Technical Document

GENERIC PROTOCOL STACK FRAMEWORK

GPF

OS – OPERATING SYSTEM INTERFACE

FUNCTIONAL INTERFACE DESCRIPTION

Document Number:	06-03-10-ISP-0003
Version:	0.12
Status:	Draft
Approval Authority:	
Creation Date:	1999-Jul-15
Last changed:	2015-Mar-08 by Mathias Punger
File Name:	os_api.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
1999-Jul-15	MP/FR et al.		0.1		1
1999-Nov-02	MP et al.		0.2		2
2000-Feb-08	ÛB et al.		0.3		3
2000-Feb-11	MP et al.		0.4		4
2000-Oct-11	MP et al.		0.5		5
2002-Jan-11	MP		0.6		6
2003-May-20	XINTEGRA		0.7	Draft	
2003-Dec-12	XINTEGRA		0.8		

2003-Dec-12	RC				7
2004-Feb-16	MP				8
2004-Jun-21	MP				9
2004-Sep-13	MP				10
2004-Nov-08	MP				11
2005-Feb-14	MP		0.12		12
2005-Apr-15	SLO				13

Notes:

1. Initial version
2. Timeout feature added
3. Condat AG change
4. New parameter mempoolhandle
5. Add T-VOID_STRUCT
6. Add os_SystemError
7. Add APIs required by L1
8. general update, added OSISR and interrupt handling
9. added functionality for L1 port to GPF
10. general cleanup
11. extend power management API
12. add os_CreatePartitionPool_fixed_pool_size()
13. Nucleus event API added

Table of Contents

Generic Protocol Stack Framework 1

GPF 1

OS – Operating System Interface 1

Functional Interface Description 1

1.1 Abbreviations 6

2 Introduction 7

3 Frame/Body Concept 8

4 Operation System Interface 9

4.1 Data Types 9
4.1.1 Base Types 9
4.1.2 T_VOID_STRUCT 9
4.1.3 OS_TIME 9
4.1.4 OS_HANDLE 9
4.1.5 OS_QDATA 10
4.1.6 OS_INT_STATE 10
4.2 Constants 10
4.2.1 Return Codes 10
4.2.2 System Error and Warning Codes 10
4.2.3 Object Identifiers 11
4.2.4 Other Constants 11
4.3 Functions 12
4.3.1 Processes 12
4.3.1.1 os_CreateTask() 12
4.3.1.2 os_DestroyTask() 13
4.3.1.3 os_StartTask() 13
4.3.1.4 os_StopTask() 14
4.3.1.5 os_SuspendTask() 14
4.3.1.6 os_DeferTask() 15
4.3.1.7 os_ResumeTask() 15
4.3.1.8 os_Relinquish() 16
4.3.1.9 os_GetTaskName() 16
4.3.1.10 os_GetTaskHandle() 17
4.3.1.11 os_MyHandle() 17
4.3.2 OS ISRs 18
4.3.2.1 os_CreateOSISR() 18
4.3.2.2 os_DeleteOSISR() 18
4.3.2.3 os_ActivateOSISR() 19
4.3.3 Interrupt Locks 20
4.3.3.1 os_SetInterruptState() 20
4.3.3.2 os_EnableInterrupts() 20
4.3.3.3 os_DisableInterrupts() 21
4.3.4 Communication 22
4.3.4.1 os_CreateQueue() 22

4.3.4.2 os_DestroyQueue() 23
4.3.4.3 os_OpenQueue() 23
4.3.4.4 os_CloseQueue() 24
4.3.4.5 os_SendToQueue() 25
4.3.4.6 os_ReceiveFromQueue() 26
4.3.5 Memory 27
4.3.5.1 os_CreatePartitionPool() 27
4.3.5.2 os_CreatePartitionPool_fixed_pool_size() 28
4.3.5.3 os_AllocatePartition() 29
4.3.5.4 os_DeallocatePartition() 30
4.3.5.5 os_CreateMemoryPool() 30
4.3.5.6 os_AllocateMemory() 31
4.3.5.7 os_DeallocateMemory() 32
4.3.5.8 os_SetPoolHandles() 32
4.3.6 Timer 34
4.3.6.1 os_CreateTimer() 34
4.3.6.2 os_DestroyTimer() 35
4.3.6.3 os_StartTimer() 36
4.3.6.4 os_StopTimer() 37
4.3.6.5 os_QueryTimer() 37
4.3.6.6 os_set_tick() 38
4.3.7 Semaphores 39
4.3.7.1 os_CreateSemaphore() 39
4.3.7.2 os_DestroySemaphore() 40
4.3.7.3 os_OpenSemaphore() 40
4.3.7.4 os_CloseSemaphore() 41
4.3.7.5 os_ObtainSemaphore() 41
4.3.7.6 os_ReleaseSemaphore() 42
4.3.7.7 os_QuerySemaphore() 42
4.3.8 Event groups 43
4.3.8.1 os_CreateEventGroup() 43
4.3.8.2 os_DeleteEventGroup() 43
4.3.8.3 os_SetEvents() 44
4.3.8.4 os_ClearEvents() 44
4.3.8.5 os_RetrieveEvents() 44
4.3.8.6 os_EventGroupInformation() 45
4.3.8.7 os_GetEventGroupHandle() 46
4.3.9 System 46
4.3.9.1 os_GetTime() 46
4.3.9.2 os_Initialize() 46
4.3.9.3 os_ObjectInformation() 47
4.3.9.4 os_SystemError() 48
4.3.9.5 os_IncrementTick() 49
4.3.9.6 os_RecoverTick() 50
4.3.9.7 os_GetInactivityTicks() 51
4.3.9.8 os_GetScheduleCount() 52

Appendices 53

A. Acronyms 53

B. Glossary 53

List of Figures and Tables

List of References

[ISO 9000:2000] International Organization for Standardization. Quality management systems - Fundamentals and vocabulary. December 2000

1.1 Abbreviations

RTOS	Real-time Operating System
VSI	Virtual System Interface

2 Introduction

G23 is a software package implementing Layers 2 and 3 of the ETSI-defined GSM air interface signaling protocol, and as such represents the part of a GSM mobile station's protocol software which is both, platform and manufacturer independent. Therefore, G23 can be viewed as a building block providing standardized functionality through generic interfaces for easy integration.

The G23 suite of products consists of the following items:

- Layers 2 and 3 for speech & short message services,
- Layers 2 and 3 for fax & data services,
- Application Control Interface/AT Command Interface,
- MMI and MMI Framework (MFW) and
- Test and integration support tools.

This document is the Functional Interface Description for the Operating System Interface (OSI).

