat” on the actual RTEMS file system.

Example Code to initialize the file systems in the generic Application code regardless of the implementation:

```

/*
** Init the Non-volatile device
*/
RetStatus = OS_mkfs(0, "/eedev0", "CF", 0, 0 );
if ( RetStatus != OS_SUCCESS )
{
    printf("Error Initializing Non-Volatile(FLASH) Volume\n");
}

RetStatus = OS_mount("/eedev0", "/cf");
if ( RetStatus != OS_SUCCESS )
{
    printf("Error Mounting Non-Volatile(FLASH) Volume\n");
}

/*
** Create the Volatile, or RAM disk device
*/
RetStatus = OS_mkfs(0, "/ramdev0", "RAM", 512, 2048 );
if ( RetStatus != OS_SUCCESS )
{
    printf("Error Initializing Volatile(RAM) Volume\n");
}

RetStatus = OS_mount("/ramdev0", "/ram");
if ( RetStatus != OS_SUCCESS )
{
    printf("Error Mounting Volatile(RAM) Volume\n");
}

```

## 3.2 File Descriptors in the OSAL

The OSAL uses abstracted file descriptors. This means that the file descriptors passed back from the OS\_open and OS\_creat calls will only work with other OSAL OS\_\* calls. The reasoning for this is as follows:

Because the OSAL now keeps track of all file descriptors, OSAL specific information can be associated with a specific file descriptor in an OS independent way. For instance, the path of the file that the file descriptor points to can be easily retrieved. Also, the OSAL task ID of the task that opened the file can also be retrieved easily. Both of these pieces of information are very useful when trying to determine statistics for a task, or the entire system. This information can all be retrieved with a single API, OS\_FDGetInfo.

Realizing that we cannot provide all of the file system calls that everyone would need, we also provide the underlying OS's file descriptor for any valid OSAL file descriptor. This way, you can manipulate the underlying file descriptor as needed.

There are some small drawbacks with the OSAL file descriptors. Because the related information is kept in a table., there is a `#define` called `OS_MAX_NUM_OPEN_FILES` that defines the maximum number of file descriptors available. This is a configuration parameter, and can be changed to fit your needs.

Also, if you open or create a file *not* using the OSAL calls (`OS_open` or `OS_creat`) then none of the other `OS_*` calls that accept a file descriptor as a parameter will work (the results of doing so are undefined). Therefore, if you open a file with the underlying OS's open call, you must continue to use the OS's calls until you close the file descriptor. Be aware that by doing this your software may no longer be OS agnostic.

## 3.3 File API

### ***OS\_creat***

**Syntax:**

int32 OS\_creat (const char \*path, int32 access);

**Description:**

Creates a file specified by const char \*path, with read/write permissions by access. The file is also automatically opened by the OS\_creat call.

**Parameters:**

path:	The absolute pathname of the file to be created.
access:	The permissions with which to open a file. Options include OS_READ_ONLY, OS_WRITE_ONLY or OS_READ_WRITE.

**Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if path is NULL  
OS\_FS\_PATH\_TOO\_LONG if path exceeds the maximum number of chars  
OS\_FS\_ERR\_NAME\_TOO\_LONG if the name of the file is too long  
OS\_FS\_ERROR if permissions are unknown or OS call fails  
OS\_FS\_ERR\_NO\_FREE\_FDS if there are no free file descriptors left in the OSAL's file descriptor table  
A file descriptor to refer to the file while it is open.

**Restrictions:**

## ***OS\_open***

### **Syntax:**

int32 OS\_open (const char \*path, int32 access, uint32 mode);

### **Description:**

This function opens a file specified by path with permissions as granted by access. Mode is unused.

### **Parameters:**

path:	The absolute pathname of the file to be opened.
access:	The permissions with which to open a file. Options include OS_READ_ONLY, OS_WRITE_ONLY or OS_READ_WRITE.
mode:	unused.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if path is NULL  
OS\_FS\_ERR\_PATH\_TOO\_LONG if path exceeds the max number of chars  
OS\_FS\_ERR\_NAME\_TOO\_LONG if the name of the file is too long  
OS\_FS\_ERROR if permissions are unknown or OS call fails  
OS\_FS\_ERR\_NO\_FREE\_FDS if there are no free file descriptors left in the OSAL's file descriptor table  
A file descriptor if success

### **Restrictions:**

## ***OS\_close***

### **Syntax:**

int32 OS\_close ( int32 filedes);

### **Description:**

This function will close the file pointed to by filedes.

