NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
GODDARD SPACE FLIGHT CENTER

Flight Software Branch (Code 582)

OS Abstraction Layer Library, v2.7

Nicholas Yanchik, Code 582
Alan Cudmore, Code 582
Ezra Yeheskeli, Code 582/Raytheon
Dwaine Molock, Code 582

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

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

Table of Contents

1 OS Abstraction Layer Introduction 7
2 Operating System API 8
2.1 Miscellaneous API 8
OS_API_Init 8
OS_printf 9
OS_Tick2Micros 10
OS_GetLocalTime 11
OS_SetLocalTime 12
OS_Milli2Ticks 13
2.2 Queue API 14
OS_QueueCreate 14
OS_QueueDelete 16
OS_QueueGet 17
OS_QueuePut 18
OS_QueueGetIdByName 19
OS_QueueGetInfo 20
2.3 Semaphore and Mutex API 21
OS_BinSemCreate 21
OS_BinSemDelete 22
OS_BinSemFlush 23
OS_BinSemGive 24
OS_BinSemTake 25
OS_BinSemTimedWait 26
OS_BinSemGetIdByName 27
OS_CountSemCreate 29
OS_CountSemDelete 30
OS_CountSemGive 31
OS_CountSemTake 32
OS_CountSemTimedWait 33
OS_CountSemGetIdByName 34
OS_MutSemCreate 36
OS_MutSemDelete 37
OS_MutSemGive 38
OS_MutSemTake 39
OS_MutSemGetIdByName 40
OS_MutSemGetInfo 41
2.4 Task Control API 42
OS_TaskCreate 42
OS_TaskDelete 43
OS_TaskExit 44
OS_TaskDelay 45
OS_TaskSetPriority 46
OS_TaskRegister 47
OS_TaskGetId 48
OS_TaskGetIdByName 49

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

OS_TaskGetInfo 50
2.5 Network API 51
OS_NetworkGetID 51
OS_NetworkGetHostName 52
3 File System API 53
3.1 Introduction 53
3.2 File Descriptors in the OSAL 55
3.3 File API 56
OS_creat 56
OS_open 57
OS_close 58
OS_read 59
OS_write 60
OS_chmod 61
OS_stat 62
OS_lseek 63
OS_remove 64
OS_rename 65
OS_cp 66
OS_mv 67
OS_ShellOutputToFile 68
OS_FDGetInfo 69
3.4 Directory API 70
OS_mkdir 70
OS_opendir 71
OS_closedir 72
OS_readdir 73
OS_rmdir 74
3.5 Disk API 75
OS_mkfs 75
OS_rmfs 76
OS_initfs 77
OS_mount 78
OS_unmount 79
OS_GetPhysDriveName 80
OS_fsBlocksFree 81
OS_chkfs 82
4 Hardware API 83
4.1 System Interrupt API 83
OS_IntAttachHandler 84
OS_IntEnable 85
OS_IntDisable 86
OS_IntLock 87
OS_IntUnlock 88
OS_IntAck 89
4.2 System Exception API 90

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

OS_ExcAttachHandler 90
OS_ExcEnable 91
OS_ExcDisable 92
4.3 System FPU Exception API 93
OS_FPUExcAttachHandler 93
OS_FPUExcEnable 94
OS_FPUExcDisable 95
4.4 Port I/O API 96
OS_PortRead8 96
OS_PortWrite8 97
OS_PortRead16 98
OS_PortWrite16 99
OS_PortRead32 100
OS_PortWrite32 101
OS_PortSetAttributes 102
OS_PortGetAttributes 103
4.5 Memory access API 104
OS_MemRead8 104
OS_MemWrite8 105
OS_EepromWrite8 106
OS_MemRead16 107
OS_MemWrite16 108
OS_EepromWrite16 109
OS_MemRead32 110
OS_MemWrite32 111
OS_EepromWrite32 112
OS_MemCpy 113
OS_MemSet 114
OS_MemSetAttributes 115
OS_MemGetAttributes 116
4.6 EEPROM Control API 117
OS_EepromWriteEnable 117
OS_EepromWriteDisable 118
OS_EepromPowerUp 119
OS_EepromPowerDown 120
4.7 PCI Bus API 121
OS_PciScanAndConfigureBus 121
OS_PciFindDevice 122
OS_PciFindSubsytemDevice 123
OS_PciFindTargetDevice 124
OS_PciReadConfigurationByte 125
OS_PciReadConfigurationWord 126
OS_PciReadConfigurationDword 127
OS_PciWriteConfigurationByte 128
OS_PciWriteConfigurationWord 129
OS_PciWriteConfigurationDword 130