3 Frame/Body Concept

The frame body concept has been designed in the context of the G23 Protocol Stack. In the case of the G23 Protocol Stack, a process represents the protocol logic of a protocol stack entity. This architecture separates the process functionality into two logical modules, the process frame and the process body. Common process functionality is located in the process frame. The main process functionality is located in the process body.

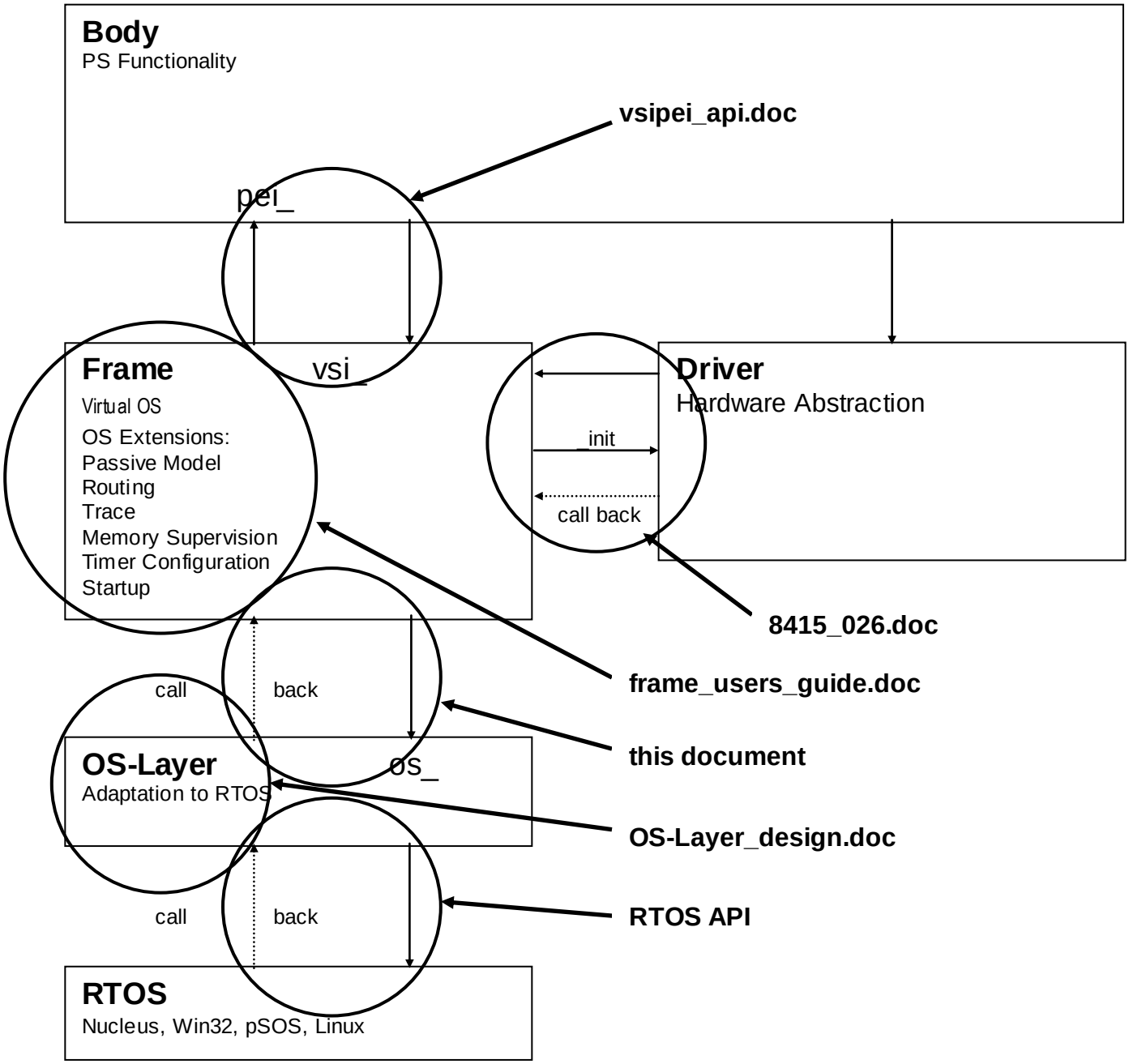

Figure 1: Protocol Stack Software Architecture and Documentation

The process frame has two interfaces. The Virtual System Interface (VSI) is the frames functional interface to be accessed by the bodies. The Operating System Interface (OS) is also a functional interface and provides the interface to the Real-time Operating System (RTOS). This interface is encapsulated in the "OS Layer" in order to keep the frame itself independent from the underlying RTOS.

In the OS Layer, the request of system resources by the protocol stack entities via the VSI is adapted to the implemented RTOS.

The intention of this interface is to provide a set of function calls that is independent of the underlying RTOS. If an RTOS does not supply all the features described in the following, e.g. the possibility of periodic timers, this must be adapted within the OS Interface.

Some functionality of the interface described in the following may not be necessary for all RTOSs and therefore some functions do not have to be filled with code. The releasing of queue handles if they are not longer used might not always be necessary when communication is closed.

4 Operation System Interface

4.1 Data Types

4.1.1 Base Types

The following type names are used as synonyms for hardware/compiler dependent integer types:

SHORT	16 bit, signed
USHORT	16 bit, unsigned
LONG	32 bit, signed
ULONG	32 bit, unsigned

4.1.2 T_VOID_STRUCT

Definition: typedef unsigned long T_VOID_STRUCT

Description: Pointers of type T_VOID_STRUCT are passed to functions in order to avoid warnings when using void pointers with a subsequent cast operation within the called function.

4.1.3 OS_TIME

Definition: This is an integral type, therefore the definition is not provided in this passage. The size may depend on the underlying RTOS, but it is at least 31 bits.

Description: This type is used for all parameters containing a time value (which is always in msec).

4.1.4 OS_HANDLE

Definition: This is an integral type, therefore the definition is not provided in this passage. The size may depend on the underlying RTOS.

Description: This type is used for all parameters containing a handle (for tasks, queues, timers, semaphores, partition pool groups, memory pools).

4.1.5 OS_QDATA

Definition:

```
typedef struct
{
  USHORT          flags;
  USHORT          data16;
  ULONG           data32;
  ULONG           len;     /* ATTENTION: only used on tool side */
  ULONG           time;
  LONG            e_id;
  T_VOID_STRUCT * ptr;
} OS_QDATA;
```

Description: This type is used by os_SendToQueue() and os_ReceiveFromQueue(), see the description of these functions for details.

4.1.6 OS_INT_STATE

Definition: This is boolean data type.

Description: This type is used by os_SetInterruptLevel(), see the description of these functions for details.

