KUKA

KUKA Roboter GmbH

SOFTWARE

RTOS Virtual Machine

Whitepaper

Edition: 2008-03-01

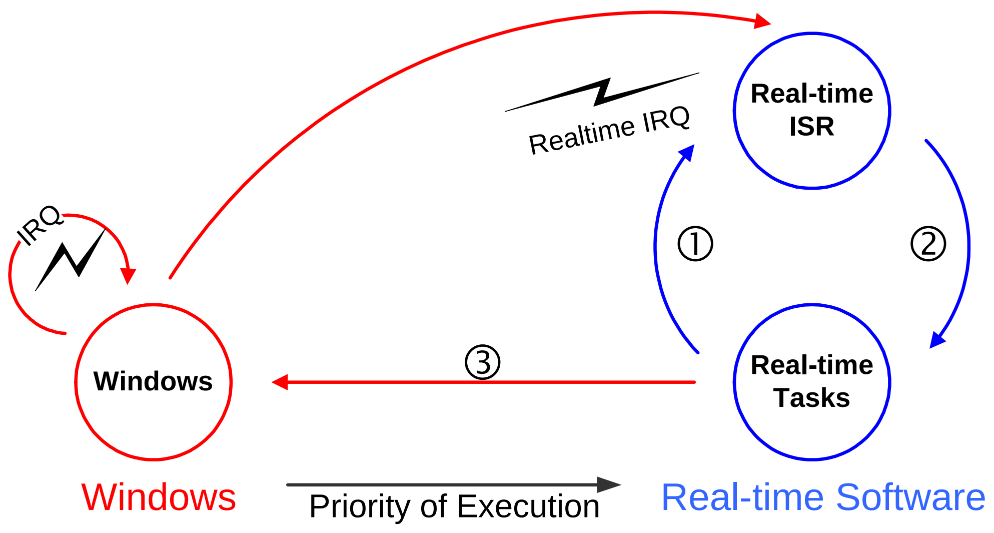
1 KUKA RTOS Virtual Machine Overview

The KUKA RTOS-VM provides a light-weight real-time virtualization platform for Windows.
On top of this platform one can very easily implement own firmware or run a custom or off-the-shelf real-time operating system.
As a result, existing real-time software can easily be adopted to run together with Windows.
When using multicore CPUs one can choose between two general operation modes.

1.1 Shared Mode Operation

Windows shall run on all CPU cores and only one CPU core shall additionally run the real-time software. If the Windows application needs a lot of CPU power (e.g. for image processing) this will be the appropriate operation mode even on multi-core CPUs. In shared mode operation Windows (on this core) will usually only get CPU time when the real-time software is idle.

The following diagram illustrates the flow of control:



<u>Operating states of the RTOS-VM in shared mode</u>

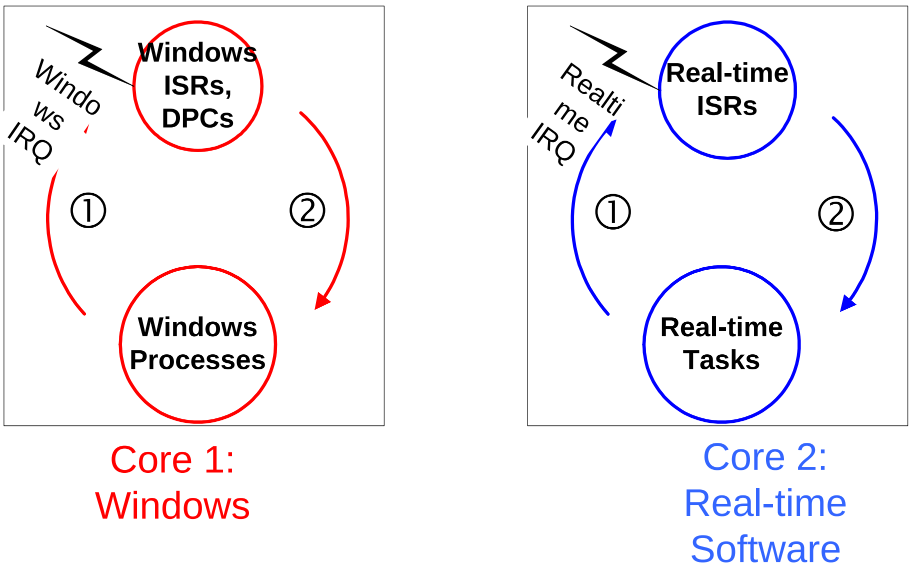
① Exception-handling or a higher priority interrupt becomes outstanding.
② Interrupt Service Routine optionally starts a new task and then finishes.
③ From the idle-state, VxWorks transfers control to Windows operating system.

Note: When running the RTOS-VM in shared mode on multiprocessor/multicore systems this state diagram is only applicable for one CPU core in the system (by default on the first core). All other CPU cores will run Windows only.

1.2 *Exclusive Mode Operation*

Windows and the real-time software shall run fully independently on different CPU cores. Using this mode will lead to much shorter interrupt and task latencies as there is no need to switch from Windows to the real-time software.

