HIGHTEC

PXROS-HR Kernel v8.2.0

User's Guide

Table of Contents

1. Introduction

The following document describes the features and application of the operating system PXROS-HR. It is subdivided into two parts:

The first part provides a general outline and describes the philosophy, special characteristics and architecture of PXROS-HR.

The second part explains in depth the practical application and programming of the system; examples show how a PXROS-HR application is constructed.

1.1. What is PXROS-HR?

PXROS-HR (**P**ortable e**X**tendible **R**eal-time **O**perating **S**ystem - **H**igh **R**eliability) is a real-time operating system for embedded systems. PXROS-HR is portable and extendible with a main emphasis on runtime safety.

Portability is restricted to processor architectures providing memory protection mechanisms via a Memory Protection Unit (MPU), which makes it possible to protect memory areas of arbitrarily small size.

1.2. Why an operating system?

An operating system is an instrument for managing the resources of a system. It helps to reduce the complexity of application development and relieves the application designer from performing standard tasks, such as memory management, thus reducing error probability and speeding up the development process.

Another advantage of an operating system is the portability of an application: The operating system abstracts hardware specific details with an additional software layer and provides standardized interfaces to access platform-specific functionality.

The basic task of an operating system is to manage the hardware resources and provide them to the application software. In essence, this means memory and process management.

Furthermore, an operating system can provide interfaces to other services, e.g. file system or network services.

1.3. What is a real-time operating system?

A real-time operating system is not only expected to perform a calculation process correctly, but also to provide the result of said calculation in due time, meaning within a predictable period of time. In general, two classes of real-time are distinguished:

Hard real-time absolutely requires to keep all time limits. If only one of the defined time limits is exceeded, this will cause a unusable result (see Figure 1).

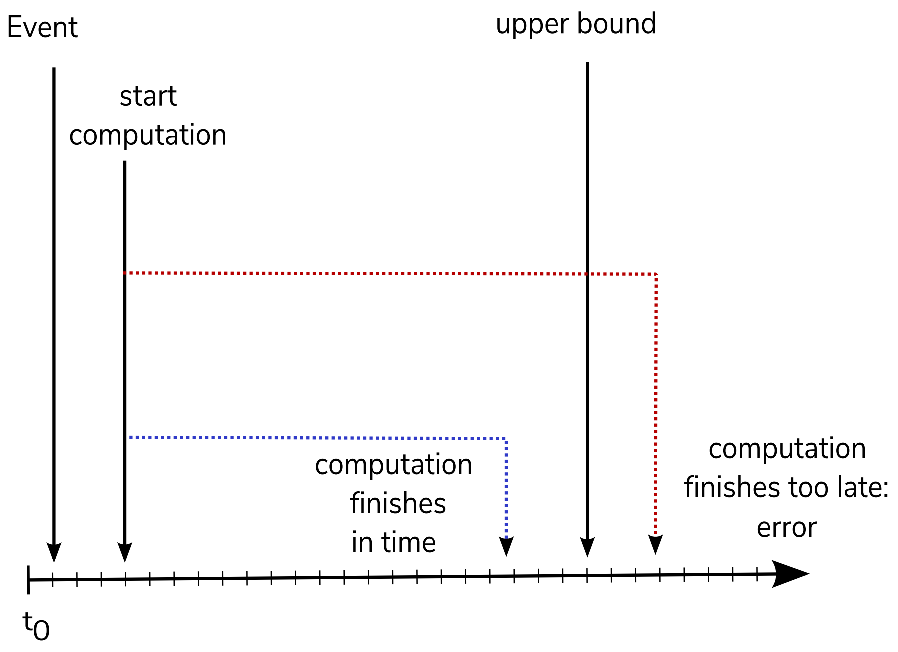
Event
upper bound
start computation
computation finishes in time
computation finishes too late: error
t_0

Figure 1. Hard real-time

Soft real-time is the more relaxed type and tolerates a few time limits to be exceeded in individual cases without causing unusable results.

A real-time operating system is characterized by the fact that it guarantees real-time conditions on operating system level. This usually means providing mechanisms that allow the application software to react to (external) events in time.

Examples:

The process of recording, processing and transferring audio data has to be performed within certain time limits. If these limits are exceeded from time to time, minor disturbances ('pops') within the audio stream will be the result, which have no negative effects. This is an example for a soft real-time requirement.

The process of reading and analyzing a measured signal, e.g. in a motor control, and the reaction to this signal have to be performed within a certain time period. If this is not the case, disruptions might occur or the motor might break down. This is an example for a hard real-time requirement.

1.4. Special features of PXROS-HR

As a multitasking operating system with hard real-time characteristics, PXROS-HR allows to run several processes, called tasks, in a quasi-parallel way. Additionally, handlers can be installed with a higher priority to react to hardware or software interrupts.

Since several program components, i.e. tasks and handlers, run on the same CPU, the available processing time has to be shared among them. This process is called *scheduling* and is performed by PXROS-HR. Priorities that are assigned to program components allow a precise adjustment of the scheduling behavior whenever certain events (e.g. receiving a message) occur.

In comparison to other real-time operating systems, PXROS-HR has some special features, such as:

- Hardware-assisted memory protection by a Memory Protection Unit (MPU)
- Permission concept for tasks
- Object-based architecture
- No interrupt locks within the kernel
- Encapsulated tasks, assisted by hardware memory protection
- Possibility of reloading and debugging tasks during runtime
- Message-based data interchange between tasks

These special features originate from the fact that PXROS-HR attaches great importance to safety. Runtime errors during development can be avoided by clear semantics and structuring of the application software. Errors, particularly erroneous memory access operations, can be detected during runtime and be prevented from having wide-ranging effects.

Hardware memory protection prevents error propagation throughout the system. The memory protection unit (MPU) is controlled by the operating system.

2. Special characteristics of PXROS-HR

The first chapter has already mentioned special characteristics distinguishing PXROS-HR from other real-time operating systems. This chapter describes these features in detail.

Hardware-based memory protection:

PXROS-HR uses the memory protection mechanism of the hardware to seal off the memory areas of different tasks from each other. This means, each task has a certain amount of memory allocated. If it tries to reach out of this area, this access violation is detected by the MPU, a trap is triggered handing control over to the operating system.

Permission concepts for tasks:

PXROS-HR arranges to grant or deny certain rights to the tasks. These include, for example, access rights to operating system or hardware resources (see Section 3.5).

Object-based system architecture:

All elements managed by the operating system are perceived as objects, such as message objects or task objects.

No interrupt locks:

Whereas it is common practice with other operating systems to lock interrupts during scheduling, PXROS-HR does without interrupt locks. This means, interrupts can be reacted to without any latencies and thus significantly increases the predictability of the system.

message-based data interchange:

Data is exchanged between tasks via message objects. In conjunction with the concept of mailboxes, these messages provide an implicit synchronization mechanism and allow restricting data access to the current message user. This method has the advantage of making conventional (explicit) synchronization mechanisms, such as semaphores, redundant.

Encapsulation:

The concept of encapsulation arises from the object-based approach in connection with hardware-assisted memory protection: A task is perceived as a self-contained capsule, which is sealed off from the exterior world. Interaction with the exterior world can only be facilitated via a well-defined, narrow interface (message objects). Accidental interference, e.g. by misrouted access operations by pointers is prevented by the hardware. An overall system consists of several such capsules. Loose connections within the system increase the manageability of complex systems considerably, while absence of reaction is ensured.

Reloading tasks:

Based on the concept of encapsulation PXROS-HR makes it possible to load and unload new tasks during system runtime in a dynamic way.

Debugging the running system:

PXROS-HR makes it possible to stop individual tasks within a running system and to debug these tasks while the overall system continues to run.

Under PXROS-HR tasks can be perceived as running each on an individual controller.

Difference between system tasks and application tasks:

Due to the permission concept it is possible to distinguish between system tasks and application tasks. System tasks usually have comprehensive privileges and provide services to application tasks that require a higher privilege level. Thus they form a system platform offering additional functionality to applications. Application tasks are potentially error-prone (e.g. supplied modules) and thus typically have lower privileges. These tasks usually implement application-specific functionalities rather than system functionalities. Application tasks can also be reloaded.

The basic system also consists of individual capsules on task level with different rights (see Figure 3).

Micro-kernel:

PXROS-HR is implemented as a micro-kernel, i.e. only the basic functionalities of the operating system are implemented in the kernel. Additional functionalities, such as the TCP/IP stack or the file system, are realized in the form of separate modules running as application tasks.

3. PXROS-HR Architecture

In this chapter the components and mechanisms used for building up a PXROS-HR system are explained.

3.1. Overview - Architecture

The architecture of PXROS-HR is object-based, i.e. all the elements of a PXROS-HR system are regarded as objects. Objects can contain other objects or be assigned to other objects (see Figure 2).

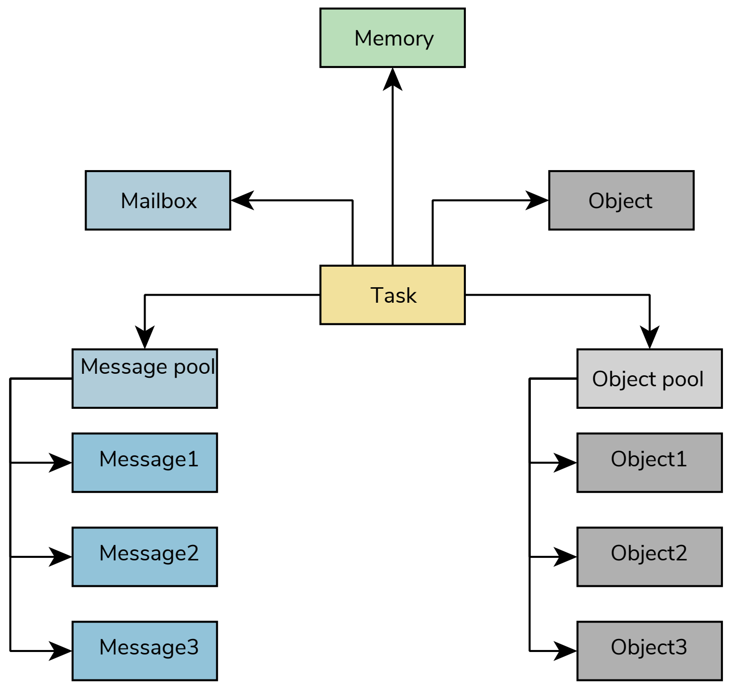

Figure 2. Object relations

The architecture design of PXROS-HR allows to define a base system providing the core functionality of an operating system, i.e. resource management and hardware abstraction. In order to extend this system with the actual application-specific functionality, application tasks can be loaded in the initialization phase or later at runtime.

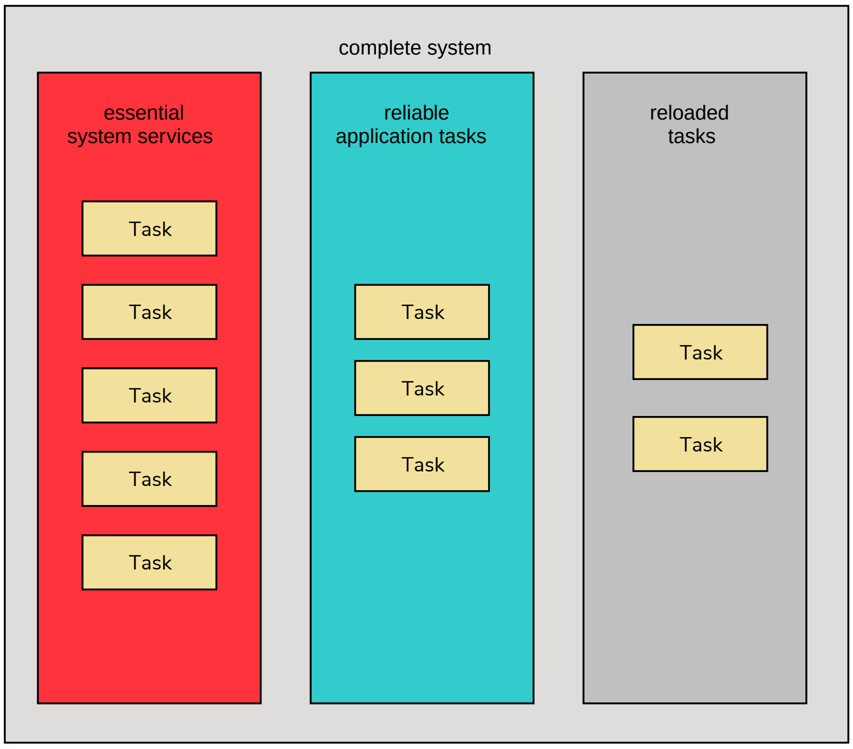

Figure 3. PXROS-HR System

3.2. Functional Units

Functional units are elements of the system that contain executable code and represent each one control flow path. In this context, a differentiation is made between tasks and handlers, as described below.

3.2.1. Tasks

Tasks are the basic elements of a PXROS-HR system. They represent concurrent processing units within an application and are prioritized among each other, yet never of higher priority than the handlers.

Tasks run in a quasi-parallel way. However, since typically several tasks share one physical processor, computing time is assigned to them by the operating system. In practice, this means that if a task is willing to run and has a higher priority than the current task, it will replace the current task.

Equal priority tasks do not supplant each other.
Exception: If time slicing is activated, a task can be supplanted by one of equal priority as soon as its time slice has expired.

A task can have any one of the following four states:

waiting

The task waits for an event to 'wake' it, e.g. the arrival of a message.

ready

The task has been woken up and is now willing to run, but it does not have the highest priority of all *ready* tasks.

active

When a task is ready and in possession of the highest priority (e.g. because a task of higher priority has changed into the *waiting* state), it becomes active and is then in possession of the processor.

suspended

A task was suspended by another task, meaning it will not become active, even if it is ready and has the highest priority.

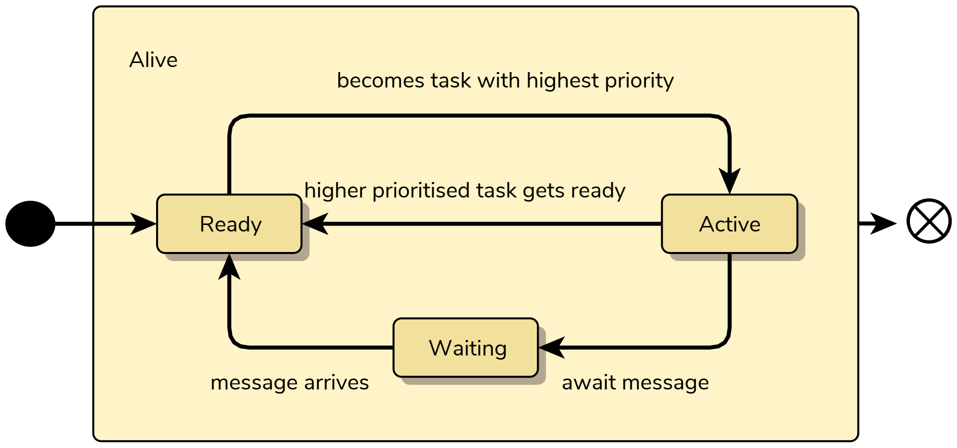
Alive
becomes task with highest priority
higher prioritised task gets ready
Ready
Active
Waiting
message arrives
await message

Figure 4. Task states

PXROS-HR uses a permission concept for tasks, i.e. tasks have to be authorized in order to access system resources. These types of permission are called privileges (hardware-based) or access rights (based on the operating system) (see Section 3.5).

