acontis technologies GmbH

SOFTWARE

RTOS Virtual Machine

Virtual Machine Framework for Windows

Product Manual
Edition: 2024-11-11

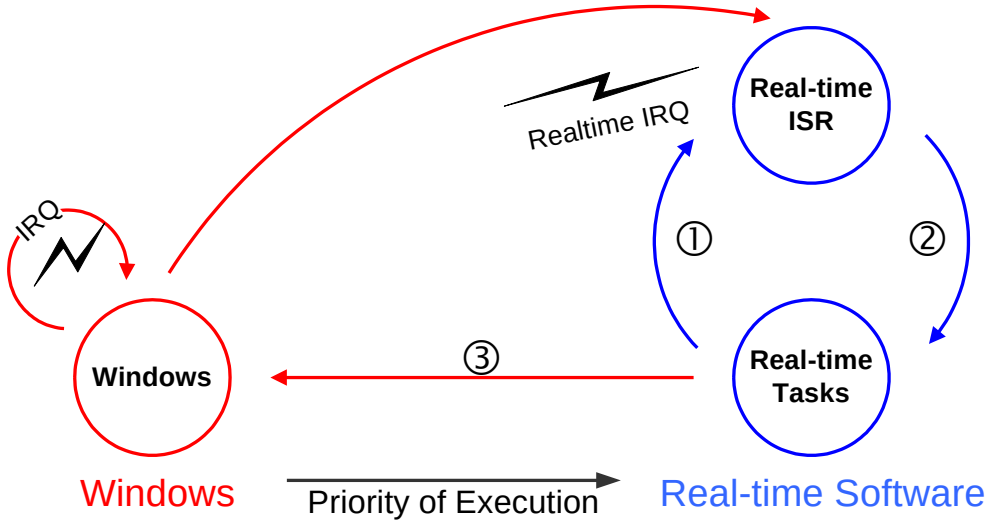
1 ACONTIS RTOS Virtual Machine Overview

The ACONTIS 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:



Operating states of the RTOS-VM in shared mode

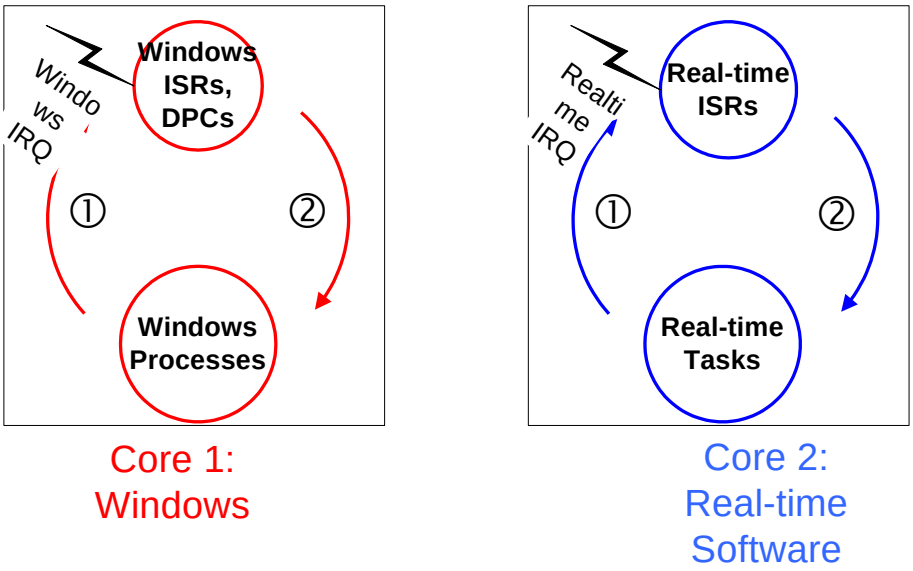
① Exception-handling or a higher priority interrupt becomes outstanding.
② Interrupt Service Routine optionally starts a new task and then finishes.
③ From the idle-state, VxWorks transfers control to Windows operating system.

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:



Core 1:
Windows

Core 2:
Real-time
Software

Operating states of the RTOS-VM in exclusive mode

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 ACONTIS RTOS-VM.
Using the ACONTIS 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