OS_PciGetResourceStartAddr 131
OS_PciGetResourceEndAddr 132
OS_PciGetResourceLen 133
OS_PciGetResourceFlags 134
OS_PciInterruptServiceRoutine 135
OS_PciDisableTargetInterrupt 137
OS_PciConnectTargetIsr 138
OS_PciDisconnectTargetIsr 139

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 Hardware 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 Hardware APIs allow port and memory based I/O access in order to provide a common way of accessing hardware resources. In addition a PCI API is present in the header files and this document, but the reference implementation is not included in this release.

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:
void 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:
N/A.

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:
mill_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

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

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_ERR_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 semaphore 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 msecs);

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

msecs: 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

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 is semid or sem_name are NULL pointers
OS_ERR_NAME_TOO_LONG if the name given is to 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:

sem_id: The id of the semaphore 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:

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 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 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 msecs);

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
msecs:	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

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 is semid or sem_name are NULL pointers
OS_ERR_NAME_TOO_LONG if the name given is to 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:

sem_id: The id of the semaphore 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:

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_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 is semid or sem_name are NULL pointers
OS_ERR_NAME_TOO_LONG if the name given is to 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:

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

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_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 th ename to 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 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:". With this File System API, volumes are represented as Unix-style paths where each volume is mounted on the root file system:

- RAM:file1.dat becomes /mnt/ram/file1.dat
- FL: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 would be "/tmp". Therefore 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, Linux or other embedded operating systems, 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.

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

- **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.
- **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,  TRUE, 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 }
};
```

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
   */
```