Under PXROS-HR it is possible to reload tasks into the system during runtime.

3.2.2. Handlers

Handlers are functions that are called whenever a hardware or software interrupt occurs. They have a higher priority than any task.

In order to speed up processing of Handlers and to avoid blocking, they have restricted access to PXROS-HR system services.

There are three types of Handlers: Context Handlers and Fast Context Handlers, which are executed in the context of the associated tasks, and Fast Handlers, running in system context.

3.2.2.1. Fast Handlers

Fast Handlers, work on interrupt level, meaning these are the fastest and have the highest priority within the system. They have to be processed as quickly as possible since otherwise interrupt processing can be delayed. They are prioritized among each other; a Handler of higher priority can interrupt one of lower priority.

Fast Handlers have all-encompassing permissions, since they run in supervisor mode.

To install a Fast Handler, a task has to have the appropriate permission.

Fast Handlers can be used, for instance, to realize equidistant sampling.

3.2.2.2. Fast Context Handlers

Similar to a Fast Handler, a Fast Context Handler also works on interrupt level and has the same priority range. In contrast to Fast Handlers, however, a Fast Context Handler is assigned to the task it was installed by. Thus, the context has to be switched at Handler entry which adds a certain delay to the actual processing time.

A Fast Context Handler is executed in the context of the creating task and has access to the address space of this task.

To install a Fast Context Handler, a task has to have the appropriate permission. The installed Handler will then have the same address space as the creating task.

3.2.2.3. Context Handlers

A Context Handler is assigned to the task by which it was installed. Context Handlers have lower priority than Fast (Context) Handlers, and higher priority than tasks.

A Context Handler is executed in the context of the creating task and has access to the address space of this task.

To install a Context Handler, a task has to have the appropriate permission. The installed Handler will not run on interrupt level but on OS level.

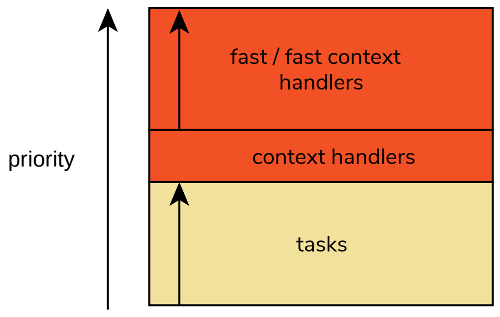

Figure 5. Distribution of Priorities

3.3. Objects

All the elements of PXROS-HR are regarded as objects. There are two basic types of objects: generic objects, which belong to an 'abstract base class' and do not implement any functionality, and specific objects, which are 'derived'.

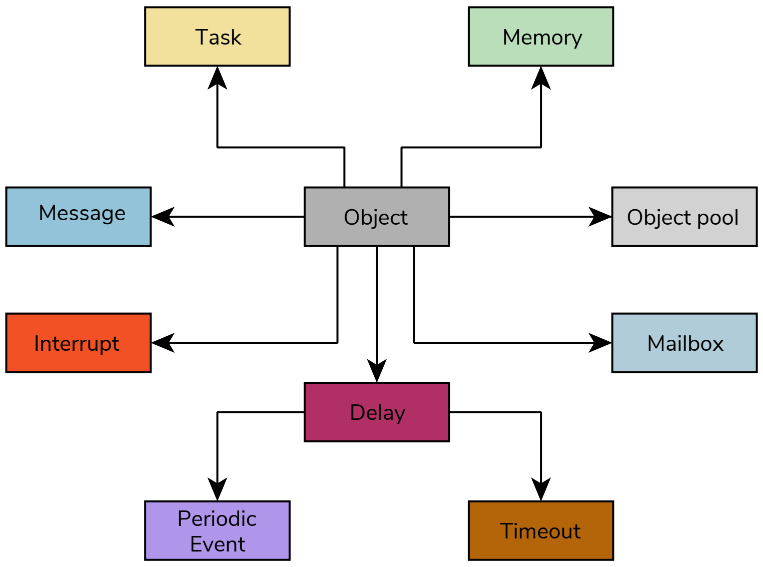

Figure 6. Object hierarchy

3.3.1. Tasks

Task objects are used for modelling PXROS-HR processes, the so-called tasks, with their properties such as priority, stack size, protected memory assigned, etc. (see also Section 3.2.1).

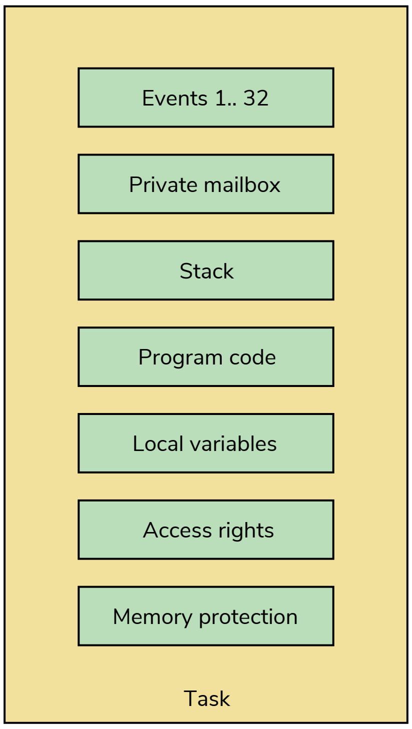

Figure 7. Components of a task object

3.3.2. MemoryClass Object

A MemoryClass object represents a memory class: the available memory can be separated into several disjoint blocks, the so-called memory classes. Tasks can be either granted exclusive access to these memory classes, or several tasks can share a memory class.

When using several memory classes, tasks can have different types of memory assigned (slow memory / fast memory). Furthermore, by assigning a memory class exclusively to one task, memory shortage during runtime can be avoided in this task, which would otherwise occur if another task utilizes all the available memory.

3.3.3. Messages

Messages are objects, which are used by tasks to exchange data. They also serve for synchronizing tasks, since a sender task can wait for a recipient task to release a Message.

As illustrated in Figure 8, the content of a Message is not copied. A Message object is given a portion of memory belonging to the owner of the Message. After sending the Message, this memory portion changes into the possession of the recipient where it remains until the recipient releases, forwards or returns it.

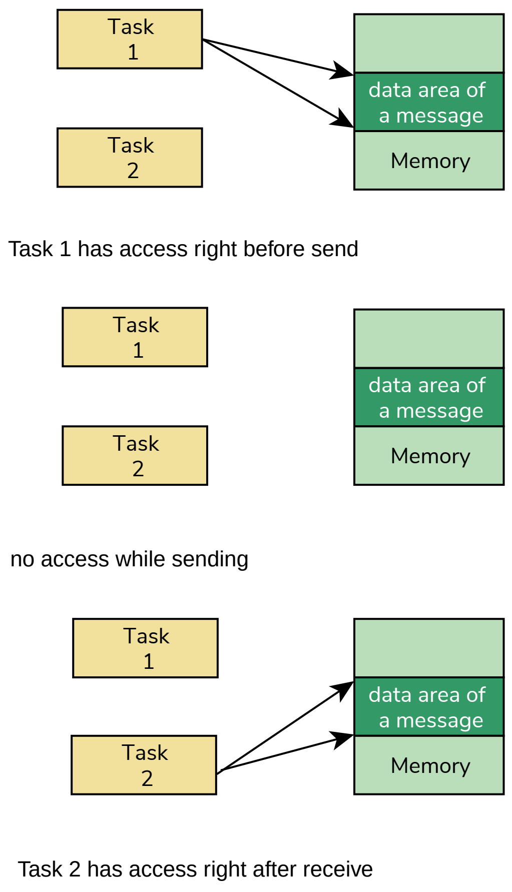

Figure 8. Access to the memory area of a Message

3.3.4. Mailboxes

Mailboxes are the communication terminals for Message exchange. A task sends a Message object to a mailbox, while a recipient can wait for arriving Messages at this mailbox.

Mailboxes can contain an arbitrary number of Message objects.

3.3.4.1. Private Mailboxes

Private mailboxes are a special feature. In general, mailboxes can be made available for any task, yet each task has its own private mailbox, which is unreadable for other tasks. The usual case of inter-task-communication is an exchange of Messages between private mailboxes.

3.3.4.2. Messagepools

Messagepools are another special feature. They are 'storehouses' for Message objects created beforehand. This means, Messages are created during initialization and stored in a Messagepool. This procedure has several advantages:

- Creating Messages during runtime can be avoided; the necessary Messages can be extracted from the Messagepool.
- There is no shortage during runtime, e.g. because at a certain time there are no more objects available.
- This shows the usefulness of the feature that several tasks can have access to the same mailbox.

3.3.4.3. Mailboxhandler

Handlers can be defined for mailboxes. If a handler was installed at a mailbox, then this handler is called whenever a Message is sent to this mailbox.

This feature is of particular usefulness if a task wishes to receive Messages from several mailboxes.

3.3.5. Objectpools

Object pools are used for storing objects until they are needed. There is a standard system pool, where all objects can be found at the time of initialisation. Other object pools can be created and filled with objects. These pools can be assigned to individual tasks in order to avoid object shortage during runtime, as they would occur with shared object pools (see Section 6.4.1).

Objects that were released after utilization, are returned to the object pool they were taken from.

3.3.6. Interrupt Objects

Interrupt objects are used for enqueing in the sysjob list, so that the according interrupt handlers are executed on OS level.

3.3.7. Objects for realising timer functionality

3.3.7.1. Delay objects

Delay objects are the most common objects for modelling timers: A Delay object makes sure that a handler function is called after a specified period.

3.3.7.2. Periodic event objects

A periodic event object (Pe) is a specialised version of a Delay object. A task creating a Pe object receives events in regular intervals.

3.3.7.3. Timeout objects

A timeout object (To) is a specialised version of a Delay object. A task creating a To object receives a single event after a defined period.

3.4. Interprocess communication

Interprocess communication, meaning communication among different tasks, is achieved by two different mechanisms: *Messages* and *Events*.

The two types of communication are different in that Messages can contain data, which can be processed by the recipient, whereas Events can only signal that an event has occurred.

3.4.1. Messages

A Message object represents an element of communication that can transfer data.

A task wishing to send a Message has to provide a data buffer for this Message.

The data buffer for a Message can be obtained in two ways: The sending task can state a memory class (see Section 3.3.2), from which the memory is to be taken, i.e. it 'borrows' the memory from this memory class and can use it as if it were its own memory for as long as it owns the Message.

The other possibility is to provide the Message object with memory of the task’s own memory.

The data buffer can be filled with arbitrary data. As soon as the Message was sent to a mailbox, the corresponding buffer is no longer part of the memory area of the task, i.e. the task may no longer access this memory area.

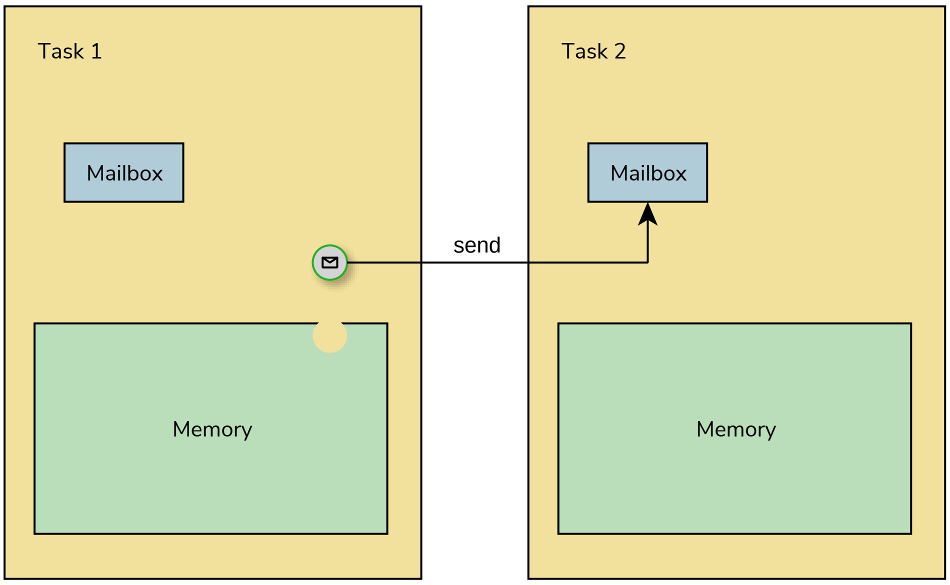

Figure 9. Sending a Message

If a recipient task reads the Message from the mailbox, the contained buffer is added to its address space. The task can access this area until it passes the Message on or releases it. Once the Message was released, the contained memory is reallocated to its original owner, i.e. the memory class or task.

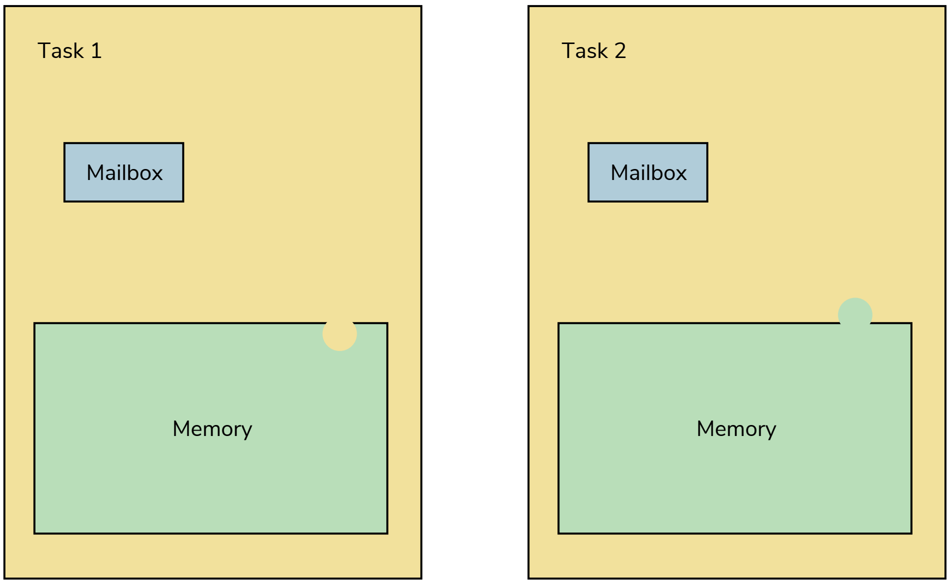

Figure 10. Receiving a Message

3.4.2. Events

An Event is used for signalling an occurrence to a task. In comparison to Messages, Events have the advantage of not using up any system resources. Up to 32 different Events can be signaled, the meaning of the events must only be agreed between sender and recipient.

A task can wait for several Events at the same time and is awakened as soon as one of the Events occurs.

3.5. Safety characteristics

The safety characteristics of PXROS-HR have the purpose of minimising the effects of runtime errors. Runtime errors should be detected so that, firstly, they can be reacted to properly via exception handling, and secondly, so that error propagation throughout the system is avoided.

3.5.1. Hardware memory protection

The PXROS-HR safety concept is based on hardware-assisted memory supervision. This concept relies upon the Memory Protection Unit (MPU) of the processor. The MPU contains registers, in which the upper and lower bounds of the memory area of a task (or the system), as well as the corresponding rights (read and/or write) are stored.