### **Parameters:**

filedes:           A positive integer that points to an entry in a file descriptor table.  
It is used to refer to a file when it is open.

### **Returns:**

OS\_FS\_ERROR if file descriptor could not be closed  
OS\_FS\_SUCCESS if success

### **Restrictions:**

## ***OS\_read***

### **Syntax:**

int32 OS\_read (int32 filedes, void\* buffer, uint32 nbytes);

### **Description:**

This function will read nbytes bytes of the file described by filedes and put the read bytes into buffer.

### **Parameters:**

- |          |   |
|----------|---|
| filedes: | A positive integer that points to an entry in a file descriptor table. It is used to refer to a file when it is open. |
| buffer:  | A pre-allocated section of memory used to store the read contents of the file   |
| nbytes:  | The number of bytes to be read from the file  |

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if buffer is a null pointer  
OS\_FS\_ERROR if OS call failed  
The number of bytes read if success

### **Restrictions:**

## ***OS\_write***

### **Syntax:**

int32 OS\_write (int32 filedes, void\* buffer, uint32 nbytes);

### **Description:**

This function will read nbytes bytes of the file described by filedes and put the read bytes into buffer.

### **Parameters:**

- |          |   |
|----------|---|
| filedes: | A positive integer that points to an entry in a file descriptor table. It is used to refer to a file when it is open. |
| buffer:  | A pre-allocated section of memory used to store the data to be written to the file                                    |
| nbytes:  | The maximum number of bytes to copy to the file   |

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if buffer is NULL  
OS\_FS\_ERROR if OS call failed  
The number of bytes written if success

### **Restrictions:**



## ***OS\_chmod***

**Syntax:**

int32 OS\_read (const char \*path, uint32 access);

**Description:**

This function is unimplemented at this time.

**Parameters:****Returns:**

OS\_FS\_ERR\_UNIMPLEMENTED

**Restrictions:**

## ***OS\_stat***

### **Syntax:**

```
int32 OS_stat (const char *path, os_fstat_t *filestats);
```

### **Description:**

This function will fill an `os_fs_stat_t` structure with information about the file specified by path.

### **Parameters:**

path:	The absolute path to the file to get information about.
filestats:	a pointer to a <code>os_fs_stat_t</code> where the information will be stored.

### **Returns:**

- OS\_FS\_ERR\_INVALID\_POINTER if path or filestats is NULL
- OS\_FS\_ERR\_PATH\_TOO\_LONG if the path is too long to be stored locally
- OS\_FS\_ERROR if the OS call failed
- OS\_FS\_SUCCESS if success

### **Restrictions:**

## ***OS\_lseek***

### **Syntax:**

int32 OS\_lseek (int32 filedes, int32 offset, uint32 whence);

### **Description:**

This function will move the read/write pointer of a file to filedes to offset.

### **Parameters:**

- |          |   |
|----------|---|
| filedes: | A positive integer that points to an entry in a file descriptor table. It is used to refer to a file when it is open.   |
| offset:  | The number of bytes to offset the read/write pointer from its position pointed to by whence.  |
| whence:  | Tells offset where to begin offsetting. Has three values:<br>OS SEEK SET – start at the beginning of the file<br>OS SEEK CUR – start at the current read/write pointer<br>OS SEEK END – start at the then of the file |

### **Returns:**

the new offset from the beginning of the file  
OS\_FS\_ERROR if OS call failed

### **Restrictions:**

## ***OS\_remove***

### **Syntax:**

int32 OS\_remove (const char \*path);

### **Description:**

This function removes the file specified by path from the drive.