Check the FSW Web site at: http://fsw.gsfc.nasa.gov/internal/(tbd) to verify this is the correct verison prior to use.

```
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, youmust 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_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_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 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_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 os_fs_stat_t 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_ERR_NAME_TOO_LONG if the name of the file is too long to be stored
OS_FS_ERROR id 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: OS_SEEK_SET – start at the beginning of the file OS_SEEK_CUR – start at the current read/write pointer 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:

old:	The absolute path to the file to be renamed.
new:	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_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_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_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_ShellOuputToFile (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 abstracte 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:

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_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 us 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_DRIVE_NOT_CREATED if the OS calls to create the drive failed
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_DRIVE_NOT_CREATED if the OS calls to create the 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_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:
Note: Currently this function only works in VxWorks.

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 Hardware 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 * InerruptHandler ,
int32 parameter) ;

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

Parameters:

InterruptNumber:	The Interrupt Number that will cause the start of the ISR
InerruptHandler:	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 Ievel) ;

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

4.4 Port I/O API

OS_PortRead8

Syntax:
int32 OS_PortRead8(uint32 PortAddress, uint8 *ByteValue) ;

Description:
Read one byte from an I/O port

Parameters:

PortAddress: The port address to be read.

ByteValue: The byte value read from this port address.

Returns:
OS_SUCESS

Restrictions:
TASK. This function can be called from application tasks or system code.

OS_PortWrite8

Syntax:
int32 OS_PortWrite8 (uint32 PortAddress, uint8 ByteValue) ;

Description:
Write one byte to port I/O

Parameters:

PortAddress: The port address to write at

ByteValue: The byte value to be written into.

Returns:
OS_SUCCESS
OS_ERROR in case of exception error.

Restrictions:
TASK. This function can be called from application tasks or system code.

OS_PortRead16

Syntax:
int32 OS_PortRead16(uint32 PortAddress, uint16 *PortValue) ;

Description:
Read 2 bytes from an I/O port

Parameters:
PortAddress: The port address to be read.

ByteValue: The 2 bytes value read from this port address.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme

Restrictions:
TASK. This function can be called from application tasks or system code.

OS_PortWrite16

Syntax:
int32 OS_PortWrite16 (uint32 PortAddress, uint16 ByteValue) ;

Description:
Write 2 bytes to port I/O

Parameters:
PortAddress: The port address to write at

ByteValue: The 2 byte value to be written into

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme

Restrictions:
TASK. This function can be called from application tasks or system code

OS_PortRead32

Syntax:
int32 OS_PortRead32 (uint32 PortAddress, uint32 *PortValue) ;

Description:
Read 4 bytes from an I/O port

Parameters:

PortAddress: The port address to be read.

ByteValue: The 4 bytes value read from this port address.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 32 bit addressing scheme

Restrictions:
TASK. This function can be called from application tasks or system code.

OS_PortWrite32

Syntax:
int32 OS_PortWrite32 (uint32 PortAddress, uint32 ByteValue) ;

Description:
Write 2 bytes to port I/O

Parameters:

PortAddress: The port address to write at

ByteValue: The 4 bytes value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 32 bit addressing scheme

Restrictions:
TASK. This function can be called from application tasks or system code.

OS_PortSetAttributes

Syntax:
int32 OS_PortSetAttributes (uint32 key, uint32 value);

Description:
This is a catch all for setting Port I/O parameters and settings. It is designed to allow needed functionality such as setting wait states, setting I/O permissions etc, that were not anticipated as part of the original API.

Parameters:

key : the Key to set. Valid Keys include
TBD

value: The value to set

Returns:
OS_SUCCESS
OS_ERROR

Restrictions:
SYSTEM: This function should be called from system and startup code only.

OS_PortGetAttributes

Syntax:
int32 OS_PortGetAttributes (uint32 key, uint32 *value);

Description:
This is a catch all for returning Port I/O parameters and settings. It is designed to allow needed functionality such as setting wait states, setting I/O permissions etc, that were not anticipated as part of the original API.