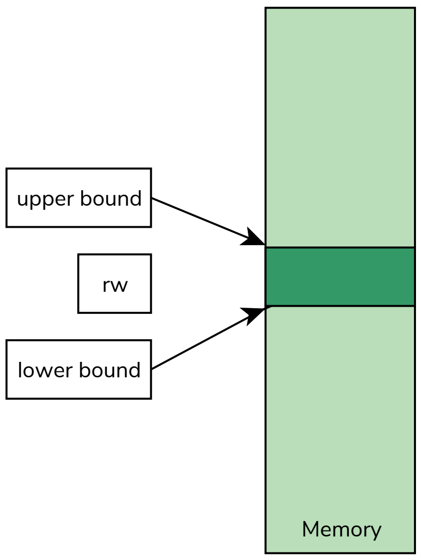

Figure 11. Memory protection register

3.5.2. Memory protection and encapsulation

A task is regarded as being enclosed in a capsule, which it cannot leave. Hardware memory protection has the purpose of preventing a task from reaching out of its capsule.

During initialisation, memory is allocated to a capsule, which it can access. The start and end addresses of this area, as well as read and write permissions, are stored in registers of the MPU, making it possible for the hardware to check memory access operations during runtime. This memory area contains the data as well as the stack memory of the task.

The MPU can even detect stack overflow if the memory is appropriately apportioned, i.e. if the stack is located at the end of the individual memory.

Apart from this, memory can be mapped onto the address area of the task if the task receives a Message object.

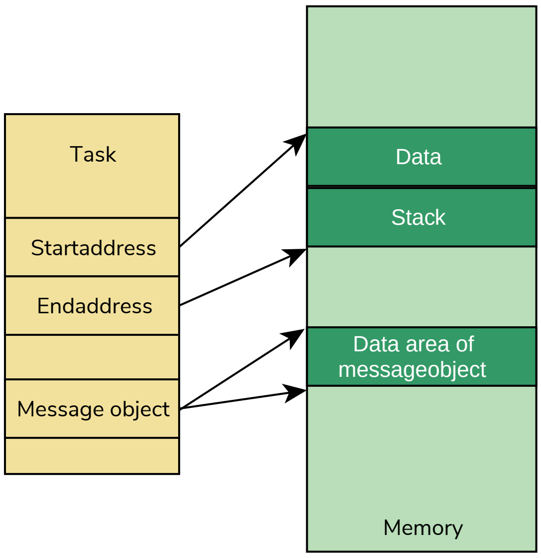

Figure 12. Memory areas of a task

In addition, extended memory areas can be defined for a task. This method is, however, subject to a loss in performance since the memory areas are stored in virtual protection registers that have to be mapped to the hardware registers before they can be accessed.

If a task reaches out of its memory areas, the MPU detects this access violation and triggers a trap, thus avoiding error propagation. The cause of the error can be detected and the erroneous module can be selectively deactivated. Functional safety of the systems is thus guaranteed.

This type of memory protection makes it possible to realise the concept of encapsulation: A task and its Context Handlers are regarded as being located within a capsule, being thus separated from the rest of the system, which in itself also consists of capsules. Communication is achieved purely via Message objects and Events.

This leads to the separation of the individual functional units. The capsules can be considered as each running on a processor of its own. Complexity of the system is thus reduced, and the probability of error propagation. Should, nevertheless, an error occur, its effects are restricted to the corresponding capsule. Thanks to these characteristics, test procedures for applications can be modularised.

3.5.3. Permissions

Tasks in general do not have any permission to access resources; such access has to be granted explicitly in individual cases. The permission concept knows two types of permissions, *privileges* and *access rights*.

Privileges are permission levels of the hardware. They can be subdivided into two categories: System privileges allow full access and are only available to the operating system and certain interrupt handlers; user privileges exist in several levels and can be assigned to parts of the application.

The following privilege levels exist:

Usermode0
allows access to the internal memory area exclusively

Usermode1
allows access to the internal memory area and to the periphery

Systemmode
allows full access (only available to the operating system and certain interrupt handlers)

Access rights are managed by the operating system. The following *access rights* can be granted:

PXACCESS_HANDLERS
for installing handlers that own system rights

PXACCESS_INSTALL_HANDLERS
for installing handlers that own the rights of the task

PXACCESS_INSTALL_SERVICES
for installing certain PXROS-HR services as handlers

PXACCESS_REGISTER
for executing system functions that have access to special registers of the processor

PXACCESS_SYSTEMDEFAULT
give access to the system memory class and system object pool

PXACCESS_RESOURCES
allows a task to access, in addition to its own resources (those that were assigned to it during creation or that it created itself), but also to resources that it does not own

PXACCESS_NEW_RESOURCES
allows a task to create object pools or memory classes

PXACCESS_SYSTEM_CONTROL
allows a task to suspend or continue other tasks

PXACCESS_MODEBITS
allows a task to set its modebits

PXACCESS_OVERRIDE_ABORT_EVENTS
: allows a task to override its aborting events

PXACCESS_TASK_CREATE
: allows a task to create another task

PXACCESS_TASK_CREATE_HIGHER_PRIO
: allows a task to create another task with a higher priority than its own priority

PXACCESS_TASK_SET_HIGHER_PRIO
: allows a task to increase its priority

PXACCESS_CHANGE_PRIO
: allows a task to lower its priority

PXACCESS_TASK_RESTORE_ACCESS_RIGHTS
: allows a task to restore its access rights to the initial value

PXACCESS_TASK_CREATE_HIGHER_ACCESS
: allows a task to create an other task without respecting the memory inheritance rule

PXACCESS_TRACECTRL
: allows a task to setup the trace mechanism by `PxTraceCtrl`

PXACCESS_GLOBAL_OBJECTS
: allows a task to allocate objects from the `PXOpoolGlobalSystemdefault` for the communication between tasks on different cores

In cases where a system service is called without having the required access rights, an error code is returned.

3.5.4. Error handling

The safety characteristics of PXROS-HR include the ability to detect and react to runtime errors, such as memory access violations.

There are, essentially, two kinds of error treatment:

First, errors are detected during the regular control flow, i.e. during system calls PXROS-HR checks, whether a call is valid and returns the corresponding error code. Invalid system calls include calls with faulty parameters, such as invalid object IDs, missing permissions or invalid system function calls within a handler.

Second, exceptional treatment aborts the regular control flow: Invalid memory access operations cause the MPU to trigger a trap. The trap can be processed by an application-specific trap handler routine.

4. PXROS-HR Multi-Core

The same programming model as described in the chapters above can be used for PXROS-HR Multi-Core systems. The tasks can be spread across several cores and can communicate with messages and events between the cores.
If events are used between tasks on different cores, an internal communication object will be allocated and sent to the corresponding core. Therefore events across cores can not be signaled by handlers and the signaling of events by tasks can fail due to object shortage.

On each core one instance of the PXROS-HR micro-kernel is running performing local administration (scheduling, object and memory management, system services, etc.) and are responsible for the inter-core communication.

Typically local memory is used for PXROS-HR and the tasks running on that core.

Tasks and other PXROS-HR objects have internal identifiers that are unique on all cores. As a result, it is not required to have any knowledge about the core assignment of the receiver task of a communication object (message or event).

4.1. Initialisation

As there is a micro-kernel running on each core, the PXROS-HR initialisation structure has to be defined per core, as well.

```
const PxInitSpec_T InitSpec_CORE0 =
    {
        .is_sysmc_type       = PXMcVarsizedAligned,
        .is_sysmc_size       = 8,
        .is_sysmc_blk        = Sysmem0,
        .is_sysmc_blksize    = SYSMEMSIZE_CORE0,

        .is_objmc_type       = PXMcVarsizedAligned,
        .is_objlmc_size      = 8,
        .is_objmc_blk        = PxObjmem_CPU0_,
        .is_objmc_blksize    = (PxSize_t)PX_OBJMEMSIZE_CPU0_,

        .is_obj_number       = NUM_OF_PXOBJS_CORE0,
        .is_obj_namelength   = PXROS_NAMESIZE,

        .is_taskmc_type      = PXMcVarsizedAdjusted,
        .is_taskmc_size      = 8,
        .is_taskmc_blk       = Taskmem_Core0,
        .is_taskmc_blksize   = TASKMEMSIZE_CORE0,

        .is_inittask         = &InitTaskSpec_CORE0,

        .is_core_start       = (unsigned int)_start,

         /* the system stack */
        .is_system_stack = PXROS_SYSTEM_STACK_BEGIN_CPU0_,
        .is_system_stack_size = (PxUInt_t)PXROS_SYSTEM_STACK_SIZE_CPU0_,
        /* all objects are global accessible */
        .is_global_obj_number = 0,

        /* the protection definition */
        .is_sys_code = &_cpu0_sys_code_protection,
        .is_sys_data = &_cpu0_sys_data_protection,
        .is_task_code = &_cpu0_task_code_protection,
    };
const PxInitSpec_T InitSpec_CORE1 =
    {
        .is_sysmc_type      = PXMcVarsizedAligned,
        .is_sysmc_size      = 8,
        .is_sysmc_blk       = Sysmem1,
        .is_sysmc_blksize   = SYSMEMSIZE_CORE1,

        .is_objmc_type      = PXMcVarsizedAligned,
        .is_objlmc_size     = 8,
        .is_objmc_blk       = PxObjmem_CPU1_,
        .is_objmc_blksize   = (PxSize_t)PX_OBJMEMSIZE_CPU1_,

        .is_obj_number      = NUM_OF_PXOBJS_CORE1,
        .is_obj_namelength  = PXROS_NAMESIZE,

        .is_taskmc_type     = PXMcVarsizedAdjusted,
        .is_taskmc_size     = 8,
        .is_taskmc_blk      = Taskmem_Core1,
        .is_taskmc_blksize  = TASKMEMSIZE_CORE1,

        .is_inittask        = &InitTaskSpec_CORE1,

        .is_core_start      = (unsigned int)_start,

         /* the system stack */
        .is_system_stack = PXROS_SYSTEM_STACK_BEGIN_CPU1_,
        .is_system_stack_size = (PxUInt_t)PXROS_SYSTEM_STACK_SIZE_CPU1_,

        /* all objects are global accessible */
        .is_global_obj_number = 0,

        /* the protection definition */
        .is_sys_code = &_cpu1_sys_code_protection,
        .is_sys_data = &_cpu1_sys_data_protection,
        .is_task_code = &_cpu1_task_code_protection,
    };
const PxInitSpec_T InitSpec_CORE2 =
    {
        ...
    };
const PxInitSpec_T InitSpec_CORE3 =
    {
        ...
    };
const PxInitSpec_T InitSpec_CORE4 =
    {
        ...
    };
const PxInitSpec_T InitSpec_CORE5 =
    {
        ...
    };


static const PxInitSpecsArray_t InitSpecsArray[] =
{
    &InitSpec_CORE0,
    &InitSpec_CORE1,
    &InitSpec_CORE2,
    &InitSpec_CORE3,
    &InitSpec_CORE4,
    &InitSpec_CORE5
};
```

The array containing all core specific initialisation structures is passed to `PxInit()` to initialize and start the PXROS-HR system.

The Inittask may call the function `PxGetCoreId()` to determine on which core it is running and can perform core specific functions or create core specific task:

```
PxUInt_t id = PxGetCoreId ();

/* insert your own Application,  e.g. initialise other tasks etc */
switch (id)
{
  case CORE_0:
      /*
       * Do core 0 specific stuff and
       * create core 0 tasks here
       */
       break;
  case CORE_1:
      /*
       * Do core 1 specific stuff and
       * create core 1 tasks here
       */
       break;
  case CORE_2:
      /*
       * Do core 2 specific stuff and
       * create core 2 tasks here
       */
       break;
}
```

5. Services of the system

In addition to the normal functionalities of an operating system, PXROS-HR provides further services to an application. These services can be subdivided into services of the core system, which can be used at any time, and supplementary services that have to be linked in the form of independent modules.

5.1. PxSysInfo services

PxSysInfo services give access to information on the internal system condition. Details to be requested include the number and type of objects used, and the number of free objects.

5.2. Tracing

The PXROS-HR tracing service allows to record system calls, message exchange and events for debugging purposes. The called function and its arguments as well as a time stamp are stored in a buffer. These stored data can then be read by the application and interpreted by an analysis tool.

5.3. PXROS-HR Service Task

A PXROS-HR Service Task is used in certain situations for calling system services. If a task terminates, the Service Task will be informed to release task specific memory like the TCB and the task object. Any task can be declared as a Service Task as long as it has the access permission `PXACCESS_SYSTEM_CONTROL`, see Section 3.5.3.

5.4. System monitor

Communication between the debugger on the host side and the PXROS-HR system requires an appropriate interface, which is provided by the system monitor. Besides the interface for the debugging functionality there is also a second interface for reloading Tasks. The system monitor is included in the functional range of the operating system.

The system monitor is capable of intercepting memory protection traps, and passing them on to the debugger.

5.5. _PxHndcall

With `_PxHndcall` PXROS-HR provides an interface to execute user functions in supervisor mode. This feature is especially helpful in the context of hardware initialization whenever register accesses require supervisor privileges. Common use-cases are the activation of peripheral modules via the clock control registers (CLC) and the execution of code sequences under temporarily disabled Endinit protection.

In order to prevent corruption of kernel data the provided user function is executed in the context of the calling task. Thus, it might be necessary to assign peripheral address ranges to the task's additional protection regions table.

Similar to most of the system service calls, functions called via `_PxHndcall` cannot be interrupted by other tasks or Context Handlers, but only by Fast Context and Fast Handlers.

6. Programming the core system

The following chapter describes how to program a PXROS-HR application. For this purpose, the basic concepts and terms, as well as 'best practice' procedures are explained and illustrated by examples.

6.1. Basic concepts

Event:
: An Event is represented by a 32 bit wide mask. Thus, several Events to be sent or to be waited for can be stored in a four bytes bitmask, i.e. a `long` value.

Priority:
: The priority of a Task can have a value between 0 and 31, wherein a lower value means a higher priority.

Ticks:
: The hardware-independent time basis of a PXROS-HR system; the attribution of ticks to milliseconds can be defined by the user.

Handles:
: PXROS-HR objects cannot be accessed directly; to use PXROS-HR objects, Handles are utilised, which represent objects at the interface (see Section 6.3.1).

6.2. The first simple application

The simple application created below does not contain any functionality, yet. It implements an executable basic PXROS-HR system consisting of the following basic elements: initialisation of PXROS-HR, the Inittask, and an empty Task.

6.2.1. Initialising PXROS-HR

Before PXROS-HR system calls can be used, PXROS-HR has to be initialised. This is done via the function `PxInit`. This function receives a parameter of the `PxInitSpec_T` type.

6.2.1.1. PXROS specification

The PXROS-HR initialisation structure `PxInitSpec_T` is defined as follows:

`PxMcType_t is_sysmc_type:`
: the type of system memory class; permitted values are `PXMcVarsized`, `PXMcVarsizedAdjusted` and `PXMcVarsizedAligned`

`PxSize_t is_sysmc_size:`
: only relevant for memory classes of type `PXMcVarsizedAligned` and `PXMcVarsizedAdjusted`
It defines the alignment or adjustment of allocated blocks

`PxAligned_t *is_sysmc_blk:`
: pointer to the memory assigned to the system memory class

`PxSize_t is_sysmc_blksize:`
: overall size of the memory block

`PxMcType_t is_objmc_type:`
: this parameter is presently not in use

`PxSize_t is_objmc_size:`
: this parameter is presently not in use

`PxAligned_t *is_objmc_blk:`
: pointer to the memory of generic PXROS-HR objects

`PxSize_t is_objmc_blksize:`
: overall size of the memory of generic PXROS-HR objects
The `size` must be at least (number of objects) * (PXOBJ_SIZE + namelength)

`PxUInt_t is_obj_number:`
: number of PXROS objects

`PxObjId_t is_obj_namelength:`
: maximum length of the object names, including zero fillers

`PxTaskSpec_ct is_inittask:`
: Task specification of the Inittask, see Section 6.2.1.3

`PxMcType_t is_taskmc_type:`
: type of memory class of tasks; permitted values are `PXMcVarsized`, `PXMcVarsizedAdjusted` and `PXMcVarsizedAligned`
This memory class will be assigned as PXMcTaskdefault to the first task.