### **Parameters:**

path:                   The absolute path to the file to be removed

### **Returns:**

OS\_FS\_SUCCESS if the driver returns OK  
OS\_FS\_ERROR if there is no device or the driver returns error  
OS\_FS\_ERR\_INVALID\_POINTER if path is NULL  
OS\_FS\_ERR\_PATH\_TOO\_LONG if path is too long to be stored locally  
OS\_FS\_ERR\_NAME\_TOO\_LONG if the name of the file to remove is too long  
to be stored locally

### **Restrictions:**

## ***OS\_rename***

### **Syntax:**

```
int32 OS_rename(const char *old, const char *new);
```

### **Description:**

This function renames the specified file `old` to a new name `new`.

### **Parameters:**

<code>old:</code>	The absolute path to the file to be renamed.
<code>new:</code>	The new absolute path of the file.

### **Returns:**

- OS\_FS\_SUCCESS if the rename works
- OS\_FS\_ERROR if the file could not be opened or renamed
- OS\_FS\_ERR\_INVALID\_POINTER if `old` or `new` are NULL
- OS\_FS\_ERR\_PATH\_TOO\_LONG if the paths given are too long to be stored locally
- OS\_FS\_ERR\_NAME\_TOO\_LONG if the new name is too long to be stored locally

### **Restrictions:**

Note that there seems to be a bug in the RTEMS version. During testing, an `OS_rename` call would fail, but a subsequent call to `OS_rename`, which depended on the first, passed. If the first call is commented out, the second will fail.

## ***OS\_cp***

### **Syntax:**

int32 OS\_cp(const char \* src, const char \*dest);

### **Description:**

This function copies the specified file *src* to a new file *dest*.

### **Parameters:**

src:	The absolute path to the file to be copied.
dest:	The new absolute path of the file.

### **Returns:**

OS\_FS\_SUCCESS if the copy works  
OS\_FS\_ERROR if the file could not be copied.  
OS\_FS\_ERR\_INVALID\_POINTER if *src* or *dest* are NULL  
OS\_FS\_ERR\_PATH\_TOO\_LONG if the paths given are too long to be stored locally  
OS\_FS\_ERR\_NAME\_TOO\_LONG if the new name is too long to be stored locally

### **Restrictions:**

## ***OS\_mv***

### **Syntax:**

```
int32 OS_mv(const char * src, const char *dest);
```

### **Description:**

This function moves the specified file *src* to a new file *dest*.

### **Parameters:**

src:	The absolute path to the file to be moved.
dest:	The new absolute path of the file.

### **Returns:**

- OS\_FS\_SUCCESS if the move works
- OS\_FS\_ERROR if the file could not be moved
- OS\_FS\_ERR\_INVALID\_POINTER if *src* or *dest* are NULL
- OS\_FS\_ERR\_PATH\_TOO\_LONG if the paths given are too long to be stored locally
- OS\_FS\_ERR\_NAME\_TOO\_LONG if the new name is too long to be stored locally

### **Restrictions:**

## ***OS\_ShellOutputToFile***

### **Syntax:**

int32 OS\_ShellOutputToFile (char \* Cmd, int32 OS\_fd);

### **Description:**

This function passes a command to the 'shell' of the underlying operating system. It directs the output from the command to the file specified by OS\_fd.

### **Parameters:**

char *Cmd:	The command to pass to the OS
int32 OS_fd:	This is the abstract file descriptor to which the output of the command is written.

### **Returns:**

N/A.

### **Restrictions:**



## ***OS\_FDGetInfo***

### **Syntax:**

```
int32 OS_TFDGetInfo (int32 filedes, OS_FDTableEntry *fd_prop);
```

### **Description:**

This function takes a file descriptor, and looks it up in the OSAL's file descriptor table. It puts all of the information known about that file descriptor into a structure pointer to by fd\_prop.

The OS\_FDTableEntry structure contains the following information:

```
int32 OSfd;                /* The underlying OS's file descriptor */
char Path [OS_MAX_PATH_LEN]; /* The absolute path to the open file */
uint32 User;               /* The task ID of the task that opened the file */
uint8 IsValid;             /* A flag showing if this FD is in use or not */
```

### **Parameters:**

filedes:           The OSAL's abstracted file descriptor to look up

task\_prop:        A pointer to a structure to hold a file descriptor's information

### **Returns:**

OS\_ERR\_INVALID\_FD if the files descriptor passed to it is invalid  
OS\_INVALID\_POINTER if the fr\_prop pointer is NULL  
OS\_FS\_SUCCESS if it copied all of the relevant info over

### **Restrictions:**

## ***OS\_FileOpenCheck***

### **Syntax:**

```
int32 OS_FileOpenCheck (char *Filename);
```

### **Description:**

This function takes a filename and determines if the file is open. The function will return success if the file is open.

