Knock Out Operating System
Programmers reference manual
Version 1.2

Contents

1 Introduction

1.1 What is a real-time kernel

The software industry has always been looking for ways to make software products correct, maintainable and reusable. One way that leads into that direction is to modularize the software.
By dividing the software into comprehensible modules the programmer can better understand the software, and can improve correctness, maintainability and reusability. Turning these modules into independent processes can further improve the software quality. All resources in such a system (like memory, process time, interrupts, peripherals devices etc.) are managed by a piece of software called the operating system.
An operating system that doesn't support all kind of fancy things but does support all needed basic functionality and is compact in design is often called a kernel.
Operating systems that service programs for human beings divide their processor time over the processes in such a way that all users get their "equal share" of processor power. This process scheduling is often called time slicing.
Processes that control hardware, most often need an other scheduling scheme where a maximum response time can be guaranteed. These real-time applications need priority based scheduling. This is what a real-time kernel offers.

1.2 Real-time kernel definitions

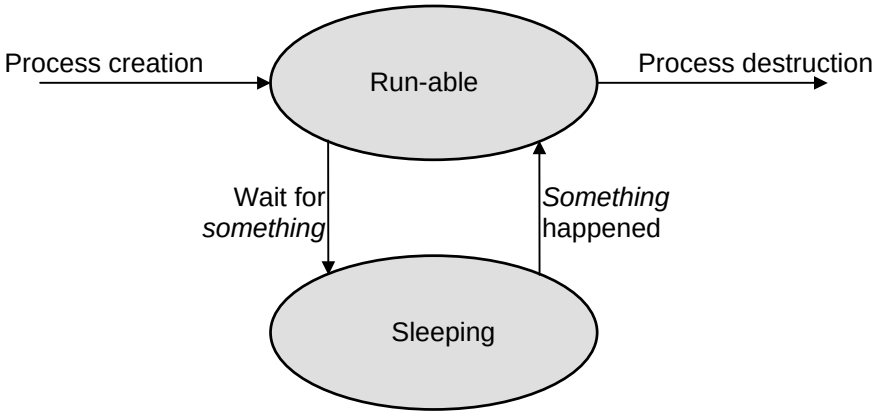
This section introduces some basic concepts about operating system technology. To have a good understanding of definitions used later in the document, it is a good idea to become familiar with these concepts before moving on to the rest of the document.

A **process** is a fundamental unit of computation that can request services from the operating system. Processes control specific functions within the application. In a **multiprocessing** environment, several processes run concurrently. In many cases, the applications will require response to physical events as they happen. Those programs that demand attention within a predictable maximum time and full access to computing resources are called **real-time applications**.

The CPU executes all instructions. However, the CPU can execute the code of only one process at a time. A multiprocessing operating system may have several processes in various states of execution at the same time. This is possible because process operation invariably involves more than CPU usage. For example, one process might wait for another process or external event to happen before gaining access to the CPU. And while it is waiting, the process does not need the CPU, and another process may be executed.

A multiprocessing operating system must know when a process has halted so that it can allow another process access to the CPU. Normally a process halts itself because it starts waiting for e.g. an external event or a timer. So the application itself notifies the operating system to put the process to sleep. A mechanism called **"interrupts**" is used to wake up the process. The occurrence of a specific interrupt might lead to the awakening of the process.

Active processes compete for CPU time according to their assigned **priority** and the availability of the processor. The priority of a process is determined by a number assigned to it. The process with the highest priority that has all the resources it needs to run (is not waiting for anything) gains control of the CPU. When that process becomes blocked (sleeping = waiting for something) the operating system chooses the run-able process with the next highest priority to use the CPU. So a process can either be run-able or sleeping, and the process with the best priority is running. This results in the following sate diagram.



So far, we have concentrated primarily on two resources available to the operating system : the processor and the interrupt. A third kind of resource is **memory**. Apart from statically allocating memory, a process can dynamically allocate and free memory for its operation. The operating system is responsible for keeping track of which parts of memory are in use and by whom.

Yet a fourth type of resource is the **Inter-process Communication (IPC) or messaging.** IPC allows a process to send packets of information to other processes. The operating system will manage the queuing of messages and synchronization of the processes.

The fifth type of resource is the **event**. An event is a coordinated synchronization between two or more processes in the system. One (or more) process can wait for an event to happen while others signal such an event (or such events). The number of signals or waits is counted. This resource is in many textbooks referred to as semaphores (or in this case counting semaphores).

Finally, the last type of resource is the **timer**. A timer is a mechanism that provides a regular “wake-up service” for a process. A process is allowed to suspend its operation for a predetermined period of time by going to sleep. The timer mechanism will awake the process after a specified time.

Resuming:
The operating system is performing all its functions on **resources.** The following resource types are known within the Knock Out Operating System (and many others) :

⇒ Process : A process is one single unit of computation within your system (it can run software). CPU time is allocated to it depending on its priority.

⇒ Memory : The operating system is responsible for keeping track of which parts of memory are in use. It is able to give a part of memory to a process for usage and eventually to get it back.

⇒ Interrupt : Is a synchronization mechanism used to respond to external events.

⇒ Messaging : This allows a process to send packets of information to other processes.

⇒ Event: A software event is a coordinated synchronization between two or more processes in the system.

⇒ Timer: A software timer is an interrupt-like resource. It allows processes to synchronize on an internal clock and suspend its own execution for a predetermined period of time.

1.3 What is KoOs ?

KoOs is a real time, multiprocessing operating system written explicitly for the 80C51XA processor but it can be ported to other architectures too. It supports functions which manage processes, handle interrupts, implement timers, control software events, allocate memory, coordinates sharing of memory and performs message passing. It is has a very low overhead, is extremely compact in size and has amazingly short response times to interrupts. It uses most features offered by the 80C51XA to achieve these goals in an effective way. Read on to learn all about it.

1.4 What KoOs does

The following text provides an overview of the functions KoOs can perform for you.

- Dynamic process creation and destruction. This means that processes are never “hanging around” using valuable memory space. Instead, the resource is created if and when you need it.
- Management of a watchdog timer on a per process basis. It’s up to the process to use it.
- Dynamic allocation and de-allocation and management of memory in variable sized memory blocks.
- Coordination when memory is shared among processes with time-out support.
- Dynamic management of software events with time-out support.
- Dynamic control of interrupts with user definable interrupt-functions and time-out support.
- Interprocess communication (messaging) with time-out support
- Dynamic allocation of timers with timer interrupt routines, flexible timer reload management and, as strange as it sounds without the details, with time-out support.
- Smart management of interrupt stacks. The process doesn’t need to know about it, neither has to take account for it on its own stack.
- Easy configuration of the operating system. Almost everything is automatic.
- Very low memory usage. KoOs is very compact. On top of that is written as many small modules and is put into a library. Only those parts that are actually needed are linked into your application.

KoOs imposes no limitations on the number of resources other than the limitations of the system itself (processing power and memory size).

1.5 KoOs internals

The following chapters provide you with a detailed explanation on the environment and the use of KoOs. If you have to modify or extend the internal structure of KoOs, you should also use the “Knock Out Operating System Design specification”. That document provides detailed information that is normally not needed to just use KoOs.

2 Using KoOs

In this chapter we will focus on how the functionality of KoOs can be used. For the sake of clarity we will skip some details. The function call specification chapter will cover them all.

We first have to introduce the usage of handles and error codes. After that we will explain the function calls that let you use the resources of KoOs. Finally we will discuss some more general issues of KoOs.

2.1 Error Codes

All functions in KoOs return a value of some kind. If this value is equal to zero (or false if you like) then the function encountered an error. You can then use the function KoOsGetError to get an error code that explains the nature of the error. Just to get a feeling for what you can expect, you will find here a list of symbolic error names taken from the headerfile KoOs.h :

KOOS_ERROR_HANDLE
KOOS_ERROR_NO_MORE_MEMORY
KOOS_ERROR_INVALID_MEMORY
KOOS_ERROR_TYPE
KOOS_ERROR_ILLEGAL_SIGNALS
KOOS_ERROR_INTERRUPT_ID
KOOS_ERROR_TIMER_FIRST
KOOS_ERROR_TIMER_RUNNING
KOOS_ERROR_TIMER_NOT_RUNNING
KOOS_ERROR_MEMORY_SIZE
KOOS_ERROR_DESTROY
KOOS_ERROR_TIMEOUT
KOOS_ERROR_NAME_TOO_LONG
KOOS_ERROR_NO_NAME
KOOS_ERROR_NOT_FOUND

Refer to the specification of the individual functions as to when to expect an error code and what they mean.