`PxSize_t is_taskmc_size:`
: only relevant for memory classes of type PXMcVarsizedAligned and PXMcVarsizedAdjusted
It defines the alignment or adjustment of allocated blocks

`PxAligned_t PXDptr_q *is_taskmc_blk:`
: pointer to the memory for this class

`PxSize_t is_taskmc_blksize:`
: overall size of the memory block

`const unsigned int is_core_start`
: the start address of this core
Before activating the core, the program counter (PC) of the core will be initialized with this address.

`PxProtectRegion_T *is_sys_code:`
  points to the specification of up to 4 (TC2x) or 5 (TC3x) protected code areas of the kernel

`PxProtectRegion_T *is_sys_data:`
  points to the specification of up to 8 (TC2x) or 9 (TC3x) protected data areas of the kernel

`PxProtectRegion_T *is_task_code:`
  points to the specification of up to 4 (TC2x) or 5 (TC3x) protected code areas of the tasks

The code protection areas `is_sys_code` and `is_task_code` must cover all memory areas which contain code executable in kernel/supervisor mode or by tasks.

The data protection areas `is_sys_data` must cover all memory areas which must be accessible in kernel/supervisor mode.

A sample initialisation of the protection areas:

```
const PxCodeProtectSet_T _cpu0_sys_code_protection =
{
    /* Range 0 the complete text section */
    .cpr[0].s = {(PxUInt_t)__TEXT_BEGIN,
                 (PxUInt_t)__TEXT_END,},
    /* Range 1 the traptab section */
    .cpr[1].s = {(PxUInt_t)__TRAP_TAB_BEGIN,
                 (PxUInt_t)__TRAP_TAB_END,},
    /* Range 2 the inttab section */
    .cpr[2].s = {(PxUInt_t)__INT_TAB_BEGIN,
                 (PxUInt_t)__INT_TAB_END,},
    .cpmr.cpxe.bits = {
                        .dp0 = 1,   /* the CPXE 0..2 executable */
                        .dp1 = 1,
                        .dp2 = 1
                      }
};

const PxCodeProtectSet_T _cpu0_task_code_protection =
{
    /* Range 0 the complete text section */
    .cpr[0].s = {(PxUInt_t)__TEXT_BEGIN,
                 (PxUInt_t)__TEXT_END,},
    .cpmr.cpxe.bits = {
                        .dp0 = 1,   /* the CPXE 0 executable */
                      }
};

const PxDataProtectSetInit_T _cpu0_sys_data_protection =
{
    /* Range 0: read only data */
    .dpr[0].s = {(PxUInt_t)PxTricSystemRodataLowerBound,
                 (PxUInt_t)PxTricSystemRodataUpperBound,},
    /* Range 1: the CSA area of CPU0 */
    .dpr[1].s = {(PxUInt_t)__CSA_BEGIN_CPU0_,
                 (PxUInt_t)__CSA_END_CPU0_,},
    /* Range 2: the KERNEL data area of CPU0 including objects and stack */
    .dpr[2].s = {(PxUInt_t)PxTricSystemDataLowerBound_CPU0_,
                 (PxUInt_t)PxTricSystemDataUpperBound_CPU0_},
    /* Range 3: System/Kernel stack of CPU0 */
    .dpr[3].s = {(PxUInt_t)PXROS_SYSTEM_STACK_BEGIN_CPU0_,
                 (PxUInt_t)PXROS_SYSTEM_STACK_CPU0_,},
    /* Range 4: the SFR area */
    .dpr[4].s = {(PxUInt_t)PERIPHERAL_MEM_BASE,
                 (PxUInt_t)PERIPHERAL_MEM_END,},
    /* Range 5:  some area */
    .dpr[5].s = {0, 0},
    /* Range 6: the other area */
    .dpr[6].s = {0,0},
    /* Range 7: used dynamically by the kernel (TC2x)
                used to get access to memory before PXROS is started
                - especially used for copy and clear functions
                For TC3x this will be range 8 and range 7 can be used otherwise
    */
    .dpr[7].s = {0,0},
    /* the DPRE 0..4,7 readable by the kernel */
    .dpmr.kernel.dpre.bits = {
                    .dp0 = 1, .dp1 = 1, .dp2 = 1, .dp3 = 1,
                    .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 1
                  },
    /* the DPWE 1..4,7 writable by the kernel */
    .dpmr.kernel.dpwe.bits = {
                    .dp0 = 0, .dp1 = 1, .dp2 = 1, .dp3 = 1,
                    .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 1
                  },

    /* the DPRE 0,3,4 readable by the system (handlers running in supervisor mode) */
    .dpmr.system.dpre.bits = {
                    .dp0 = 1, .dp1 = 0, .dp2 = 0, .dp3 = 1,
                    .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 0
                  },
    /* the DPWE 3,4 writable by the system (handlers running in supervisor mode) */
    .dpmr.system.dpwe.bits = {
                    .dp0 = 0, .dp1 = 0, .dp2 = 0, .dp3 = 1,
                    .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 0
                  },
};
```

The operating system must have at least access to the CSA area, its local data, the objects, the supervisor stack and the peripheral space.

6.2.1.2. Init function

Calling `PxInit` starts the Inittask and only returns if an error occurred during initialisation. In this case, the function returns a corresponding error code:

`PXERR_INIT_ILLALIGN`
: Erroneous alignment of system memory class

`PXERR_INIT_ILLMCTYPE`
: System memory class is not of type `PXMcVarsized` or `PXMcVarsizedAdjusted`

`PXERR_INIT_NOMEM`
: Not enough memory available

`PXERR_PROT_ILL_REGION`
: Invalid memory protection areas defined

`PXERR_INIT_SCHEDULE_FAILED`
: the scheduling of the init task failed

`PXERR_MC_ILLSIZE`
: size for PXMcSystemdefault is too small

`PXERR_INIT_NOMEM`
: not enough memory for initialization

`PXERR_OBJECT_SHORTAGE`
: not enough objects given in initstruct

`PXERR_GLOBAL_ILLEGAL_CORE`
: number of cores not supported

`PXERR_ILL_NULLPOINTER_PARAMETER`
: invalid system stack specification

`PXERR_PROT_PERMISSION`
: memory protection unit cannot be activated

`PXERR_ILLEGAL_ACCESS`
: incorrect access permission for _initspecs elements

`PXERR_GLOBAL_OBJLIST_INCONSISTENCY`
: inconsistency between global and local init

After a successful `PxInit` call, the PXROS-HR system services are available.

6.2.1.3. The Inittask

The Inittask is the first Task of the system. It is automatically started after system initialization and corresponds in layout and handling to a normal Task (see Section 6.2.3). The differences to other tasks are:

- it is created and started by `PxInit`
- it can only use resources available after `PxInit`

The Inittask can be used for initialising the application and for creating and starting the application Tasks. Typically, the priority of the Inittask is set to lowest priority after the initialization phase, i.e. it is only activated if no other Task or Handler is active (Idletask).

The existence of an Idletask is not compulsory. The Inittask could also be terminated after execution, yet it could also be used to do real work, e.g doing background calculations, system controlling etc.

6.2.1.4. Sample code

The initialization of the PXROS-HR system for only one core is exemplified by the below code:

```
// for more information on this structure, see chapter
// "Layout and initialisation of a Task"
static const PxTaskSpec_T InitTaskSpec =
{
        .ts_name                     = "InitTask",
        .ts_fun                      = InitTask_Func,
        .ts_mc                       = PXMcTaskdefault,
        .ts_opool                    = PXOpoolSystemdefault,
        .ts_prio                     = 0,
        .ts_privileges               = PXUser1Privilege,
        .ts_context                  = &InitTaskContext,
        .ts_taskstack.stk_type       = PXStackAlloc,
        .ts_taskstack.stk_size       = INITTASK_STACKSIZE,
        .ts_taskstack.stk_src.mc     = PXMcTaskdefault,
        .ts_inttaskstack.stk_type    = PXStackDontCheck,
        .ts_inttaskstack.stk_size    = 0,
        .ts_inttaskstack.stk_src.stk = 0,
        .ts_abortstacksize           = 0
};

const PxInitSpec_T InitSpec_CORE0 =
    {
        .is_sysmc_type      = PXMcVarsizedAligned,
        .is_sysmc_size      = 8,
        .is_sysmc_blk       = Sysmem0,
        .is_sysmc_blksize   = SYSMEMSIZE_CORE0,

        .is_obj_number      = NUM_OF_PXOBJS_CORE0,
        .is_obj_namelength  = PXROS_NAMESIZE,
        .is_inittask        = &InitTaskSpec,

        .is_objmc_type      = PXMcVarsizedAligned,
        .is_objlmc_size     = 8,
        .is_objmc_blk       = PxObjmem_CPU0_,
        .is_objmc_blksize   = (PxSize_t)PX_OBJMEMSIZE_CPU0_,

        .is_taskmc_type     = PXMcVarsizedAdjusted,
        .is_taskmc_size     = 8,
        .is_taskmc_blk      = Taskmem_Core0,
        .is_taskmc_blksize  = TASKMEMSIZE_CORE0,

        .is_core_start      = (unsigned int)_start,

         /* the system stack */
        .is_system_stack = PXROS_SYSTEM_STACK_BEGIN_CPU0_,
        .is_system_stack_size = (PxUInt_t)PXROS_SYSTEM_STACK_SIZE_CPU0_,

        /* all objects are global accessible */
        .is_global_obj_number = 0,

        /* the protection definition */
        .is_sys_code = &_cpu0_sys_code_protection,
        .is_sys_data = &_cpu0_sys_data_protection,
        .is_task_code = &_cpu0_task_code_protection,
    };

// Since a memory class for tasks is defined
// the inittask gets PXMcTaskdefault as the memory class

static const PxInitSpecsArray_t InitSpecsArray[] =
{
    &InitSpec_CORE0,
    0,
    ...
};


PxError_t error;

// the initialisation of PXROS-HR; if PxInit returns,
// the corresponding error code is logged in error

error = PxInit(InitSpecsArray, 1);
```

6.2.2. Startup-initialisation

Before the initialisation of PXROS-HR, possible hardware configurations might be executed. Since this usually has to be done with supervisor privileges, an application cannot carry out these procedures since it runs in user mode. This problem can be circumvented if the application passes arbitrary functions, which can be used to configure and initialize the hardware.

`_PxInitcall(function, params…)` adds the function `function` with its parameters `params` to an internal table; `params` is an arbitrary list of parameters.

The user can start the execution of these functions by calling `PxInitializeBeforePxInit()`. The functions are executed with system permission and in the order, in which they were entered in the table.

The function `PxInitializeBeforePxInit` has to be called before the initialization of PXROS-HR, i.e. before `PxInit`!

6.2.3. Layout and initialisation of a Task

6.2.3.1. General

Essentially, a PXROS-HR Task consists of a Task function, which is called when the Task is started. After its creation a Task is started by sending an activation event, i.e. the Task function is called, provided one or more activation events were defined. If no activation event was defined when the Task was created, the Task is immediately started. Normally, this Task function does never return, i.e. it contains an infinite loop.

In exceptional cases a Task can be terminated by calling the system service `PxDie`. This function does not return. To be able to use `PxDie`, a Service Task has to be initialised beforehand.

The initialization of the Task is carried out during creation via the function `PxTaskCreate`. The necessary parameters for initializing the Task are passed to this function:

opool
: the object pool, from which the Task object is taken

taskspec
: the Task specification, which contains the necessary data for initialization (see Section 6.2.3.2)

prio
: the priority of the Task

actevents
: Events activating the Task; usually only one

6.2.3.2. Task specification

Task specification is a structure of the type `PxTaskSpec_T` (the type designators `PxTaskSpec_t` and `PxTaskSpec_ct` mentioned in the reference manual relate to pointers or constant pointers to `PxTaskSpec_T`). This structure is defined as follows:

`PxUChar_t *ts_name:`
: name of the Task; a string

`void (*ts_fun)(PxTask_t task, PxMbx_t mbx, PxEvents_t events):`
: pointer to the Task function

`PxMc_t ts_mc:`
: Default memory class of the Task

`PxOpool_t ts_opool:`
: Default objectpool of the Task

`PxStackSpec_T ts_taskstack:`
: Stack specification of the Task (see stack specification in this section)

`PxStackSpec_T ts_inttaskstack:`
: Interrupt stack specification of the Task (see stack specification in this section)

`PxUChar_t ts_prio:`
: Initial priority of the Task; this parameter is kept only for compatibility; it is ignored

`PxEvents_t ts_actevents:`
: Activation events for the Task, this parameter is kept only for compatibility; it is ignored

`PxTicks_t ts_timeslices:`
: size of the time slices for this Task; if this value is set to 0, no time slices are used

`PxSize_t ts_abortstacksize:`
: size of the abort stacks, stated by the number of frames

`PxTaskSchedExt_t ts_sched_extension:`
: not implemented

`PxArg_t ts_sched_initarg:`
: not implemented

`PxArg_t ts_privileges:`
: Privileges of the Task

`PxUInt_t ts_accessrights:`
: Access rights of the Task

`PxTaskContext_ct ts_context:`
: Description of the address space of the Task (see task context specification in this section)

`PxProtectRegion_ct ts_protect_region:`
: extended memory areas of the Task

Notes:

- A default memory class can be assigned via the macro `PXMcTaskdefault`, i.e inherit the memory class from the creator.
- The default object pool of the Task can be assigned via the macro `PXOpoolTaskdefault`, i.e inherit the object pool from the creator.
- Signalising an activation event starts the Task, i.e. after its creation the Task waits until it receives an activation event; usually only one activation event is stated
- The size of the Abort stacks determines the maximum nesting depth of `PxExpectAbort` calls (see Section 6.10)
- `ts_protect_region` defines additional memory protection areas of the Task. It is interpreted by PXROS-HR as a pointer to an arbitrarily large array terminated by an element whose lower bound address is set to 0.
  The specified memory areas serve as an extension to the static memory context of the Task and are applied to the MPU dynamically. However, using additional memory areas for heavily accessed data should be avoided due to the runtime overhead (granting permissions for a currently non-active area always requires a protection trap resolution). Common use cases for this mechanism are for instance register accesses and accessing configuration data.

The type `PxStackSpec_T` is used for describing Task and interrupt stacks. It contains information on the stack type (ascending/descending), stack size, and from where the memory is taken (for details, see the PXROS-HR reference manual).

The `PxTaskContext_t` structure describes the memory context of the Task. It contains two protection regions (see below, `PxProtectRegion_t`) which can be inherited from the creator or defined individually for each task. If 0 is assigned as the lower bound address, the area will be inherited.

The structure `PxProtectRegion_t` defines protected memory areas, which can be made available to the Task. It contains the upper and lower boundary of the memory area (the first valid and the first invalid address), as well as the access permission: no access, read only, write only, read and write.