The following diagram, illustrates the flow of control on a dual core system:



<u>Operating states of the RTOS-VM in exclusive mode</u>

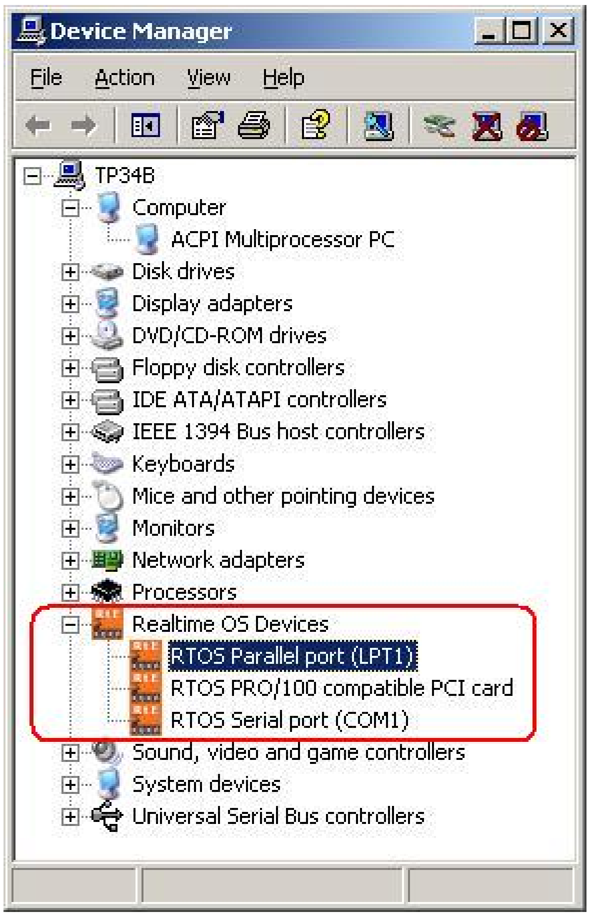
① Exception-handling or a higher priority interrupt becomes outstanding.

② Interrupt Service Routine optionally starts a new task and then finishes.

Note: When running the RTOS-VM in exclusive mode Windows will never be interrupted. Application and interrupt processing run concurrently and independently on both CPU cores. There is no need in the real-time software to enter the idle state.

1.3 Real-time Device Management

To achieve real-time behavior the RTOS will have to directly access its hardware devices. In fact, hardware devices are never emulated, neither in Windows nor in the RTOS. Every specific device, e.g. a PCI network adapter card will, then either be used by Windows or by the RTOS exclusively.
All hardware devices which shall be used by the RTOS will be managed by the Windows RtosPnp driver shipped with the KUKA RTOS-VM.
Using the KUKA Real-Time Device Manager tool the user can select which device shall be used by the RTOS and which by Windows.
Within the Windows Device Manager all RTOS devices will then appear in the “Realtime OS Devices” tree:



Within the RTOS there are two methods for detecting whether a device can be accessed or not. For PCI devices usually the PCI vendor and device ID can be used. For other devices (as well as for PCI devices) the device name (e.g. RTOS PRO/100 PCI card) can be used.

1.4 Virtual Machine Framework

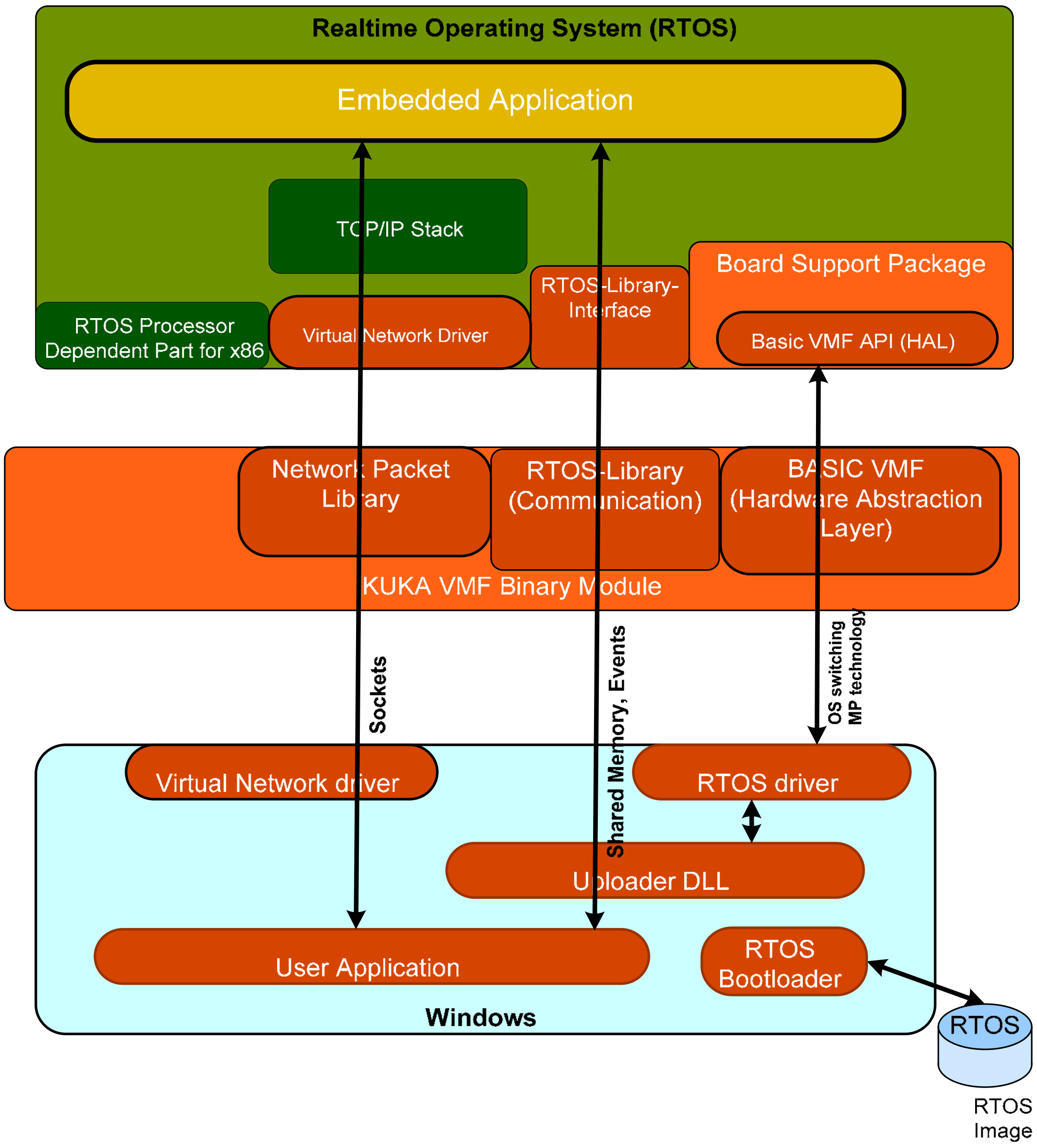
Using the KUKA RTOS-VM there is no need to understand the complex hardware of modern PC systems. The basic hardware components of the PC (architecture specific processor registers, timer, interrupt controller, memory handling/partitioning) can be accessed in the real-time software by simply calling the appropriate functions that the RTOS-VM hardware abstraction layer (HAL) provides.
Besides the HAL functions the RTOS-VM provides additional services, especially for communication with Windows:

- Shared Memory: Direct access to shared memory areas
- Shared Events: Notification using named events
- Data Access Synchronization: Interlocked Data Access
- Date and Time Synchronization
- Virtual Serial Channel
- Network Packet Library: basic Ethernet data transfer service
- RTOS configuration services (e.g. for dynamically setting the IP address of the virtual network)

The application interface between the real-time software and the RTOS-VM is called the Virtual Machine Framework (VMF).
When calling VMF hardware functions the hardware will be directly accessed and not emulated. These functions are called the VMF Hardware Abstraction Layer (HAL) functions.

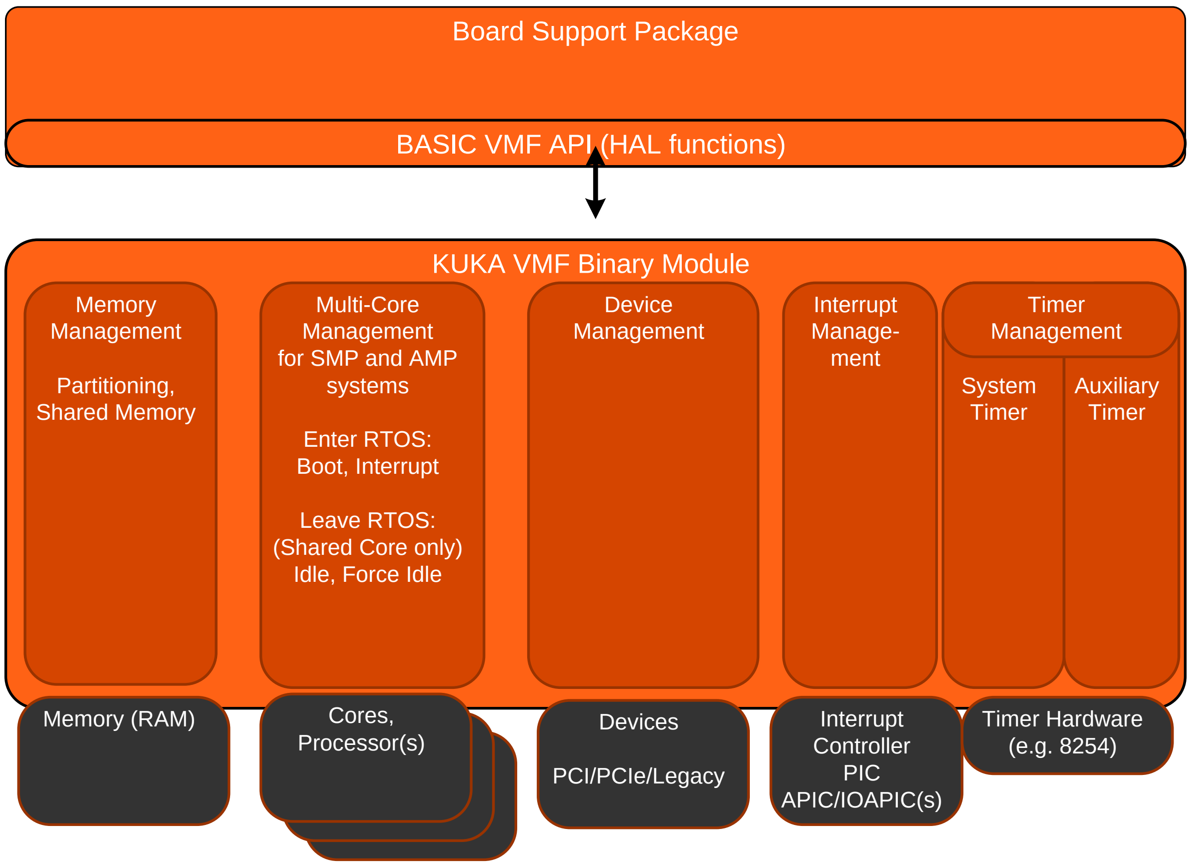
1.4.1 VMF Architecture

The following figure shows the general architecture of the VMF when a RTOS is embedded within Windows. Besides the basic VMF API (the HAL) which usually is required to build a RTOS BSP (Board Support Package) the VMF contains functions for communication between Windows and the RTOS (e.g. shared memory, events, network packet library). On top of the network packet library a virtual network driver can be built which will then provide a virtual network connection between Windows and the RTOS.



1.4.2 Basic VMF Services (Hardware Abstraction Layer)

The basic VMF services provide a simple programming interface to access the otherwise complex PC hardware.
The following figure shows in more detail the basic VMF services which usually are used within a RTOS Board Support Package.