### **Parameters:**

Filename	The name of the file to check
----------	-------------------------------

### **Returns:**

OS_INVALID_POINTER	if the Filename pointer is NULL
OS_FS_SUCCESS	if the file is open
OS_FS_ERROR	if the file is not open

### **Restrictions:**

## ***OS\_CloseAllFiles***

**Syntax:**

int32 OS\_CloseAllFiles (void);

**Description:**

This function closes all files that are open in the OSAL. These files must have been opened through the OSAL.

**Parameters:**

none

**Returns:**

OS\_FS\_SUCCESS if the close operations returned without error

OS\_FS\_ERROR if there was an error returned while closing any of the files

**Restrictions:**

## ***OS\_CloseFileByName***

### **Syntax:**

```
int32 OS_CloseFileByName ( char *Filename);
```

### **Description:**

This function will close the file with the given filename.

### **Parameters:**

Filename:        A string that matches the name that was used to open  
                  The file in the OSAL OS\_open call.

### **Returns:**

OS\_FS\_ERROR if the file could not be closed  
OS\_FS\_SUCCESS if the file was found and closed

### **Restrictions:**

The file must be currently open through the OSAL and the path/filename string must match the name used when opening the file. For example: If the OS\_open function was passed “/eeprom/dir1/myfile.dat”, this functions Filename string must match, it cannot be “myfile.dat”.

## 3.4 Directory API

### ***OS\_mkdir***

**Syntax:**

```
int32 OS_mkdir (const char *path, uint32 access);
```

**Description:**

This function will create a directory specified by path.

**Parameters:**

path:	The absolute pathname of the directory to be created.
access:	unused.

**Returns:**

- OS\_FS\_ERR\_INVALID\_POINTER if path is NULL
- OS\_FS\_ERR\_PATH\_TOO\_LONG if the path is too long to be stored locally
- OS\_FS\_ERROR if the OS call fails
- OS\_FS\_SUCCESS if success

**Restrictions:**

## ***OS\_opendir***

**Syntax:**

```
os_dirp_t OS_opendir( const char *path);
```

**Description:**

This function will open the specified directory for reading.

**Parameters:**

path:                The absolute pathname of the directory to be opened for reading

**Returns:**

NULL if path is NULL, path is too long, OS call fails  
a pointer to a directory if success

**Restrictions:**

## ***OS\_closedir***

### **Syntax:**

```
int32 OS_closedir( const char *path);
```

### **Description:**

This function will close the specified directory.

### **Parameters:**

path:                The absolute pathname of the directory to be closed.

### **Returns:**

OS\_FS\_SUCCESS if success  
OS\_FS\_ERROR if close failed

### **Restrictions:**

## ***OS\_readdir***

### **Syntax:**

os\_dirent\_t\* OS\_readdir( os\_dirp\_t directory);

### **Description:**

This function will return a pointer to a os\_dirent\_t structure which will hold all of the information about a directory.

### **Parameters:**

directory:      A directory descriptor pointer that was returned from a call to OS\_opendir.

### **Returns:**

A pointer to the next entry for success  
NULL if error or end of directory is reached

### **Restrictions:**



## ***OS\_rewinddir***

**Syntax:**

```
void OS_rewinddir( os_dirp_t directory);
```

**Description:**

This function will reset the directory pointer to the beginning on the currently open directory.

**Parameters:**

directory:      A directory descriptor pointer that was returned from a call to OS\_opendir.

**Returns:**

N/A

**Restrictions:**

## ***OS\_rmdir***

### **Syntax:**

```
int32 OS_rmdir( const char *path);
```

### **Description:**

This function will remove the specified directory from the file system.