Sample code for creating and initialising the Task specification:

```
// Definition of the memory context of the Task; since the elements of
// protection[0] are set to 0, the memory area is inherited
// from the creator of the Task

static const PxTaskContext_T Task1_Context = {

        // data area, inherited from the creator of the Task
        .protection[0].lowerBound = 0,
        .protection[0].upperBound = 0,
        .protection[0].prot       = NoAccessProtection,

        // read-write data area
        .protection[1].lowerBound = (PxUInt_t)Task1_data_base,
        .protection[1].upperBound = (PxUInt_t)Task1_data_end,
        .protection[1].prot       = WRProtection
};

#define TASK1_STACKSIZE     100         // stacksize in 4-byte units

// Creating the Task specification
const PxTaskSpec_T Task1_Spec = {
        // Name of the Task
        .ts_name                        = "Task1",

        // Task function (see above)
        .ts_fun                         = Task1_Func,

        // Memory block of the Task
        .ts_mc                          = PXMcTaskdefault,

        // Objectpool of the Task
        .ts_opool                       = PXOpoolTaskdefault,

        // Start priority of the Task, this Parameter exists only
        // for compatibility; it is ignored
        .ts_prio                        = 0,

        // Privileges of the Task
        .ts_privileges                  = PXUser0Privilege,

        // A pointer to the memory context of the Task
        .ts_context                     = &Task1_Context,

        // Stack type, will be allocated
        .ts_taskstack.stk_type          = PXStackAlloc,

        // Stack size in 4-byte units to be checked;
        .ts_taskstack.stk_size          = TASK1_STACK_SIZE,

        // Stack will be allocated from PXMcTaskdefault
        .ts_taskstack.stk_src.mc       = PXMcTaskdefault,

        // Abort stack size; is ignored in this case
        .ts_abortstacksize              = 0
};
```

The Task is created as follows:

```
// Creates a Task with the predefined specification,
// priority TASK1_PRIO and the activation events TASK1_ACTIVATION_EVENTS.
// The return value is the ID of the created Task
Task1_Id = PxTaskCreate(opool, &Task1_Spec, TASK1_PRIO,  TASK1_ACTIVATION_EVENTS);
```

6.2.3.3. The Task function

A Task function has to conform to the following prototype:

void Task_Func(PxTask_t myID, PxMbx_t myMailbox, PxEvents_t myActivationEvents)

These parameters are passed from the system as soon as the function is called, i.e. if the corresponding Task is activated for the first time. The parameters have the following meaning:

`myID`
: ID of the Task

`myMailbox`
: Private mailbox of the Task

`myActivationEvents`
: Mask of the received activation events

If activation events were defined when creating the Task (see Section 6.2.3.2), the Task will only become active if one or more of these events are received.

The Task function can contain almost any code, yet it may not be terminated. This means, it must either contain an infinite loop or be terminated via a blocking system call (e.g. `PxAwaitEvents(0)` or `PxDie()`).

A typical Task function consists of a declaration and initialization section and an infinite loop. In this loop the function waits for events and/or messages. If an Event or a message is received, the Task becomes ready for execution. If it has the highest priority of all Tasks in ready state, it is executed.

Example for a Task function:

```
void InitTask_Func(PxTask_t myID,
                   PxMbx_t myMailbox,
                   PxEvents_t myActivationEvents)
{
  // Declarations and initialisations
    int foo = 0;
    int bar;

    if(myActivationEvents == EV_ACT_EV1)
    {
        bar = 1;
    }
    else
    {
        bar = 42;
    }

    // Main loop
    while(1)
    {
      // Parallel waiting for events and messages
        PxMsgEvent_t msgev;
        msgev = PxMsgReceive_EvWait(myMailbox,EV_EVENT_MASK);
        if(PxMsgIdIsValid(msgev.msg))
        {
          // message received
            do_sth_with_msg(msgev.msg);
        }

        if(msgev.events != 0)
        {
          // event received
            do_sth_with_events(msgev.events);
        }
    }
}
```

6.2.4. Summary

Summary of the Initialisation of a PXROS-HR system:

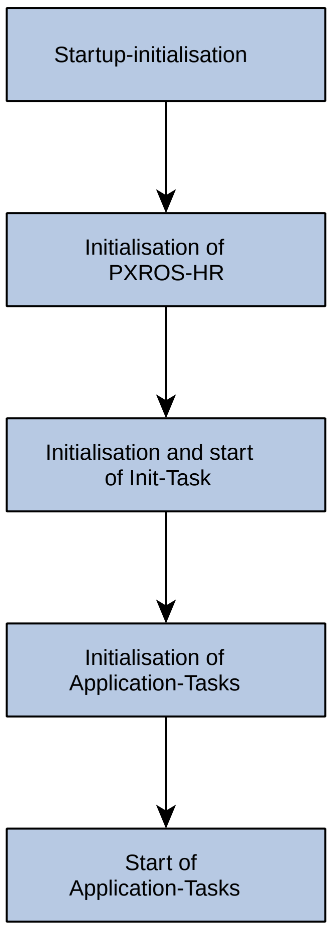
Startup-initialisation
Initialisation of PXROS-HR
Initialisation and start of Init-Task
Initialisation of Application-Tasks
Start of Application-Tasks

Figure 13. Initialisation

A complete module for implementing the initialisation of PXROS-HR and for creating a Task can be found in Appendix A.

6.3. Utilising system services

6.3.1. Basics

In principle, every PXROS-HR system service returns a value that should be evaluated. This value is either an object handle or a numeric type. A numeric value might be an error code, an event bit mask or a priority.

Handles are interface representations of PXROS-HR objects. If objects are requested from the system or passed on to system services, this is done via Handles.

If a service returns an error code, this code should be regarded in any case. If a service returns an object handle, the user should always check if the Handle is valid or if an error has occurred. `PxMsgIdIsValid(handle)` can be used for checking if a requested message object is valid.

6.3.2. Blocking and non-blocking system services

PXROS-HR system services can be blocking or non-blocking. Non-blocking system services return immediately after having been called, while blocking services can set a Task to the *waiting* state, which can cause scheduling. There are three reasons why a system call can be blocking:

1. Waiting for an event: the function returns as soon as it receives an event
2. Waiting at a mailbox: the function returns as soon as the mailbox contains message objects
3. Waiting at an object pool: if an object (e.g. a message) is requested from an object pool, a system service can block if the object pool is empty; it will then only return if objects are available again

6.3.3. System Calls in Tasks and Handlers

Due to the execution state of the PXROS-HR system the service calls are devided into two classes. Handlers as well as functions called via `_PxHndcall` (see Section 5.5) may only call system functions with the suffix _Hnd. Tasks, in contrast, may only call functions, which do *not* have the _Hnd suffix. If a call occurs at the wrong place, an error message will be returned: `PXERR_HND_ILLCALL` in case a Handler service was called in a Task, or `PXERR_TASK_ILLCALL` in the opposite case.

6.4. Object management

6.4.1. Object pools

PXROS-HR objects can, whenever they become necessary, be requested from an object pool.

A standard object pool is passed to a Task during its creation, yet the Task can also access other object pools if they are made available and if the Task has an access right to such resources (see Section 3.5.3).

An object pool can be used by several Tasks; thus a flexible distribution of objects can be achieved. The disadvantage of this method is that in the event of an object shortage, all the Tasks that have access to this object pool might be affected.

A typical application for object pools could be that Tasks, which are important for the system, have their own object pools assigned, while less important Tasks share a common object pool.

6.4.2. Requesting PXROS-Objects

PXROS-HR objects can be requested as specialized objects via functions. E.g. `PxPeRequest` will request an object of type `PxPe_t`.

Requesting objects is only possible within a Task; within a Handler this is not permitted (see

Section 6.3.3).

6.5. Memory management

Memory management under PXROS-HR is based on memory classes. A memory class is an object of the type `PxMc_t`. A memory class can manage memory in blocks of fixed or variable size. Fixed block size allows a quicker access to memory blocks, while variable block size makes it possible to access memory blocks in exactly the size in which they are needed, thus making memory utilisation more efficient. Memory classes are not usually used for requesting memory explicitly, but mainly for making memory available to message objects. By default, a Task has two memory classes at its disposal: `PXMcSystemdefault`, the standard memory class of the system, and `PXMcTaskdefault`, the standard memory class of the Task, which was assigned to it during creation. To use `PXMcSystemdefault` the Task must have the appropriate rights (see Section 3.5.3).

6.6. Communication

Communication and synchronisation of different Tasks is achieved via Events and messages. Messages are objects containing data, while Events are flags signalling events.

6.6.1. Events

An Event is a flag signalling a certain event. A Task can wait for certain Events by the function `PxAwaitEvents(PXEvents_t events)`. The parameter `events` is a bitmask stating for which Events the Task shall wait. `PxAwaitEvents` blocks until one or more of the events defined in `events` occur. If `events` is set to zero, `PxAwaitEvents` will not return.

```
...

#define EV_EV1    1
#define EV_EV2    (1<<1)
#define EV_EV3    (1<<2)

...

PXEvents_t ret_ev;

//  wait for Event EV_EV1, EV_EV2 or EV_EV3
ret_ev = PxAwaitEvents(EV_EV1|EV_EV2|EV_EV3);

// check, which event has arrived
if( ret_ev & EV_EV1)
{
  ...
}
if( ret_ev & EV_EV2)
{
  ...
}
if( ret_ev & EV_EV3)
{
  ...
}

...
```

`PxGetSavedEvents` returns the Events received but not yet processed by the Task. This function returns immediately.

The non-blocking resetting of the Events of a Task is carried out by `PxResetEvents(PXEvents_t events)`. The parameter `events` states, which Events are to be reset. This function returns a bitmask stating, which Events were reset.

```
...

PXEvents_t ret_ev;

// get all received Events
ret_ev = PxGetSavedEvents();

...

// all Events of ret_ev are reset
ret_ev = PxResetEvents(ret_ev);

...
```

6.6.2. Messages

Messages are PXROS-HR objects containing a data buffer and are used for data interchange between Tasks. A Task creating a message object is called the Owner of the message; the Task currently having access to a message object is called the *User* of the message.

6.6.2.1. Requesting and releasing message objects

Message objects can be requested in two different ways: They can either be created by `PxMsgRequest` or by `PxMsgEnvelop`. The difference between these two methods is that with `PxMsgRequest` the data memory of the message is taken from a memory class, while with the envelope mechanism the memory has to be provided by the requesting Task.

Message objects can be requested by Tasks but not by Handlers.

There are several ways of requesting message objects via `PxMsgRequest`:

PxMsg_t PxMsgRequest(PxSize_t msgsize, PxMc_t mcId, PxOpool_t opoolId)
: This function returns a message object taken from the object pool `opoolId`. It contains a data buffer of the size `msgsize`. The data buffer is taken from the memory class `mcId`. Should the object pool be empty, this function will block until objects are available again.

PxMsg_t PxMsgRequest_NoWait(PxSize_t msgsize, PxMc_t mcId, PxOpool_t opoolId)
: This function returns a message object taken from the object pool `opoolId`. It contains a data buffer of the size `msgsize`. The data buffer is taken from the memory class `mcId`. Should the object pool be empty, the returned Handle is invalid. This function returns immediately.

PxMsgEvent_t PxMsgRequest_EvWait(PxSize_t msgsize, PxMc_t mcId, PxOpool_t opoolId, PxEvents_t events)
: This function returns a message object taken from the object pool `opoolId`. It contains a data buffer of the size `msgsize`. The data buffer is taken from the memory class `mcId`. Should the object pool be empty, this function will block until objects are available again or until one of the Events stated in `events` occurs.

If the envelop mechanism is to be used, one of the following functions can be executed:

PxMsg_t PxMsgEnvelop(PxMsgData_t data_area, PxSize_t msgsize, PxOpool_t opoolid)
: This function packs the buffer `data_area` from the memory area of the Task into a message object taken from the object pool `opoolid` and returns this package. Should the object pool be empty, this function will block until objects are available again.

PxMsg_t PxMsgEnvelop_NoWait(PxMsgData_t data_area, PxSize_t msgsize, PxOpool_t opoolid)
: This function packs the buffer `data_area` from the memory area of the Task into a message object taken from the object pool `opoolid` and returns this package. Should the object pool be empty, this function will return an invalid Handle.

PxMsgEvent_t PxMsgEnvelop_EvWait(PxMsgData_t data_area, PxSize_t msgsize, PxOpool_t opoolid, PxEvents_t events)
: This function packs the buffer `data_area` from the memory area of the Task into a message object taken from the object pool `opoolid` and returns this package. Should the object pool be

empty, this function will block until objects are available again or until one of the Events specified in `events` occurs.

If the envelop mechanism is used, the data area of the message is not taken from a memory class; instead the creating Task has to provide a buffer from its own memory area.

A message can be released by the function `PxMsgRelease` or `PxMsgRelease_Hnd`. If a message object is released, the Task or Handler abandons its access right to the data area of the message. By default the message object is stored in its original object pool, and its data buffer is returned to the memory class where it was taken from or, if the message was created via `PxMsgEnvelop`..., data buffer is reallocated to the memory area of the original Owner of the message.

To be precise, the memory buffer packed via `PxMsgEnvelop` is not really taken from the memory area of the Task; the Task could, theoretically, still access this area. The buffer should, however, be treated as if it were extracted from the address space of the Task.

The behaviour of `PxMsgRelease`... can vary:

- If a message is fitted with a Release mailbox (via the function `PxMsgInstallRelmbx(PxMsg_t msgid, PxMbx_t mbxid)`), the behaviour of `PxMsgRelease`... changes: The corresponding message object is not released but stored in the mentioned Release mailbox instead (see Section 6.6.2.2).
- After a message was sent, the owner of the message can wait for the release via the function `PxMsgAwaitRel(PxMsg_t msg)`. To do so, `PxMsgSetToAwaitRel(PxMsg_t msg)` has to be called before sending. Then, if the recipient of the message calls `PxMsgRelease`, the message object is not released but returned to the sender instead.

If a message was marked via `PxMsgSetToAwaitRel` or requested by `PxMsgEnvelop`, the function `PxMsgAwaitRel(PxMsg_t msg)` has to be used to wait for the message release.

6.6.2.2. Mailboxes

Mailboxes are a central component of the communication mechanism. A message is not sent directly to the recipient Task, but to a mailbox, from which the recipient can read it. Mailboxes are organised according to the FIFO principle (first in, first out). A mailbox has unlimited capacity; an arbitrary number of messages can stored within it.

Apart from message interchange between Tasks, mailboxes can also serve as message pools or release mailboxes.

Message pools
: are mailboxes containing prefabricated message objects, which can be extracted and reused by a

Task (see also in A.3). In comparison to the requesting of message objects, this has several advantages:

- it is quicker since the message object already exists instead of having to be requested from the system first
- message objects are available even if no PXROS-HR objects are available and a request for an object from the object pool would fail
- the consumption of PXROS-HR objects is restricted, i.e. any object shortage caused by excessive use of message objects is of local relevance only

The disadvantages of using message pools:

- the number of stored message objects may exceed the number of required ones
- the size of the data areas of the message objects must be large enough for all use cases

Release Mailboxes
are mailboxes, where message objects are stored after release by the user, instead of being released. To utilise a release mailbox, the mailbox has to be assigned to the message via `PxMsgInstallRelmbx`.