Board Support Package
BASIC VMF API (HAL functions)
KUKA VMF Binary Module
Memory Management
Partitioning, Shared Memory
Multi-Core Management for SMP and AMP systems
Enter RTOS: Boot, Interrupt
Leave RTOS: (Shared Core only) Idle, Force Idle
Device Management
Interrupt Manage-ment
Timer Management
System Timer
Auxiliary Timer
Memory (RAM)
Cores, Processor(s)
Devices
PCI/PCIe/Legacy
Interrupt Controller PIC APIC/IOAPIC(s)
Timer Hardware (e.g. 8254)

When porting system software (e.g. a RTOS Board Support Package) to run with the KUKA RTOS-VM there is no need to directly access PC hardware like timers or interrupt controllers.
The VMF as well provides a generic method for booting the system software (e.g. a RTOS) and for setting up the RTOS memory context (virtual memory).
When running on multi-core systems the VMF also provides methods for executing a RTOS which supports Symmetric Multiprocessing (SMP).
Summarized, using the VMF one gets the following advantages:

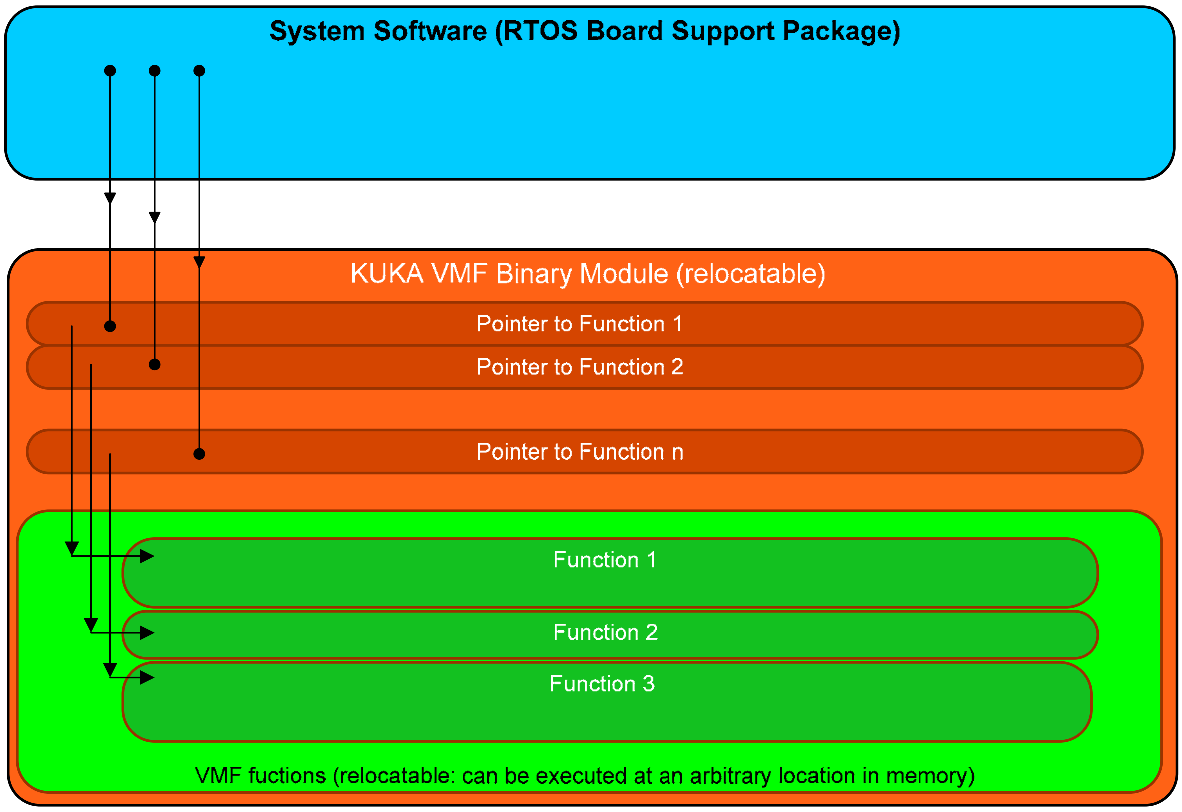
- Fully virtualized hardware access (via Hardware Abstraction Layer functions). No need to understand the complex PC hardware.
- Either run the RTOS and Windows together on one single core or use dedicated cores exclusively for each operating system.
- The **same** RTOS image can be run either on a shared or a non-shared CPU core.
- Sophisticated Multi Core Support
  - Run the RTOS on one single or on multiple cores (SMP)
  - A RTOS can run in SMP mode even on dual core CPUs

1.5 Portability

When using standard frameworks or libraries the customer usually gets either source-code which in a first step would have to be ported to his specific environment (operating system, compiler, linker).
In cases where the supplier does not want to ship the source-code the customer would have to wait until a version for the framework/library is available for his environment.

To avoid these implications the KUKA VMF is not shipped as a library or source code but as a relocatable binary module. This binary module will be loaded by the KUKA RTOS-VM at an arbitrary location in the memory (the VMF code can be executed at any location in memory!).

Every call to a VMF function will then be redirected via well-known locations inside a jump table, this jump table is stored at a well-defined location inside the binary module.
Thus there is no need to port one single line of C language or assembly language code (and no need to add the VMF as an additional library to the customer's environment).
The only requirement is to include one single header file. Within this header file the VMF functions are simply defined as macros which call the appropriate functions using the function pointer in the jump table.