2.2 Handles

You will see that whenever you ask KoOs to create for you a resource, it will return you a handle. A handle is a (non-zero) number that has no specific meaning to you but for KoOs it uniquely specifies a resource. In all subsequent calls to KoOs you use this handle to specify on what resource the action is performed.

2.3 Processes

If you create a process, you have to specify the environment of the process and what code it should execute.

During the creation of the process, some memory will be reserved for use as a process stack. The size, which you have to specify, should account for all automatic variables and function calls that the process uses plus a small amount of overhead (some 20 bytes).

A process has a process priority (not to be confused with the processor priority) which is used in scheduling activities. This priority is a number in the range 0..255. The higher the number, the better the priority. See chapter 2.15 - Scheduling, for more details.

The code that a process executes is just a function that you specify in the process creation request. When the end of that function is reached, the process will end and return all its resources to KoOs. You can also end a process prematurely with the same mechanism by calling the function KoOsProcessExit. The death of a process is discussed in more detail in chapter 2.11 - The death of a process.

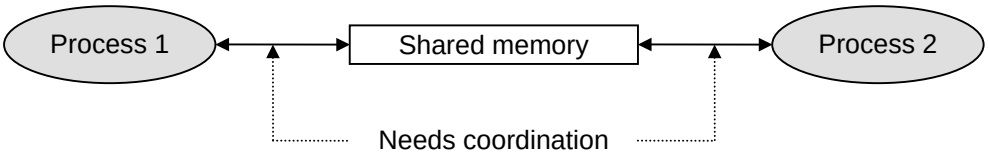
When you create a process, you have to specify one parameter (a word) that is given to the process (function). This could be used e.g. to let multiple processes share the same code. In this case the parameter could configure the code (e.g. 10 lamp drivers, each running the same code, but the parameter specifies what lamp should be controlled).

2.4 Memory

A process can request for variable sized blocks of memory that are managed by KoOs. The request is done with the call KoOsMemoryCreate with the size as a parameter. As soon as the process has no need anymore for the block it can return it to KoOs with the call KoOsMemoryDestroy.
As with all resources, KoOs will give you a handle. With the functions KoOsMemoryAddress and KoOsMemorySize you can obtain the begin-address and the size of the memory block.

2.5 Regions

When you want to share memory amongst processes you need some coordination on that usage. Suppose one process is filling a block of memory with new data. During this time the operating system decides that an other process (with an higher priority) needs to run. If this second process is going to use the block of memory, it will find partly old and partly new data in it. To overcome this problem you can use a region instead of memory.



Comparable with the memory resource you can create a region through the call KoOsRegionCreate with the size as a parameter and, if you no longer need it, destroy it with the KoOsRegionDestroy call. You can also obtain the size with the KoOsRegionSize call.

As soon as you are going to use the region you have to use the call KoOsRegionLock. This call will return you the address of the memory and give you exclusive access to the memory region. If, during this time, other processes try to lock this region, they will be put in the sleep state (= they will wait) until the region is available.

With the call KoOsRegionUnlock you signal that you don't need the memory region anymore (at least for the moment) so an other process can lock it. Multiple processes can request a lock on the region. These requests will be queued on a first in first out basis.

A more advanced usage of the region allows the actual address of the memory region to change before your lock or after your unlock call. That's why it is important to always use the newly obtained address from every KoOsRegionLock call.

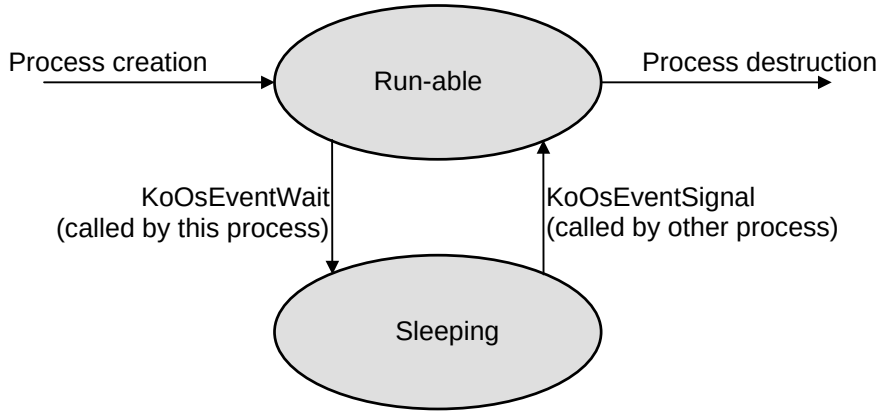
A region is actually a combination of a memory block with a semaphore to coordinate the usage of that memory block. The semaphore, or event as it is called in KoOs, is the next resource to discuss.

2.6 Events

An event is a mechanism that is used to synchronize two or more processes and as such it is a resource that is shared with other processes (see more on resource sharing in 2.10 - Resource hierarchy and ownership). The event is created with the KoOsEventCreate call and destroyed with the KoOsEventDestroy call.

A process can wait for an event to ‘happen’ with the call KoOsEventWait. This will put the process to sleep (the waiting state) unless the event already happened, in which case the call immediately returns.

A process can make an event ‘happen’ by using the call KoOsEventSignal.



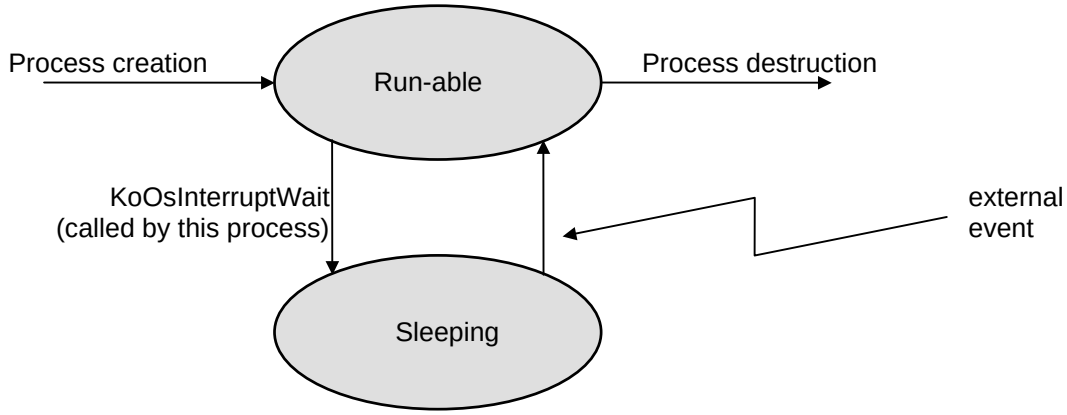
The number of signals and waits is counted, so that extra signals and waits don’t get lost. The waiting list of an event resource (and all other resources that wait) has a first-in-first-out order. This behavior is in many textbooks described as a counting semaphore.

2.7 Interrupts

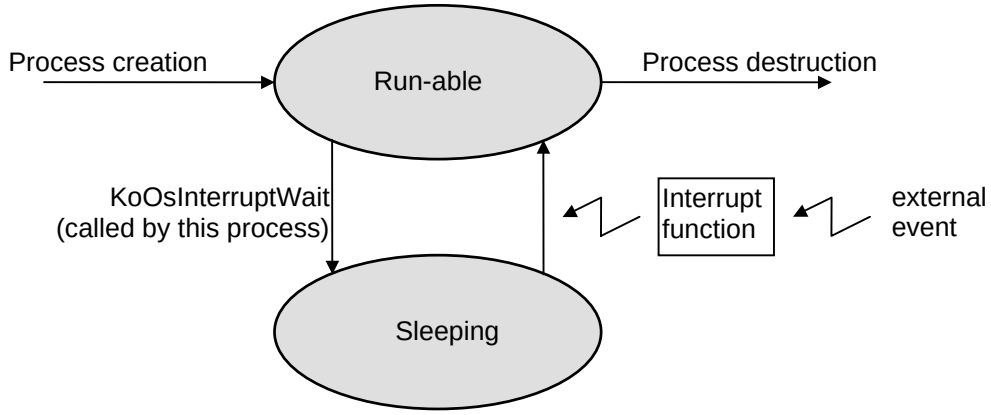
Interrupts are used to synchronize processes to external events. It behaves partly like an event. You can create it with the call KoOsInterruptCreate and destroy it with the call KoOsInterruptDestroy.

A process can wait for an interrupt to ‘happen’ with the call KoOsInterruptWait. This will put the process to sleep (the waiting state) unless the interrupts already happened, in which case the call immediately returns.

The difference with events comes from the signal side. In its simplest form, the interrupt resource is signaled by the external event. As with the event resource, multiple interrupts are recorded.