Parameters:

key: the Key to get. Valid Keys include
TBD

value: A pointer to write the key value into.

Returns:
OS_SUCCESS
OS_ERROR

Restrictions:
SYSTEM: This function should be called from system and startup code only.

4.5 Memory access API

OS_MemRead8

Syntax:
int32 OS_MemRead8(uint32 MemoryAddress, uint8 *ByteValue) ;

Description:
Read one byte of memory

Parameters:
MemoryAddress: address to be read.

ByteValue: The byte value read from this address.

Returns:
OS_SUCCESS

Restrictions:

OS_MemWrite8

Syntax:
int32 OS_MemWrite8 (uint32 MemoryAddress, uint8 ByteValue) ;

Description:
Write one byte of memory

Parameters:

MemoryAddress:	The address to write at
ByteValue:	The byte value to be written into.

Returns:
OS_SUCCESS

Restrictions:

OS_EepromWrite8

Syntax:
int32 OS_EepromWrite8 (uint32 MemoryAddress, uint8 ByteValue) ;

Description:
Write one byte of eeprom memory

Parameters:

MemoryAddress:	The address to write at
ByteValue:	The byte value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_TIMEOUT write operation did not go through after a specific timeout

Restrictions:

OS_MemRead16

Syntax:
int32 OS_MemRead16(uint32 MemoryAddress, uint16 *uint16Value) ;

Description:
Read 2 bytes of memory

Parameters:

MemoryAddress: The address to be read.

Uint16Value: The 2 bytes value read from this address.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme OS_ERROR in case of exception error.(memory protected area)

Restrictions:

OS_MemWrite16

Syntax:
int32 OS_MemWrite16 (uint32 MemoryAddress, uint16 uint16Value) ;

Description:
Write 2 bytes of memory

Parameters:

MemoryAddress:	The address to write at.
uint16Value:	The 2 byte value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme

Restrictions:

OS_EepromWrite16

Syntax:
int32 OS_EepromWrite16 (uint32 MemoryAddress, uint16 uint16Value) ;

Description:
Write 2 bytes of eeprom memory

Parameters:

MemoryAddress: The address to write at..

Uint16Value: The 2 bytes value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme
OS_ERROR_TIMEOUT write operation did not go through after a specific timeout

Restrictions:

OS_MemRead32

Syntax:
int32 OS_MemRead32(uint32 MemoryAddress, uint32 *uint32Value) ;

Description:
Read 4 bytes of memory

Parameters:

MemoryAddress: The address to be read.

Uint32Value: The 4 bytes value read from this address.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 32 bit addressing scheme

Restrictions:

OS_MemWrite32

Syntax:
int32 OS_MemWrite32 (uint32 MemoryAddress, uint32 uint32Value) ;

Description:
Write 4 bytes of memory

Parameters:

MemoryAddress:	The address to write at..
uint32Value:	the 4 bytes value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 32 bit addressing scheme

Restrictions:

OS_EepromWrite32

Syntax:
int32 OS_EepromWrite32 (uint32 MemoryAddress, uint32 uint32Value) ;

Description:
Write 4 bytes of eeprom memory

Parameters:

MemoryAddress:	The address to write at.
uint32Value:	The 4 bytes value to be written into.

Returns:
OS_SUCCESS
OS_ERROR_ADDRESS_MISALIGNED The Address is not aligned to 16 bit addressing scheme
OS_ERROR_TIMEOUT write operation did not go through after a specific timeout

Restrictions:

OS_MemCpy

Syntax:
int32 OS_MemCpy (void *dest, void *src, size_t n) ;

Description:
Copies n bytes from one address location to another address location

Parameters:

dest: A pointer to the memory area to copy TO

src: A pointer to the memory area to copy FROM.

n: Number of Bytes to copy.

Returns:
OS_SUCCESS

Restrictions:
Overlap is not allowed .

OS_MemSet

Syntax:
int32 OS_MemSet (void *dest, int value, size_t n) ;

Description:
Copies the byte *value* to the first *n* bytes of the address pointed to by dest

Parameters:

dest: A pointer to the memory area to fill

value: The byte value to fill. Note that the lower 8 bits are used.

n: Number of Bytes to fill.

Returns:
OS_SUCCESS

Restrictions:

OS_MemSetAttributes

Syntax:
int32 OS_MemSetAttributes (uint32 key, uint32 value);

Description:
This is a catch all for setting Memory parameters and settings. It is designed to allow needed functionality such as setting wait states, setting I/O permissions etc, that were not anticipated as part of the original API.

Parameters:

key : the key to set. Valid Keys include TBD

value: The value to set

Returns:
OS_SUCCESS
OS_ERROR

Restrictions:
SYSTEM: This function should be called from system and startup code only.

OS_MemGetAttributes

Syntax:
int32 OS_MemGetAttributes (uint32 key, uint32 *value);

Description:
This is a catch all for returning Memory I/O parameters and settings. It is designed to allow needed functionality such as setting wait states, setting I/O permissions etc, that were not anticipated as part of the original API.

Parameters:

key: the Key to get. Valid Keys include TBD

value: A pointer to write the key value into.

Returns:
OS_SUCCESS
OS_ERROR

Restrictions:
SYSTEM: This function should be called from system and startup code only

4.6 EEPROM Control API

OS_EepromWriteEnable

Syntax:
int32 OS_EepromWriteEnable();

Description:

Enable the eeprom for write operation

Parameters:
none

Returns:
OS_SUCCESS

Restrictions:

OS_EepromWriteDisable

Syntax:
int32 OS_EepromWriteDisable();

Description:
Disable the eeprom from write operation