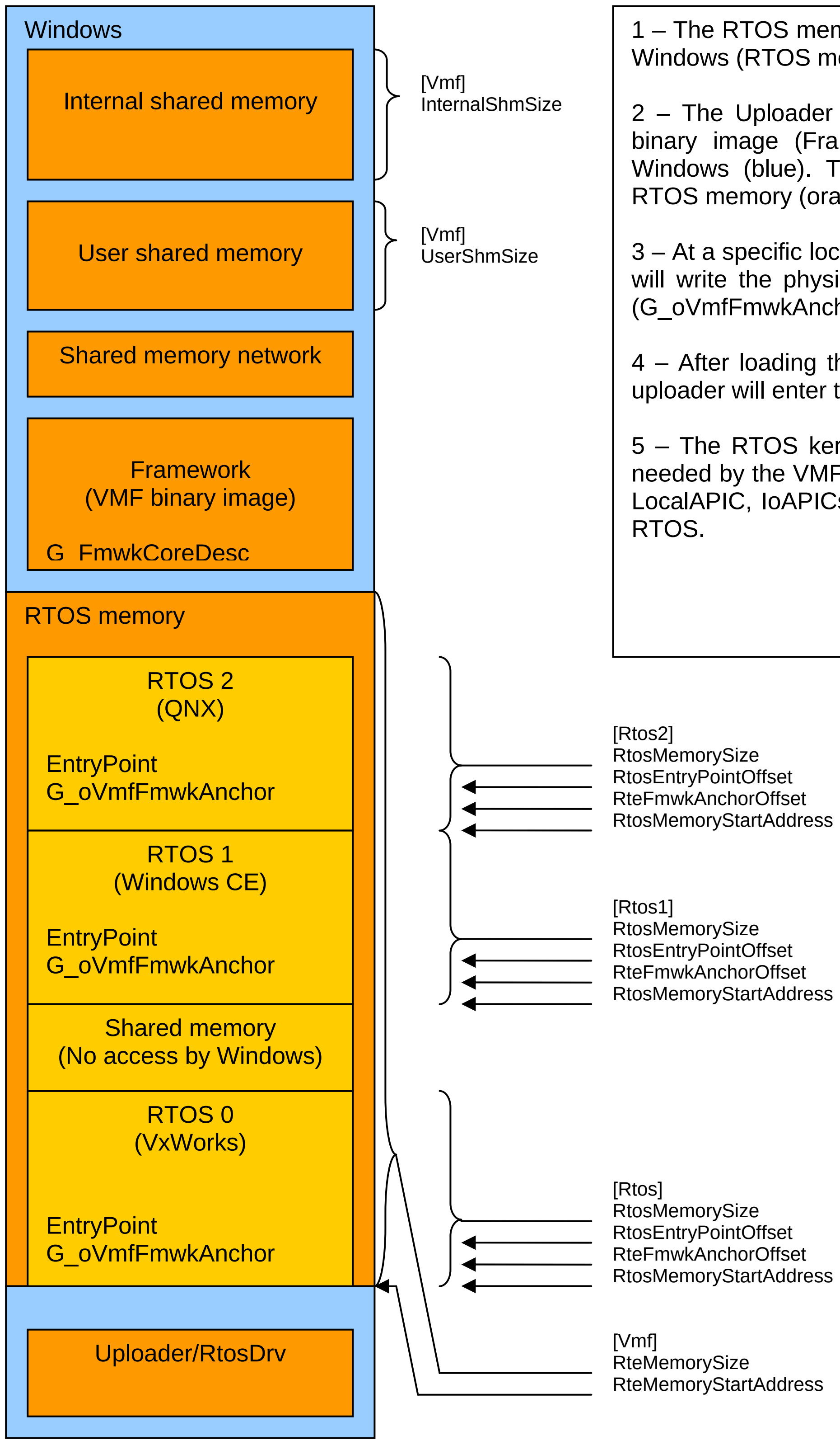

System Software (RTOS Board Support Package)
KUKA VMF Binary Module (relocatable)
Pointer to Function 1
Pointer to Function 2
Pointer to Function n
Function 1
Function 2
Function 3
VMF fuctions (relocatable: can be executed at an arbitrary location in memory)

Summarized, using the VMF binary module leads to the following advantages:

- No porting necessary, just include a C header file.
- No change necessary in the system software when new VMF versions are released (just exchange the binary module by the new one).
- The same binary VMF module will be used together with different RTOSes; this ensures a higher quality than if the VMF code would have been ported individually for any RTOS.

1.6 Memory Layout

VMF = Virtual Machine Framework
RTOS Framework = RTOS interface (VMF interface functions)



1 – The RTOS memory area (orange) will not be used by Windows (RTOS memory configuration)

2 – The Uploader (RTOS Bootloader) copies the VMF binary image (Framework) into an area allocated by Windows (blue). The RTOS image is copied into the RTOS memory (orange).

3 – At a specific location in the RTOS image the uploader will write the physical base address of the VMF image (G_oVmfFmwkAnchor).

4 – After loading the RTOS image into the memory the uploader will enter the RTOS boot entrypoint.

5 – The RTOS kernel will then boot. All memory areas needed by the VMF (Internal / User Shm, virtual network, LocalAPIC, IoAPICs etc.) will have to be mapped by the RTOS.

1.7 General

The VMF defines a virtual machine platform to run one or multiple secondary operating systems (RTOS) on top of a primary operating system (Windows).
The VMF is a binary module which is loaded at a predefined physical address. The interface function entry points are located at fixed offsets within this binary module.
All functions of the VMF are fully relocatable, thus the VMF may be located at any physical address and mapped into the RTOS memory context at an arbitrary location without to be recompiled or relinked.

1.8 VMF management anchor

Some information about the VMF is needed within the RTOS, e.g. the physical base address of the framework binary image. This data is located at a specific location inside the RTOS memory.
After loading the RTOS image into the memory the uploader will copy the VMF management data at the appropriate location inside the RTOS memory

2 Example Implementation: Mini RTOS

The example implementation shows how to use the VMF functions to get a RTOS running. Typically the Board Support Package of the RTOS has to be adjusted to use VMF functions. Therefore the Mini RTOS example is split into two parts:

a) Board Support Package: Mini BSP
b) RTOS: Mini RTOS

The Mini BSP contains all the low level functions which initialize the hardware (timer, interrupt controller). The Mini RTOS is merely a placeholder for a real RTOS. It doesn't provide any real functionality.