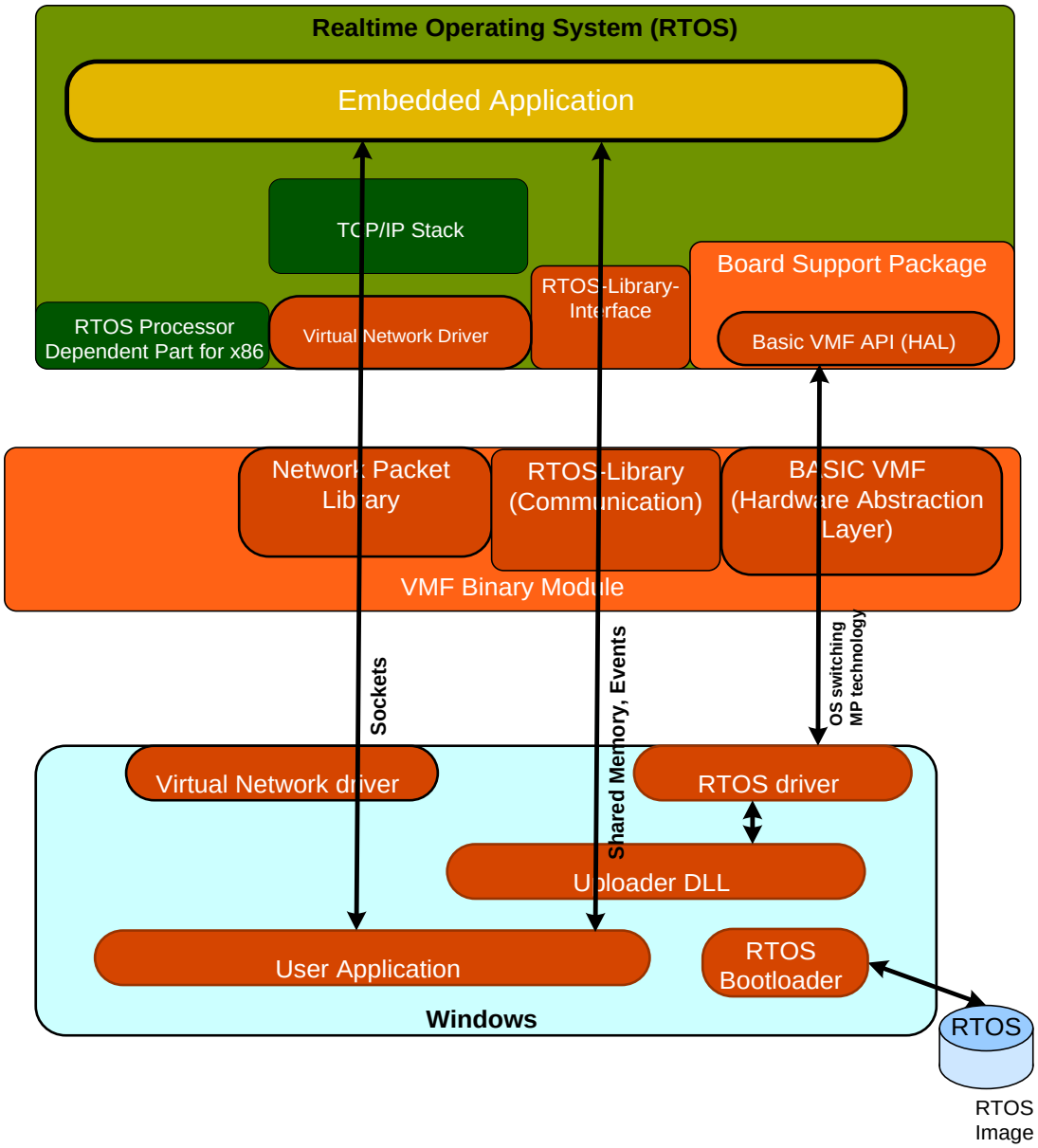
Using the ACONTIS RTOS-VM there is no need to understand the complex hardware of modern PC systems. The basic hardware components of the PC (architecture specific processor registers, timer, interrupt controller, memory handling/partitioning) can be accessed in the real-time software by simply calling 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.