Often one has to respond very quickly on the external event. If at that time one or more other processes with higher priority are running, the response has to wait till these processes block. This may result in too long response times (too high interrupt latency).
To overcome this problem you can specify in the KoOsInterruptCreate call an interrupt function. This function is executed as a result of the external event without waiting for the processes with any priority. However if all interrupt processing is done in these interrupt functions of the various interrupt events, we just moved the response time problem from the processes to the interrupt functions. We can however, use a combination of the interrupt function and the waiting process. The interrupt function does the response time critical work and the process does the time consuming work.
The interrupt function can even contain a decision as to signal the waiting process or not.



As an example think of a serial port that receives characters from a terminal or keyboard. The serial port receive event can trigger an interrupt routine that stores the received characters in a buffer avoiding an overrun of received characters.
Only when a complete line is received, the waiting process is triggered. The process can then interpret the line and execute the commands that are on it.

The prototype for an interrupt function is as follows :

```
unsigned InterruptFunction( int SignalCount );
```

The return value of the function signifies if the waiting process has to be signaled or not. The parameter that the function receives contains the current signal count of the interrupt. A positive value means that there where more signals than waits, so nobody is waiting for it and the value indicates that that many KoOsInterruptWait calls will return immediately without waiting. A negative value means that there where more waits then signals, so there are that many processes waiting on the interrupt. A value of zero means that the number of signals and waits where equal, so no process is waiting.
This SignalCount can maybe be used to make a more intelligent decision whether to wake up the process or not.

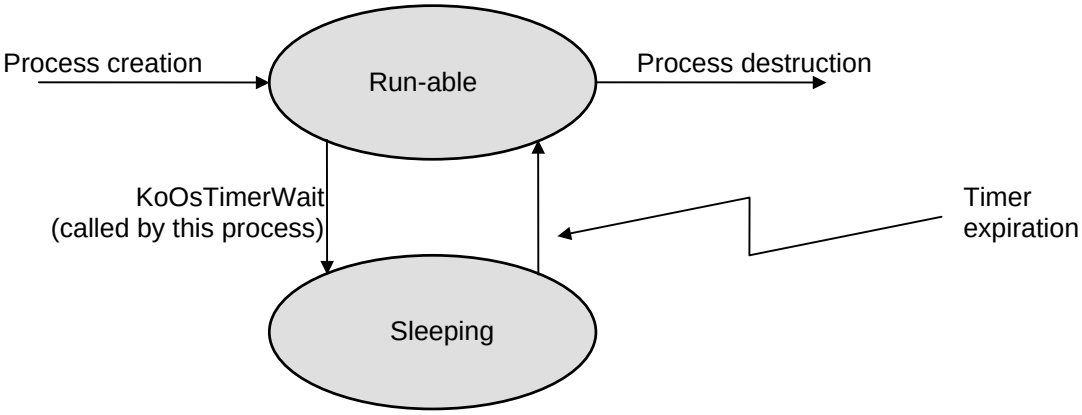
2.8 Timers

Timer resources behave very similar to interrupt resources. The triggering event is not something external, but the expiration of a timer. The extras have to do with the control of the timer.

You can wait for it with the function KoOsTimerWait, create it with the function KoOsTimerCreate and destroy it with the function KoOsTimerDestroy. The KoOsTimerCreate also offers you the possibility to specify an ‘interrupt function’ that is now called a ‘timer expiration function’ or ‘timer function’ for short.

Once you have a timer, it is not yet running. You do this with the KoOsTimerStart function. You can always stop it with the KoOsTimerAbort function. As the name suggests this is a fairly rude way and there is another, more gentle option.

With the KoOsTimerStart function you specify a timer expiration interval in clock ticks. The duration of a clock tick in KoOs is configurable, but it is usually in the range between 1 and 100 mS with a typical value of something like 10 mS. When you start the timer, you could just have missed the clock tick or it might almost going to happen. This means that you have an uncertainty of one clock tick for the first expiration of the timer. The KoOsTimerStart function offers you the possibility to specify the first and the successive expiration intervals as two separate values (called the first and reload values).

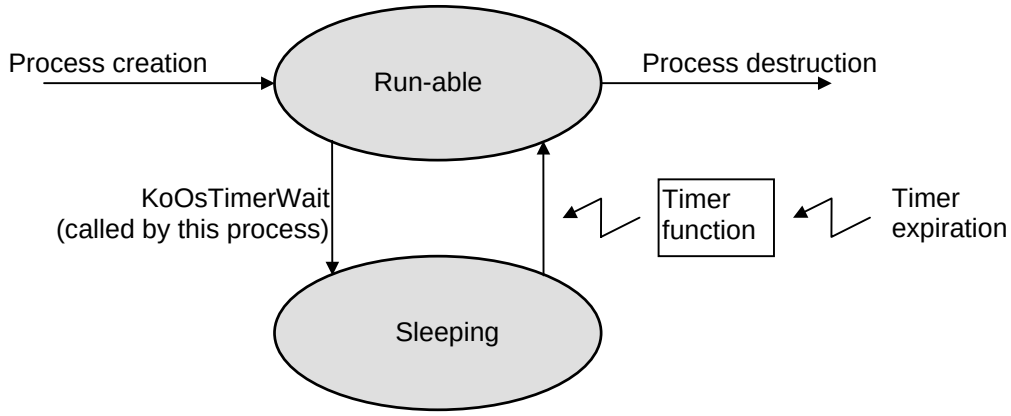


Once the timer is started, the timer will expire (apart from the first time) at a regular interval that you specified as reload value. If you specify the reload value as 0, then the timer will not be reloaded after the first expiration and thus stops. This is the one-shot mode. But there is more : you can influence the timer from the timer function.

The timer function has the following prototype :

```
unsigned TimerFunction( int SignalCount, unsigned *Reload );
```

The return value and the SignalCount parameter are used exactly the same as with the interrupt function.



The timer function is called when the timer expires, just before it is reloaded. The timer function receives a pointer to that reload variable and thus can change it to any value it likes. This way you can make enormously complicated timer expiration schemes. You can also change the reload value to 0 which will stop the time gracefully.

All these features make the timer resource very powerful yet simple to use.

In its simplest form you can create a timer without a timer function, start it (one-shot or continuous) and wait in the process for its expiration. It is as simple as that.
In a complex form, you create a timer with a timer function and start it. The timer function does some response time critical work (e.g. some data acquisition), tweaks the reload value and has some kind of intelligence that decides to wake the waiting process or not, maybe also based on the current value of the SignalCount. That is complex functionality, managed in a fairly easy way.

2.9 Mailbox

The KoOs implements inter-process communication in the form of mailboxes. You can create a mailbox with the call KoOsMailboxCreate and destroy it wit the call KoOsMailboxDestroy.

A mailbox can hold multiple messages, but if you use the call KoOsMailboxReceive you get the oldest mail message from the mailbox (there is a first in first out mechanism for mail).
If, at the time you issue a KoOsMailboxReceive, there is no mail in the mailbox, you will wait for it (your task is put to sleep).

You can put messages in a mailbox with the call KoOsMailboxSend. If there is no process waiting for it, the message will be queued. If there was a process waiting, it will wake up and receive the message (as soon as it has the right priority to run).

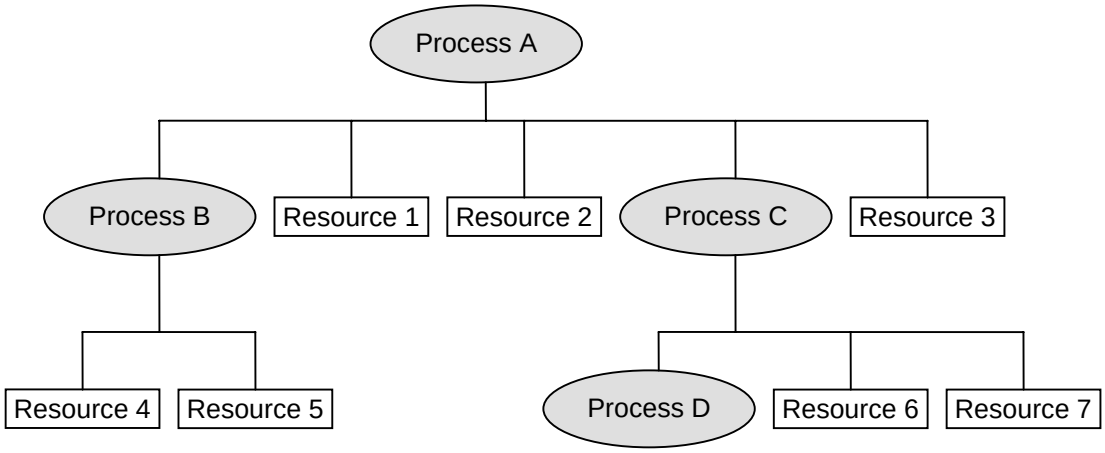
Multiple processes can wait for a message from a mailbox. They will be put in a queue with a first in first out characteristic.