### **Parameters:**

path:                The absolute pathname of the directory to be removed.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if path is NULL  
OS\_FS\_ER\_PATH\_TOO\_LONG

### **Restrictions:**

## 3.5 Disk API

### ***OS\_mkfs***

**Syntax:**

```
int32 OS_mkfs (char* address, char *devname, char *volname, uint32 blocksize,  
               uint32 numblocks);
```

**Description:**

This function will make a drive on the target with a dos file system.

**Parameters:**

- |            |  |
|------------|--|
| address:   | The address at which to start the new disk. If address == 0, then space will be allocated by the OS. |
| devname:   | The name of the “generic” drive.   |
| volname:   | The name of the volume – only used in VxWorks.   |
| blocksize: | The size of a single block on the drive.   |
| numblocks: | The amount of blocks to allocated for the drive.   |

**Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if devname is NULL  
OS\_FS\_ERR\_DRIVE\_NOT\_CREATED if the OS calls to create the drive failed  
OS\_FS\_ERR\_DEVICE\_NOT\_FREE if the volume table is full  
OS\_FS\_SUCCESS on creating the disk

**Restrictions:**

## ***OS\_rmfs***

### **Syntax:**

```
int32 OS_rmfs (char *devname);
```

### **Description:**

This function will remove the target file system.

### **Parameters:**

devname:      The name of the “generic” drive.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if devname is NULL  
OS\_FS\_ERROR if the devname cannot be found in the table  
OS\_FS\_SUCCESS on removing the filesystem

### **Restrictions:**

## ***OS\_initfs***

### **Syntax:**

```
int32 OS_initfs (char* address, char *devname, char *volname, uint32 blocksize,  
                uint32 numblocks);
```

### **Description:**

This function will initialize (without reformatting) a drive on the target with a dos file system.

### **Parameters:**

- |            |  |
|------------|--|
| address:   | The address at which to start the new disk. If address == 0, then space will be allocated by the OS. |
| devname:   | The name of the “generic” drive.   |
| volname:   | The name of the volume – only used in VxWorks.   |
| blocksize: | The size of a single block on the drive.   |
| numblocks: | The amount of blocks to allocated for the drive.   |

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if devname is NULL  
OS\_FS\_ERR\_PATH\_TOO\_LONG if the name is too long  
OS\_FS\_ERR\_DEVICE\_NOT\_FREE if the volume table is full  
OS\_FS\_ERR\_DRIVE\_NOT\_CREATED if the OS calls to create the drive failed  
OS\_FS\_SUCCESS on creating the disk

### **Restrictions:**

## ***OS\_mount***

### **Syntax:**

int32 OS\_mount (const char \*devname, char\* mountpoint);

### **Description:**

This function will mount a disk to the filesystem

### **Parameters:**

devname:      The name of the drive to mount. devname is the same from OS\_mkfs

mountpoint:   The name to call this disk from now on.

### **Returns:**

OS\_FS\_SUCCESS

OS\_FS\_ERROR

OS\_FS\_DRIVE\_NOT\_CREATED

### **Restrictions:**

Note: In RTEMS, there is no concept of “mount” because RTEMS mounts its file system on initialization, and cannot recognize other filesystems.

## ***OS\_unmount***

### **Syntax:**

int32 OS\_unmount (const char \*mountpoint);

### **Description:**

This function will unmount a drive from the file system and make all open file descriptors useless.

### **Parameters:**

mountpoint:                      The name of the drive to unmount.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if name is NULL  
OS\_FS\_ERR\_PATH\_TOO\_LONG if the absolute path given is too long  
OS\_FS\_ERROR if the OS calls failed  
OS\_FS\_SUCCESS if success

### **Restrictions:**

## ***OS\_GetPhysDriveName***

### **Syntax:**

int32 OS\_GetPhysDriveName ( char \* PhysDriveName, char \* MountPoint);

### **Description:**

This function will return the name of the physical drive underlying the abstracted file system given the abstracted mount point of that drive.

### **Parameters:**

PhysDriveName:      The name of the physical drive is copied into this pointer

MountPoint:          The mountpoint of the drive in the OS Abstraction Layer

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if either parameter is NULL

OS\_FS\_ERROR if the mount point was not found

OS\_SUCCESS on getting the name of the drive

### **Restrictions:**



## ***OS\_fsBlocksFree***

### **Syntax:**

int32 OS\_fsBlocksFree (const char \*name);

### **Description:**

This function will return the number of blocks free in the file system.