Parameters:
none

Returns:
OS_SUCCESS

Restrictions:

OS_EepromPowerUp

Syntax:
int32 OS_EepromPowerUp();

Description:
Power up the eeprom

Parameters:
none

Returns:
OS_SUCCESS

Restrictions:

OS_EepromPowerDown

Syntax:
int32 OS_EepromPowerDown();

Description:
Power down the eeprom

Parameters:
none

Returns:
OS_SUCCESS

Restrictions:

4.7 PCI Bus API

OS_PciScanAndConfigureBus

Syntax:
int OS_PciScanAndConfigureBus (void);

Description:
This service is used to scan the PCI bus to detect and configure all PCI targets currently present.

Parameters:
none

Returns:
This service returns the number of PCI targets detected and configured on the PCI bus.

Restrictions:
This function should only be called by system code during initialization.

OS_PciFindDevice

Syntax:
int OS_PciFindDevice (uint16 vendor, uint16 device, int index, uint32 *target);

Description:
This function is used to scan the list of installed devices looking for a device matching the specified vendor and device ids.

Parameters:

vendor:	Indicates vendor id of PCI target.
device:	Indicates device id of PCI target.
index:	Indicates the instance of the PCI target if the PCI device table contains more than one entry matching the vendor/id identifier.
target:	32 Bit unsigned integer pointer to the location where the PCI target id of the matching PCI target should be stored.

Returns:
Returns –1 if no entries were found matching the vendor/id identifier.
Returns 0 if PCI target found matching the vendor/id identifier.

Restrictions:

OS_PciFindSubsytemDevice

Syntax:
int OS_PciFindSubsystemDevice (uint16 subvendor, uint16 subdevice, int index,
uint32 *target);

Description:
This function is used to scan the list of installed devices looking for a device matching the specified subsystem vendor and device ids.

Parameters:

subvendor: Indicates subsystem vendor id of PCI target.

subdevice: Indicates subsystem device id of PCI target.

index: Indicates the instance of the PCI target if the PCI device table contains more than one entry matching the subsystem vendor/id identifier.

target : 32 Bit unsigned integer pointer to the location where the PCI target id of the matching PCI target should be stored.

Returns:
Returns –1 if no entries were found matching the subsystem vendor/id identifier.
Returns 0 if PCI target found matching the subsystem vendor/id identifier.

Restrictions:

OS_PciFindTargetDevice

Syntax:
int OS_PciFindTargetDevice (uint16 vendor, uint16 device, uint16 subvendor, uint16 subdevice, int index, uint32 *target);

Description:
This function is used to scan the list of installed devices looking for a device matching the specified vendor/device and subsystem vendor/device ids.

Parameters:

vendor:	Indicates vendor id of PCI target.
device:	Indicates device id of PCI target.
subvendor:	Indicates subsystem vendor id of PCI target.
subdevice:	Indicates subsystem device id of PCI target.
index:	Indicates the instance of the PCI target if the PCI device table contains more that one entry matching the vendor/id and subsystem vendor/id identifier.
target:	32 Bit unsigned integer pointer to the location where the PCI target id of the matching PCI target should be stored.

Returns:
Returns –1 if no entries were found matching the vendor/id and subsystem vendor/device identifier.
Returns 0 if PCI target found matching the vendor/id and subsystem vendor/device identifier.

Restrictions:

OS_PciReadConfigurationByte

Syntax:
int OS_PciReadConfigurationByte (uint32 target, int offset, uint8 *ptr);

Description:
This function is used to read one byte from the configuration space of the PCI target identified by ***target***.

Parameters:

target:	Indicates the PCI target to access.
offset:	Indicates the byte offset into the PCI target's configuration space.
ptr:	Pointer to the location to store the value read from the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciReadConfigurationWord

Syntax:
int OS_PciReadConfigurationWord (uint32 target, int offset, uint16 *ptr);

Description:
This function is used to read two bytes from the configuration space of the PCI target identified by ***target***.

Parameters:

target:	Indicates the PCI target to access.
offset:	Indicates the word (16-bit) offset into the PCI target's configuration space.
ptr:	Pointer to the location to store the value read from the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciReadConfigurationDword

Syntax:
int OS_PciReadConfigurationDword (uint32 target, int offset, uint32 *ptr);

Description:
This function is used to read four bytes from the configuration space of the PCI target identified by ***target***.

Parameters:

target:	Indicates the PCI target to access.
offset:	Indicates the dword (32-bit) offset into the PCI target's configuration space.
ptr:	Pointer to the location to store the value read from the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciWriteConfigurationByte

Syntax:
int OS_PciWriteConfigurationByte (uint32 target, int offset, uint8 value);

Description:
This function is used to write one byte to the configuration space of the PCI target identified by ***target***.

Parameters:

Target:	Indicates the PCI target to access.
Offset:	Indicates the byte offset into the PCI target's configuration space.