The message is actually a memory resource that can contain anything you like and can have any size you like, as long as the sender and the receiver agree with each other. Along with the message you can send a handle. This handle can contain anything as long as the sender and the receiver agree. You can e.g. send a handle of a mailbox to send a reply to, or a handle of an event to acknowledge the reception of the message.

As the sender, you have to be the owner of the memory resource (see more about ownership in chapter 2.10 - Resource hierarchy and ownership). If you send the message (memory resource), you transfer the ownership of that message to the receiver.

2.10 Resource hierarchy and ownership

If a process creates a resource of any kind, it becomes the owner. Since processes are also resources, you can create a complete hierarchy. The following chart is an example of such a hierarchy.



Process A created resources 1, 2, 3 and processes B and C. It owns these resources and processes B and C are its “child processes”.
Process B created resources 4 and 5.
Process C created resources 6, 7 and process D. It owns these resources and processes D is its “child process”.

In order to destroy a resource you have to be the owner. The only exception to that rule is the process. Although a process is created (and owned) by its parent, it can only destroy (end) itself (see chapter 2.11 - The death of a process).

A process can use all the resources that it owns, but also the resources of its parent and its grand-parent …….and so on. In the given example process D can use all its own resources (none in this case), the resources of its parent (resources 6 and 7) and the resources of its grant-parent (resources 1, 2 and 3), but it can NOT use the resources owned by its uncles (resources 4 and 5).
Strictly speaking, Process D can also use process D, C and B because processes are also resources. However, since the only thing you can do with a process is creating it, this usage is impractical.

Because of this hierarchy you can divide the system in “main processes”. These “main processes” can divide their work over a number of “sub-processes”. Within such a process group you can share resources, but they are hidden for other process groups. On the top level you can have resources that are shared by everyone. This hierarchy helps a lot in modularizing your system.

The parent of all processes (in our example Process A) is part of KoOs. It is the system process which has the following unique features :

- It is always there,
- It is always run-able,
- It has the lowest priority in the system (the reserved value of zero),
- It doesn’t do anything apart from measuring the systems “idle time” or switching the processor to the idle mode to conserve power,
- It can own resources that are shared by the whole system.

At system start-up, the System process is the only process running. It creates resources that are shared by the whole system and creates some other processes. These other processes do the actual work (they might defer it to their child processes). The system process will, after this start-up code, consume all processor time that is not needed by all other processes. This time is referred to as the idle time of the system.

2.11 The death of a process

If a process ends (by either reaching the end of the code or calling KoOsProcessExit) it automatically destroys all the resources it owns and finally destroys itself. It cannot always do this immediately because :

- It cannot destroy any child processes,
- It should not destroy any resources that are potentially used by its child processes.

For this reason it will :

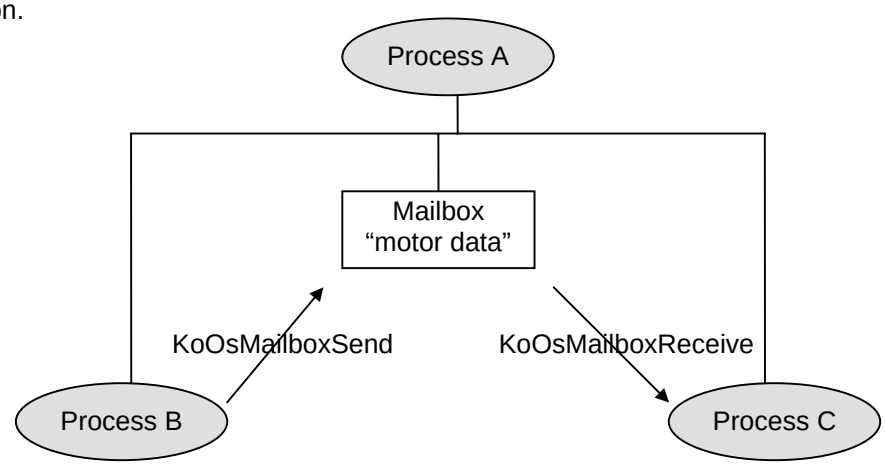
- change its priority to very low (a value of 1),
- wait until all child processes are destroyed.

Then it will wake up and, as a low priority process, it will clean-up and vanish. This might take a fair amount of CPU time, but because of the low priority it will not interfere with the real-time behavior of the rest of the system (it steals idle time).

2.12 Resource naming

If a process is going to use a resource that it didn't create itself, it doesn't know the handle of that resource. The creator of that resource can of course put the handle in a global variable, but there is also an other way that preserves the modularity of the system.

When a resource of any kind is created, you can (as a parameter of the create call) give it a name. Another process can then use the name of the resource to find its handle with the call KoOsResourceFind. This call will first check the resources of the process itself, then the resources of the parent, then the resources of the grand-parent …… and so on. Since this call is expensive you only use it once to get the handle and use the handle from that moment on.



The name of the resource is a normal "C-string" with a maximum string length of 255 characters (including the terminating zero) and can contain anything you like. The resource name also doesn't have to be unique. This can be used to your advantage to overrule resources that are higher in the resource hierarchy for a group of sub processes.

Be aware that long descriptive names might be a very good programming practice, but they also cost memory so you have to find the right trade off.

2.13 Time-out support on blocking calls

All blocking calls (calls that put the process in a sleeping state) can due to failing external stimuli or even errors in the application software wait forever. To detect this phenomena KoOs supports an optional time-out mechanism. With each blocking call you can specify a maximum waiting time. If this maximum is reached the waiting process is made run-able, but it returns from the waiting call with an error indication (see chapter 2.1 - Error Codes).

The time-out time is given in a number of time-out clock ticks. The time-out clock tick is derived from the timer clock tick by means of a prescaler value which is configurable in KoOs. Typical values for the time-out clock tick interval are 100ms or 1 s, but can go as low as 1 ms in special cases.

2.14 The process Watchdog timer

A process can due to a program error, hang up in some kind of an endless loop. Since other processes are still (more or less) normally running this defect could go undetected for a hardware watchdog like the one that is available on the 80C51XA-G3.

To overcome this problem each process can have its own software watchdog. A process has to "kick" its software watchdog on a regular basis by giving it a time-out value as a number of time-out ticks. If the process fails to "kick" the watchdog within that given time-out time, a KoOsTaskWatchdogFailure interrupt will be issued. This interrupt can be handled as any

other interrupt. So some kind of “supervising process” can be created that handles system failures.

The watchdog counter continues to run even if the process is waiting in a blocking call. The blocking calls also have their own time-out mechanism, so you can check on that too (see chapter 2.13 - Time-out support on blocking calls).

2.15 Scheduling

The scheduler of KoOs determines which run-able process is running. Waiting (sleeping) processes are not considered. The scheduler is always making the run-able process with the best priority running. If the running process changes, we speak about a task switch. Processes with equal priority will be handled on a ‘first come first served’ basis considering the moment that their state changed to run-able. A change in who has the best priority only occurs in the following situations :

1. The currently running process starts waiting for something. This is always due to a request (call) to KoOs. These types of calls are called blocking calls.
2. The currently running process ends (see chapter 2.11 - The death of a process).
3. The currently running process does a call to KoOs that wakes a process with a better (not worse and not the same) priority. These types of calls are called waking calls.
4. An interrupt or timer function wakes a task with a better (not worse and not the same) priority than the priority of the currently running task.

A task switch due to 1, 2 or 3 is always predictable. A task switch due to 4 can happen at any time. This real-time behavior (real world behavior ?) can also be described as :

The work with the highest priority always gets done first but there might be dependencies.

You can adopt the software behavior to the real world behavior by :

A. carefully assigning the priorities to processes and,
B. implement process dependencies through synchronizing resources (event, region, mailbox) and,
C. respond timely to external events (interrupts and timers) and,
D. implement (of course) the right algorithm.

KoOs helps you by providing the right tools for A, B and C. There are sometimes cases where you can take a shortcut and don’t need all those tools. Suppose two (or even more processes share a block of memory. Your first idea could be that you need a region instead of an ordinary memory resource since this sharing needs coordination. This could be wrong because the processes never use the memory concurrently and so you don’t need the coordination.