4.2 Constants

4.2.1 Return Codes

OS_OK	0	successful execution
OS_WAITED	1	successful execution; returned by some OS functions to indicate that the calling process was blocked *)
OS_PARTITION_FREE	2	partition is freed, this is checked before is primitive is sent *)
OS_ALLOCATED_BIGGER	3	bigger partition allocated than requested, because no partition of requested size available *)
OS_ERROR	-1	error
OS_TIMEOUT	-2	requested service not available during specified time
OS_PARTITION_GUARD_PATTERN_DESTROYED	-3	partition guard pattern destroyed

*) These return codes are not required for correct functionality but for the generation of additional traces in case of abnormal system states.

4.2.2 System Error and Warning Codes

These system error and warning codes are passed to the function os_SystemError() in case of abnormal system state. The do not necessarily have to be implemented for proper system function. There may be used to generate additional debug information in case of an abnormal system state.

OS_SYST_ERR_QUEUE_CREATE	0x8001	Error at queue creation
OS_SYST_ERR_MAX_TIMER	0x8002	Number of timers exceeds MAX_TIMER
OS_SYST_ERR_MAX_TASK	0x8003	Number of tasks exceeds MAX_TASKS
OS_SYST_ERR_STACK_OVERFLOW	0x8004	Task stack guard pattern destroyed
OS_SYST_ERR_PCB_PATTERN	0x8005	Partition guard pattern destroyed
OS_SYST_ERR_NO_PARTITION	0x8006	No partition available

OS_SYST_ERR_STR_TOO_LONG	0x8007	String to be traced is too long
OS_SYST_ERR_OVERSIZE	0x8008	The size of the new primitive exceeds the partition size at REUSE
OS_SYST_ERR_TASK_TIMER	0x8009	The number of task timers exceeds the number of timers requested in pei_create()
OS_SYST_ERR_SIMUL_TIMER	0x800A	The number of simultaneous running timers exceeds MAX_SIMULTANEOUS_TIMER
OS_SYST_ERR_QUEUE_FULL	0x800B	Queue full, write attempt failed
OS_SYST_ERR_MAX_SEMA	0x800C	The number of semaphores exceeds MAX_SEMPHORES
OS_SYST_ERR_NO_MEMORY	0x800D	No dynamic memory available
OS_SYST_ERR_BIG_PARTITION	0x800E	Requested size exceeds size of biggest partition
OS_SYST_WRN_WAIT_PARTITION	0x0001	Waited for requested partition
OS_SYST_WRN_WAIT_QUEUE	0x0002	Waited for space in queue
OS_SYST_WRN_BIG_PARTITION	0x0003	Bigger partition returned than requested
OS_SYST_WRN_MULTIPLE_FREE	0x0004	Partition freed more than once
OS_SYST_WRN_REQ_TRUNCATED	0x0005	Allocation request truncated
OS_SYST_WRN_FREE_FAILED	0x0006	Invalid pointer passed to deallocation API.

4.2.3 Object Identifiers

OS_OBJSYS	0	to get system information (see os_ObjectInformation())
OS_OBJTASK	1	to get task information (see os_ObjectInformation())
OS_OBJQUEUE	2	to get queue information (see os_ObjectInformation())
OS_OBJPARTITIONGROUP	3	to get partition group information (see os_ObjectInformation())
OS_OBJMEMORYPOOL	4	to get memory pool information (see os_ObjectInformation())
OS_OBJTIMER	5	to get timer information (see os_ObjectInformation())
OS_OBJSEMAPHORE	6	to get semaphore information (see os_ObjectInformation())

4.2.4 Other Constants

RESOURCE_NAMELEN	8	max. length of os 'object' names
OS_NOTASK	0	special task handle to indicate to the OS that the call is performed from outside any task (e.g. by an interrupt routine)
OS_NO_EVENT	0	No RTOS events pending (see
OS_EVENT	1	RTOS events pending
OS_QNORMAL	1	normal message priority (see os_SendToQueue)
OS_QURGENT3	2	urgent message priority (see os_SendToQueue)
OS_QFPARTITION	0x0001	flag: The memory for the message was obtained by os_AllocatePartion()

4.3 Functions

4.3.1 Processes

4.3.1.1 os_CreateTask()

Function definition:

LONG os_CreateTask(OS_HANDLE caller, char * name, void (*TaskEntry)(OS_HANDLE,ULONG), ULONG stacksize, ULONG priority, OS_HANDLE * taskhandle, OS_HANDLE mempoolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	task name	IN
void (*TaskEntry)(OS_HANDLE,ULONG)		task entry function, see os_StartTask()	IN
ULONG	stacksize	task stacksize	IN
ULONG	priority	task priority (0=low;255=high)	IN
OS_HANDLE *	taskhandle	handle of created task (!= 0)	OUT
OS_HANDLE	mempoolhandle	handle of memory pool for task stack	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateTask() creates a task with the specified name, stacksize and priority.

The memory for the stack is allocated form the previously created memory pool (use os_CreateMemoryPool(), 4.3.5.5) specified by the parameter 'mempoolhandle'. The task stack is initialized to 0xfe. This initialization is necessary for the stack check mechanism.

An adaptation of the task priorities in the G23 Protocol Stack to the priority order the RTOS has implemented is done in this function. The priorities of the applications tasks are passed to the os_CreateTask() function in the following manner:

0 ... lowest priority, 255 ... highest priority

An adaption is necessary as the RTOSs may have a different enumeration for priorities:

NUCLEUS: 0 ... highest priority, 255 ... lowest priority

pSOS: 0 ... lowest priority, 255 ... highest priority

4.3.1.2 os_DestroyTask()

Function definition:

LONG os_DestroyTask(OS_HANDLE caller, OS_HANDLE handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	handle	handle of the task to be destroyed	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DestroyTask() deletes a previously created task with the specified task handle.

4.3.1.3 os_StartTask()

Function definition:

LONG os_StartTask(OS_HANDLE caller, OS_HANDLE taskhandle, ULONG value)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	taskhandle	handle of the task to be started	IN
ULONG	value	used as 2nd parameter of the task entry function	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_StartTask() starts the task specified by the task handle. The task starts as if its entry function is called with the parameters taskhandle and value. The entry function should never return.

4.3.1.4 os_StopTask()

Function definition:

LONG os_StopTask(OS_HANDLE caller, OS_HANDLE taskhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	taskhandle	handle of the task to be stopped	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_StartTask() stops the task specified by the task handle.

4.3.1.5 os_SuspendTask()

Function definition:

LONG os_SuspendTask(OS_HANDLE caller, OS_TIME time)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_TIME	time	suspend time in ms	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_SuspendTask() suspends the calling task for the suspend time in milliseconds.