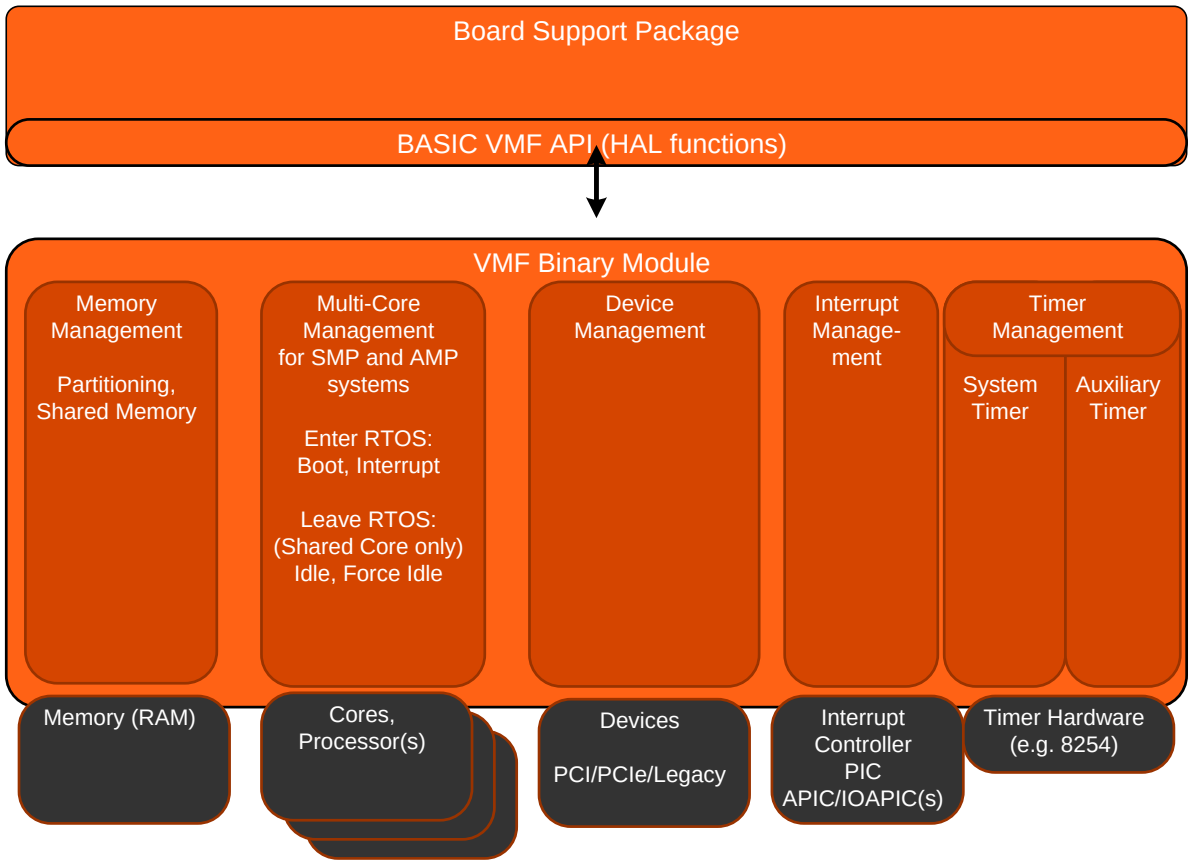
1.4.1 VMF Architecture

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.



When porting system software (e.g. a RTOS Board Support Package) to run with the ACONTIS 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:

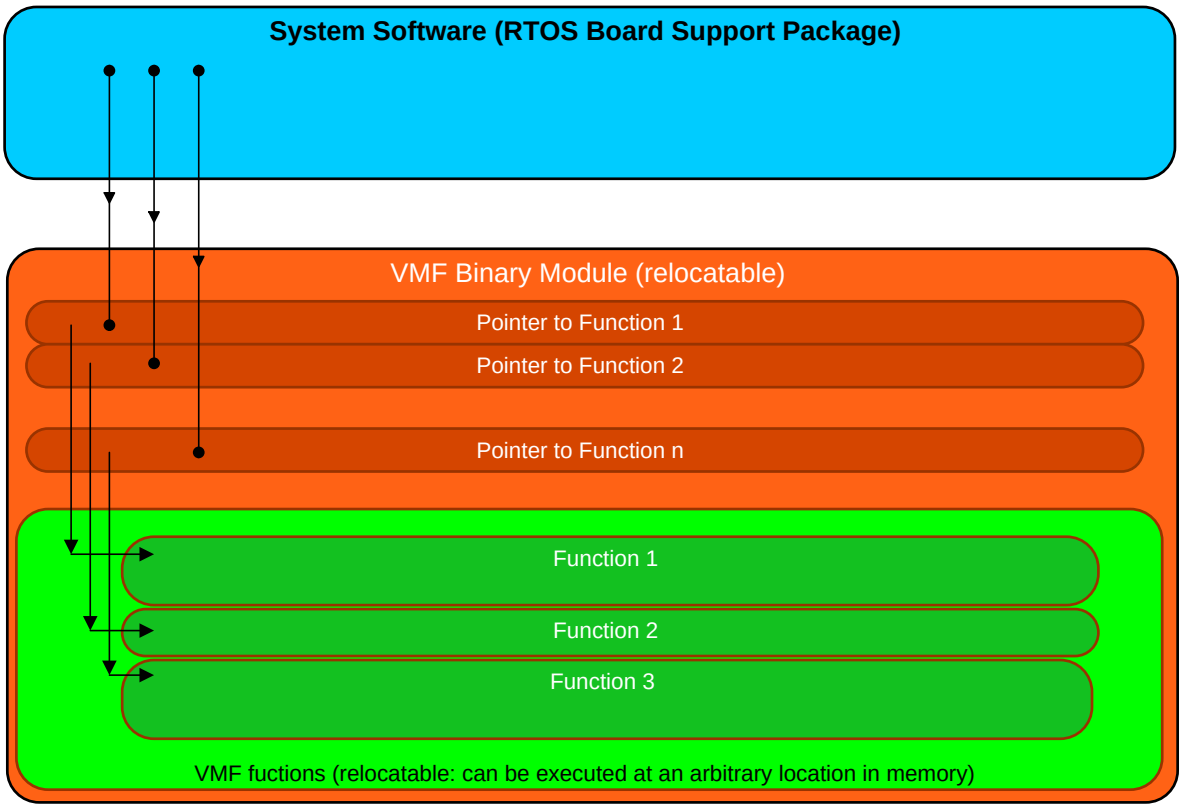
- Fully virtualized hardware access (via Hardware Abstraction Layer functions). No need to understand the complex PC hardware.
- Either run the RTOS and Windows together on one single core or use dedicated cores exclusively for each operating system.
- The **same** RTOS image can be run either on a shared 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 ACONTIS VMF is not shipped as a library or source code but as a relocatable binary module. This binary module will be loaded by the ACONTIS 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.