Another interesting situation occurs if all the processes that share the memory have the same priority. In this case they will be pre-empted by processes with a higher priority, but they will never be pre-empted by each other. So as far as this memory is concerned a process will have exclusive access to the memory as long as it doesn’t do a blocking call.

We already stated that the priority is a number in the range 0..255. The higher the number, the better the priority. You specify this value when you create the process but you cannot use all possible values. The value of 0 is reserved for the system process, and 1 is reserved for dying processes. The value of 255 is reserved for future enhancements (possibly a system debugger / monitor ?).

3 Function reference

This chapter describes all available function calls to kernel functionality in alphabetical order.

The specification starts of with a prototype of the function as taken from file KoOs.h.

The description under the heading Parameters briefly explains the parameters of the function and possible limitations to their value.

The description under the heading Return value explains the meaning of the return value of the function. Most of the time a return value of zero means there was an error.

Under Description you find an explanation of the offered functionality

If the function reported an error (return value equals zero) you can use the function KoOsGetError to get an error code indicating the type of error. Under the heading Errors you can find the possible errorcodes associated with this function and an explanation what they mean in this specific situation.

Under the heading Function call type the function is classified under one of the following categories :

category	meaning
signaling	due to this function call an other sleeping (waiting) process might become runable.
waiting	due to this function call, the current process might become sleeping (waiting). This is sometimes called a blocking function call.
creating	This function creates a new resource.
destroying	This function destroys (removes, returns to the kernel) a resource.
other	All other functions.

The Cycle count is calculated as the number of cycles needed to execute all instructions and the possible call overhead in processor clock cycles. The clock cycles for the individual instructions are taken from the “16 bit 80C51XA Microcontrollers Data Handbook IC25”. This method does not take the influence of slow memories and/or 8/16 bit busses into account, but is independent of the used processor clock frequency.

3.1 KoOsEventCreate

```
KoOsHandle   KoOsEventCreate(   char    *Name,
                                int     InitialSignals  )
```

Parameters

Name	Zero terminated string to name the event, or NULL for no name.
InitialSignals	Initial number of signals given to the event. Must be positive or 0.

Return value

The handle of the just created event, or 0 in case of an error.

Description

This function will create an event resource and, if successful, will return the handle of that event resource. You can initially signal the event (see KoOsEventSignal) any number of times. A signal value of 0 indicates an initially not signaled event resource.
The event resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_ILLEGAL_SIGNALS	InitialSignals is not allowed to be negative
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.2 KoOsEventDestroy

unsigned KoOsEventDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the event resource

Return value

0 in case of an error.

Description

This function will destroy (remove) the event resource indicated by the Handle. All processes that are waiting for this event are released with an error (KOOS_ERROR_DESTROY). You have to be the owner of the event resource to be able to destroy it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an event
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d.

3.3 KoOsEventSignal

unsigned KoOsEventSignal(KoOsHandle Handle)

Parameters

Handle Handle of the event resource

Return value

0 in case of an error.

Description

This function will signal an event resource. If there was a process (or processes) waiting for this event, it will be made run-able. If there was no waiting process (or processes) the signal is remembered for later use.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an event

Function call type

☑ signaling ☐ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.4 KoOsEventWait

```
unsigned    KoOsEventWait(    KoOsHandle    Handle
                              unsigned      Timeout    )
```

Parameters

Handle	Handle of the event resource
Timeout	Number of system time-out timer ticks that the function will wait before it will abort. 0 for no time-out support.

Return value

0 in case of an error.

Description

This function will wait for the event resource to be signaled. If there where signals without waiting processes, then these signals are remembered, and this function call will be completed immediately without waiting.
The time-out support is a fail-safe mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other than zero is given. If, in this case, the time-out time has elapsed, this function is completed with an error.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not valid user of resource
KOOS_ERROR_TYPE	Resource is not an event
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_DESTROY	While waiting, the event was destroyed by the owner

Function call type

☐ signaling ☑ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d. (… + waiting time)

3.5 KoOsGetError

unsigned char KoOsGetError(void)

Parameters

none.

Return value

Error code.

Description

If a kernel function indicates that there was an error, which is in general indicated by a return value of zero, then you can use this function to get an error code to indicate the nature of the error. If the KoOsGetError function is used after a kernel function that didn't report an error, then the return value of KoOsGetError is invalid.

Errors

Possible errorcodes	Reason
KOOS_ERROR_HANDLE	Illegal handle or not valid user of resource
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to complete function
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted
KOOS_ERROR_TYPE	Resource is of the wrong type
KOOS_ERROR_ILLEGAL_SIGNALS	Number of Signals is not allowed to be negative
KOOS_ERROR_INTERRUPT_ID	Interrupt already in use or illegal ID
KOOS_ERROR_TIMER_FIRST	Invalid timer start value
KOOS_ERROR_TIMER_RUNNING	Invalid because timer is running
KOOS_ERROR_TIMER_NOT_RUNNING	Invalid because timer is not running
KOOS_ERROR_MEMORY_SIZE	Invalid memory size
KOOS_ERROR_DESTROY	While waiting, the resource was destroyed by the owner
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_NAME	No name given
KOOS_ERROR_NOT_FOUND	Resource not found

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

19 clock cycles

3.6 KoOsInterruptCreate

```
KoOsHandle    KoOsInterruptCreate(    char        *Name,
                                      unsigned    *InterruptID,
                                      unsigned    (*InterruptFunction)( int SignalCount ) )
```

Parameters

Name	Zero terminated string to name the interrupt, or NULL for no name.
InterruptID	Predefined interrupt ID. List of names is included in KoOs.H and varies with the processor type.
InterruptFunction	Interrupt function to be invoked when specified interrupt occurs, or NULL for no Interrupt function.

Return value

The handle of the just created interrupt resource, or 0 in case of an error.

Description

This function will create an interrupt resource and, if successful, will return the handle of that interrupt resource.
The interrupt resource binds the physical interrupt source as indicated by InterruptID with a function as given by InterruptFunction and a counting semaphore type of resource that a process (or processes) can use to wait for.
This function does NOT initialize the interrupt source (UART, PCA… etc.) and does NOT enable the interrupt.
Its use with the InterruptID's DOES however handle the creation of interrupt vectors. Interrupt vectors that are not used through their Interrupt ID's and are not reserved by the kernel are left uninitialized.
Note that the kernel startup code will set the interrupt priority register (IPAn or IPBn) to the value as given in the KoosConfig.src configuration file.
The interrupt resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.
Note that the InterruptFunction is a normal C function without any *_interrupt()* and *_using()* type modifiers.

Interrupt function

If the interrupt becomes active, the interrupt function is invoked. The return value of this interrupt function is evaluated to see if the interrupt semaphore has to be signaled (return value non-zero) or not signaled (return value zero).
The interrupt function can use the given parameter SignalCount to make a more intelligent decision about signaling or not signaling of the interrupt semaphore. The value of SignalCount has the following meaning :

SignalCount	meaning
> 0	There are "SignalCount" number of signals without waiting processes
0	There are no waiting processes and no access signals
< 0	There are "-SignalCount" waiting processes

In case there is no interrupt function, the interrupt semaphore is signaled as if there where a function that always returned a non-zero value.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_INTERRUPT_ID	Interrupt in use or illegal ID
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.7 KoOsInterruptDestroy

unsigned KoOsInterruptDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the interrupt resource

Return value

0 in case of an error.

Description

This function will destroy (remove, return to the kernel) the interrupt resource indicated by the Handle. All processes that are waiting for this interrupt are released with an error (KOOS_ERROR_DESTROY). You have to be the owner of the interrupt resource to be able to destroy it.
This function does NOT de-initialize the interrupt source (UART, PCA… etc.) and does NOT disable the interrupt.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an event
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d.

3.8 KoOsInterruptWait

```
unsigned    KoOsInterruptWait(    KoOsHandle    Handle
                                  unsigned      Timeout    )
```

Parameters

Handle	Handle of the interrupt resource
Timeout	Number of system time-out timer ticks that the function will wait before it will abort. 0 for no time-out support.

Return value

0 in case of an error.

Description

This function will wait for the interrupt resource to be signaled by the associated interrupt function or directly by the interrupt source (if there is no interrupt function). If there where signals without waiting processes, then these signals are remembered, and this function call will be completed immediately without waiting.
The time-out support is a fail-safe mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other then zero is given. If, in this case, the time-out time has elapsed, this function is completed with an error.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not valid user of resource
KOOS_ERROR_TYPE	Resource is not an interrupt
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_DESTROY	While waiting, the interrupt was destroyed by the owner

Function call type

☐ signaling ☑ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d. (… + waiting time)

3.9 KoOsMailboxCreate

KoOsHandle KoOsMailboxCreate(char *Name)

Parameters

Name Zero terminated string to name the mailbox, or NULL for no name.