4.3.1.6 os_DeferTask()

Function definition:

LONG os_DeferTask (OS_HANDLE task_handle, OS_TIME time)

Parameters:

Type	Name	Meaning	
OS_HANDLE	task_handle	task to be suspended	IN
OS_TIME	time	suspend time in ms	IN

Return:

Type	Meaning	
LONG	OS_OK	success
LONG	OS_ERROR	error

Description:

The function os_DeferTask() suspends the task specified by *task_handle* for the suspend time in milliseconds. A task that is suspended with os_DeferTask() can be resumes with os_ResumeTask(), see 4.3.1.7.

OS_ERROR is returned in case of an invalid parameter *task_handle*.

4.3.1.7 os_ResumeTask()

Function definition:

LONG os_ResumeTask (OS_HANDLE task_handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	task_handle	task to be suspended	IN

Return:

Type	Meaning	
LONG	OS_OK	success
LONG	OS_ERROR	error

Description:

The function os_ResumeTask() resumes the task specified by *task_handle* that was previously suspended with os_DeferTask(), see 4.3.1.6.

OS_ERROR is returned in case of an invalid parameter *task_handle*.

4.3.1.8 os_Relinquish()

Function definition:

LONG os_Relinquish (void)

Parameters: none

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_Relinquish() allows all other ready tasks with the same priority as the calling task to run.

4.3.1.9 os_GetTaskName()

Function definition:

LONG os_GetTaskName(OS_HANDLE caller, OS_HANDLE handle, char * name)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	handle	task handle	IN
char *	name	buffer address for requested task name	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_GetTaskName() returns the name of the task specified by the task handle.

4.3.1.10 os_GetTaskHandle()

Function definition:

LONG os_GetTaskHandle(OS_HANDLE caller, char * name, OS_HANDLE * handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	task name or NULL	IN
OS_HANDLE *	handle	task handle	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_GetTaskHandle() returns the handle of the task specified by the task name.

If name is NULL, the handle of the calling task is returned, even if the value of caller is invalid. This call can therefore be used to obtain the task handle when it is unknown.

4.3.1.11 os_MyHandle()

Function definition:

OS_HANDLE os_MyHandle (void)

Parameters: none

Return:

Type	Meaning	
OS_HANDLE	> 0	task handle of the currently running GPF task
	= 0	non GPF task or interrupt running

Description:

The function os_MyHandle() returns the handle of the currently running task. If called in the context of a non GPF based task or interrupt handling (either ISR or OSISR) os_MyHandle() returns zero.

4.3.2 OS ISRs

4.3.2.1 os_CreateOSISR()

Function definition:

LONG os_CreateOSISR(char * name, void (*OSISREntry)(void), ULONG stacksize, ULONG priority, OS_HANDLE * osisr_handle)

Parameters:

Type	Name	Meaning	
char *	name	OSISR name	IN
void (*OSISREntry)(void)	OSISR entry function, to call when OS ISR is activated		IN
ULONG	stacksize	OSISR stacksize	IN
ULONG	priority	OSISR priority (0=low;2=high)	IN
OS_HANDLE *	osisr_handle	handle of created OS ISR (!= 0)	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateOSISR() creates/registers a deferred ISR (OS ISR) with the specified name, stacksize and priority. An OSISR is approximately half way between an interrupt handler, and an OS task. An OS ISR is a context from which it is safe to make non-blocking OS calls, for example sending a message to a queue, or raising a semaphore.

The memory for the stack is allocated for the memory pool specified by the handle passed to the parameter ‘int_pool_handle’ of the function os_SetPoolHandles() () .os_SetPoolHandles() need to be called before os_CreateOSISR(). The allocated stack is initialized to 0xfe. This initialization is necessary for the stack check mechanism (check not yet implemented).

The priorities of the OS ISRs are passed to the os_CreateOSISR() function in the following manner:

0 ... lowest priority, 2 ... highest priority

An adaptation is necessary as the RTOSs may have a different enumeration for priorities. Indeed some RTOS may not even have a concept of a deferred ISRs, in which case it may be emulated by either calling the OSISR handler function directly from the context of the os_ActivateOSISR() caller, or by using a task of higher priority than a normal task (see os_CreateTask()), so that a regular task cannot preempt a OSISR. The exact manner for emulating a OSISR would depend on what contexts can make (non-blocking) OS calls.

4.3.2.2 os_DeleteOSISR()

Function definition:

LONG os_DeleteOSISR(OS_HANDLE osisr_handle)

Parameters:

Type	Name	Meaning

OS_HANDLE	osisr_handle	OS ISR handle returned by os_CreateOSISR()	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DeleteOSISR() deletes the OS ISR specified by the parameter *osisr_handle.*

The memory of the OS ISR stack is freed.

4.3.2.3 os_ActivateOSISR()

Function definition:

LONG os_ActivateOSISR(OS_HANDLE osisr_handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	osisr_handle	OS ISR handle returned by os_CreateOSISR()	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_ActivateOSISR() activates a OS ISR. What this function actually does depends on the underlying RTOS. For an RTOS that requires OS ISRs to be scheduled, this schedules the OS ISR and returns. For other RTOS's, this may invoke the OS ISR entry directly, or may schedule the high priority task that emulates the OS ISR. This function should be called as the last thing that an ISR does. Depending on how OS ISR's are implemented by the underlying RTOS, this function may not return.

An activation counter ensures that the registered OS ISR handler is called as often as os_ActivateOSISR() is called even if more than one call to os_ActivateOSISR() is done before the OSISR handler was called. The activation counter is either maintained in the RTOS or needs to be implemented in the OS abstraction layer.

4.3.3 Interrupt Locks

Interrupt locks provide a means of implementing a critical section that works for tasks, HISRs, and interrupt handlers. In general, one should be careful while using interrupt locks, since they block out all interrupts. Critical sections implemented using interrupt locks should be kept as short as possible.

4.3.3.1 os_SetInterruptState()

Function definition:

LONG os_SetInterruptState(OS_INT_STATE new_state, OS_INT_STATE *old_state)

Parameters:

Type	Name	Meaning	
OS_INT_STATE	new_state	interrupt state to be set	IN
OS_INT_STATE *	old_state	previous interrupt state	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_SetInterruptState() will set the interrupt state corresponding to the value specified by the parameter *new_state*. The previous interrupt state is returned in the parameter **old_state*.

When using this it is strictly recommended to use this API function as shown in the following example:

```
OS_INT_STATE old_state;

os_DisableInterrupts(&old_state);

/* critical section */

os_SetInterruptState(old_state, …);
```

4.3.3.2 os_EnableInterrupts()

Function definition:

LONG os_EnableInterupts(OS_INT_STATE *old_state)

Parameters:

Type	Name	Meaning	
OS_INT_STATE *	old_state	previous interrupt state	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_EnableInterupts() will enable all interrupts and return the previous interrupt state in **old_state*.