2.1 Content, directories

The example is split into the following directories:

2.1.1 DEFS

Header files needed when building a Board Support Package for the RTOS.

2.1.2 MiniBsp

Example BSP. This example BSP can be used as a starting point when creating a Board Support Package for the target operating system.
The following list explains the functionality of the most important files:

- rtosBspAsm.s
  → boot entry point, function bspInit().
- rtosBoot.c
  → example RTOS boot sequence. Entry point at function rtosInit().
- rtosBsp.c
  → Main BSP functions
- rtosBspTimer.c
  → example timer implementation (the first timer is usually used for the RTOS clock tick timer, the second timer is a auxiliary timer to be used by the application).
- device.config
  → Real-Time device management configuration file (for devices which are configured manually)
- rtos.config
  → RTOS configuration file (required for booting the RTOS)

2.1.3 MiniRtos

This is a placeholder for a real RTOS. The Mini BSP needs some of these functions.

2.2 Build the Mini RTOS

The Mini RTOS is built using the shipped GNU tools.
The following steps have to be executed:

- Start a Windows command shell
- Change into the examples directory
- Set the appropriate environment by running setenv.bat
- Start the build using the BuildMiniRtos.bat file

As result the file MiniRtos.bin will be generated.
This file can be loaded and started by the RTOS-VM Uploader tool.

2.3 Start the Mini RTOS

Using the RTOS-VM Uploader tool the Mini RTOS binary image (MiniRtos.bin) can be loaded into memory and is then started automatically.
The following additional files are required (located in the same directory):

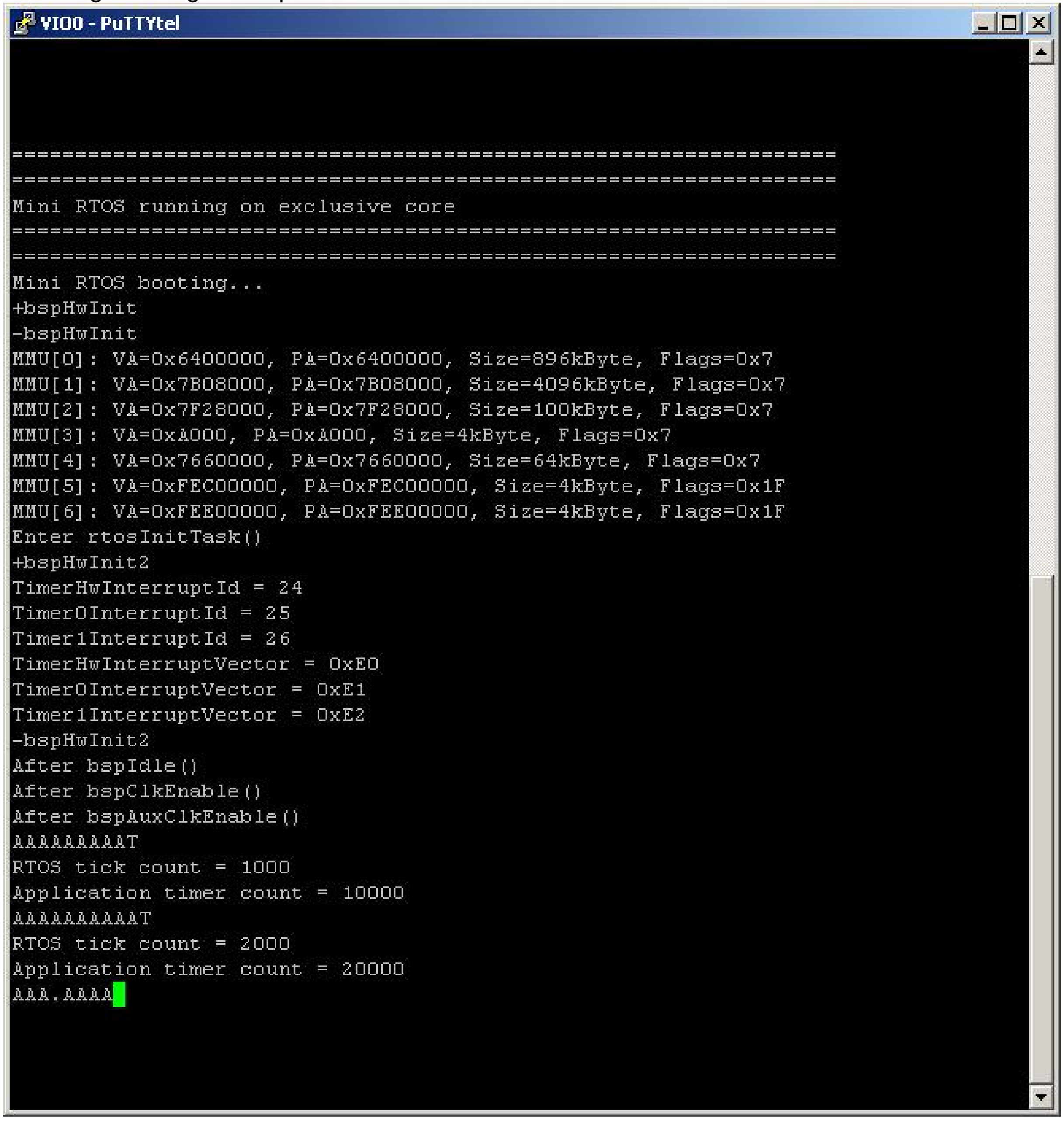
- rtos.config RTOS configuration file including information for the Uploader
- device.config RTOS device configuration file
- vmf.bin VMF binary image
- Windows runtime environment for the RTOS-VM: The Uploader tool, the RTOS service and the RTOS control application.