Return value

The handle of the just created mailbox resource, or 0 in case of an error.

Description

This function will create a mailbox resource and, if successful, will return the handle of that mailbox resource.
The mailbox resource combines a fifo for memory blocks (the messages) and a counting semaphore type of resource that a process (or processes) can use to wait for.
The mailbox resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.10 KoOsMailboxDestroy

unsigned KoOsMailboxDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the mailbox resource

Return value

0 in case of an error.

Description

This function will destroy (remove) the mailbox resource indicated by the Handle. All processes that are waiting for this mailbox are released with an error (KOOS_ERROR_DESTROY). All waiting Memory blocks in the queue (the to be received messages) are also destroyed (removed, returned to the kernel). You have to be the owner of the mailbox resource to be able to destroy it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an mailbox
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d.

3.11 KoOsMailboxReceive

```
KoOsHandle   KoOsMailboxReceive(  KoOsHandle   ToHandle,
                                  unsigned     TimeOut,
                                  KoOsHandle   *FromHandle   )
```

Parameters

ToHandle	Handle of the mailbox resource
Timeout	Number of system time-out timer ticks that the function will wait before it will abort. 0 for no time-out support.
FromHandle	Handle of the response mailbox. Null trashes this information.

Return value

Handle of the received memory block (the message). 0 in case of an error.

Description

This function will wait for the mailbox resource to contain a message. If there where message(s) without waiting process(es), then these message(s) are queued, and this function call will be completed immediately without waiting.
The time-out support is a fail-safe mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other then zero is given. If, in this case, the time-out time has elapsed, this function is completed with an error.
The value stored in location *FromHandle is the value given in the KoOsMailboxSend function and is passed on by the kernel without any checking. This feature is intended to send a “return address” along with the message, but the application is free to use this feature any way it likes.
Note that the receiving process becomes the owner of the just received memory block. So sending messages by mailbox is actually transferring ownership of memory blocks between processes in a synchronized way.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal ToHandle or not valid user of the mailbox
KOOS_ERROR_TYPE	Resource is not a mailbox
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_DESTROY	While waiting, the mailbox was destroyed by the owner

Function call type

☐ signaling ☑ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d. (… + waiting time)

3.12 KoOsMailboxSend

```
unsigned    KoOsMailboxSend(    KoOsHandle    FromHandle,
                                KoOsHandle    ToHandle,
                                KoOsHandle    MemHandle    )
```

Parameters

FromHandle	Handle used as “return address”
ToHandle	Handle of the mailbox to send the message to
MemHandle	Handle of a memory block to send (the message)

Return value

0 in case of an error.

Description

This function will signal a mailbox resource by sending a memory block to it (the message). If there was a process (or processes) waiting for this mailbox, it will be made run-able. If there was no waiting process (or processes) the signal and the message are queued for later use. The value given by FromHandle is passed-on by the kernel to the receiving process (function KoOsMailboxReceive) without any checking. This feature is intended to send a “return address” along with the message, but the application is free to use this feature any way it likes.
The sending process gives away its ownership of the memory block to the receiving party. While the memory block is in the queue of the mailbox, nobody owns the memory block.
The difference with the KoOsMailboxSendUrgent function is that this function places the messages at the end of the message queue (first in, first out).

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of memory resource, Illegal handle or not valid user of the mailbox resource.
KOOS_ERROR_TYPE	ToHandle resource is not a mailbox resource, Memhandle resource is nor a memory resource.

Function call type

☑ signaling ☐ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.13 KoOsMailboxSendUrgent

```
unsigned    KoOsMailboxSendUrgent(    KoOsHandle    FromHandle,
                                      KoOsHandle    ToHandle,
                                      KoOsHandle    MemHandle    )
```

Parameters

FromHandle	Handle used as “return address”
ToHandle	Handle of the mailbox to send the message to
MemHandle	Handle of a memory block to send (the message)

Return value

0 in case of an error.

Description

This function will signal a mailbox resource by sending a memory block to it (the message). If there was a process (or processes) waiting for this mailbox, it will be made run-able. If there was no waiting process (or processes) the signal and the message are queued for later use. The value given by FromHandle is passed-on by the kernel to the receiving process (function KoOsMailboxReceive) without any checking. This feature is intended to send a “return address” along with the message, but the application is free to use this feature any way it likes.
The sending process gives away its ownership of the memory block to the receiving party. While the memory block is in the queue of the mailbox, nobody owns the memory block.
The difference with the KoOsMailboxSend function is that this function places the messages at the start of the message queue (last in, first out).

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of memory resource, Illegal handle or not valid user of the mailbox resource.
KOOS_ERROR_TYPE	ToHandle resource is not a mailbox resource, Memhandle resource is nor a memory resource.

Function call type

☑ signaling ☐ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.14 KoOsMemoryAddress

void *KoOsMemoryAddress(KoOsHandle Handle)

Parameters

Handle Handle of the memory resource

Return value

Address of the memory block, NULL in case of an error.

Description

This function will return the address of the memory block indicated by the Handle.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not a valid user of the resource
KOOS_ERROR_TYPE	Resource is not a memory resource

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.15 KoOsMemoryCreate

```
KoOsHandle   KoOsMemoryCreate(   char       *Name,
                                 unsigned   Size        )
```

Parameters

Name	Zero terminated string to name the memory block, or NULL for no name.
Size	Size of the requested memory block

Return value

The handle of the just created memory block, or 0 in case of an error.

Description

This function will create a memory resource of a given Size and, if successful, will return the handle of that memory resource.
The event resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_MEMORY_SIZE	Blocks of 0 bytes not allowed
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.16 KoOsMemoryDestroy

unsigned KoOsMemoryDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the memory resource

Return value

0 in case of an error.

Description

This function will destroy (remove, return to the kernel) the memory resource indicated by the Handle. You have to be the owner of the event resource to be able to destroy it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not a memory resource
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d.

3.17 KoOsMemorySize

unsigned KoOsMemorySize(KoOsHandle Handle)

Parameters

Handle Handle of the memory resource

Return value

Size of the memory block, 0 in case of an error.

Description

This function will return the Size in bytes of the memory block indicated by the Handle.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not a valid user of the resource
KOOS_ERROR_TYPE	Resource is not a memory resource

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.18 KoOsProcessCreate

```
KoOsHandle    KoOsProcessCreate(    char          *Name,
                                    void          (*Main)( unsigned ),
                                    unsigned      StackSize,
                                    unsigned      Priority,
                                    KoOsHandle    Parameter          )
```

Parameters

Name	Zero terminated string to name the new process, or NULL for no name
Main	Function that makes the code for the new process
StackSize	Size of the stack for this new process (minimum 32 bytes)
Priority	Priority of the new process (range 1…255)
Parameter	Parameter passed on to the Process code

Return value

The handle of the just created process, or 0 in case of an error.

Description

This function will create a new process with a given StackSize and Priority. It will start the new process giving it the supplied parameter. If successful, the function will return the handle of that new process.
The StackSize must be 32 byte or larger. If a smaller size is given, 32 bytes is used.
The Priority must be in the range 1…255. The higher the number the better (higher) the priority. If 0 is given (which is reserved for the idle system process), 1 is used. If larger then 255 is given, 255 is used.
The Parameter is passed on to the process without any checking. Its purpose is to differentiate multiple instances of the same code. It can be just a number or a handle to a memory resource containing more information.
The event resource can be named so that it can be found by sibling processes through that name (see KoOsResourceFind).

The Main process function

The Main process function should be a function in the form :

```
void    NewProcess(    KoOsHandle    parameter    )
{
        // code for the process
}
```

The process ends if you reach the end of the function (or if you call the function KoOsProcessExit). Then it will go through the following termination steps.

- Change its priority to 1 (= extremely low, but just above the idle system process),
- Wait till all its sibling processes (if any) are terminated,
- Destroy all owned resources,
- Destroy the process itself.

Note that during these steps a possible running watchdog timer continues to run and could result in a watchdog failure.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.19 KoOsProcessExit

void KoOsProcessExit(void)

Parameters

none

Return value

This function will never return.

Description

This function will end the current process by going through the following termination steps :

- Change its priority to 1 (= extremely low, but just above the idle system process),
- Wait till all its sibling processes (if any) terminated,
- Destroy all owned resources,
- Destroy the process itself.

Note that during these steps a possible running watchdog timer continues to run and could result in a watchdog failure.

Errors

none, since it cannot report them.