### **Parameters:**

name:           The name of the drive to check for free blocks.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if name is NULL

OS\_FS\_ERROR if the OS call failed

The number of blocks free in a volume if success

### **Restrictions:**

This function should work for vxWorks, Linux, and OS X. On RTEMS 4.10, this call will work on RFS file systems. It will not work on the RTEMS DOS file systems.

## ***OS\_fsBytesFree***

### **Syntax:**

int32 OS\_fsBytesFree (const char \*name, uint64 \*bytes\_free);

### **Description:**

This function will return the number of bytes free in the file system.

### **Parameters:**

name:	The name of the drive to check for free blocks. This can also be the name of an existing file in that file system.
bytes_free:	The number of bytes available in the file system. This will be filled out by the function.

### **Returns:**

- OS\_FS\_ERR\_INVALID\_POINTER if name or bytes\_free is NULL
- OS\_FS\_ERROR if the underlying OS call failed
- OS\_FS\_SUCCESS if the call completed successfully

### **Restrictions:**

This function should work for vxWorks, Linux, and OS X. On RTEMS 4.10, this call will work on RFS file systems. It will not work on the RTEMS DOS file systems.

## ***OS\_chkfs***

### **Syntax:**

os\_fshealth\_t OS\_chkfs (const char \*name, boolean repair);

### **Description:**

This function will check the file system integrity, and may or may not repair it, depending on repair.

### **Parameters:**

name:                The name of the drive to check integrity.

### **Returns:**

OS\_FS\_ERR\_INVALID\_POINTER if name is NULL

OS\_FS\_SUCCESS if success

OS\_FS\_ERROR if the OS calls fail

### **Restrictions:**

Note: Currently this function only works in VxWorks.

## 4 Interrupt/Exception API

### 4.1 System Interrupt API

Notes :

The following API definitions use the 'Interrupt Number' parameter. The Abstraction Layer will translate this value to a vector number or to a Mask number – all depends on the specific architecture.

The IntDisable/Enable functions are a good way of abstracting the architecture, but the mask/unmask functions may still be needed. They can be removed if not needed.

The Exception functions may not be supported on all architectures. Some processors do not have the ability to enable or disable processor exceptions.

## ***OS\_IntAttachHandler***

### **Syntax:**

```
int32 OS_IntAttachHandler ( uint32 InterruptNumber, void * InterruptHandler ,  
                           int32 parameter ) ;
```

### **Description:**

The call associates a specified C routine to a specified interrupt number. Upon occurring of the InterruptNumber , the InterruptHandler routine will be called and passed the *parameter*.

### **Parameters:**

InterruptNumber:	The Interrupt Number that will cause the start of the ISR
InterruptHandler:	The ISR associated with this interrupt
parameter:	The parameter that is passed to the ISR

### **Returns:**

OS\_SUCCESS

OS\_INVALID\_INT\_NUM

OS\_INVALID\_POINTER

OS\_ERROR , i.e. the maximum number of registered LISRs has been exceeded (Nucleus)

### **Restrictions:**

The attached routine must not invoke certain OS system functions that may block

## ***OS\_IntEnable***

### **Syntax:**

int32 OS\_IntEnable ( int32 level ) ;

### **Description:**

Enable the corresponding interrupt number.

### **Parameters :**

IntLevel:	The Interrupt Number to be enabled ENABLE_ALL_INTR (-1)
-----------	--

### **Returns:**

OS\_SUCCESS  
OS\_INVALID\_INT\_NUM  
OS\_ERROR other errors

### **Restrictions:**

SYSTEM

## ***OS\_IntDisable***

### **Syntax:**

int32 OS\_IntDisable ( int32 Level) ;

### **Description:**

Disable the corresponding interrupt number.

### **Parameters:**

Level:	The Interrupt Number to be disabled DISABLE_ALL_INTR ( -1)
--------	---

### **Returns:**

OS\_SUCCESS  
OS\_INVALID\_INT\_NUM  
OS\_ERROR other errors

### **Restrictions:**

SYSTEM

## ***OS\_IntLock***

**Syntax:**

int32 OS\_IntLock (void) ;

**Description:**

Locks out all interrupts.