ATTENTION: All interrupts are enabled regardless of the state when interrupts were previously disabled. It is recommended to use the function only during initialization. To re-enable interrupts at the end of a critical section please use os_SetInterruptState(), refer to example in 4.3.3.1.

4.3.3.3 os_DisableInterrupts()

Function definition:

LONG os_DisableInterupts(void)

Parameters:

Type	Name	Meaning	
OS_INT_STATE *	old_state	previous interrupt state	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DisableInterupts() will disable all interrupts and return the previous interrupt state in **old_state*.

4.3.4 Communication

4.3.4.1 os_CreateQueue()

Function definition:

LONG os_CreateQueue(OS_HANDLE caller, OS_HANDLE comhandle, char * name, ULONG entries, OS_HANDLE * acthandle, OS_HANDLE mempoolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	queuehandle	handle of queue to be created	IN
char *	name	queue name	IN
ULONG	entries	(max) number of queue entries	IN
OS_HANDLE *	acthandle	handle of created queue	OUT
OS_HANDLE	mempoolhandle	handle of memory pool for queue memory	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateQueue() creates a message queue with the specified name and number of entries. If the parameter queuehandle is set to zero, the next available handle is used. The parameter ‘mempoolhandle’ specifies the handle of a previously created memory pool where the queue memory is allocated from.

4.3.4.2 os_DestroyQueue()

Function definition:

LONG os_DestroyQueue(OS_HANDLE caller, OS_HANDLE queuehandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	queuehandle	queue handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DestroyQueue() deletes a previously created queue with the specified queue handle.

4.3.4.3 os_OpenQueue()

Function definition:

LONG os_OpenQueue(OS_HANDLE caller, char * name, OS_HANDLE * queuehandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	queue name	IN
OS_HANDLE *	queuehandle	queue handle	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_OpenQueue() retrieves the handle of the queue specified by the queue name. This handle is needed to open communication with a protocol stack entity.

This function may also be used to register the calling task for communication via the specified queue.

4.3.4.4 os_CloseQueue()

Function definition:

LONG os_CloseQueue(OS_HANDLE caller, OS_HANDLE queuehandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	queuehandle	queue handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CloseQueue() releases the handle of the queue.

This function may be used to deregister the calling task from communication via the specified queue.

4.3.4.5 os_SendToQueue()

Function definition:

LONG os_SendToQueue(OS_HANDLE caller, OS_HANDLE queuehandle, ULONG priority, OS_TIME timeout, OS_QDATA * data)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	queuehandle	queue handle of the receiver	IN
ULONG	priority	priority of message	IN
OS_TIME	timeout	timeout in ms for send attempt	IN
OS_QDATA *	data	pointer to message data to be sent	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error
	OS_WAITED	success, but waited for free queue element
	OS_TIMEOUT	timeout occurred

Description:

The function os_SendToQueue() writes the message identified by the pointer data into the queue specified by the queue handle.

The elements of data are treated as follows:

flags	special flags (bit coded) to be interpreted. Currently there is only one Flag defined: OS_QFPARTITION (bit 0): This flag should be set, if ptr points to memory obtained by os_AllocatePartition(). All other Bits are reserved and should be 0.
data16	transported to the receiver as is
data32	transported to the receiver as is
len	transported to the receiver as is; this is the length of the message referred by ptr
time	time in ms since reset when the message is written into the queue
e_id	the handle of the destination entity (needed to deliver the message to the correct entity in case several entities share one task)
ptr	the message referred by ptr is transported to the receiver, either by reference (ptr itself) or by copy (i.e. the receiver gets a pointer to a copy of the message)

The calling task is suspended until the request can be satisfied or the specified time is expired. If the function has to wait for a free queue element, OS_WAITED is returned.

If the calling process is a non-task thread (caller = OS_NOTASK), the function returns immediately regardless of whether or not the request can be satisfied. In this case, OS_ERROR is returned if the queue is full.

Two priorities are implemented: OS_QNORMAL and OS_QURGENT. The messages are transported in FIFO order within each priority class. All OS_QURGENT messages (if any) are transported before any OS_QNORMAL message.

4.3.4.6 os_ReceiveFromQueue()

Function definition:

LONG os_ReceiveFromQueue(OS_HANDLE caller, OS_HANDLE queuehandle, OS_QDATA * data, OS_TIME timeout)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	queuehandle	queue handle of the receiver	IN
OS_QDATA *	data	pointer data struct, filled by this function	IN
OS_TIME	timeout	timeout for receive request	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error
	OS_TIMEOUT	timeout occurred

Description:

The function os_ReceiveFromQueue() waits for received messages in the queue specified by the queue handle.

For the elements of data see os_SendToQueue().

The calling task is suspended until a message is received or the specified time is expired.

4.3.5 Memory

4.3.5.1 os_CreatePartitionPool()

Function definition:

LONG os_CreatePartitionPool(OS_HANDLE caller, char * name, void * addr, ULONG num , ULONG size, OS_HANDLE * grouphandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	name of pool group	IN
void *	addr	start address of pool	IN
ULONG	num	number of partitions in pool	IN
ULONG	size	size of one partition in pool	IN
OS_HANDLE *	grouphandle	handle of the pool group created	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreatePartitionPool() creates a partition memory pool inside a memory area specified by the caller that contains fixed-sized memory buffers.

The number of buffers and its size is defined by the parameters ‘num’ and ‘size’.

ATTENTION: It has to be considered that the size of the required memory will exceed the product of ‘num’ and ‘size’ because of internal overhead.

The pool is created at the start address specified by addr. A group handle is returned for future pool access.

The partition memory pools are put together to groups identified by the group name. This enables the caller of os_AllocatePartition() to access a specific a group of partition pools. Due to this organization, it is possible to separate the different groups for different applications such as the exchange GSM primitives or sending of traces to the test interface.

The function returns the handle of the group referred by name via grouphandle. If a group with the given name does not exists, then it will be created automatically.

4.3.5.2 os_CreatePartitionPool_fixed_pool_size()

Function definition:

LONG os_CreatePartitionPool(OS_HANDLE caller, char * name, void * addr, ULONG pool_size, ULONG partition_size, OS_HANDLE * grouphandle, ULONG *num_created)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	name of pool group	IN
void *	addr	start address of pool	IN
ULONG	pool_size	size of pool in bytes	IN
ULONG	partition_size	size of one partition in bytes	IN
OS_HANDLE *	grouphandle	handle of the pool group created	OUT
ULONG *	num_created	number of created partitions	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreatePartitionPool() creates a partition memory pool inside a memory area specified by the caller that contains fixed-sized memory buffers. This number of partitions is determined by the size of the partitions and the size of the pool defined by the parameter 'pool_size'.