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d. (… + time to destroy resources + waiting time)

3.20 KoOsProcessWatchdog

void KoOsProcessWatchdog(unsigned Timeout)

Parameters

Timeout	Number of system time-out timer ticks for the processes watchdog counter. 0 for no time-out support.

Return value

none

Description

This function will reload the current processes watchdog counter. A value of 0 (the default for the watchdog counter when a process is created) will disable the watchdog counter for this process.
The time-out support is a fail-safe mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other then zero is given.
Note that the system time-out tick mechanism introduces a one tick uncertainty in the time-out time.

Watchdog failure

When a process watchdog failure is detected, a KoOsTaskWatchdogFailure interrupt is generated. You can use KoOsInterruptCreate and related support to catch that interrupt and handle it.

Errors

none.

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

19 clock cycles

3.21 KoOsRegionCreate

```
KoOsHandle   KoOsRegionCreate(   char       *Name,
                                 unsigned   Size       )
```

Parameters

Name	Zero terminated string to name the region resource, or NULL for no name.
Size	Size of the requested memory block in the region resource

Return value

The handle of the just created region resource, or 0 in case of an error.

Description

This function will create a region resource of a given Size and, if successful, will return the handle of that memory resource. The region resource is a memory block with a semaphore to coordinate the usage by multiple processes.
The region resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_MEMORY_SIZE	blocks of 0 bytes not allowed
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.22 KoOsRegionDestroy

unsigned KoOsRegionDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the region resource

Return value

0 in case of an error.

Description

This function will destroy (remove, return to the kernel) the region resource indicated by the Handle. All processes that are waiting for this region are released with an error (KOOS_ERROR_DESTROY). You have to be the owner of the region resource to be able to destroy it. To be save you first have to acquire the region for use with the KoOsRegionLock function

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an region
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐other

Cycle count

t.b.d.

3.23 KoOsRegionLock

```
void        *KoOsRegionLock(    KoOsHandle    Handle
                                unsigned      Timeout     )
```

Parameters

Handle	Handle of the region resource
Timeout	Number of system time-out timer ticks that the function will wait before it will abort. 0 for no time-out support.

Return value

Address of the memory block of the region, NULL in case of an error.

Description

This function will wait for the region resource to be available for exclusive use by this process. If the region resource is currently not in use, this function call will be completed immediately without waiting. You should release your lock on the region as soon as possible with the function KoOsRegionUnlock.
The time-out support is a fail-save mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other then zero is given. If, in this case, the time-out time has elapsed, this function is completed with an error.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not valid user of resource
KOOS_ERROR_TYPE	Resource is not a region
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_DESTROY	While waiting, the region was destroyed by the owner

Function call type

☐ signaling ☑ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d. (… + waiting time)

3.24 KoOsRegionSize

unsigned KoOsRegionSize(KoOsHandle Handle)

Parameters

Handle Handle of the region resource

Return value

Size of the memory block in the region, 0 in case of an error.

Description

This function will return the Size in bytes of the memory block in the region indicated by the Handle. This function is only meaningful if the current process has the region locked.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not a valid user of the resource
KOOS_ERROR_TYPE	Resource is not a region resource

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.25 KoOsRegionUnlock

unsigned KoOsRegionUnlock(KoOsHandle Handle)

Parameters

Handle Handle of the region resource

Return value

0 in case of an error.

Description

This function will remove the lock on the specified region resource. If there was a process (or processes) waiting for this event, it will be made run-able. After this call, you are no longer allowed to access the memory block of the region. If you need it again, you can lock it again with the KoOsRegionLock function

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not a valid user of the resource
KOOS_ERROR_TYPE	Resource is not an region

Function call type

☑ signaling ☐ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.26 KoOsResourceFind

```
KoOsHandle   KoOsResourceFind(   char       *Name,
                                 unsigned   Type       )
```

Parameters

Name	Zero terminated string with the name of the resource to find.
Type	Resource type of the resource to find (see KoOs.H for resource types).

Return value

The handle of the found resource, or 0 in case of an error.

Description

This function will find a named resource with the given name and of the given type. If successful it will return the handle of that resource. The function will only find resources of which it has valid user rights. It will first search its owned resources, then the resources of its parent process, than of the parent's parent process ….. and so on.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_NO_NAME	No name given
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NOT_FOUND	Resource with given name and of given type and valid user rights for this process not found

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.27 KoOsTimeOutSupport

void KoOsTimeOutSupport(void)

Parameters

none

Return value

none

Description

This function will install and activate time-out support for waiting system calls and the process watchdog timers. It should only be used in the initialization phase of the application (in the system *main* function).
The time-out support only functions if the system timer support is activated with function like the KoOsTimerInterruptOnTimer0 or KoOsTimerInterruptOnTimer1 or KoOsTimerInterruptOnTimer2 or with custom timer support.

Errors

none

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.28 KoOsTimerAbort

unsigned KoOsTimerAbort(KoOsHandle Handle)

Parameters

Handle Handle of the timer resource

Return value

0 in case of an error.

Description

This function will abort (stop) a running timer.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an timer
KOOS_ERROR_TIMER_NOT_RUNNING	Timer was not running

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.29 KoOsTimerCreate

```
KoOsHandle  KoOsTimerCreate(  char      *Name,
                              unsigned  (*TimerFunction)(  int       SignalCount,
                                                           unsigned  *Reload )  )
```

Parameters

Name Zero terminated string to name the timer, or NULL for no name.
TimerFunction Timer function to be invoked when the created timer expires, or NULL for no timer function.

Return value

The handle of the just created timer resource, or 0 in case of an error.

Description

This function will create a timer resource and, if successful, will return the handle of that timer resource.
The timer resource binds a software timer running on the system timer tick interrupt with a function as given by TimerFunction and a counting semaphore type of resource that a process (or processes) can use to wait for.
The timer resource can be named so that sibling processes through that name (see KoOsResourceFind) can find it.
Note that the TimerFunction is a normal C function without any *_interrupt()* and *_using()* type modifiers although it will run as an interrupt routine.

Timer function

When the timer expires, the timer function is invoked. The return value of this timer function is evaluated to see if the timer semaphore has to be signaled (return value non-zero) or not signaled (return value zero).
The timer function can use the given parameter SignalCount to make a more intelligent decision about signaling or not signaling of the timer semaphore. The value of SignalCount has the following meaning :

SignalCount	Meaning
> 0	There are “SignalCount” number of signals without waiting processes
0	There are no waiting processes and no access signals
< 0	There are “-SignalCount” waiting processes

In case there is no timer function, the timer semaphore is signaled as if there where a function that always returned a non-zero value.

The timer function is called at the point that the timer has expired and is about to be reloaded with the reload value. The timer function can change this reload value through parameter Reload. This way you are able to make very complex timing sequences without restarting a timer (which always has a 1 tick uncertainty at start-up).
A graceful way of stopping the timer is to have a reload value of 0. This can be done in the timer function or with the KoOsTimerStart function.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_NAME_TOO_LONG	Resource name > 254 characters
KOOS_ERROR_NO_MORE_MEMORY	Not enough memory left to create the resource

Function call type

☐ signaling ☐ waiting ☑ creating ☐ destroying ☐ other

Cycle count

t.b.d.

3.30 KoOsTimerDestroy

unsigned KoOsTimerDestroy(KoOsHandle Handle)

Parameters

Handle Handle of the timer resource

Return value

0 in case of an error.

Description

This function will destroy (remove) the timer resource indicated by the Handle. All processes that are waiting for this timer are released with an error (KOOS_ERROR_DESTROY). If the timer was running, it is aborted (stopped) first. You have to be the owner of the timer resource to be able to destroy it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an timer
KOOS_ERROR_INVALID_MEMORY	Memory management administration corrupted

Function call type

☐ signaling ☐ waiting ☐ creating ☑ destroying ☐ other

Cycle count

t.b.d.