The Mini RTOS can then be started with the following command line:

```
UploadRTOS.exe MiniRtos.bin
```

2.4 Debug Console

The virtual I/O channel is used as debug console.
The shipped putty Telnet client also supports the virtual I/O channel (using the command line option –vio).
After starting the Mini RTOS the putty application can be started with the –vio option and the following messages are printed.


VIO0 - PuTTYtel

```
=================================================================
=================================================================
Mini RTOS running on exclusive core
=================================================================
=================================================================
Mini RTOS booting...
+bspHwInit
-bspHwInit
MMU[0]: VA=0x6400000, PA=0x6400000, Size=896kByte, Flags=0x7
MMU[1]: VA=0x7B08000, PA=0x7B08000, Size=4096kByte, Flags=0x7
MMU[2]: VA=0x7F28000, PA=0x7F28000, Size=100kByte, Flags=0x7
MMU[3]: VA=0xA000, PA=0xA000, Size=4kByte, Flags=0x7
MMU[4]: VA=0x7660000, PA=0x7660000, Size=64kByte, Flags=0x7
MMU[5]: VA=0xFEC00000, PA=0xFEC00000, Size=4kByte, Flags=0x1F
MMU[6]: VA=0xFEE00000, PA=0xFEE00000, Size=4kByte, Flags=0x1F
Enter rtosInitTask()
+bspHwInit2
TimerHwInterruptId = 24
Timer0InterruptId = 25
Timer1InterruptId = 26
TimerHwInterruptVector = 0xE0
Timer0InterruptVector = 0xE1
Timer1InterruptVector = 0xE2
-bspHwInit2
After bspIdle()
After bspClkEnable()
After bspAuxClkEnable()
AAAAAAAAAT
RTOS tick count = 1000
Application timer count = 10000
AAAAAAAAAAAT
RTOS tick count = 2000
Application timer count = 20000
AAA.AAAA
```

Using the debug console is very helpful in the bring-up phase of the RTOS. Messages can be printed out at a very early stage.
The file rtosBsp.c contains helpful routines like DbgPrintf() which prints a formatted message similar to printf() but without needing the RTOS.
The function DbgWait() can be used to stop processing until the user presses a key at the debug console.

3 RTOS configuration

The rtos.config file contains several entries where memory and CPU settings are defined. RTOS specific settings may also be stored herein; these settings will have to be processed by the RTOS. Settings for a RTOS are stored beyond a RTOS specific key.
The first RTOS gets the key “[Rtos]”, the second gets the key “[Rtos1]” etc.
The following entries are defined:

• MemoryStartAddress	RTOS memory physical base address
• MemorySize	RTOS memory size
• ImageOffset	Offset where the RTOS image has to be copied by the uploader
• EntryPointOffset	Boot entrypoint offset of the RTOS
• VmfAnchorOffset	VMF management anchor offset. After loading the RTOS image the uploader will copy the VMF management information data at this location.
• ProcessorMask	CPU mask to determine where the RTOS shall run (multiple bits could be set in case of a SMP system)

Example:

```
 [Rtos]                              ; OsId = 0
    "MemoryStartAddress"=dword:1000000  ; Note: value is hexadecimal!
    "MemorySize"=dword:1000000          ; Note: value is hexadecimal!

    "ImageOffset"=dword:8000            ; RTOS image offset
    "EntryPointOffset"=dword:8000       ; RTOS boot entrypoint offset
    "VmfAnchorOffset"=dword:8010        ; RTOS management anchor offset

    "ProcessorMask"=dword:0x0001        ; VxWin on shared BP

    "Bootline" = "shm(0,1)pc:vxWorks h=192.168.0.1 e=192.168.0.2 u=target pw=vxworks"

[Rtos1]                              ; OsId = 1
    "MemoryStartAddress"=dword:2000000  ; Note: value is hexadecimal!
    "MemorySize"=dword:1000000          ; Note: value is hexadecimal!
    "ImageOffset"=dword:8000            ; RTOS image offset
    "EntryPointOffset"=dword:8000       ; RTOS boot entrypoint offset
    "VmfAnchorOffset"=dword:8010        ; RTOS management anchor offset

    "ProcessorMask"=dword:0x000E        ; run in SMP mode on AP1, AP2, AP3

    "Bootline" = "vnet(0,1)pc:vxWorks h=192.168.0.1 e=192.168.0.2 u=target pw=vxworks"
```

4 Using the VMF API functions

Every framework function uses a pointer to the framework and a pointer to a data area. These two pointers shall be globally defined. The names of the pointer variables are fix and must be pFmwkDesc and pvFmwkData.

If paging is disabled pFmwkDesc points to a physical address, otherwise it must point to a virtual address. The physical base address of the VMF can be found at a specific location in memory, the VMF anchor descriptor (VMF_ANCHOR_DESC). This location is determined as follows:

Pointer to VMF_ANCHOR_DESC = MemoryStartAddress + VmfAnchorOffset

The values of MemoryStartAddress and VmfAnchorOffset can be found in the rtos configuration file.

The size of the data area at which pvFmwkData points is fix, it is set by the macro VMF_FMWK_DATADESC_SIZE.

The VMF can only be used after the variables pFmwkDesc and pvFmwkData are initialized. See section 5.2 for more details.

Prior to using VMF functions you have to include the vmfInterface.h header file.
The header file rteOs.h contains data types used by the framework functions.
Error definitions are located in rteError.h.

Example:
```
#include <vmfInterface.h>
#incluede <rteError.h>
```

5 Booting the RTOS

5.1 Pre-boot steps

The following steps are executed by the RTOS uploader prior to enter the boot entrypoint of the RTOS.