ATTENTION: It has to be considered that internally some bytes for check purposes are added to the 'partition_size'. This means the number of created partitions may be smaller than the result of the division of 'pool_size' by 'partition_size'. The number of created partitions is returned in the parameter '*num_created'.

The pool is created at the start address specified by addr. A group handle is returned for future pool access.

The partition memory pools are put together to groups identified by the group name. This enables the caller of os_AllocatePartition() to access a specific a group of partition pools. Due to this organization, it is possible to separate the different groups for different applications such as the exchange GSM primitives or sending of traces to the test interface.

The function returns the handle of the group referred by name via grouphandle. If a group with the given name does not exists, then it will be created automatically.

4.3.5.3 os_AllocatePartition()

Function definition:

LONG os_AllocatePartition(OS_HANDLE caller, T_VOID_STRUCT ** buffer, ULONG size, OS_TIME timeout, OS_HANDLE grouphandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
T_VOID_STRUCT **	buffer	address of allocated partition	OUT
ULONG	size	number of bytes to store in partition	IN
OS_TIME	timeout	timeout in ms for allocation request	IN
OS_HANDLE	grouphandle	group handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error
	OS_WAITED	success, but waited for free partition
	OS_TIMEOUT	no partition available in specified time

Description:

The function os_AllocatePartition() allocates a partition from the memory pool specified by the group handle.

If no partition is available at time of calling, the calling task is suspended until a partition becomes available. In this case, OS_WAITED is returned. If the specified time expired before a partition is available, OS_TIMEOUT is returned.

If the calling process is a non-task thread, the function returns immediately regardless of whether or not the request can be satisfied. In this case, OS_ERROR is returned if no partition is available.

4.3.5.4 os_DeallocatePartition()

Function definition:

LONG os_DeallocatePartition(OS_HANDLE caller, T_VOID_STRUCT * buffer)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	caller	task handle of the caller	IN
T_VOID_STRUCT *	buffer	address of partition to be freed	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DellocatePartition() frees a previously allocated partition.

4.3.5.5 os_CreateMemoryPool()

Function definition:

LONG os_CreateMemoryPool(OS_HANDLE caller, char * name, void * addr, ULONG poolsize, OS_HANDLE * poolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	name of pool	IN
void *	addr	start address of pool	IN
ULONG	poolsize	total size of pool	IN
OS_HANDLE *	poolhandle	pool handle	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateMemoryPool() creates a dynamic memory pool at the start address specified by addr. A pool handle is returned for future pool access.

4.3.5.6 os_AllocateMemory()

Function definition:

LONG os_AllocateMemory(OS_HANDLE caller, T_VOID_STRUCT ** buffer, ULONG size, OS_TIME timeout, OS_HANDLE poolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
T_VOID_STRUCT **	buffer	address of allocated buffer	OUT
ULONG	size	number of bytes to be allocated	IN
OS_TIME	timeout	timeout in ms for allocation request	IN
OS_HANDLE	poolhandle	pool handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error
	OS_WAITED	success, but waited for buffer
	OS_TIMEOUT	no buffer available in specified time

Description:

The function os_AllocateMemory() allocates a buffer of the specified number of bytes from a dynamic memory pool specified by the pool identifier.

If no buffer of the requested size is available at time of calling, the calling task is suspended until a buffer is available. In this case, OS_WAITED is returned. If the specified time expired before a memory buffer is available, OS_TIMEOUT is returned.

If the calling process is a non-task thread, the function returns immediately regardless of whether or not the request can be satisfied. In this case, OS_ERROR is returned if no buffer is available.

4.3.5.7 os_DeallocateMemory()

Function definition:

LONG os_DeallocateMemory(OS_HANDLE caller, T_VOID_STRUCT * buffer)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
T_VOID_STRUCT *	buffer	address of memory to be freed	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DellocateMemory() frees a previously allocated buffer.

4.3.5.8 os_SetPoolHandles()

Function definition:

LONG os_SetPoolHandles(OS_HANDLE ext_pool_handle, OS_HANDLE int_pool_handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	ext_pool_handle	handle of memory pool in external RAM	IN
OS_HANDLE	int_pool_handle	handle of memory pool in internal RAM	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_SetPoolHandles() is called to inform the OS layer about handles of memory pools that are needed for internal use. It need to be called before the OS API functions that need to allocate memory for internal use. These functions are:

- os_CreateOSISR(), the stack of a created OS ISR is allocated from the pool specified by 'int_pool_handle'
- os_CreatePartitionPool(), the memory allocated temporarily for the initialization of the partition guard patterns is allocated form the pool specified by 'ext_pool_handle'.

The intention of providing two pool handles is to allow the usage of pools in (fast) internal and (slower) external RAM.

ATTENTION: os_SetPoolHandles() is called by the GPF frame layer, but needs to be called explicitly if the OS layer is used ‘stand alone’.

4.3.6 Timer

4.3.6.1 os_CreateTimer()

Function definition:

LONG os_CreateTimer(OS_HANDLE caller, void (*timeoutProc)(OS_HANDLE,ULONG), OS_HANDLE * timerhandle, OS_HANLDE mempoolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
void *	timeoutProc()	called at expiration (see os_StartTimer())	IN
OS_HANDLE *	timerhandle	timer handle	OUT
OS_HANDLE	mempoolhandle	handle of memory pool for queue memory	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateTimer() creates an application timer and returns its handle.

The parameter mempoolhandle is currently not used.

4.3.6.2 os_DestroyTimer()

Function definition:

LONG os_DestroyTimer(OS_HANDLE caller, OS_HANDLE timerhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	timerhandle	timer handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DestroyTimer() deletes the application timer specified by its handle.

4.3.6.3 os_StartTimer()

Function definition:

LONG os_StartTimer(OS_HANDLE caller, OS_HANDLE timerhandle, ULONG id, OS_TIME initial_time, OS_TIME reschedule_time)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	task handle of the caller	IN
OS_HANDLE	timerhandle	timer handle	IN
ULONG	id	timer identifier	IN
OS_TIME	initial_time	initial time in ms	IN
OS_TIME	reschedule_time	rescheduling time in ms	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_StartTimer() starts an application timer specified by its handle.

The initial time is the time in milliseconds until the first timeout occurs. If the rescheduling time is set to zero, the timer expires only once, otherwise the timer will expire periodically. The period is specified by the rescheduling time.

The function os_StartTimer() transforms the timer values from milliseconds to the timer ticks required by the RTOS.

If the timer expires, the timeout procedure is called with three parameters: the handle of the entity that started the timer (caller), the handle of the task in which context the timer was started (internally set) and the timer identifier (id).

4.3.6.4 os_StopTimer()

Function definition:

LONG os_StopTimer(OS_HANDLE caller, OS_HANDLE timerhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	timerhandle	timer handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_StopTimer() stops an application timer specified by its handle.

4.3.6.5 os_QueryTimer()

Function definition:

LONG os_QueryTimer(OS_HANDLE caller, OS_HANDLE timerhandle, OS_TIME * remaining_time)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	timerhandle	timer handle	IN
OS_TIME *	remaining_time	remaining time	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_QueryTimer() returns the remaining time until (next) expiration. When the timer already expired and it is not currently running, then the remaining time is set to zero.

4.3.6.6 os_set_tick()

Function definition:

LONG os_set_tick(int tick)

Parameters:

Type	Name	Meaning	
int	tick	information about tick duration	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_set_tick() is called to inform the OS layer about the duration of the RTOS timer tick. Currently the tick duration 10ms and TDMA frame length (4.615ms) is supported. To select 10ms tick the parameter 'tick' needs to be set to SYSTEM_TICK_10_MS, to select a TDMA frame tick duration the parameter 'tick' needs to be set to SYSTEM_TICK_TDMA_FRAME.

ATTENTION: os_set_tick() is called by the in the function Initialize_Application() provided by a GPF based software package in the function xxxinit.c, but needs to be called explicitly if the OS layer is used 'stand alone'.

4.3.7 Semaphores

4.3.7.1 os_CreateSemaphore()

Function definition:

LONG os_CreateSemaphore(OS_HANDLE caller, char * name, ULONG count, OS_HANDLE * semhandle, OS_HANDLE mempoolhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	semaphore name	IN
ULONG	count	initial count (e.g. 1 for a binary sem.)	IN
OS_HANDLE *	semhandle	semaphore handle	OUT
OS_HANDLE	mempoolhandle	handle of memory pool for queue memory	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CreateSemaphore() creates a semaphore and returns its handle. The semaphores are counting semaphores with an initial value count.

The parameter mempoolhandle is currently not used.

4.3.7.2 os_DestroySemaphore()

Function definition:

LONG os_DestroySemaphore(OS_HANDLE caller, OS_HANDLE semhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	semhandle	semaphore handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_DestroySemaphore() destroys a previously created semaphore.

4.3.7.3 os_OpenSemaphore()

Function definition:

LONG os_OpenSemaphore(OS_HANDLE caller, char * name, OS_HANDLE * semhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
char *	name	semaphore name	IN
OS_HANDLE *	semhandle	semaphore handle	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_OpenSemaphore() retrieves the handle of the semaphore specified by the semaphore name.

This function may also be used to register the calling task for using the specified semaphore.

4.3.7.4 os_CloseSemaphore()

Function definition:

LONG os_CloseSemaphore(OS_HANDLE caller, OS_HANDLE semhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	semhandle	semaphore handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_CloseSemaphore() releases the handle of the semaphore specified by the semaphore name.

This function may be used to deregister the calling task from using the specified semaphore.

4.3.7.5 os_ObtainSemaphore()

Function definition:

LONG os_ObtainSemaphore(OS_HANDLE caller, OS_HANDLE semhandle, OS_TIME timeout)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	semhandle	semaphore handle	IN
OS_TIME	timeout	timeout in ms	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error
	OS_TIMEOUT	timeout occurred

Description:

The function os_ObtainSemaphore() obtains the semaphore specified by its handle, i.e. the counter is decremented, if it is greater than zero.

If the counter is equal to zero, then the calling task is suspended until the counter is incremented by another task (os_ReleaseSemaphore()) or the specified time is expired. If the calling process is a non-task thread (caller = OS_NOTASK) the function returns immediately regardless of whether or not the request can be satisfied. In this case, OS_ERROR is returned if the counter was already zero.

4.3.7.6 os_ReleaseSemaphore()

Function definition:

LONG os_ReleaseSemaphore(OS_HANDLE caller, OS_HANDLE semhandle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	semhandle	semaphore handle	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_ReleaseSemaphore() releases the semaphore specified by its handle, i.e. the counter is decremented.

4.3.7.7 os_QuerySemaphore()

Function definition:

LONG os_QuerySemaphore(OS_HANDLE caller, OS_HANDLE semhandle, ULONG * count)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_HANDLE	semhandle	semaphore handle	IN
ULONG *	count	current value of semaphore counter	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_QuerySemaphore() returns the current counter value of the semaphore.

4.3.8 Event groups

4.3.8.1 os_CreateEventGroup()

Function definition:

LONG os_CreateEventGroup(char *evt_grp_name, OS_HANDLE *evt_grp_handle)

Parameters:

Type	Name	Meaning	
char *	evt_grp_name	event group name	IN
OS_HANDLE*	evt_grp_handle	handle of the new event group	OUT

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_CreateEventGroup() returns the handle of a newly created event group with the given name.

4.3.8.2 os_DeleteEventGroup()

Function definition:

LONG os_DeleteEventGroup(OS_HANDLE evt_grp_handle)

Parameters:

Type	Name	Meaning	
OS_HANDLE	evt_grp_handle	handle of the event group to be deleted	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_DeleteEventGroup() deletes a previously created event group of the given handle .

4.3.8.3 os_SetEvents()

Function definition:

LONG os_SetEvents(OS_HANDLE evt_grp_handle, unsigned event_flags)

Parameters:

Type	Name	Meaning	
OS_HANDLE	evt_grp_handle	handle of the event group to be set	IN
unsigned	event_flags	pattern to be set	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_SetEvents() writes an event group with the given value.

4.3.8.4 os_ClearEvents()

Function definition:

LONG os_ClearEvents (OS_HANDLE evt_grp_handle, unsigned event_flags)

Parameters:

Type	Name	Meaning	
OS_HANDLE	evt_grp_handle	handle of the event group to be set	IN
unsigned	event_flags	pattern to be cleared	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_ClearEvents () deletes the given bit pattern in an event group.

4.3.8.5 os_RetrieveEvents()

Function definition:

LONG os_RetrieveEvents (OS_HANDLE evt_grp_handle, unsigned event_flags,
char option, unsigned *retrieved_events, unsigned suspend)

Parameters:

Type	Name	Meaning	
OS_HANDLE	evt_grp_handle	handle of the event group addressed	IN
unsigned	event_flags	pattern to be checked against	IN
char	option	option of logical operation	IN
unsigned*	retrieved_events	bit pattern encountered	OUT
unsigned	suspend	option whether to suspend	IN

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_ClearEvents () checks an event group for the bit pattern in ‘event_flags’, applying the operation specified in ‘option’ (AND/OR).

The calling task can be suspended or not, should the required pattern not be met, depending on the ‘suspend’ option.

The encountered bit pattern is returned, anyway, in ‘retrieved_events’.