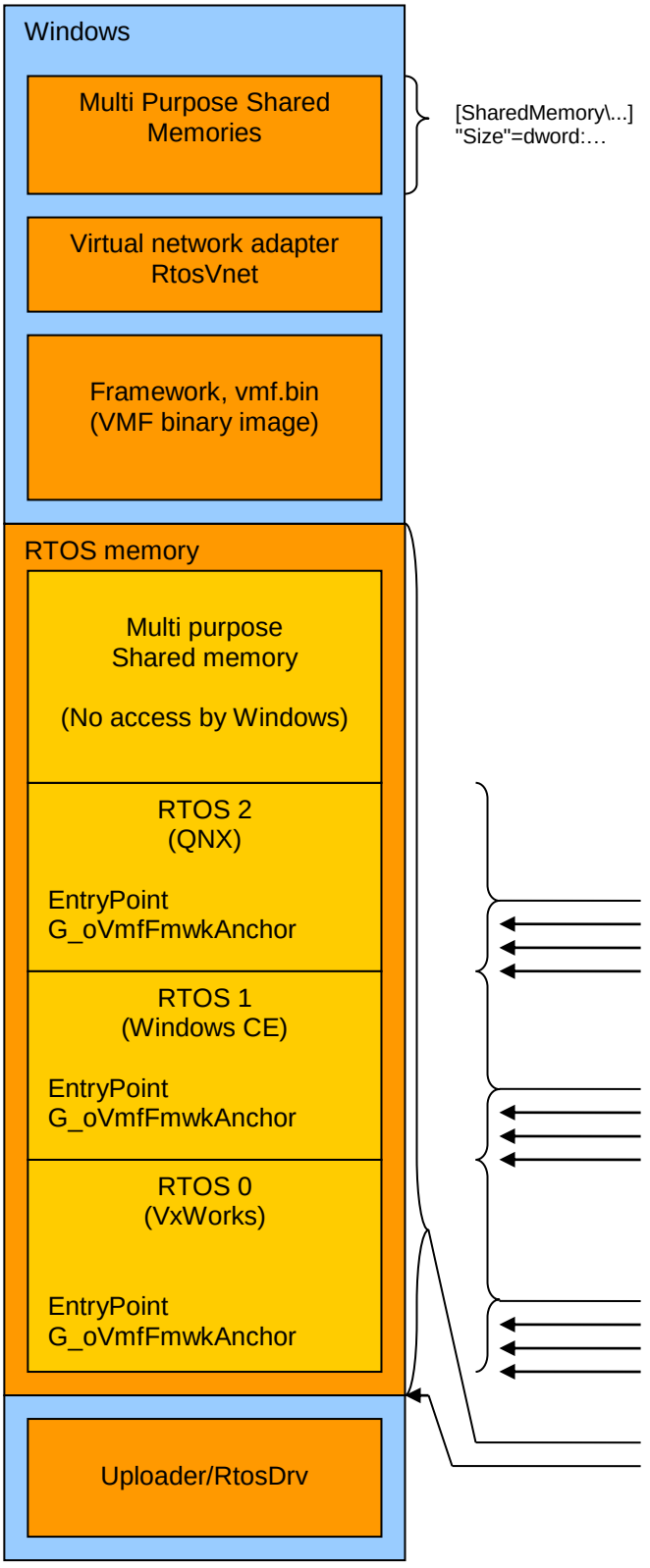


Summarized, using the VMF binary module leads to the following advantages:

- 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

2 – The Uploader (RTOS Bootloader) copies the VMF binary image (Framework) file vmf.bin 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 area (the anchor, G_oVmfFmwkAnchor) some basic VMF information is located, among other information the uploader will store the physical base address of the VMF image here.

4 – After loading the RTOS image into 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 VmfWin SDKs and example implementations

When using the VmfWin Software Development Kit an off-the-shelf RTOS can be adjusted to co-existently run and communicate with Windows.
The SDKs offers basic services to adjust the Board Support Package of the RTOS (for example the hardware abstraction layer) as well as higher-level Windows/RTOS communication services.

After installing VmfWin it is possible to load and execute the shipped Mini RTOS example (see chapter 2.7). By default VmfWin is located below C:\Program Files\acontis_technologies\VmfWin.

2.1 RTOS SDK

The RTOS SDK is required to adjust the RTOS. It contains basic functions like the VMF hardware abstraction layer as well as communication services like the network packet library or shared memory access functions.
The RTOS SDK is located below the VmfWin\SDK directory. It contains the header files required to use the Virtual Machine Framework. As the VMF is a binary module which has not to be linked with the RTOS there is no need for a library file to include.

2.1.1 Mini RTOS example implementation

The Mini RTOS 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 hardware abstraction layer functions instead of directly access the hardware.
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 OS functionality.

2.1.2 Virtual network packet library example implementation

The VxWorks network driver is shipped as an example implementation to demonstrate the usage of the virtual network packet library. See chapter 6.12 for more information about the vnet library.

2.2 The RTOS Library

VMF communication service functions (e.g. for shared events) only provide the basic services without any synchronization, some of them also have to be called within a well-defined memory context (ring 0 context, kernel context).
The Windows/RTOS communication services are therefore summarized within the RTOS library which is based on VMF services. This library is split into two parts, an OS independent part and an OS dependent part.
Synchronization (e.g. interrupt locking or mutexes) are part of the OS dependent part.
A detailed description of the RTOS library can be found in chapter 8.
The RTOS library is the application side counterpart of the Windows SDK (see chapter 2.3).