To fulfill their purpose, message pools have to be specified as release mailboxes for the contained message objects, so that they can return to the pool after release by the message recipient.

The following code exemplifies how a message pool can be created and used:

```
...

  PxMbx_t mpool;
  PxMsg_t msg;

  msgpool = PxMbxRequest(my_opool); // request mailbox object

  if(PxMbxIdIsValid(msgpool))       // check, if object valid
  {

    PxMsg_t tmpmsg;
    for(i=0;i<10;i++)               // request 10 messages
    {
      tmpmsg = PxMsgRequest(MSG_SIZE, my_memclass,my_opool);
      if(PxMsgIdIsValid(tmpmsg))
      {
        PxMsgInstallRelmbx(tmpmsg,msgpoool); // set release mailbox
        tmpmsg = PxMsgSend(tmpmsg,msgpool)   // send message to message pool
        if(PxMsgIdIsValid(tmpmsg))
                do_error_handling();
      }
      else
      {
        do_error_handling();
      }
    }
  }
  else
  {
    do_error_handling();
  }

 ...

 msg = PxMsgReceive(msgpool); // a message is requested from the
                              // message pool

 fill_msg_data();

 PxMsgSend(msg,targetmbx)    // the message is sent to the mailbox
                             // targetmbx
 ...
```

6.6.2.3. Sending and receiving messages

Sending and receiving messages is always done via mailboxes. The sender sends a message to a mailbox, and the recipient receives the message from this mailbox.

A message is sent via the function PxMsgSend, the necessary parameters are the message object to be sent and the mailbox to which the message is to be sent. After sending, the sender does no longer have access to the message object, meaning the Handle of the message is invalid after sending. In addition, the sender passes the exclusive access right to the data area of the message to the recipient, i.e. a pointer to this area is invalid.

A message object and the corresponding data area belong exclusively to one user, i.e. they are allocated to one Task.

A Task can receive a message by taking it from a mailbox. `PxMsgReceive(PxMbx_t mbx)`` returns the next available message in accordance with the FIFO principle; prioritised messages are an exception to this rule. A prioritised message is sent to the mailbox in question via the function `PxMsgSend_Prio`. A prioritised message 'overtakes' the normal messages in the mailbox.

Messages can also be received in a non-blocking way: `PxMsgReceive_NoWait(PxMbx_t mbx)` returns immediately, either with a message object or with an invalid Handle if the mailbox is empty.

`PxMsgReceive_EvWait(PxMbx_t mbx, PxEvents_t ev)` makes it possible to wait for messages and Events at the same time. The function returns an object of the type `PxMsgEvent_t`. The following check helps find out whether a message or an Event (or both) was returned:

```
...

PxMsgEvent_t retval = PxMsgReceive_EvWait(mbx, EV_MASK);
if(PxMsgIdIsValid(retval.msg))
{
  // a message was received
  do_sth();
}

if(retval.events != 0)
{
  // one or more Events were received

  if(retval.event & EV1)
  {
    ...
  }

  if(retval.event & EV2)
  {
    ...
  }

  ...
}
```

6.6.2.4. Accessing the content of a message

A Task can access the content of a message as soon as the message was requested from the object pool or taken from a mailbox. The function `PxMsgGetData(PxMsg_t msg)` returns a pointer to the data area of the message object `msg`, `PxMsgGetBuffersize(PxMsg_t msg)` returns the size of the data buffer and `PxMsgGetSize(PxMsg_t msg)` returns the size of the data within the buffer. If handler want to access the message data `PxMsgGetData_Hnd(PxMsg_t msg)` must be used.

If access to a data area is no longer required, yet the message is not to be released or sent, the access to the data area should be released via the function `PxMsgRelDataAccess(PxMsg_t msg)` since parallel access is limited to a maximum of four message buffers.

If access to a data area was released via `PxMsgRelDataAccess(PxMsg_t msg)`, a pointer to this area may only be used after it was regained via `PxMsgGetData`.

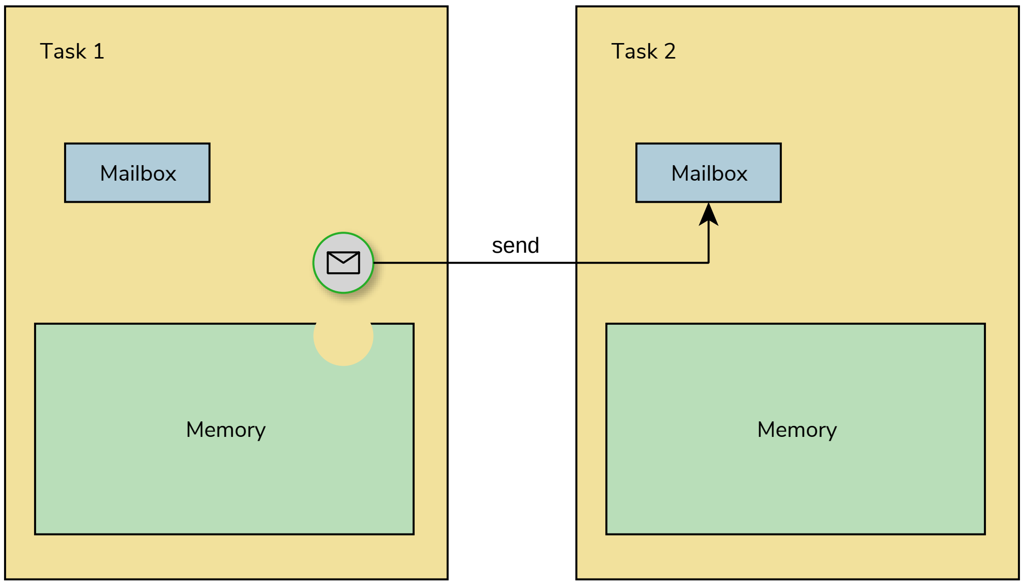

Figure 14. Creating and sending a message

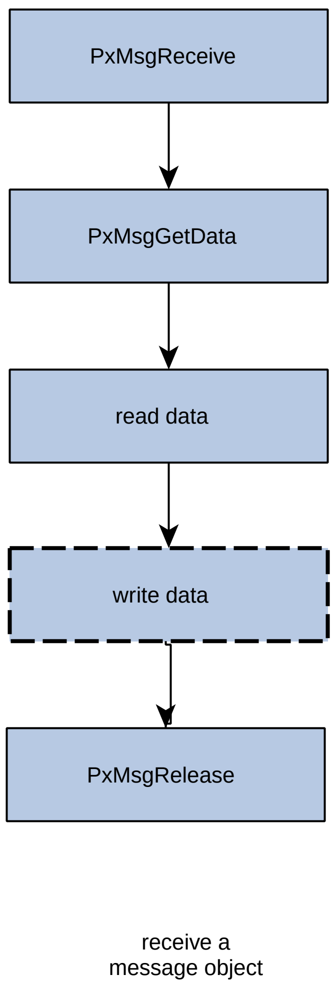

Figure 15. Receiving a message

6.6.2.5. Sample Code

The following example shows how to create, send and receive a message and how to access the received data.

Task1 creates and sends a message to Task2:

```
// Task1
...

 struct comm_data data;
 PxMsg_t msg;
 PxError_t error;

...

 // create message with data as data area of the message
 msg = PxMsgEnvelop(&data,sizeof(data),my_opool);

 if(!PxMsgIdIsValid(msg))
 {
   do_error_handling();
   return;
 }

 error = PxMsgSetToAwaitRel(msg);
 if(error != PXERR_NOERROR)
 {
   do_error_handling();
   return;
 }

 // Send the message; access to data and msg
 // is no longer permitted
 msg = PxMsgSend(msg,mbx_task2);

 // if sending was successful, the message Handle
 // is invalid
 if(PxMsgIdIsValid(msg))
 {
   do_error_handling();
   return;
 }

 // wait for recipient to release the message
 msg = PxMsgAwaitRel(msg);

 if(!PxMsgIdIsValid(msg))
 {
   do_error_handling();
   return;
 }

 ...

 // the message is finally released
 PxMsgRelease(msg);
```

Task2 waits at its mailbox for messages from Task1:

```
// Task2
...

 struct comm_data* pdata;
 PxMsg_t msg;

 // wait for message
 msg = PxMsgReceive(my_mbx);
 if(!PxMsgIdIsValid(msg))
 {
   do_error_handling();
   return;
 }

 // read data area of the message
 pdata = (struct comm_data*)PxMsgGetData(msg);

...

 // Release message; access to msg and pdata
 // is no longer permitted
 PxMsgRelease(msg);

...
```

6.6.2.6. Access restriction

Access to the data area can be restricted for a user of the message. Either the type of access or the size of the accessible area can be restricted.

WRProtection:
: permits unrestricted access to the data area, i.e. the user has the right to read and write

WriteProtection:
: permits the user to write into the data buffer

ReadProtection:
: permits the user to read from the data buffer

NoAccessProtection:
: permits no access to the data buffer

Access protection can only be changed by the owner of the message.

The user of a message can restrict the data area of a message; this is achieved via `PxMsgSetData` and `PxMsgSetSize`. The start address of the data area is changed via `PxMsgSetData`; the new address has to lie within the original data area. The size of the data area is automatically adjusted, so that the last address lies within the original data area.

The size of the data area can be reset via `PxMsgSetSize`; care has to be taken that the last address of the new data area lies within the original data area.

An example:

Task1 sends a message to Task2:

```
...
struct foo
{
  int x;
  int y;
  struct bar b;
  int z;
} data;

...
// create message and pack data into message
msg = PxMsgEnvelop(&data,sizeof(data),defaultopool);
PxMsgSetToAwaitRel(msg);

// protect data area of the message; the recipient
// has only read rights to the buffer; he may not
// write into it
PxMsgSetProtection(msg, ReadProtection);

...
// send to Task2
PxMsgSend(msg,task2mbx);
...
// wait for release of message
msg = PxMsgAwaitRel(msg);

// reset to original data area
PxMsgSetData(msg,0);

// reset to original size of the data area
PxMsgSetSize(msg,sizeof(data));
...
```

Task2 passes the message on to Task3; Task3 shall only have access to the component b of the structure foo:

```
...
struct foo* pData;
...
// receive message
msg = PxMsgReceive(task2mbx);
...
pData = PxMsgGetData(msg);
...
// restrict start address and size of the data area to Element b of
//  the structure foo
// Set start address to b
PxMsgSetData(msg,&pData->b);

// Set size to size of b, i.e. z is removed
PxMsgSetSize(msg,sizeof(pData->b));

// Note: If only the beginning of the data area is changed, the
// size of the remaining area is automatically adjusted

...
// send to Task3
PxMsgSend(msg,task3mbx);
...
```

Task3 receives the message and then releases it:

```
...
struct bar* pData;
PxSize_t size;
...
// receive message
msg = PxMsgReceive(task3mbx);
...
// request data area and size of the area
pData = PxMsgGetData(msg);
size = PxMsgGetSize();
...
PxMsgRelease(msg);
...
```

6.7. Interrupt and Trap Handler

Handlers are functions reacting to asynchronous events, i.e. to hardware or software interrupts or to traps. They are generally of higher priority than Tasks, meaning a running Task can always be interrupted by a Handler.

There are three types of interrupt Handlers: Context Handlers, Fast Context Handlers and Fast Handlers. Depending on the individual use case the according type should be chosen (see Section 3.2.2).

When starting the system, interrupts are enabled.

6.7.1. Fast Handlers

Fast Handlers are the fastest and highest prioritised interrupt Handlers. They run with system privileges in interrupt mode and are activated as soon as an interrupt occurs.

A Task can only install a Fast Handler if it has the appropriate permission (`PXACCESS_HANDLERS`, see Section 3.5.3).

A Fast Handler is installed via the function

PxError_t PxIntInstallFastHandler(PxUInt_t intno, **void**(* inthandler)(PxArg_t), PxArg_t arg)

`intno`
: the interrupt number to be handled

`inthandler`
: the Handler function

`arg`
: the argument of the Handler function

The function returns the error code `PXERR_REQUEST_INVALID_PARAMETER` if `intno` lies outside the specification, otherwise `PXERR_NOERROR`.

The argument `arg` can be used for exchanging data between the Handler and a Task. An example for data exchange between Handler and Task can be found in Section A.2.

Instead of a self-implemented Handler, a service provided by PXROS-HR can also be installed as a Handler. To do so, the function

PxError_t PxIntInstallService(PxUInt_t intno, PxIntSvEnum_t service, PxArg_t arg, PxEvents_t events)
should be used. The parameters have the following meaning:

`intno`
: the interrupt number to be handled

`service`
: the service to be installed

`arg`
: the argument of the service function

`events`
: the Events possibly sent by the service function

The function returns the error code `PXERR_REQUEST_INVALID_PARAMETER` if `intno` lies outside the specification, or `PXERR_ILLEGAL_SERVICE_CALLED` if an invalid service was defined, otherwise `PXERR_NOERROR`.

To be able to install a service, the Task has to have the permission `PXACCESS_INSTALL_SERVICES`.

The following services are available:

`PxTickDefine_IntHnd_SvNo`
: calls the function `PxTickDefine_Hnd` (see Section 6.8.1), `arg` and `events` are ignored

`PxTaskSignalEvents_IntHnd_SvNo`
: calls the function `PxTaskSignalEvents_Hnd`; the Handle of the recipient Task has to be passed in `arg`, the Events to be sent in `events`

6.7.2. Context Handlers

Context Handlers are executed with the privileges of the Task which installed it, and run in the address space of that Task. The Task must have the right to install Context Handlers (`PXACCESS_INSTALL_HANDLERS`, see Section 3.5.3).

A Context Handler is not immediately called if the corresponding interrupt has occurred; instead it is entered into a list, which is processed after returning from interrupt level to system level.

> Due to the fact that a Context Handler is first entered into a list, no assumption can be made regarding its execution time. Real time behaviour is not given!

Before a Context Handler can be installed, an interrupt object has to be requested via `PxInterruptRequest`. A Handler is installed via:

PxError_t PxIntInstallHandler(PxUInt_t intno, PxInterrupt_t intObj, **void**(*inthandler)(PxArg_t), PxArg_t arg)

`intno`
: the interrupt number to be handled

`intobj`
: the Handle of the corresponding interrupt object

`inthandler`
: the Handler function

`arg`
: the argument of the Handler function

The function returns the error code `PXERR_REQUEST_INVALID_PARAMETER` if `intno` lies outside the specification, otherwise `PXERR_NOERROR`.

6.7.3. Fast Context Handlers

Fast Context Handlers are executed with the privileges of the Task which installed it, and run in the address space of that Task. The Task must have the right to install Handlers (`PXACCESS_INSTALL_HANDLERS`, see Section 3.5.3).

A Fast Context Handler is called immediately if the corresponding interrupt has occurred, depending on the hardware priority of the interrupt.

A Handler is installed via:

PxError_t PxIntInstallFastContextHandler(PxUInt_t intno, PxIntHandler_t inthandler, PxArg_t arg)

`intno`
: the interrupt number to be handled

`inthandler`
: the Handler function

`arg`
: the argument of the Handler function

The function returns the error code `PXERR_REQUEST_INVALID_PARAMETER` if `intno` lies outside the specification, otherwise `PXERR_NOERROR`.

6.7.4. Trap handler

A trap handler is similar to an interrupt handler and, in principle, has the same functionality. The difference in the two lies in that an interrupt handler is triggered by a hardware or software interrupt, while a trap handler is triggered by an exception, such as a faulty memory access operation. If a trap handler is called, all hardware interrupts are blocked.