**Parameters:**

None

**Returns:**

Previous state of interrupt locking before OS\_IntLock was called

**Restrictions:**

SYSTEM



## ***OS\_IntUnlock***

**Syntax:**

```
int32 OS_IntUnlock (int32 IntLevel) ;
```

**Description:**

Enables previous state of interrupts

**Parameters:**

IntLevel:        The level of interrupts to restore. This is usually what is returned from OS\_IntLock

**Returns:**

Previous state of interrupt locking before OS\_IntLock was called

**Restrictions:**

SYSTEM

## ***OS\_IntAck***

**Syntax:**

```
int32 OS_IntAck ( int32 InterruptNumber );
```

**Description:**

Acknowledge the corresponding interrupt number.

**Parameters:**

InterruptNumber:     The Interrupt Number to be Acknowledged.

**Returns:**

OS\_SUCCESS

OS\_INVALID\_INT\_NUM

OS\_ERROR other errors

**Restrictions:**

SYSTEM

## 4.2 System Exception API

### ***OS\_ExcAttachHandler***

**Syntax:**

```
int32 OS_ExcAttachHandler ( uint32 ExceptionNumber, void * ExceptionHandler ,  
                           int32 parameter ) ;
```

**Description:**

The call associates a specified C routine to a specified exception number. Upon occurring of Exception Number , the ExceptionHandler routine will be called and passed the *parameter*.

**Parameters:**

InterruptNumber:	The Exception Number that triggers the call.
InerruptHandler:	The handler for this exception
parameter:	The parameter that is passed to the Exception handler.

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_INVALID\_POINTER  
OS\_ERROR

**Restrictions:**

The attached routine must not invoke certain OS system functions that may block.

## ***OS\_ExcEnable***

**Syntax:**

int32 OS\_ExcEnable ( int32 ExceptionNumber ) ;

**Description:**

Enable/unmask the corresponding exception number.

**Parameters:**

InterruptNumber:	The Exception Number to be enabled ENABLE_ALL_EXC (-1)
------------------	---

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_ERROR other errors

**Restrictions:**

SYSTEM

## ***OS\_ExcDisable***

**Syntax:**

```
int32 OS_ExcDisable ( int32 ExceptionNumber ) ;
```

**Description:**

Disable/mask the corresponding exception number.

**Parameters:**

InterruptNumber:	The Exception Number to be disabled DISABLE_ALL_EXC ( -1)
------------------	--

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_ERROR other errors

**Restrictions:**

SYSTEM

## 4.3 System FPU Exception API

### ***OS\_FPUExcAttachHandler***

**Syntax:**

```
int32 OS_FPUExcAttachHandler ( uint32 ExceptionNumber, void * ExceptionHandler,  
                               int32 parameter ) ;
```

**Description:**

The call associates a specified C routine to a specified FPU exception number. When the specified FPU Exception occurs , the ExceptionHandler routine will be called and passed the *parameter*.

**Parameters:**

InterruptNumber:	The Exception Number that triggers the call.
InerruptHandler:	The handler for this exception
parameter:	The parameter that is passed to the Exception handler.

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_INVALID\_POINTER  
OS\_ERROR

**Restrictions:**

The attached routine must not invoke certain OS system functions that may block.

## ***OS\_FPUExcEnable***

**Syntax:**

int32 OS\_FPUExcEnable ( int32 ExceptionNumber ) ;

**Description:**

Enable/unmask the corresponding exception number.

**Parameter:**

InterruptNumber:	The Exception Number to be enabled ENABLE_ALL_EXC (-1)
------------------	---

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_ERROR other errors

**Restrictions:**

SYSTEM

## ***OS\_FPUExcDisable***

**Syntax:**

```
int32 OS_FPUExcDisable ( int32 ExceptionNumber ) ;
```

**Description:**

Disable/mask the corresponding exception number.

**Parameters:**

InterruptNumber:	The Exception Number to be disabled DISABLE_ALL_EXC ( -1)
------------------	--

**Returns:**

OS\_SUCCESS  
OS\_INVALID\_EXC\_NUM  
OS\_ERROR other errors

**Restrictions:**

SYSTEM