2.2.1 Windows

A windows application cannot directly call VMF functions but uses RTOS library functions instead. These functions are provided with the Windows SDK (see below).

2.2.2 RTOS

Some of the VMF communication services have to be synchronized to work properly (e.g. shared events). In some real-time operating systems it is also necessary to change into kernel mode when calling such functions in the application layer.
Therefore the RTOS side should not use those VMF functions directly but similar to Windows by using RTOS library functions instead.
VmfWin provides a portable RTOS library which is split into two parts, an OS independent part and an OS dependent part which has to be adjusted.

2.2.3 VMF Interface serialization

On some operating systems it is necessary to serialize VMF functions.
If for example on Windows or Windows CE a VMF function shall be called in user mode (Ring 3) the call has to be serialized first. In a second step the serialized data have to be transferred to a kernel driver (using some kind of IoControl call). Then the kernel driver has to de-serialize the data and call the appropriate VMF functions. The results then will have to be transferred back to the user mode application.
For this purpose VmfWin contains a portable serialization module. Most of the serialization can be handled in a generic way, only a few simple functions have to be implemented in a separate operating system specific adaptation layer. See chapter 7 for more information.

2.3 Windows SDK

The Windows SDK is required for writing Windows applications that shall use the communication services (e.g. shared memory, shared events) of VmfWin.
The Windows SDK is located below the VmfWin\SDK\inc\Windows and VmfWin\SDK\lib\Windows directories. It provides the RTOS library for Windows (RtosLib32.h \ RtosLib64.h) together with the appropriate header files (e.g. rtosLib.h).
This library has to be linked together with the Windows application to access the RTOS library functions of the dynamic link library RtosLib32.dll (32 bit) or RtosLib64.dll (64 bit) (located in the VmfWin\Bin\Windows\x86 (32 bit) or VmfWin\Bin\Windows\x64 (64 bit)).

2.4 Content, directories

2.4.1 RTOSWin runtime setup

Below "…\RtosWin Runtime" the RTOSWin runtime setup can be found. This will install all drivers and configuration files required prior to installing the final RTOSWin solution.

2.4.2 VmfWin

All VmfWin components are located below this directory. It contains all components required to build the RTOSWin product on top of the RTOS Virtual Machine and the Virtual Machine Framework.

2.4.3 SDK

The SDK is located below …\VmfWin\SDK. It consists of the following parts.

a) The Virtual Machine Framework (VMF)
   The VMF is the key part of the product used to port an RTOS on top of the RTOS Virtual Machine. It provides a hardware abstraction layer as well as basic communication and diagnostic functions.
b) The RTOS Library (rtoslib)
   In contrast to the VMF, which is required to porting an RTOS and usually only used in the RTOS' Board Support Package, the rtoslib provides functions which the end user application may use (e.g. communication services).
c) Examples
   The MiniBSP/MiniRTOS example shows how to use the VMF hardware abstraction layer, the RtosVnet example shows how to use the VMF's network packet library and several other examples show the usage of the rtoslib.

2.4.3.1 Virtual Machine Framework API

Only three header files located below …\SDK\Defs are required to port the RTOS on top of the RTOS Virtual Machine. No library has to be linked to the RTOS Board Support Package.

2.4.3.2 VMF interface serialization

The VMF interface serialization is provided for cases when the interface is not available to be called directly (e.g. in Windows when it is called in user mode it has to be serialized and transferred to a kernel mode driver).
The generic (portable) source code of the VMF interface serialization can be found in …\SDK\VmfInterfaceSerializing\KernelMode (for the kernel mode part) and in …\SDK\VmfInterfaceSerializing\UserMode (for the user mode part). Below these directories there are some example implementations for the OS adaptation layer (Windows CE, Windows).

2.4.3.3 RTOS Library

The generic (portable) source code of the RTOS Library is located in …\SDK\RtosLib. Beneath this directory there are some example implementations for the OS adaptation layer (VxWorks, Windows CE, Windows).

2.4.4 Examples

2.4.4.1 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).

2.4.4.2 MiniRtos

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

2.4.4.3 Virtual Network Packet Library

A VxWorks network driver serves as an example how to use the virtual network packet library. This network driver is used in the ACONTIS VxWin product.

2.4.4.4 RTOS Library

There are several example applications for Windows CE and VxWorks on the RTOS side which use the RTOS library for communication with Windows.

2.5 OEM Files

Some parts of the VmfWin product will also be part of the final RTOSWin product. These are described in the sections below

2.5.1 Windows runtime files

The Windows runtime files are mandatory to run the RTOS VM.

a) Uploader Utility etc. (…\Bin\Windows\x86)
   This utility is required for starting and stopping the RTOS. The Uploader DLL also provides the rtoslib functions.
b) Drivers (…\Bin\Windows\Drivers)
   3 drivers are required to run the RTOS VM. The Network driver RtosVnet.sys provides a virtual network connection between Windows and the RTOS. The RTOS driver RtosDrv.sys is the central driver for managing the RTOS VM. The RtosPnp driver and the Rtos-Inf files are together the main parts of the RTOS Device Management.