Value:	Indicates the byte value to write to the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciWriteConfigurationWord

Syntax:
int OS_PciWriteConfigurationWord (uint32 target, int offset, uint16 value);

Description:
This function is used to write two bytes to the configuration space of the PCI target identified by ***target***.

Parameters:

target:	Indicates the PCI target to access.
offset:	Indicates the word (16-bit) offset into the PCI target's configuration space.
value:	Indicates the word (16-bit) value to write to the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciWriteConfigurationDword

Syntax:
int OS_PciWriteConfigurationDword (uint32 target, int offset, uint32 value);

Description:
This function is used to write four bytes to the configuration space of the PCI target identified by ***target***.

Parameters:

target:	Indicates the PCI target to access.
offset:	Indicates the dword (32-bit) offset into the PCI target's configuration space.
value:	Indicates the dword (32-bit) value to write to the PCI target's configuration space.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target is present in device entry table.

Restrictions:

OS_PciGetResourceStartAddr

Syntax:
unsigned long OS_PciGetResourceStartAddr (uint32 target, int bar);

Description:
This function is used to get the starting address of the base/bar segment for the specified PCI target.

Parameters:

target:	Indicates the PCI target to access.
bar:	Indicates the region or base address register (ranging from 0 to 5 inclusive).

Returns:
Indicates the starting address of the memory-mapped address assigned to this bar/base address.

Restrictions:

OS_PciGetResourceEndAddr

Syntax:
unsigned long OS_PciGetResourceEndAddr (uint32 target, int bar);

Description:
This function is used to get the ending address of the base/bar segment for the specified PCI target.

Parameters:

target:	Indicates the PCI target to access.
bar:	Indicates the region or base address register (ranging from 0 to 5 inclusive).

Returns:
Indicates the ending/last usable address of the memory-mapped address assigned to this bar/base address.

Restrictions:

OS_PciGetResourceLen

Syntax:
unsigned long OS_PciGetResourceLen (uint32 target, int bar);

Description:
This function is used to get the length of the base/bar segment for the specified PCI target.

Parameters:

target:	Indicates the PCI target to access.
bar:	Indicates the region or base address register (ranging from 0 to 5 inclusive).

Returns:
Indicates the length of the memory-mapped address assigned to this bar/base address.

Restrictions:

OS_PciGetResourceFlags

Syntax:
unsigned long OS_PciGetResourceLen (uint32 target, int bar);

Description:
This function is used to get the flags associated with this base/bar segment for the specified PCI target.

Parameters:

target:	Indicates the PCI target to access.
bar:	Indicates the region or base address register (ranging from 0 to 5 inclusive).

Returns:

OS_PCI_RESOURCE_NOT_PRESENT	Indicates the region is not present for this PCI device.
OS_PCI_RESOURCE_IO	Indicates the region is IO mapped.
OS_PCI_RESOURCE_MEM	Indicates the region is memory mapped.

Restrictions:

OS_PciInterruptServiceRoutine

Syntax:
void OS_PciInterruptServiceRoutine (unsigned long parameter);

Description:
This function is the interrupt service routine for the main PCI interrupt.

Parameters:

parameter:	Indicates a parameter that can be passed into the interrupt service routine during execution.

Returns:
none

Restrictions:

OS_PciEnableTargetInterrupt

Syntax:
int OS_PciEnableTargetInterrupt (uint32 target);

Description:
This function is used to enable the PCI target interrupt at the PCI master interface.

Parameters:
target: Indicates the PCI target interrupt to enable.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target interrupt is enabled.

Restrictions:

OS_PciDisableTargetInterrupt

Syntax:
int OS_PciDisableTargetInterrupt (uint32 target);

Description:
This function is used to disable the PCI target interrupt at the PCI master interface.

Parameters:
target: Indicates the PCI target interrupt to disable.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target interrupt is disabled.

Restrictions:

OS_PciConnectTargetIsr

Syntax:
int OS_PciConnectTargetIsr (uint32 target, void (*handler)(unsigned long parameter), unsigned long parameter);

Description:
This function is used to connect an interrupt service routine the specified PCI target.

Parameters:

target: Indicates the PCI target to install interrupt handler for.

handler: The pointer to interrupt service routine to be installed for the specified PCI target.

parameter: Indicates a parameter that can be passed into the interrupt service routine during execution.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target interrupt service routine has been installed.

Restrictions:

OS_PciDisconnectTargetIsr

Syntax:
int OS_PciDisconnectTargetIsr (uint32 target);

Description:
This function is used to disconnect an interrupt service routine the specified PCI target.

Parameters:

target:	Indicates the PCI target for interrupt service routine being disconnected.

Returns:
Returns –1 if PCI target is not present in device entry table.
Returns 0 if PCI target interrupt service routine has been uninstalled.

Restrictions: