FIELDBUS FOUNDATION Field Device Software
PROFIBUS PA Device Software

OSIF User Manual

Version 2.41

Date: November, 2011

SOFTING Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Phone (++49) 89 45656-0
Fax (++49) 89 45656-399

Table of Contents

Terms and Abbreviations

CPU	Central Processing Unit
EEP	EEPROM Handling Component
EEPROM	Electrically Erasable Read Only Memory
FD	Field Device
ID	Identifier
IF	Interface
OS	Operating System
OSIF	(Softing FD) Operating System Interface
RTOS	Real Time Operating System

References

/1/ embOS User's Manual; Segger Microcontroller Systeme GmbH

1 Introduction

The Softing Field Device (FD) Software is developed as a widely portable – in terms of supported CPU types, compilers, operating systems etc. – software, which allows easy adoption to its runtime environment. As one of its features, it shall not only support, but even take advantage of different real-time multitasking operating systems. Additionally, it shall also be able to run without any (predefined) operating system.

This is achieved by an OS abstraction layer called OSIF, which allows to assimilate the properties and features of different commercial operating systems to a common base, but also to upgrade missing features. This ends up being an OS replacement, if there is no underlying system available.

OSIF defines a common set of functions, which are used by the FD software, and also a system startup strategy to initialize and start the several components as part of the resulting system. The application programmer may of course make use of these functions for his own benefit; if he wants to interface with certain FD software components, he needs to make use of these functions, since some of the FD interfaces are based on OSIF.

This manual does not deal with OSIF porting issues, i.e. does not go into OSIF internals and how to map them to underlying operating systems. It will just show, how to make use of the OSIF concepts and functions, which may be a requirement for any device application programmer to interface the Softing FD software.

2 OSIF concepts

Similar to general RTOS systems, the basic OSIF units are tasks, which communicate by events. Events may be connected to mailboxes to be able to transport more structured data than just simple flags. Events may be generated by tasks themselves, but also by timers or interrupts in general. Additionally, for inter-task synchronization issues, resource semaphores are provided, which may be seized by tasks and may lead to the suspension of a task, if already blocked.

2.1 OSIF Tasks and Events

In contrary to standard RTOS systems, an OSIF task consists of a routine, which is called only, if an event addressing this task is received. The task gains the event(s), that occurred, passed as parameter when called. There is no need to actively wait for the events by the task routine; this is already handled by OSIF. The task functions, as provided by the OSIF user, act just as an event handler for the expected event.

Tasks are identified by their ID, some kind of a handle, which has a range between 1 and MAX_TASK_ID. Additionally, there are some ‘virtual’ task IDs outside of this range, which are needed e.g. to specify an interrupt service routine as the initiator of a message.

When ‘creating’ the task (see below ...), the events that are expected by the handling function are to be defined. The handler will be called only for these events; any other are ignored at this time (but not deleted). As soon as the handler is called, i.e. the event is passed to handler, it is cleared by means of OSIF. Thus, if during event handling another event is received, the handler will be called again immediately after return.

Events are not queued, but only passed in parallel to the handler; It is not possible to determine, which of the events, if more than one is received, has been generated first. For OSIF there is no event prioritizing. It is a matter of the handling function, in which sequence it serves the events.

All tasks must be configured to let OSIF know, what events they expect to deal with. Events sent to a task which is not configured to handle these events will lead to an exception.

One special event, which needs not to be configured, as it will be sent to every task independent of it configuration, is OS_IF_EVENT_SYSSTART. For further details refer to chapter “2.6 System Start-up”.

2.2 Messages

To allow more structured inter-task-communication, messages may be used. By task configuration (during compile time) a number of mailboxes may be created for any task; each mailbox will be connected to certain event, which will be generated for the task, when it receives a mail at the according mailbox. Events are not duplicated, if more than one message appears in the mailbox; the mailbox has to be polled, until all messages are received.

The functions for message handling are non-blocking; thus, due to the OSIF event handling concept, it is not possible for directly awaiting a message, but instead awaiting the event connected to that mailbox.

The maximum number of messages per mailbox is a configuration constant, which has to be set during compile time. The number of mailboxes per task is limited by the number of different events, the underlying operating system is able to deal with, and therefore is an implementation constant.

2.3 Timer and Interrupts

Interrupt handling in general does not differ from ‘normal’ systems, i.e. those not using OSIF. However, if interrupt routines have to interact with OSIF mechanisms, some certain precautions have to be met.

As soon as OSIF functions are to be called from an interrupt context, the system must be informed about that fact. Usually, this is done by calling a certain function at interrupt entry, and another function right before exit. For certain operating systems, this must be done inside special interrupt handlers, which have to be written in assembler, and thus are not able to be implemented in general by OSIF. In the following, the use of an operating system like embOS (/1/) is shown, which allows an easier interaction that could be mapped to common-purpose OSIF functions.

In general, only the functions to generate events ("*osif_set_event ()*") or to send a message ("*osif_send_message ()*") are allowed to be used from an interrupt context. The call of any other function inside an interrupt service routine may lead to a FATAL exception in the best case, to unpredictable results in the worst case.

OSIF timers work in a quite simple way: after being created they can be started and, if needed, stopped. When the timer expires, it sends the event OS_IF_EVENT_TIMER to the related task.

Timers may be considered as a special form of interrupts; since most operating systems provide a certain interface for timing functions, and the purpose of timers has a smaller range than other interrupts in general, OSIF provides an easy-to-use interface to deal with timers. Of course, this also implies some limitations for their functionality; if needed, the appropriate OS functions may be used, which may interact with OSIF similar to interrupts.

Within OSIF, timers are connected to tasks in a one-to-one relationship, i.e. each timer sends the event to exactly one task, and each task may receive timer events from exactly one timer. The relationship between timer and task ID is configured at creation of the timer by call of "*osif_create_timer ()*".

When starting a timer, the timeout value has to be specified in units of milliseconds. OSIF will convert the value to internal timer ticks automatically. Additionally it must be specified, whether the timer shall stop after expiration ("one-shot" mode) or restart with its original start value and thus produce the OS_IF_EVENT_TIMER periodically ("cyclic" mode).

2.4 Resource Semaphores

Resource semaphores provide a special way of inter-task-synchronization, simultaneously to events. OSIF implements semaphores completely independent of other mechanisms. To handle concurrent access (i.e. from different tasks) to one and the same resource, semaphores are a feasible way. The usage of a semaphore may be done from anywhere in a task context, but not from an interrupt context.

There are two possibilities to deal with semaphores: the one is to check for the availability of a certain resource and, if denied, continue with something else. The other option is to wait, until the resource is available, thus suspending further task execution for that time.

To avoid deadlock situations, most operating systems provide a mechanism so called priority inversion, which boosts the priority of the task currently holding a semaphore, which a higher priority task is waiting for. This mechanism, if existent, is not affected in any way by OSIF and will work as usual.

2.5 OSIF Configuration

The way in which tasks are created and configured, differs in a wide range from one operating system to another. To keep the interface within OSIF identical and easy for all supported systems, the aforementioned issue is subdivided into two parts: the configuration aspect is separated in one or more files which takes care to configure the several tasks during compile time. In a second step during runtime, the task is activated through a common interface, with only a few variable parameters. This is done by calling function "*osif_create_task ()*", which allows only to set the task's priority and the address of the event handler (which is in fact the task's main function). Thus, the priority system of the underlying operating system is harmonized in a way, that valid priorities range from 2 to 254, with the effective priority being higher, the higher the priority number. The priorities 0 and 1 as well as 255 are reserved for OSIF internal purposes.

The task configuration consists at least of the files `OSIF_CFG.C`, `OSIF_CFG.H` and some global definitions in `OSIF.H`. Depending on the operating system these files may be accompanied by further files due to OS specific configuration aspects. For details of task configuration please refer to chapter 3.8.

2.6 System Start-up

The start-up process divides into two main parts: the component initialization, where the component "*nnn_init ()*" is called by OSIF, and the start sequence, where a special event is sent to each activated task.

The component initialization has the main goal to set the appropriate component into a working state, i.e. initialize its internal data structures, create the task(s) by connecting the event handlers and set up the component specific interrupt system (as far as not to be done in the global processor initialization). There must not be any interaction between the components or tasks until all initializations have completed. Since the task system is not running in this state, no OSIF functions concerning task execution etc. must be called.

After control is passed to the OS scheduler, all tasks – with the sequence according to the task priorities - are accessed with the special event OS_IF_EVENT_SYSSTART. This event is generated exactly once for each task; no other event will be generated beforehand (even though other events may be generated at the same time; thus, the OS_IF_EVENT_SYSSTART should be served with highest priority in the event handler, i.e. before all other events). Now the component interactions may be set up, e.g. accessing the EEPROM, exchanging events and messages and using semaphores. The start-up phase has completed after the last (i.e. lowest priority) task has served its event OS_IF_EVENT_SYSSTART.

If ever possible, OSIF contains in module `OSIF_MN.C` the system's "*main ()*" function. After the necessary initializations for the operating system are done, the initialization functions for the EEP-IF ("*eep_init ()*") and the communication layer ("*cif_init ()*") are called. Finally, the application component is initialized by call of the user provided function "*appl_init ()*".

2.7 Exception Handling

Since, for most problem situations, there is no useful error recovery method on embedded devices (as far as the errors cannot be communicated to the host system and handled there or even result from errors cause by the communication network), errors may lead to an exception in most cases. The Softing FD software distinguishes four classes of exceptions:

- Warnings
- Assertions
- Errors
- Fatal Errors

Warnings in general just want to give information about an unusual situation, but do not affect program execution in general – apart from the execution time of the exception handler when possibly outputting a related message. For release version, warnings are skipped in general.

Assertions are used to provide checkpoints within the code during development. The source file name and the line of code, where the assertion was generated, are passed as parameters to the exception handler. They are also skipped in release versions.

Errors indicate problem situations that refrain from proper code execution. Some additional information will be passed to the exception handler.

Fatal errors are generated in situations, where no further code execution is possible. Information passing is similar to 'normal' errors. Fatal errors in general lead to the restart of the device and are always active.

Depending on the generation target (e.g. emulator version, release version etc.), special reactions are possible: e.g. for the M16C CPU environment, there is a fixed breakpoint inserted in the exception handler, causing the emulator to stop when an exception other than WARNING occurs.

The File `EXCEPT.H` defines the data structures and exception encoding to be passed to the exception handler. Additional codes may be entered, but existing definitions must not be changed or removed.

One sample exception handler with name "*default_exception_handler ()*" is contained in module `OSIF_MN.C` beneath the "*main ()*" function. The Softing FD software generates exceptions by call of "*exception_handler ()*"; during the linker process, the symbol "*exception_handler ()*" is connected to function "*default_exception_handler ()*". This provides maximum flexibility to the device application programmer, as he needs not to enter changes to the original "*default_exception_handler ()*" function, but may write his own exception handler according to his requirements and just connect it during link time instead of the default handler.

3 OSIF Interface

3.1 Data Types and Definitions

The most important definitions concerning OSIF and its usage are found in file OSIF.H. Here, the IDs of the several tasks as created by the Softing FD software are fixed as well as their priorities. Next, some definitions for events are made.

The user may introduce additional definitions, but must not change any of those already stated here; otherwise, the FD software won't work properly. Since the number of events is very limited, they can only be used according to the following rule: the event definitions have to be unambiguous as seen from each task, that it has to deal with. In other words: different events may have the same value, as long as they are not used by one and the same task. E.g. this rule applies to OSIF_EVENT_EEPROG and OS_IF_EVENT_REQRES: OSIF_EVENT_EEPROG is used by the EEP task only, which on the other hand does not deal with OS_IF_EVENT_REQRES; thus, the two events may have the same value without interfering. This must be verified by checking the allowed events for the tasks concerned.

Next, the data structure needed to transport messages is defined. Each message transported by OSIF must start with the data structure T_OS_MSG_ HDR, which is used to route the message to the appropriate mailbox:

```
typedef struct _T_OS_MSG_HDR
{
    USIGN8    message_id;                /* Message-Identifier      */
    USIGN8    snd_task_id;               /* send task identifier    */
    USIGN8    rvc_task_id;               /* receive task identifier */
    USIGN8    user_reserved;             /* free for user           */
} T_OS_MSG_HDR;
```

The data structure T_USR_MSG_BLOCK, which is used for the FD internal message exchange, may serve as a sample of how to build message structures.

Depending on the functionality provided by the OS used, some of the functions specified in the following may be realized as macros and thus also defined in OSIF.H, which serves in these cases just as a mapping and renaming instance.

Additional definitions concerning configuration issues are made in file OSIF_CFG.H. For further details please refer to chapter "3.9 OSIF Configuration".

3.2 OSIF Tasks and Events

As already mentioned, OSIF tasks are identified by their ID, which is basically valid in the range from 1 to MAX_TASK_ID. Additionally, for all functions (except "*osif_create_task ()*", of course), the according task must be existent (i.e. described in OSIF_CFG.C) and created (by call of "*osif_create_task ()*").

There is one special ID named CURRENT_TASK_ID, which may be used in some cases, where the currently active task is addressed. In general, this is allowed only, when this task ID is unambiguously known, e.g. for the functions "*osif_wait_event ()*", "*osif_get_msg ()*", "*osif_send_msg ()*" for the sending task and if not from an interrupt service routine, as well as the semaphore handling functions.

3.2.1 osif_create_task ()

```
FUNCTION GLOBAL VOID osif_create_task
  (
    IN USIGN8 task_id,
    IN USIGN8 task_prio,
    IN VOID   (FAR_C *task_main_func) (T_EVENT),
    IN T_EVENT event_mask
  );
RETURN_VALUES:
  None
```

This function is used to create a task. Parameter *task_id* passes the ID of the task to be created, its priority in a range of 2 to 254 (the higher the number, the higher the effective priority). Parameter *task_main_function* points to the task's event handler, which expects to receive events as specified in parameter *event_mask*. The task cannot be deleted after creation. If parameter *task_prio* is set to 0, the predefined priority as specified in the task configuration will be used.

In case of any error the global exception handler is called.

3.2.2 osif_get_current_task_id ()

```
FUNCTION GLOBAL USIGN8 osif_get_current_task_id (VOID)

RETURN_VALUES:
  ID of the current running task
```

Evaluates the ID of the currently running task.

3.2.3 osif_disable_task ()

```
FUNCTION GLOBAL VOID osif_disable_task
  (
    IN USIGN8   task_id
  );
RETURN_VALUES:
  None
```

The task addressed by parameter *task_id* is refrained from being executed by setting its priority to 0. To bring the task back to normal execution the function "*osif_enable_task ()*" (see below) is provided.

3.2.4 osif_enable_task ()

```
FUNCTION GLOBAL VOID osif_enable_task
  (
    IN USIGN8   task_id
  );
RETURN_VALUES:
  None
```

Re-enables a previously disabled task by restoring its original priority level. The task affected is identified by the parameter *task_id*. The function does not check, if the task really has been disabled before.

3.2.5 osif_disable_all_tasks ()

```
FUNCTION GLOBAL VOID osif_disable_all_tasks (VOID);

RETURN_VALUES:
  None
```

Refrains all other than the calling task from being executed by stopping the task scheduler. Scheduling will be restarted by use of "*osif_enable_all_tasks ()*" (see below). Interrupts are not affected, i.e. will continue to work. In any case, it must not be used from an interrupt service routine – it does not make any sense, as scheduling is disabled during interrupt handling anyway, and scheduling must be re-enabled in any case within the interrupt handler; otherwise the system will probably run into a deadlock situation.

As the use of this function has a profound impact on the global timing and latency behavior of the whole system, the time period while having task scheduling disabled should be kept as short as possible (comparable to disabling interrupts).

3.2.6 osif_enable_all_tasks ()

```
FUNCTION GLOBAL VOID osif_enable_all_tasks (VOID);

RETURN_VALUES:
  None
```

Restarts the task scheduler after being disabled by "*osif_disable_all_tasks ()*". It has to be used in conjunction with "*osif_disable_all_tasks ()*".

3.2.7 osif_set_event ()

```
FUNCTION GLOBAL VOID osif_set_event
  (
    IN USIGN8 task_id,
    IN USIGN8 event
  );
RETURN_VALUES:
  None
```

This function is used to send the event(s) as given in parameter *event* to the task specified by parameter *task_id*.

The function may be called from an interrupt service routine to pass events to the task system. However, parameter *task_id* must address an existing and configured task only; virtual task IDs are not allowed here.

3.2.8 osif_wait_event ()

```
FUNCTION GLOBAL T_EVENT osif_wait_event
  (
    IN  USIGN8 task_id,
    IN T_EVENT event_mask
  );
RETURN_VALUES:
  event data
```

"*osif_wait_event ()*" is used to receive an event addressed to the calling task. Due to the task structure of OSIF, the explicit use of the function is normally not needed, as it is part of the OSIF task frame. For special cases of course, where a task wants to wait for events outside the standard event handler, it may be used anywhere in the task context.

The call to "*osif_wait_event ()*" is blocking, i.e. the function does not return, until one or more events matching the expected ones as defined in parameter *event_mask* are received by the task identified by parameter *task_id*. By return of the function, the events, as indicated by the result value, are OSIF internally cleared, i.e. no more events exist on subsequent calls to "*osif_wait_event ()*" (until the according events are received again). The result contains only events that have been specified to be expected; any other event will not be indicated, but will also not be cleared; thus it is possible to serve other events with another call to "*osif_wait_event ()*" by specifying those events in parameter *event_mask*.

3.3 Messages

Messages in general may be of any size and structure in OSIF, since only pointers to the certain messages are transferred. Consequently, OSIF does not care about the message data block itself. The OSIF user is responsible to provide accessible memory for the message and to dispose it after use.

Messages are stored in mailboxes, which are directly connected to input events of the according task. Thus, each task must be configured properly, which events have an underlying mailbox and how many messages can be stored in each of these mailboxes. This configuration is done at compile time in file `OSIF_CFG.C`.

3.3.1 osif_send_msg ()

```
FUNCTION GLOBAL VOID osif_send_msg
  (
    IN USIGN8 msg_event,
    IN USIGN8 rcv_task_id,
    IN USIGN8 snd_task_id,
    IN T_OS_MSG_HDR FAR_D *msg_ptr
  );
RETURN_VALUES:
  None
```

A data block addressed by parameter *msg_ptr* is transferred from the task addressed by parameter *snd_task_id* (which may also be a virtual task ID in case if called from an interrupt service routine or CURRENT_TASK_ID) to the mailbox indicated by parameter msg_event of the task specified by parameter *rcv_task_id*. The according event is generated automatically.

3.3.2 osif_get_msg ()

```
FUNCTION GLOBAL T_OS_MSG_HDR *osif_get_msg
  (
    IN USIGN8 task_id,
    IN T_EVENT event
  );
RETURN_VALUES:
  pointer to the received message
  NULL if no (more) message is available
```

Messages sent by "*osif_seng_msg ()*" are received by the function "*osif_get_msg ()*". Parameter *task_id* identifies the task (CURRENT_TASK_ID is a valid value here), while parameter *event* specifies the appropriate mailbox of the task. Reception of a message is in general combined with reception of an event, which at the same time indicates the mailbox, where the message resides. While the event reception is signaled implicitly by calling the task event handler respectively return of function "*osif_wait_event ()*", according messages have to be retrieved explicitly by call of "*osif_get_msg ()*".

Events are not duplicated, if more than one message is present in the mailbox. Thus, after receiving the according message event, the mailbox must be polled until all messages are read out, i.e. until "*osif_get_msg ()*" returns NULL. The event handler must be prepared, that even no message may be present in the mailbox.

3.4 Timer

Functions controlling timers may be accessed only from within a valid task context or during initialization. When used in the initialization context, care must be taken, that the related task will be created at latest, when the timer expires. Otherwise the timer will try to send an event to a non-existing task, which will lead to an OSIF exception.

3.4.1 osif_create_timer ()

```
FUNCTION GLOBAL USIGN8 osif_create_timer
  (
    USIGN8      task_id
  );
RETURN_VALUES:
  Timer ID the timer is accessible through
```

Creates a timer and connects it to task identified by parameter *task_id*, which will receive an OS_IF_EVENT_TIMER, when the timer expires. Only one timer may be connected to a single task; a single timer can refer to one task only. Once created, the timer cannot be deleted any more. If the new timer could be created, its ID is returned; otherwise an exception is generated.

3.4.2 osif_start_timer ()

```
FUNCTION GLOBAL VOID osif_start_timer
  (
    IN USIGN8   timer_id,
    IN USIGN32  timer_value,
    IN BOOL     cyclic
  );
RETURN_VALUES:
  None
```

Starts the timer addressed by parameter *timer_id* with the timeout value in parameter *timer_value* in units of milliseconds. If parameter *cyclic* is TRUE, the timer will work in a periodic manner, i.e. keep generating events until the timer is stopped. If parameter *cyclic* is set to FALSE, the timer stops automatically after expiration, i.e. works as a one-shot time with generating the event only once.

If the timer is still running, it will be restarted with the new timeout (not generating an event, however).

3.4.3 osif_stop_timer ()

```
FUNCTION GLOBAL VOID osif_stop_timer
  (
    USIGN8      timer_id
  )
RETURN_VALUES:
  None
```

Stops the timer addressed by parameter *timer_id*. As the timer won't expire any more, no event will be generated. Nothing happens, if the timer does not run, either due a previous call to "*osif_stop_timer ()*" or because the timer already expired.

3.5 Interrupts

3.5.1 DISABLE_INTERRUPTS ()

```
FUNCTION GLOBAL VOID DISABLE_INTERRUPTS (VOID);

RETURN_VALUES:
  None
```

This function disables all interrupts by using methods conformed to the underlying operating system and the CPU type used.

3.5.2 RESTORE_INTERRUPTS ()

```
FUNCTION GLOBAL VOID RESTORE_INTERRUPTS (VOID);

RETURN_VALUES:
  None
```

This function restores the interrupt enable status to the value before the call of the associated "*DISABLE_INTERRUPTS ()*" function by using methods conform to the underlying operating system and the CPU type used. This allows usage of "*DISABLE_INTERRUPTS ()*" and "*RESTORE_INTERRPTS ()*" as nested pairs without conflicting with preceding calls.

3.5.3 ENTER_IRQ ()

```
FUNCTION GLOBAL VOID ENTER_IRQ (VOID);

RETURN_VALUES:
  None
```

The call of this function indicates the start of an interrupt service to the operating system. This should arise as the first instruction in an interrupt handler, which makes use of OSIF mechanisms.

3.5.4 LEAVE_IRQ ()

```
FUNCTION GLOBAL VOID LEAVE_IRQ (VOID);

RETURN_VALUES:
  None
```

This function indicates the completion of an interrupt service to the operating system. This should arise as the last instruction in an interrupt handler, which makes use of OSIF mechanisms.

3.6 Resource Semaphores

3.6.1 osif_ini_sema ()

```
FUNCTION GLOBAL VOID osif_init_sema (VOID)

RETURN_VALUES:
  None
```

This function is used to initialize the semaphore management.

3.6.2 osif_create_sema ()

```
FUNCTION GLOBAL USIGN8 osif_create_sema
  (
    IN  USIGN8 task_id
  );
RETURN_VALUES:
  ID of newly created semaphore
```

Creates a resource semaphore for a task specified by parameter *task_id*. The resource is not blocked after creation.

3.6.3 osif_delete_sema ()

```
FUNCTION GLOBAL USIGN8 osif_create_sema
  (
    IN  USIGN8 task_id,
    IN  USIGN8 sema_id
  );
RETURN_VALUES:
  None
```

Deletes a resource semphore specified by parameter *task_id* and parameter *sema_id*.

3.6.4 osif_use_sema ()

```
FUNCTION GLOBAL VOID osif_use_sema
  (
    IN  USIGN8 task_id,
    IN  USIGN8 sema_id
  );
RETURN_VALUES:
  None
```

Claims the resource semaphore addressed by parameter *sema_id* for the task specified by parameter *task_id*. CURRENT_TASK_ID is a valid value for parameter *task_id* here. If the resource is already blocked by another task, the calling task is suspended and will resume execution as soon as the semaphore is released and is not being used by any higher priority task.

Depending on the underlying operating system a priority inversion mechanism may be provided, boosting the priority of the task currently owning the resource, if a higher priority task is waiting.

3.6.5 osif_request_sema ()

```
FUNCTION GLOBAL USIGN8 osif_request_sema
  (
    IN  USIGN8 task_id,
    IN  USIGN8 sema_id
  );
RETURN_VALUES:
  0     => Resource was not available
  1     => Resource was available, is now blocked by calling task
```

Requests to claim resource semaphore addressed by parameter *sema_id* for the task specified by parameter *task_id*, where CURRENT_TASK_ID is a valid value. If the semaphore is still free, "*osif_request_sema ()*" claims it and returns 1 as result. If not, 0 is returned immediately. This call is non-blocking, i.e. the task continues execution in any case.

3.6.6 osif_check_sema ()

```
FUNCTION GLOBAL USIGN8 osif_check_sema
  (
    IN  USIGN8 sema_id
  );
RETURN_VALUES:
  0     => Resource is not available, i.e. blocked by at least one task
  1     => Resource is available
```

Checks the current status of the resource semaphore addressed by parameter *sema_id* without attempting to claim it. The result value 0 indicates, the semaphore is used; if it is available, 1 is returned.

3.6.7 osif_release_sema ()

```
FUNCTION GLOBAL VOID osif_release_sema
  (
    IN  USIGN8 task_id,
    IN  USIGN8 sema_id
  );
RETURN_VALUES:
  None
```

Releases the resource semaphore identified by parameter *sema_id* previously used by the task specified by *task_id*. CURRENT_TASK_ID may be used as value for parameter *task_id*.

3.7 Memory Management

3.7.1 osif_malloc ()

```
FUNCTION GLOBAL VOID * osif_malloc
  (
    IN USIGN16 mem_size
  );
RETURN_VALUES:
  pointer to the new memory block
```

This function allocates memory in a thread-safe way. The size of the requested memory is specified by *mem_size*.

3.7.2 osif_free ()

```
FUNCTION GLOBAL VOID osif_free
  (
    IN VOID * mem_block
  );
RETURN_VALUES:
  None
```

This function frees memory in a thread-safe way. The allocated memory is addressed by parameter *mem_block*.

3.8 System Time

3.8.1 osif_get_system_time()

```
FUNCTION GLOBAL VOID osif_get_system_time
  (
    OUT T_SYS_TIME * sys_time
  );
PARAMETERS:
  system_time     Location where to store the time value

RETURN_VALUES:
  None
```

In FF devices this function provides the current application clock time. It is updated by system management, and the application clock synchronization service is used to keep it coordinated with the application clocks of other devices.

Its value represents the number of 1/32 millisecond ticks since January 1, 1972.

The structure of the time value is:

```
struct {
  USIGN32 high;   most  significant part  (units of 2^27 ms (ca. 37,3h))
  USIGN32 low;    least significant part  (units of 2^-5 ms (=1/32 ms) )
} T_SYS_TIME;
```

As PROFIBUS PA does not know an application clock time, in PA devices this function provides a constant value of 0.

3.9 OSIF Configuration

The configuration of OSIF depends largely on the properties of the underlying operating system. Thus, only basic rules and some samples for a concrete implementation may be given here. The configuration aspects of OSIF are condensed mainly in files `OSIF_CFG.C` with some common definitions in `OSIF_CFG.H`.

Due to the event structure of OSIF, it is important to define the list of allowed events for each task (defined as NNN_EVENTS) as well as the connected mailboxes (NNN_MSGEVT). All events with mailboxes, asserted of course, must be contained in the list of allowed events. For memory management reasons, also the max. number of messages per mailbox must be specified, separately for each task (not for each mailbox).

The number of different events also depends on the operating system and is abstracted to type T_EVENT. Common to most real time operating systems is the usage of events as a field of single bits, which are combined into one unit (octet or short integer etc.).

All tasks to be used are described in an array of element type TASK_CFG, which is evaluated during start-up in function "*osif_init ()*". Besides for the information already mentioned, settings concerning stack configuration, debug support like task names etc are also entered here.

3.10 System Start-up

OSIF itself contains several initialization functions, which are executed during the start-up process. The most important one is "*osif_init ()*", which reads the task configuration array, runs some consistency checks and prepares all task's events and mailboxes. Additionally, the system's idle task is created, if needed.

The timer and the semaphore module contain initialization functions of their own, which are not necessarily called during start-up, but when the first use of the according component is executed. This allows for the linking of modules only in the case, when it is actually accessed from outside of the OSIF.