2.5.2 SDK for the RTOS Library

(…\SDK\Defs)
These files are required when an Windows application shall be written that wants to use the rtoslib.

2.5.3 Windows demo runtime

In case a demo version of the RTOSWin product shall be provided a demo version of the RTOS driver can be used (located in …\Bin\Windows\x64\Drivers\Eval). (64 Bit)
…\Bin\Windows\x86\ Drivers.Win7\Eval (32 Bit)
If this driver is used instead of the full version the following restrictions will be activated:

– After 30 minutes the RTOS stops for 30 seconds.
– 30 days after starting the RTOS operation the first time the system will not be able to be started at all.

2.5.4 Additional OEM files

Below the …\SDK\OemSetup directory additional files that may be provided by the RTOSWin product are located.

2.5.4.1 OEM Setup

Currently an (pseudo) example setup batch file is provided. In case an RTOSWin customer wants to create his own setup he can find here:
– how to install the RTOS VM Windows device drivers
– how to install the RTOS VM Windows services and Uploader tool
– what environment is necessary to run the RTOS VM (registry settings)
– how the memory configuration has to be done
– …

2.6 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

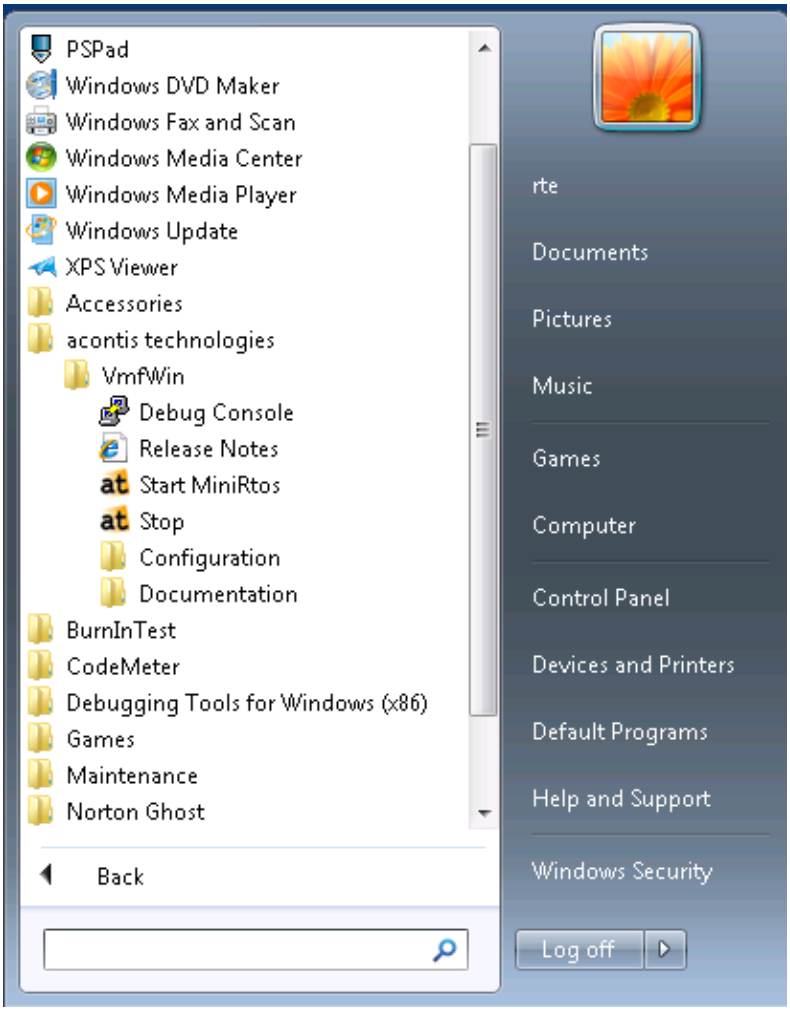
As result the file MiniRtos.bin will be generated below a newly created obj sub-directory.
This file can be loaded and started by the RTOS-VM Uploader tool.

2.7 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:

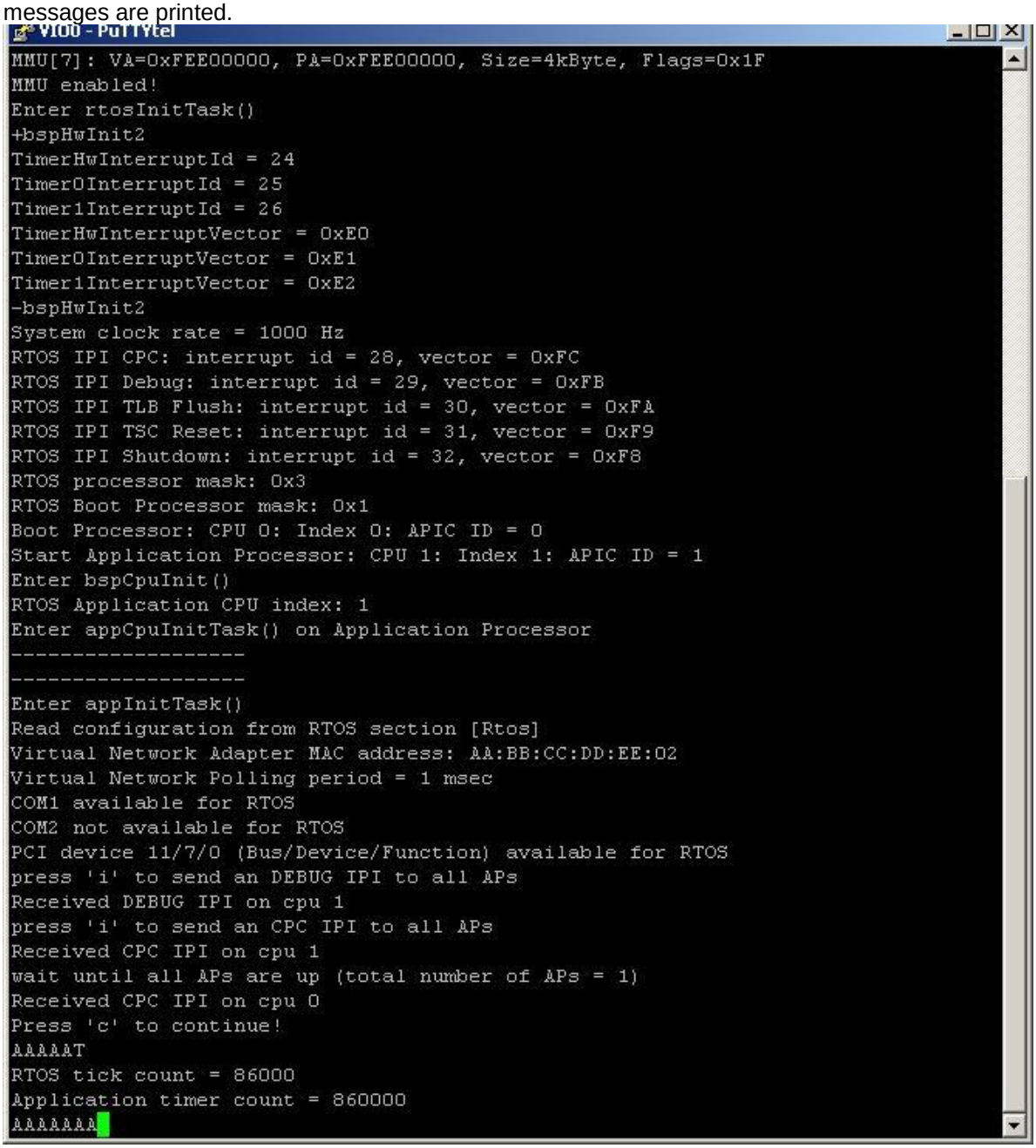
- rtos.config RTOS configuration file including information for the Uploader
- device.config RTOS device configuration file
- vmf.bin VMF binary image
- Windows runtime environment for the RTOS-VM: The Uploader tool, the RTOS service and the RTOS control application.

The Mini RTOS can then be started with the following command line:
RtosUpload.exe MiniRtos.bin or by using the shortcut in the Windows program menu:



2.8 Debug Console

The Virtual Machine Framework provides a virtual I/O channel for transferring data between Windows and the RTOS (similar to a serial line interface).
The virtual I/O channel may be used as debug console.
The Mini RTOS example used this channel to print out debug information and to wait for some user input.
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.



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. Details can be found in the RTOS VM User Manual.

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 fixed 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 value of MemoryStartAddress can be found in the rtos configuration file.
The value of VmfAnchorOffset will be taken from the VMF_OSIMAGE_INFO structure, which should be compiled into the OS binary file. If a file does not contain the structure the anchor offset will be taken from the configuration file.

The size of the data area, at which pvFmwkData points, is fixed. 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. This file itself includes rteOs.h, containing data types used by the framework functions, and rteError.h, containing error definitions.

Example:
```
#include <vmfInterface.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.

- 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 fixed 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 value of MemoryStartAddress can be found in the rtos configuration file.
The value of VmfAnchorOffset will be taken from the VMF_OSIMAGE_INFO structure, which should be compiled into the OS binary file. If a file does not contain the structure the anchor offset will be taken from the 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>

/*****************************************************************************
*                                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: `aKernelMapTable[]`
➔ The anchor descriptor (see chapter 5.2) contains the aKernelMapTable[] member, 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) including information about executable / writable / cached
- 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.

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. Superseded by VMF_KERNELMAP_SHM
VMF_KERNELMAP_VNET	→ Shared memory area for the virtual network
VMF_KERNELMAP_INTERRUPT_PROCESSOR	→ Local APIC memory
VMF_KERNELMAP_INTERRUPT_IOAPIC	→ I/O APIC memory
VMF_KERNELMAP_PROCESSORBOOTCODE	→ Memory area with processor boot code
VMF_KERNELMAP_TIMESTAMP_HPET	→ HPET memory area
VMF_KERNELMAP_SHM	→ Shared memory area(s)

Besides the type there are additional informations (executable / writable / cached) included:

VMF_KERNELMAP_NOEXECUTE	→ Not executable / executable
VMF_KERNELMAP_READONLY	→ Read only / read and writable
VMF_KERNELMAP_UNCACHED	→ Uncached / cached

VMF_KERNELMAP_TYPEMASK is a mask to be used for separating the memory type from the additional information.

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 located in `aKernelMapTable[]` 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 ( (VMF_KERNELMAP_TYPEMASK && 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. The RTOS should not run any other task on the same or a lower priority because it might never get CPU time.

6 Virtual Machine Framework Interface

6.1 VMF function call synchronization

Some of the VMF functions are not re-entrant and their calls should be synchronized. Please take care to the synchronization notes in the function group description.

6.2 VMF Versioning

6.2.1 Using OS image information

The VMF_OSIMAGE_INFO contains information about the OS requirements regarding its memory address, required framework version, entry point offset and more.
It can be located anywhere within the OS image file, but a position at the beginning should be preferred to minimize search time.