4.3.8.6 os_EventGroupInformation()

Function definition:

LONG os_EventGroupInformation (OS_HANDLE evt_grp_handle, char *Name, unsigned* mask_evt,
unsigned* tasks_waiting, OS_HANDLE* first_task)

Parameters:

Type	Name	Meaning	
OS_HANDLE	evt_grp_handle	handle of the event group	IN
char*	name	name of the event group	IN
unsigned*	mask_evt	event bit pattern	OUT
unsigned*	tasks_waiting	number of tasks waiting	OUT
OS_HANDLE*	first_task	pointer onto first task waiting	OUT

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_EventGroupInformation() retrieves information on an event group, such as the current pattern in ‘event_flags’, the number of waiting tasks, and the entry into the first waiting task.

4.3.8.7 os_GetEventGroupHandle()

Function definition:

LONG os_GetEventGroupHandle(char *evt_grp_name, OS_HANDLE *evt_grp_handle)

Parameters:

Type	Name	Meaning	
char*	evt_grp_name	event group name	IN
OS_HANDLE*	evt_grp_handle	calling task	OUT

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_GetEventGroupHandle() finds the handle of a previously created event group with the given name.

4.3.9 System

4.3.9.1 os_GetTime()

Function definition:

LONG os_GetTime(OS_HANDLE caller, OS_TIME * time)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
OS_TIME *	time	system time in ms	OUT

Return:

Type	Meaning	
LONG	OS_OK	success

Description:

The function os_GetTime() returns the system time in milliseconds from the start of the system.

4.3.9.2 os_Initialize()

Function definition:

LONG os_Initialize(void)

Parameters: none

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_Initialize() may be used for general initialization purposes and must be called at system start.

4.3.9.3 os_ObjectInformation()

Function definition:

LONG os_ObjectInformation(OS_HANDLE caller, ULONG id, OS_HANDLE handle, ULONG len, char * buffer)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	calling task	IN
ULONG	id	object identifier	IN
OS_HANDLE	handle	object handle	IN
ULONG	len	length of the buffer	IN
char *	buffer	buffer to write information	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_ObjectInformation() may be used to get information about the OS (and RTOS) 'objects' that exist in the system. Each object is defined by an object identifier (OS_OBJTASK, OS_OBJQUEUE, OS_OBJPARTITIONGROUP, OS_MEMORYPOOL, OS_OBJTIMER, OS_OBJSEMAPHORE) and an object handle.

The functions returns OS_ERROR if the identifier is unknown or if the handle exceeds the max handle for objects of this type. If the handle is lower than that maximum but no object exists for this handle, then an empty string ("") is written to the buffer.

There is an extra identifier (OS_OBJSYS) to get system information, the handle is ignored in this case.

The information and the format written to the buffer depends on the information provided by the RTOS.

4.3.9.4 os_SystemError()

Function definition:

void os_SystemError(OS_HANDLE Caller, ULONG cause, const char *file, int line)

Parameters:

Type	Name	Meaning	
OS_HANDLE	caller	task handle of the caller	IN
ULONG	cause	error/warning code	IN
consr char *	file	file where error occurred	IN
int	line	line where error occurred	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_SystemError() may be used for the exception handling in case of abnormal system states detected by the frame. The frame can detect a subset of all possible errors like a lack of resources which dimensions are under control of the frame. The frame cannot detect CPU exceptions like ‘abort data’. The error and warning codes that may be passed by the frame to os_SystemError() are listed in 4.2.2.

The function os_SystemError() does not necessarily have to be filled with any error handling code for proper system functionality. It may be used to generate additional debug information in case of abnormal system states detected by the frame.

os_SystemError() will reset the system in case an the passed *cause* parameter indicates an error.

4.3.9.5 os_IncrementTick()

Function definition:

LONG os_IncrementTick (OS_TICK ticks)

Parameters:

Type	Name	Meaning	
OS_TICK	ticks	ticks to be added to the current OS ticks	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_IncrementTick() is used to increment the RTOS system clock by the number of ticks specified by the parameter *ticks*.

ATTENTION: Compared to the rest of the OS abstraction layer API the function os_IncrementTick() expects ticks instead of milliseconds.

4.3.9.6 os_RecoverTick()

Function definition:

LONG os_RecoverTick (OS_TICK advanced_ticks)

Parameters:

Type	Name	Meaning	
OS_TICK	advanced_ticks	ticks to be added to the current OS ticks	IN

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_RecoverTick() is used to adjust the RTOS system clock and timer management services when leaving a sleep mode in which the system tick was stopped. This system ticks is the reference clock for the RTOS and in case it is stopped, also the timers in the RTOS are no nonger running.

The RTOS system clock is set to its current value (ticks when system tick was stopped before entering sleep) plus the value specified by the parameter *advanced_ticks*. The value of *advanced_ticks* needs to be retrieved from the power management that controls the sleep modes.

ATTENTION: Compared to the rest of the OS abstraction layer API the function os_RecoverTick() expects ticks instead of milliseconds.

4.3.9.7 os_GetInactivityTicks()

Function definition:

LONG os_GetInactivityTicks (int *next_event, OS_TICK *next_event_ticks)

Parameters:

Type	Name	Meaning	
int *	next_event	next event known	OUT
OS_TICK *	next_event_ticks	ticks until next action in RTOS	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_GetInactivityTicks() delivers information about the next synchronous event in the RTOS. If **next_event* returns OS_NO_EVENT there is no next event pending in the RTOS. If **next_event* is OS_EVENT the number of OS ticks until the next application or task timer in the RTOS are returned in **netx_event_ticks*.

ATTENTION: Compared to the rest of the OS abstraction layer API the function os_GetInactivityTicks() returns ticks instead of milliseconds.

4.3.9.8 os_GetScheduleCount()

Function definition:

LONG os_ GetScheduleCount (int task_handle, int *schedule_count)

Parameters:

Type	Name	Meaning	
int	task_handle	task handle	IN
int *	schedule_count	number of time a task was scheduled	OUT

Return:

Type	Meaning	
LONG	OS_OK	success
	OS_ERROR	error

Description:

The function os_GetScheduleCount() returns the number of times a task specified by *task_handle* was scheduled by the RTOS since boot time in the parameter **schedule_count*. If the *task_handle* is set to OS_NOTASK the counter value for the currently running task is returned.

Appendices

A. Acronyms

DS-WCDMA Direct Sequence/Spread Wideband Code Division Multiple Access

B. Glossary

International Mobile Telecommunication 2000 (IMT-2000/ITU-2000) Formerly referred to as FPLMTS (Future Public Land-Mobile Telephone System), this is the ITU's specification/family of standards for 3G. This initiative provides a global infrastructure through both satellite and terrestrial systems, for fixed and mobile phone users. The family of standards is a framework comprising a mix/blend of systems providing global roaming. <URL: http://www.imt-2000.org/>