- Clear the RTOS memory area
- Copy the RTOS image file at the appropriate offset inside the RTOS memory area
- Store the VMF management information at the anchor offset address inside the RTOS memory area

After these preparing steps the boot entrypoint of the RTOS will be called.
The boot entrypoint is called in 32 bit protected mode with valid GDT and SS, DS and CS selectors that allow 4 GByte of memory to be addressed. Paging is turned off.

5.2 Framework initialization, Framework memory context mapping

Every framework function uses a pointer to the framework and a pointer to a data area. These two pointers may be globally defined. The names of the pointer variables are fix and must be pFmwkDesc and pvFmwkData.

If paging is disabled pFmwkDesc points to a physical address, otherwise it must point to a virtual address. The physical base address of the VMF can be founded at a specific location in memory, the VMF anchor descriptor (VMF_ANCHOR_DESC). This location is determined as follows:

Pointer to VMF_ANCHOR_DESC = MemoryStartAddress + VmfAnchorOffset

The values of MemoryStartAddress and VmfAnchorOffset can be found in the rtos configuration file.

The size of the data area at which pvFmwkData points is fix, it is set by the macro VMF_FMWK_DATADESC_SIZE.

Prior to using VMF functions you have to include the vmfInterface.h header file. The header file rteOs.h contains data types used by the framework functions.

Example:
```
#include <vmfInterface.h>
#incluede <rteOS.h>

/**************************************************************************
*                                 DEFINES
*/

#define RTOS_BASE_ADDR  0x1000000   /* base physical address where the RTOS is linked to */
#define RTOS_VMF_ANCHOR_OFFSET  0x2000  /* VMF anchor offset in RTOS memory */

/**************************************************************************
*                                 GLOBALS
*/

PVMF_FMWK_DESC  pFmwkDesc   = NULL;
VOID*           pvFmwkData  = NULL;
UINT8   G_oVmfFmwkData[VMF_FMWK_DATADESC_SIZE];
VMF_ANCHOR_DESC* G_pVmfAnchorDesc = (VMF_ANCHOR_DESC*)(RTOS_BASE_ADDR +
                                                       RTOS_VMF_ANCHOR_OFFSET);
void FrmwkInit{void)

       /* VMF pointers initialization */
       pFmwkDesc  = (PVMF_FMWK_DESC)G_pVmfAnchorDesc->dwFmwkAddrPhys;
       pvFmwkData = &G_oVmfFmwkData[0];

       // now framework functions can be used (after VMF initialization!)

}
```

Prior to calling any framework functions some memory area mappings have to be initialized. Afterwards the basic framework initialization has to be executed with paging turned on.

Step 1: vmfCoreGetKernelMapTablePhys()
➔ This call will return a mapping table with physical memory areas which have to be mapped at an arbitrary virtual memory location. The mapping table contains the following information for each memory area:

- Memory type (e.g. IOAPIC memory area, internal shared memory area, virtual network memory area)
- Physical base address
- Memory size

The virtual memory location must be calculated and returned back to the Framework.
The following macros are used in the Framework to identify the type of memory.
Memory types are mainly divided into two types, cached and uncached:

Cached:

VMF_KERNELMAP_RTOSMEMORY	→ RTOS memory area
VMF_KERNELMAP_FRAMEWORK	→ VMF binary module memory area
VMF_KERNELMAP_INTERNALSHM	→ Internal shared memory
VMF_KERNELMAP_USERSHM	→ User shared memory
VMF_KERNELMAP_PROCESSORBOOTCODE	→ Memory area with processor boot code
VMF_KERNELMAP_VNET	→ Shared memory area fort he virtual network

Uncached:

VMF_KERNELMAP_INTERRUPT_PROCESSOR	→ Local APIC memory
VMF_KERNELMAP_INTERRUPT_IOAPIC	→ I/O APIC memory

The RTOS (BSP) has to map these memory areas at an arbitrary virtual address location. The virtual base addresses then have to be stored in the mapping table.
Later, calls to some of the framework function will then require a pointer to the mapping table to be able to access these memory areas – the framework function will then use the virtual base address provided by the RTOS.

Optional step 1b: map memory regions, enable paging
➔ In this step the MMU will be enabled and all memory areas returned by vmfCoreGetKernelMapTablePhys() will have to be mapped, the virtual addresses in the mapping table will then have to be set to the appropriate values.
The areas to be mapped are (see above):

- RTOS framework binary image
- Internal shared memory
- Additional framework memory regions

Step 2: call vmfCoreInit()
➔ basic framework hardware initialization (e.g. initializing framework data descriptor)

Optional step 2b: map memory regions, enable paging (if not done in 1b)
➔ After the MMU is enabled the virtual addresses in the kernel mapping table have to be set to the appropriate values.

```
...
for (nLoop = 0; nLoop < nUsedEntries ; nLoop++)
{
   if (aVmfKernelMap[nLoop].dwType == VMF_KERNELMAP_FRAMEWORK)
   {
      pFmwkDesc = aVmfKernelMap[nLoop].dwVirtAddress;
      break;
   }
}
vmfCoreInit(G_pVmfAnchorDesc, aVmfKernelMap, nUsedEntries);
```

Step 3: call vmfCoreHwInit()
➔ basic hardware initialization (e.g. interrupt controller, timer)

5.3 RTOS boot

The last step is to finish booting the RTOS.

5.4 Shared processor core: RTOS idle loop

If the RTOS is running in shared mode (shared with a primary operating system on the same processor core, e.g. Windows) it has to return every time when entering the idle loop. Otherwise the primary operating system will never be executed.
A call to vmfCoreIdleRoutine() will return to the primary OS (e.g. Windows).
If the RTOS does not provide a specific idle loop a task with lowest priority has to be created which has to call this function.