A trap handler is installed by the function

PxError_t PxTrapInstallHandler(PxUInt_t trapno, PxBool_t(* traphandler)(PxTrapTin_t, PxUInt_t, PxUInt_t, PxUInt_t, PxUInt_t *, TC_CSA_t *), PxUInt_t arg)

The three parameters are defined as:

trapno
: the trap number the handler is installed for

traphandler
: the handler which is installed for the trap

arg
: the user defined argument which is passed to the handler

The trap handler is defined by the prototype

PxBool_t TrapHandler(PxTrapTin_t trapTin, PxArg_t arg, PxUInt_t runtaskid, PxUInt_t dstr, PxUInt_t *deadd, TC_CSA_t *csa)

The five parameters are defined as:

TrapTin
: The trap number and the TIN of the trap

arg
: the user defined argument passed to the trap handler

runtaskid
: the object ID of the active Task at the time the trap was triggered

dstr
: the content of the register DSTR

deadd
: the content of the register DEADD

csa
: a pointer to the stored CSA (Context Save Area)

The handler function should return TRUE, if the trap condition has been solved by the handler, FALSE otherwise.

6.8. Timer functionality

The time unit used by PXROS-HR is a *Tick*. A Tick is an abstract time unit, the length of which has to be defined by the application.

6.8.1. Basic timer functionality

The internal time management of PXROS-HR is based on an internal, system-independent time unit, the Tick. The consecutive number of Ticks having passed since the start-up of the timer system is internally administered and can be read or incremented via system services.

The allocation of Ticks to a tangible time basis is done via function `PxTickSetTicksPerSecond`, which states how many Ticks correspond to one second. `PxTickSetTicksPerSecond(100)`, for example, sets the length of a Ticks to 10 milliseconds.

> It should be ensured that the time basis is the same for the initialization of the timer interrupts and when calling `PxTickSetTicksPerSecond`.

The continued counting of Ticks is performed by the function `PxTickDefine_Hnd`. This function has to be called within the Handler of the timer interrupts and registered as a service for the timer interrupt (see installation of an interrupt service).

The number of Ticks since the first call of `PxTickDefine_Hnd` can be queried via `PxTickGetCount`.

The time passed since the first call of `PxTickDefine_Hnd` can be queried via `PxTickGetTimeInMilliSeconds`.

The function `PxGetTicksFromMilliSeconds(unsigned long ms)` converts `ms` milliseconds to Ticks.

6.8.2. Delay Jobs

A delay job executes a function as a Handler after a predefined period of time. A Delay Job object can be requested from an object pool via `PxDelayRequest` (and `PxDelayRequest_NoWait` and `PxDelayRequest_EvWait`).

With the system services

PxError_t PxDelaySched(PxDelay_t delayId, PxTicks_t ticks, **void** (*handler)(PxArg_t), PxArg_t arg)
and

PxError_t PxDelaySched_Hnd(PxDelay_t delayId, PxTicks_t ticks, **void** (*handler)(PxArg_t), PxArg_t arg)
a function can be registered for subsequent processing. In this context, `delayId` is the Handle of the delay object, `ticks` the number of Ticks after which the function is to be executed, `handler` the pointer to the function to be executed, and `arg` the argument passed to the function.

In the below example, a Handler function is periodically called by repeatedly triggering execution from within in the function itself:

```
typedef struct _HndArg {
PxDelay_t delay;
PxTicks_t ticks;
} HndArg_t;

void handler_function(PxArg_t arg)
{
    HndArg_t *hndarg = (HndArg_t *)arg;
    PxError_t error;

  ...

   error = PxDelaySched_Hnd(hndarg->delay,hndarg->ticks,handler_function,arg);

  ...
}

...

void TaskFunc()
{
    ...

    PxDelay_t delay;
    PxTicks_t ticks;
    PxError_t error;
    HndArg_t  hndArg;

    ...

    delay = PxDelayRequest(defaultopool);

    if(PxDelayIdIsValid(delay))
    {
        ticks = PxGetTicksFromMillis(100);
        hndArg.delay = delay;
        hndArg.ticks = ticks;
        error = PxDelaySched(delay,ticks,handler_function,(PxArg_t)&hndArg);
        if(error != PXERR_NOERROR)
        {
          do_error_handling();
        }

        ...

    }

    ...
}
```

6.8.3. Timeout objects and periodical timers

A typical application for a Delay Job Handler is sending a timeout event to a Task. Timeout objects can be used to avoid having to implement this action over and over again. A Timeout object can be requested in the following way:

PxTo_t to = PxToRequest(PxOpool_t opool,PxTicks_t ticks,PxEvents_t ev)
The parameters have the following meaning:

`opool`
: the object pool from which the timeout object is to be taken

`ticks`
: the number of Ticks after which the Events are to be sent

`ev`
: the Events to be sent after expiry of the timeout

The timer starts after the function call `PxToStart(PxTo_t to)`. It can be terminated via `PxToStop(PxTo_t to)` if it is not yet expired.

Periodical timers (`PxPe_t`) differ from timeout objects in that they do not just send a single timeout Event but periodical ones.

6.9. Access to peripherals

There are two ways of accessing peripheral registers: A Task in `Usermode0` can access peripheral registers via system calls if it has explicit permission; a Task in `Usermode1` has full access to peripheral registers.

6.9.1. Access in Usermode1

In `Usermode1` a Task has direct access to peripheral registers insofar as this is permitted by the hardware. This has the advantage that peripheral registers can be read and written in a fast way.

The disadvantage of this method is that the Task has access to all the registers to which the hardware has granted read/write permission, not only to those the Task actually needs. Erroneous access operations are thus possible.

6.9.2. Access via system calls

In `Usermode0` a Task can access peripheral registers for read/write operations solely via system calls. Thus, peripheral access is slower in `Usermode0` than in `Usermode1`.

In `Usermode0` a Task can have read/write permission granted for certain registers. Under safety aspects this is an advantage over direct access.

If a Task wishes to access peripheral registers via system calls, it has to have access permission `PXACCESS_REGISTERS` (see Section 3.5.3). Furthermore, the address areas of the desired registers must be defined with the required access rights (read and/or write) in the additional memory areas table of the Task (`PxTaskSpec_T.ts_protect_region`, see task specification)

The following system function is available for reading a peripheral register:

PxULong_t PxRegisterRead(**volatile** PxULong_t *addr)

This function receives the address of the desired register as a parameter and returns the content of the register, zero in case of an error.

If the function returns zero, it cannot be determined whether this is the actual content of the register or an erroneous access operation.

The function

PxError_t PxRegisterWrite(**volatile** PxULong_t *addr, PxULong_t value)

writes the value `value` into the register with the address `addr`. If the access was successful, the function will return `PXERR_NOERROR`; in case of an error, it will return either `PXERR_PROT_ILL_REGION` or `PXERR_ACCESS_RIGHT`.

With the function

PxError_t PxRegisterSetMask(**volatile** PxULong_t *addr, PxULong_t mask, PxULong_t value)

the bits masked in `mask` in the register with address `addr` are set to the value stated in `value`. If the access was successful, the function will return `PXERR_NOERROR`; in case of an error, it will return either `PXERR_PROT_ILL_REGION` or `PXERR_ACCESS_RIGHT`.

6.10. The abort mechanism

`PxExpectAbort` makes it possible to abort the execution of a function from within another function. The function must have been started via `PxExpectAbort(ev, func, params …)`. The parameters have the following meaning:

`ev`
    the Events causing the abort of the function

`func`
    the function to be executed. `func` may have arbitrary parameters, yet it may not return any values

`params`
    a list of the parameters passed to `func`

`PxExpectAbort` returns the Event, which has caused the abort, or zero if the function was regularly terminated.

In the following example the function `func` is called in `Task1` via `PxExpectAbort`. By sending the Event `EV_ABORT` Task2 aborts the execution of the function.

```
...

void func(int i)
{
  ...
}

void Task1Func()
{
  ...

  PxEvents_t retval;

  ...

  retval = PxExpectAbort(EV_ABORT,func,42);

  if(retval == 0) // function was regularly terminated
  {
    ...
  }
  else // the function was aborted
  {
    ...
  }

  ...
}

void Task2Func()
{

  ...

  PxTaskSignalEvents(Task1,EV_ABORT);

  ...
}
...
```

6.11. Suspend and resume tasks

Sometimes it may be useful, to suspend a task from the execution. A suspended task remains in the condition when it was suspended, but is no longer part of the scheduling.

To suspend a task the function `PxTaskSuspend(PxTask_t)` is used:

```
PxError_t   err;
PxTask_t    Task_ID;
    err = PxTaskSuspend(Task_ID);
```

A suspended task may be resumed, if it is necessary to let it be part of the scheduling again.

To resume a task the function `PxTaskResume(PxTask_t)` is used:

```
PxError_t   err;
PxTask_t    Task_ID;
    err = PxTaskResume(Task_ID);
```

The task must have the access permission `PXACCESS_SYSTEM_CONTROL` (see Section 3.5.3) to execute these functions.

Appendix A: Example code

A.1. For starters: a simple application for TC2x

```
#include <pxdef.h>

#undef _STRINGIFY
#define _STRINGIFY(x)   #x
#define _DEF_SYM(n,v)                         \
    __asm (".global       " #n );           \
    __asm (".set  " #n "," _STRINGIFY(v));  \
    __asm (".type " #n ",STT_OBJECT")

#define DEF_SYM(name,val)   _DEF_SYM(name,val)


// Priority of the Inittask after initialisation
#define MIN_PRIO       31

// Priority of Task1
#define TASK1_PRIO      4


#define PXROS_NAMESIZE          12
#define NUM_OF_PXOBJS           100
#define SYSMEMSIZE              2000
#define TASKMEMSIZE            10000

#define INITTASK_STACKSIZE      200
#define INITTASK_INTSTACKSIZE   0

#define TASK1_STACKSIZE         200
#define TASK1_INTSTACKSIZE      32

/* define the symbol __PXROS_NAMESIZE__
 * to make the object name size accessible by the linker
 */
DEF_SYM(__PXROS_NAMESIZE__, PXROS_NAMESIZE);

/* declaration of object pool sizes (for linker) */
DEF_SYM(__NUM_OF_PXOBJS__CPU0_, NUM_OF_PXOBJS);


/* start address for slave cores (started by master core) */
extern void _start(void);

/* data area description for the system */
/* these symbols are defined in the system linker script Multi.ld */

/* PXROS object memory for each CPU */
extern PxMemAligned_t  PxObjmem_CPU0_[];
extern unsigned int PX_OBJMEMSIZE_CPU0_[];

/* description of system stack (for each CPU) */
extern const PxUInt_t PXROS_SYSTEM_STACK_BEGIN_CPU0_[];
extern const PxUInt_t PXROS_SYSTEM_STACK_SIZE_CPU0_[];

/* memory class for system objects */
#pragma section ".CPU0.systemmemory" awBc0 8
static PxMemAligned_t Sysmem[(SYSMEMSIZE + sizeof(PxMemAligned_t) - 1) / sizeof(PxMemAligned_t)]
PXMEM_ALIGNED;
#pragma section

#pragma section ".CPU0.taskmemory" awBc0 8
/* memory class for task this will also be defined as PXMcTaskdefault to the inittask */
static PxMemAligned_t Taskmem[(TASKMEMSIZE + sizeof(PxMemAligned_t) - 1) / sizeof(
PxMemAligned_t)] PXMEM_ALIGNED;
#pragma section

// Function prototypes

// Task function of the Inittask
static void InitTask_Func(PxTask_t myID,
                          PxMbx_t myMailbox,
                          PxEvents_t myActivationEvents);


// Task function of Task1
static void Task1_Func(PxTask_t myID,
                       PxMbx_t myMailbox,
                       PxEvents_t myActivationEvents);


// Code Area
extern PxUInt_t __TEXT_BEGIN[];
extern PxUInt_t __TEXT_END[];

// context save area
extern PxUInt_t __CSA_BEGIN_CPU0[];
extern PxUInt_t __CSA_END_CPU0[];

// System data areas
extern PxUInt_t   PxTricSystemRodataLowerBound[];
extern PxUInt_t   PxTricSystemRodataUpperBound[];

extern PxUInt_t   PxTricSystemDataLowerBound_CPU0_[];
extern PxUInt_t   PxTricSystemDataUpperBound_CPU0_[];


/* CPU0 Protection description */
const PxCodeProtectSet_T _cpu0_sys_code_protection =
{
    /* Range 0 the complete text section */
    .cpr[0].s = {(PxUInt_t)__TEXT_BEGIN,     (PxUInt_t)__TEXT_END,},
    .cpmr.cpxe.bits =
    {
        .dp0 = 1,       /* the CPXE 0 executable */
    }
};

const PxCodeProtectSet_T _cpu0_task_code_protection =
{
    /* Range 0 the complete text section */
    .cpr[0].s = {(PxUInt_t)__TEXT_BEGIN, (PxUInt_t)__TEXT_END,},
    .cpmr.cpxe.bits =
    {
        .dp0 = 1,       /* the CPXE 0 executable */
    }
};

const PxDataProtectSetInit_T _cpu0_sys_data_protection =
{
    /* Range 0: read only data */
    .dpr[0].s = {(PxUInt_t)PxTricSystemRodataLowerBound,
                 (PxUInt_t)PxTricSystemRodataUpperBound,},
    /* Range 1: the CSA area of CPU0 (global address) */
    .dpr[1].s = {(PxUInt_t)__CSA_BEGIN_CPU0_,
                 (PxUInt_t)__CSA_END_CPU0_,},
    /* Range 2: the KERNEL data area of CPU0 (global address) */
    .dpr[2].s = {(PxUInt_t)PxTricSystemDataLowerBound_CPU0_,
                 (PxUInt_t)PxTricSystemDataUpperBound_CPU0_},
    /* Range 3: the SFR area */
    .dpr[3].s = {(PxUInt_t)PERIPHERAL_MEM_BASE,
                 (PxUInt_t)PERIPHERAL_MEM_END,},
    /* Range 4: the supervisor stack */
    .dpr[4].s = {(PxUInt_t)PXROS_SYSTEM_STACK_BEGIN_CPU0_,
                 (PxUInt_t)PXROS_SYSTEM_STACK_CPU0_,},
    /* Range 7: used dynamically by system
                used to get access to memory before PXROS is started
                - especially used for copy and clear functions
    */
    .dpr[7].s = {0,0},
    /* the DPRE 0..4,7 readable */
    .dpmr.kernel.dpre.bits =
    {
        .dp0 = 1, .dp1 = 1, .dp2 = 1, .dp3 = 1,
        .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 1
    },
    /* the DPWE 1..4,7 writable */
    .dpmr.kernel.dpwe.bits =
    {
        .dp0 = 0, .dp1 = 1, .dp2 = 1, .dp3 = 1,
        .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 1
    },
    /* the DPRE 0,3,4 readable */
    .dpmr.system.dpre.bits =
    {
        .dp0 = 1, .dp1 = 0, .dp2 = 0, .dp3 = 1,
        .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 0
    },
    /* the DPWE 3,4 writable */
    .dpmr.system.dpwe.bits =
    {
        .dp0 = 0, .dp1 = 0, .dp2 = 0, .dp3 = 1,
        .dp4 = 1, .dp5 = 0, .dp6 = 0, .dp7 = 0
    },
};

#define INITTASK_ACCESS_RIGHTS  PXACCESS_HANDLERS               \
                              | PXACCESS_INSTALL_HANDLERS       \
                              | PXACCESS_INSTALL_SERVICES       \
                              | PXACCESS_REGISTERS                  \
                              | PXACCESS_SYSTEMDEFAULT              \
                              | PXACCESS_RESOURCES                  \
                              | PXACCESS_NEW_RESOURCES              \
                              | PXACCESS_SYSTEM_CONTROL             \
                              | PXACCESS_MODEBITS                   \
                              | PXACCESS_TASK_CREATE                \
                              | PXACCESS_TASK_CREATE_HIGHER_PRIO    \
                              | PXACCESS_TASK_SET_HIGHER_PRIO       \
                              | PXACCESS_CHANGE_PRIO                \
                              | PXACCESS_TASK_RESTORE_ACCESS_RIGHTS \
                              | PXACCESS_TASK_CREATE_HIGHER_ACCESS


// Memory context of Inittask
static const PxTaskContext_T InitTaskContext =
{
    .protection[0].lowerBound = (PxUInt_t)&PxTricSystemRodataLowerBound,
    .protection[0].upperBound = (PxUInt_t)&PxTricSystemRodataUpperBound,
    .protection[0].prot       = ReadProtection,

    .protection[1].lowerBound = 0,
    .protection[1].upperBound = 0,
    .protection[1].prot       = NoAccessProtection,
};


// Specification of Inittask
static const PxTaskSpec_T InitTaskSpec =
{
    .ts_name                     = "InitTask",
    .ts_fun                      = InitTask_Func,
    .ts_mc                       = PXMcTaskdefault,
    .ts_opool                    = PXOpoolSystemdefault,
    .ts_prio                     = 0,
    .ts_privileges               = PXUser1Privilege,
    .ts_context                  = &InitTaskContext,
    .ts_taskstack.stk_type       = PXStackAlloc,
    .ts_taskstack.stk_size       = INITTASK_STACKSIZE,
    .ts_taskstack.stk_src.mc     = PXMcTaskdefault,
    .ts_abortstacksize           = 0.
    .ts_accessrights             = INITTASK_ACCESS_RIGHTS,
};


// Memory context of Task1
static const PxTaskContext_T Task1Context =
{
    .protection[0].lowerBound = 0,
    .protection[0].upperBound = 0,
    .protection[0].prot       = NoAccessProtection,

    .protection[1].lowerBound = 0,
    .protection[1].upperBound = 0,
    .protection[1].prot       = NoAccessProtection,
};
```

```
// Specification of Task1
static const PxTaskSpec_T Task1Spec =
{
    .ts_name                        = "Task1",
    .ts_fun                         = Task1_Func,
    .ts_mc                          = PXMcTaskdefault,
    .ts_opool                       = PXOpoolTaskdefault,
    .ts_prio                        = 0,
    .ts_privileges                  = PXUser0Privilege,
    .ts_context                     = &Task1Context,
    .ts_taskstack.stk_type          = PXStackAlloc,
    .ts_taskstack.stk_size          = TASK1_STACKSIZE,
    .ts_taskstack.stk_src.mc        = PXMcTaskdefault
    .ts_abortstacksize              = 0
};


// PXROS-HR system specification
static const PxInitSpec_T InitSpec =
{
    .is_sysmc_type          = PXMcVarsized,
    .is_sysmc_size          = 0,
    .is_sysmc_blk           = Sysmem,
    .is_sysmc_blksize       = SYSMEMSIZE,
    .is_obj_number      = NUM_OF_PXOBJS_CORE0,
    .is_obj_namelength  = PXROS_NAMESIZE,
    .is_inittask        = &InitTaskSpec_CORE0,

    .is_objmc_type      = PXMcVarsizedAligned,
    .is_objlmc_size     = 8,
    .is_objmc_blk       = PxObjmem_CPU0_,
    .is_objmc_blksize   = (PxSize_t)PX_OBJMEMSIZE_CPU0_,

    .is_taskmc_type     = PXMcVarsizedAdjusted,
    .is_taskmc_size     = 8,
    .is_taskmc_blk      = Taskmem,
    .is_taskmc_blksize  = TASKMEMSIZE,

    /* the system stack */
    .is_system_stack = PXROS_SYSTEM_STACK_BEGIN_CPU0_,
    .is_system_stack_size = (PxUInt_t)PXROS_SYSTEM_STACK_SIZE_CPU0_,

    /* the protection definition */
    .is_sys_code = &_cpu0_sys_code_protection,
    .is_sys_data = &_cpu0_sys_data_protection,
    .is_task_code = &_cpu0_task_code_protection,
};

static const PxInitSpecsArray_t InitSpecsArray =
{
    &InitSpec,
    0,
    0,
};



// Code of Inittask
```

```
static void InitTask_Func(PxTask_t myID,
                          PxMbx_t myMailbox,
                          PxEvents_t myActivationEvents)
{
    /* create task1 */
    PxTask_t taskID = PxTaskCreate(PXOpoolSystemdefault,
                                   &Task1Spec,
                                   TASK1_PRIO,
                                   0);
    /* minimize own priority */
    PxTaskSetPrio(myID,MIN_PRIO);

    /* loop forever */
    while(1) ;
}



// Code of Task function Task1_Func
static void Task1_Func(PxTask_t myID,
                       PxMbx_t myMailbox,
                       PxEvents_t myActivationEvents)
{
  // Declarations and initialisation
    int foo = 0;
    int bar;

    if(myActivationEvents == EV_ACT_EV1)
    {
        bar = 1;
    }
    else
    {
        bar = 42;
    }


    //  Main loop
    while(1)
    {
      // parallel waiting for events and messages
        PxMsgEvent_t msgev;
        msgev = PxMsgReceive_EvWait(myMailbox,EV_EVENT_MASK);
        if(PxMsgIdIsValid(msgev.msg))
        {
          // message received
            do_sth_with_msg(msgev.msg);
        }

        if(msgev.events != 0)
        {
          // event received
            do_sth_with_events(msgev.events);
        }
    }
}
```

```
int main(void)
{
    PxError_t error;

    error = PxInit(&InitSpecArray, 1);
    if (error != PXERR_NOERROR)
    {
      PxPanic();
      return 1;
    }
    return 0;
}
```