3.31 KoOsTimerInterruptOnTimer0

void KoOsTimerInterruptOnTimer0(unsigned Reload)

Parameters

Reload Reload value for Timer 0 in 16 bit auto reload up-counter/timer

Return value

none

Description

This function will install and activate system timer support. It should only be used in the initialization phase of the application (in the system *main* function).
To set the system timer tick interrupt you have to take the XA clock frequency and the prescaler (PT1 and PT0 bits in the SCR register) into account to calculate the reload value. A possible solution might be :

```
#define FOSCILATOR          14745600
#define PRESCALER           4
#define TICKS_PER_SECOND    100
#define TIMER_RELOAD        -FOSCILATOR/(TICKS_PER_SECOND*PRESCALER)

KoOsTimerInterruptOnTimer0(TIMER_RELOAD );
```

Errors

none

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.32 KoOsTimerInterruptOnTimer1

void KoOsTimerInterruptOnTimer1(unsigned Reload)

Parameters

Reload Reload value for Timer 1 in 16 bit auto reload up-counter/timer

Return value

none

Description

This function will install and activate system timer support. It should only be used in the initialization phase of the application (in the system *main* function).
To set the system timer tick interrupt you have to take the XA clock frequency and the prescaler (PT1 and PT0 bits in the SCR register) into account to calculate the reload value. A possible solution might be :

```
#define FOSCILATOR          14745600
#define PRESCALER           4
#define TICKS_PER_SECOND    100
#define TIMER_RELOAD        -FOSCILATOR/(TICKS_PER_SECOND*PRESCALER)

KoOsTimerInterruptOnTimer1(TIMER_RELOAD );
```

Errors

none

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.33 KoOsTimerInterruptOnTimer2

void KoOsTimerInterruptOnTimer2(unsigned Reload)

Parameters

Reload Reload value for Timer 2 in 16 bit auto reload up-counter/timer

Return value

none

Description

This function will install and activate system timer support. It should only be used in the initialization phase of the application (in the system *main* function).
To set the system timer tick interrupt you have to take the XA clock frequency and the prescaler (PT1 and PT0 bits in the SCR register) into account to calculate the reload value. A possible solution might be :

```
#define FOSCILATOR          14745600
#define PRESCALER           4
#define TICKS_PER_SECOND    100
#define TIMER_RELOAD        -FOSCILATOR/(TICKS_PER_SECOND*PRESCALER)

KoOsTimerInterruptOnTimer2(TIMER_RELOAD );
```

Errors

none

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.34 KoOsTimerStart

```
unsigned    KoOsTimerStart(    KoOsHandle    Handle,
                               unsigned      First,
                               unsigned      Reload      )
```

Parameters

Handle	Handle of the timer resource
First	value first time loaded in the timer when started, 0 not allowed
Reload	value reloaded in the timer if the timer expires, 0 stops the timer

Return value

0 in case of an error.

Description

This function will start a non-running timer. The timer will be counted in units of the system timer tick. Because there is always a start-up inaccuracy of 1 system timer tick, you can separately specify the first loaded value and the successive reload value. A reload value of 0 will stop the reloading process. This opens the possibility to create a “one-shot” timer. Changing the reload value can also be done on the fly when running by the timer function (see KoOsTimerCreate).

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not owner of resource
KOOS_ERROR_TYPE	Resource is not an timer
KOOS_ERROR_TIMER_FIRST	Value for First is 0
KOOS_ERROR_TIMER_RUNNING	Timer was already running

Function call type

☐ signaling ☐ waiting ☐ creating ☐ destroying ☑ other

Cycle count

t.b.d.

3.35 KoOsTimerWait

```
unsigned    KoOsTimerWait(    KoOsHandle    Handle
                              unsigned      Timeout    )
```

Parameters

Handle	Handle of the timer resource
Timeout	Number of system time-out timer ticks that the function will wait before it will abort. 0 for no time-out support.

Return value

0 in case of an error.

Description

This function will wait for the timer resource to be signaled by the associated timer function or directly by the timer source (if there is no timer function). If there where signals without waiting processes, then these signals are remembered, and this function call will be completed immediately without waiting.
The time-out support is a fail-safe mechanism that will be used if the timer support is included and system time-out support is included and a time-out value other then zero is given. If, in this case, the time-out time has elapsed, this function is completed with an error.
Time-out support on a timer at first sounds ridicules, but since you can create extremely wild timing sequences, and the timer function doesn't always have to signal the timer semaphore, there might be situations where it is useful. On top of that all waiting kernel functions have time-out support, so why not make that available if you have it.

Errors

KoOsGetError errorcode	Reason
KOOS_ERROR_HANDLE	Illegal handle or not valid user of resource
KOOS_ERROR_TYPE	Resource is not an timer
KOOS_ERROR_TIMEOUT	Time-out time elapsed
KOOS_ERROR_DESTROY	While waiting, the timer was destroyed by the owner

Function call type

☐ signaling ☑ waiting ☐ creating ☐ destroying ☐ other

Cycle count

t.b.d. (… + waiting time)

4 Starting an application

KoOs provides an environment in which you should provide a main() function that initializes and starts your application. The main function should then terminate and the scheduling of processes will start. So in its simplest form an application looks something like this :

```
#include “KoOs.H”

void MyApplicationProcess( KoOsHandle Parameter )
{

  // do your application work

}

void main( void )
{
  KoOsProcessCreate( 0 /*name*/, MyApplicationProcess, 64 /*stack*/, 10 /*priority*/, 0 );
}
```

To go into more detail we have to consider the following area’s :

- KoOs code before the main function,
- the main function,
- KoOs code after the main function.

4.1 KoOs code before the main function

After a reset of the processor, KoOs does the following initialization :

- Initialize the system stack pointer,
- initialize the memorypool,
- Initialize the SystemProcess and run in its context but in SystemMode, so no scheduling takes place,
- disable the processor hardware watchdog,
- initialize the processor priority registers with values given in the KoOsConfig.src file,
- initialize all statically declared variables (with 0 or the given value).

At this point the main function is called. As you can see interrupts are globally disabled and scheduling is inhibited.

4.2 The main function

At this point you should initialize your application. This involves at a minimum the creation of one process, but is usually more. This is the place to create global (system wide) resources. You have to consider the following limitations :

- interrupts are not yet enabled, so interrupts are not yet serviced,
- scheduling is not yet started, so you shouldn’t do any blocking calls.

This is the right place to start your timer tick support and your time-out & watchdog support if your application needs these functions.
When your initialisation is done, you should exit the main function to continue.

4.3 KoOs code after the main function

After the main function returns, KoOs finalizes its initialization by doing the following :

- continue the SystemProcess in user mode at processor priority level 0.
- enable the global interrupt (EA bit),
- enable scheduling .

At this point the application processes will be scheduled in and the application is running.

At some point in time non of the user processes will be runable and so the SystemProcess will run its idle time processing code. By default this will switch the processor in the idle mode which conserves power without degrading the interrupt response times.

4.4 Create an application with the Tasking EDE environment

If you are going to create an application using the Tasking EDE environment, you might find the following design hints useful.

First you have to create a project and assign all your project files to it. This not only includes your source file with the main function, but should also include the KoOs.h header file and the KoOs_s.a (small memory model) or KoOs_m.a (medium memory model) library file.

Now you have to specify how you want your project to be build. This is done with various commands under the “EDE” menu entry. The following entries are of special interest in relation to KoOs :

- Processor options…
  tab : Processor : select a target CPU
  tab : Memory : you probably have no on chip EPROM/ROM,
  you probably have external memory,
  you probably have a separate code and data bus,
- C Compiler options
  submenu : Project Options :
  Memory model = small or medium
- Linker Options…
  tab : Linker : Link default C libraries = off[1],
  tab : Stack : System stack size = … (KoOs’ system stack),
  tab : Heap : Heap size = … (KoOs’ dynamic memory)

To help you to get “up and running”, 5 example projects are given. They are kept as small as possible, they do hardly any error checking and are not useful as application. They are just there to show that “things are working”.

The example projects assume that you have something like an LED on your target hardware that you can switch on and off in a very easy way. You might have to adapt it to your target hardware.

Project KoOsTst1

[1] You cannot use the default tasking libraries ‘as is’, because they contain a module start.asm that conflicts with the startup code of KoOs (more detail : the mandatory label __START conflicts).

This project is there to demonstrate that you can compile, link, locate and run an application. You should see in the .map file all sorts of names with KoOs in it. If not, you are not using the KoOs.a library, but likely the default tasking library.

Project KoOsTst2

This project is there to demonstrate that you can start a process, and so your basic KoOs is up and running.

Project KoOsTst3

This project is there to demonstrate that timer support is working

Project KoOsTst4

This project is there to demonstrate that time-out support is working

Project KoOsTst5

This project is there to demonstrate that watchdog support and mailboxes are working and you can run multiple processes.

If you have no target hardware yet, you can run the code with the simulator[2]. Add the following code to the example projects :

```
int _simo(unsigned stream, char *port, unsigned len)
{
        return len + stream + *port;    /* names used by CrossView */
}

char TestText[] = "This is a test\n";
```

Now you can replace the LED toggle statement with the following statement :

```
_simo( 0, TestText, sizeof( TestText ) );
```

Don't forget to activate virtual I/O stream 0 for output in the simulator.

[2] You need version 1.48 or later. This version supports software interrupts and timers.