A.2. Data exchange between Handler and Task

```
struct
{
    PxTask_t    commTask;  // TaskID
    PxEvents_t commEv;     // Data-Ready event
    PxUChar     commMem[100];  // Memory areas of the communication structure
} TaskHandlerComm;

#define MY_HANDLER_EVENT    1

#define COMM_BUFFER_SIZE    512

void HandlerFunc(PxArg_t arg);


TaskFunc(PxTask_t myID, PxMbx_t myMailbox, PxEvents_t myActivationEvents)
{
    struct TaskHandlerComm myTaskHandlerComm;

...

    // Initialising the communication object
    myTaskHandlerComm.commTask = myID;
    MyTaskHandlerComm.commEv   = MY_HANDLER_EVENT;


    // Installing Handler with communication structure
    // as an argument
    PxIntInstallFastContextHandler(COMM_INTERRUPT,
                                   HandlerFunc,
                                   &myTaskHandlerComm);

    while (1)
    {
...
        if (ev & MY_HANDLER_EVENT)
        {
          //  Handler has received data and entered
          //  them in communication structure
          ProcessInputData(myTaskHandlerComm);
        }
...
```

```
    }
}


void HandlerFunc(PxArg_t arg)
{
    struct TaskHandlerComm *myTaskHandlerComm;
    PxUChar_t byte;

    myTaskHandlerComm  = (struct TaskHandlerComm *) arg;

     // Store input data in communication structure
     byte = GetInputByte();
     StoreInputData(myTaskHandlerComm->commMem);

     if (DataReadyForTask())
     {
        // Complete set of data received, Task is
        // informed
        PxTaskSignalEvents_Hnd(myTaskHandlerComm->commTask,
                               myTaskHandlerComm->commEv);
      }
}
```

A.3. Messagepools

A.3.1. Creating a Messagepool

```
//  creates a new Mailbox as a Messagepool
//  and enters "cnt" new Messages with data buffers
//  of "size" bytes size in this Mailbox
//  Objects are extracted from "srcpool", memory from "srcmc"
//  Messages have "poolmbx" as their release Mailbox,
//  i.e. they are stored here after release.

PxMbx_t MsgPoolCreate(PxUInt_t cnt,
                     PxSize_t size,
                     PxMc_t srcmc,
                     PxOpool_t srcopool)
{
   PxMbx_t    poolmbx;
   PxError_t  err;
   PxTmode_t  oldmode;
   PxMsg_t    msg;

   //   Avoid interruptions
   oldmode = PxSetModebits(PXTmodeDisableAborts);

   //   Create new Mailbox
   poolmbx = PxMbxRequest_NoWait(srcopool);

   if (!PxMbxIdIsValid(Poolmbx))
   {
       // Creation error
```

```
        PxSetError(poolmbx.error);
        PxClearModebits(~oldmode);
        return poolmbx;
    }

    //   Request "cnt" Messages and store in Mailbox
    while (cnt--)
    {
        // Request Message
        msg = PxMsgRequest_NoWait(size, srcmc, srcopool);
        if (PxMsgIdIsValid(msg))
        {
            // Install release Mailbox
            PxMsgInstallRelmbx(msg, poolmbx);

            // Message is stored in Messagepool
            PxMsgSend(msg, poolmbx);
        }
        else
        {
            // Error has occurred, Messagepool is
            // deleted
            err             = msg.error;

            // Release existing Messages
            poolmbx         = MsgPoolDelete(Poolmbx);
            poolmbx.error   = err;
            PxSetError(poolmbx.error);
            break;
        }
    }

    //   Reset mode bits
    PxClearModebits(~oldmode);

    //   Return Messagepool
    return poolmbx;
}
```

A.3.2. Deleting a Messagepool

```
//  Deletes a Messagepool, which was created via
// MsgPoolCreate. All concerned Messages must
//  be present within the Messagepool.
//  If a pool Message is deleted after
//  deletion of the Messagepool, a
//  "PXERR_MBX_ILLMBX" will occur.
//  Tasks may not wait at this Mailbox,
// otherwise Mailbox cannot be deleted.
PxMbx_t MsgPoolDelete(PxMbx_t poolmbx)
{
    PxMsg_t      msg;
    PxTmode_t    oldmode;

    // Avoid interruptions
    oldmode = PxSetModebits(PXTmodeDisableAborts);

    // Release all messages
    for (msg = PxMsgReceive_NoWait(poolmbx); PxMsgIdIsValid(msg); )
    {
        PxMsgInstallRelmbx(msg, PxMbxIdInvalidate());
        PxMsgRelease(msg);
    }

    // Reset mode bits
    PxClearModebits(~oldmode);

    // Delete Mailbox
    return PxMbxRelease(poolmbx);
}
```

A.4. Dynamical creation of a C++ task

To create a C++ task dynamically a class describing the task object must be defined.

```
extern "C" {
void Task1_Entry(PxTask_t myID, PxMbx_t myMailbox, PxEvents_t myActivationEvents);
}
/*
The Class of Task1
This class defines all data and functions of the task Task1
*/
class Task1
{
    PxStackAligned_t TaskStack[(TASK_STACKSIZE / (sizeof(PxStackAligned_t) / sizeof(PxInt_t)))];
    // add 6 dummy bytes to avoid protection disruption by using "st.d" at the end of the data
area
    // normaly an access to this pad bytes will lead to an protection fault
    // this 6 byte have to be always the last elements in the data area of a task
    // don't at any members behind this protection pad
    PxChar_t    __protectionPad[PXALLOC_SECURITY_PAD] PXMEM_ALIGNED;
public:
    PxTask_t TaskCreate(PxPrio_t prio, PxEvents_t actev);
    friend PxTask_t CreateTask1(PxPrio_t prio, PxEvents_t actevents);
    friend void Task1_Entry(PxTask_t myID, PxMbx_t myMailbox, PxEvents_t myActivationEvents);
} __attribute__ ((aligned(64)));
```

The task has to provide an API function to create it:

```
PxTask_t TaskCreate(PxPrio_t prio, PxEvents_t events)
{
    return Task1Obj.TaskCreate(prio, events);
}
```

The task creation is almost the same as the static task creation. The only difference is that the second entry in the task context must cover the task's object:

```
PxTask_t TaskCreate (PxPrio_t prio, PxEvents_t events, PxMc_t memClass, PxOpool_t objPool)
{
    PxTaskSpec_T    ts;
    PxTaskContext_T context;

    memset((PxUChar_t *)&ts, 0, sizeof(ts));

    context.protection[0].lowerBound    = 0;
    context.protection[0].upperBound    = 0;
    context.protection[0].prot          = NoAccessProtection,
    context.protection[1].lowerBound    = (unsigned int)this;
    context.protection[1].upperBound    = ((unsigned int)this) + sizeof(Task1);
    context.protection[1].prot          = WRProtection;

    ts.ts_name                          = (const PxChar_t *)"Task1";
    ts.ts_fun                           = TaskEntry;
    ts.ts_mc                            = memClass;
    ts.ts_opool                         = objPool;
    ts.ts_privileges                    = PXUser1Privilege;
    ts.ts_accessrights                  = THISTASK_PXACCESS;
    ts.ts_context                       = &context;
    ts.ts_protect_region                = 0;

    /* Specify the task's stack. */
    ts.ts_taskstack.stk_type            = PXStackFall;
    ts.ts_taskstack.stk_size            = TASK_STACKSIZE;
    ts.ts_taskstack.stk_src.stk         = &TaskStack[(TASK_STACKSIZE / (sizeof(PxStackAligned_t)
/ sizeof(PxInt_t)))];

    return PxTaskCreate (PXOpoolTaskdefault, &ts, prio, events);
}
```

The creating task must allocate the memory needed for the task object. The easiest method is to request memory from default memory class. The pointer to the allocated memory block is stored in a task variable to be able to release the memory when the task is removed. Then the API function to create the task is called:

```
PxMemAligned_t *taskArea;
PxTask_t CreateTask1(PxPrio_t prio, PxEvents_t actEv)
{
    Task1 *Task1Ptr;
    PxTask_t taskId = PxTaskIdInvalidate();

    taskArea = (PxMemAligned_t *)PxMcTakeBlk(PXMcTaskdefault,sizeof(Task1));
    if (taskArea == 0)
    {
        PxTaskIdSetError(taskId, PXERR_MC_NOMEM;
        return taskId;
    }
    memset ((void *)taskArea, 0, sizeof(Task1));
    Task1Ptr = (Task1 *)taskArea;
    taskId = Task1Ptr->TaskCreate (prio, actEv, PXMcTaskdefault, PXOpoolTaskdefault);
    if (PxTaskIdError(taskId) != PXERR_NOERROR)
    {
        PxMcReturnBlk(PXMcTaskdefault, taskArea);
    }
    return taskId;
}
```

A.5. Remove a dynamically created task

There are only 2 steps necessary to remove a dynamically created task:

1. Terminate the task
2. Release the memory used by the task

```
PxError_t err;
PxTask_t taskId;

    err = PxTaskForceTermination(taskId);
    if (err == PXERR_NOERROR)
    {
        taskId = PxTaskIdInvalidate();
        taskArea = PxMcReturnBlk(PXMcTaskdefault, taskArea);
    }
```

Disclaimer

HIGHTEC

HighTec EDV-Systeme GmbH
Europaallee 19, D-66113 Saarbrücken
info@hightec-rt.com
+49-681-92613-16
www.hightec-rt.com