```
typedef struct _VMF_OSIMAGE_INFO
{
    UINT8  aSignature[16];          /* must be VMF_OSIMAGE_INFO_SIGNATURE_DATA */

    UINT32 dwSize;                  /* must be sizeof(VMF_OSIMAGE_INFO) */
    UINT32 dwProductVersion;        /* must be VMF_PRODUCT_VERS_NO */
    UINT32 dwOsVersion;             /* OS version number */
    UINT32 dwFmwkVersionRequired;   /* must be VMF_FMWK_VERSION */

    UINT32 dwFmwkAnchorOffset;      /* offset from config file entry "MemoryStartAddress" to
                                       VMF anchor desc */
    UINT32 dwFmwkDataOffset;        /* offset from config file entry "MemoryStartAddress" to
                                       VMF data desc */
    UINT32 dwEntryPointOffset;      /* offset from config file entry "MemoryStartAddress" to
                                       entry point */
    UINT32 dwImageLoadOffset;       /* offset from config file entry "MemoryStartAddress" to
                                       OS image load address */

    UINT64 qwImageLoadAddress;      /* image load address - will be used to verify config file
                                       entry "MemoryStartAddress" */
    UINT64 qwMemoryMinimumSize;     /* minimum RAM requirement - will be used to verify config
                                       file entry "MemorySize" */

    UINT32 aFlags[4];               /* see VMF_OSIMAGE_INFO_FLAG_... */
    CHAR   aOsName[16];             /* can be any string – last character must be '\0' */

    VMF_OSIMAGE_RESOURCE_REQUEST aResourceRequests[VMF_OSIMAGE_INFO_RESOURCE_REQUEST_MAX];
} VMF_PACKED(1) VMF_OSIMAGE_INFO, *PVMF_OSIMAGE_INFO;
```

Flags:

- VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_IMAGELOADOFFSET,
  VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_ENTRYPOINTOFFSET,
  VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_FMWKVERSION,
  VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_IMAGELOADADDRESS,
  VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_MEMORYMINIMUMSIZE
  Setting one of these flags means that the corresponding value will be used as OS image file offset to the real value.
  Example:
  - aFlags[0] contains VMF_OSIMAGE_INFO_FLAG_0_ASOFFSET_FMWKVERSION
  - dwFmwkVersionRequired contains a value of 132
  This causes the required framework version will be read from OS image file at offset 132.
- VMF_OSIMAGE_INFO_FLAG_1_MULTISMP
  When aFlags[1] contains this flag the OS can handle multiple SMP configuration – for example Windows using processor mask 0x3 and RTOS 0xC.
  For such a configuration the OS must be capable to drain off some interrupts.

Special values:

- VMF_OSIMAGE_INFO_FMWKDATAOFFSET_UNKNOWN
  Can be used if the framework data offset can not be determined.
- VMF_OSIMAGE_INFO_IMAGELOADADDRESS_UNKNOWN
  Can be used if the image load address can not be determined.
- VMF_OSIMAGE_INFO_IMAGELOADADDRESS_RELOCATABLE
  Must be used if the image is relocatable.
- VMF_OSIMAGE_INFO_IMAGEMINIMUMRAM_UNKNOWN
  Can be used if the minimum ram requirement can not be determined.

The resource request structure is currently reserved for further use.

6.2.2 Without OS image information

The four bytes of a version are representing MAJOR.MINOR.SERVICEPACK.BUILD number.

It is possible to replace the VMF binary with a newer version – for example to increase functionality. To detect incompatibilities the Uploader compares the version of the framework binary to be loaded and the version required by the OS image to be started with the version expected.

The version expected is VMF_FMWK_VERSION, which is defined in vmfInterface.h.

- Major number is required to be equal.
- Minor number of VMF binary must be greater or equal to the expected one
- Minor number of OS binary must be less or equal to the expected one

The Uploader will read four bytes from the OS image file at an offset given by the config file. This offset is defined in the OS section using the name "VmfVersionOffset".

Example:

```
[Rtos]
        ; …
        "VmfVersionOffset"=dword:80      ; remind: value is hex!
        ; …
```

For compatibility this check can be disabled using an offset value of FFFFFFFF.

6.3 VMF anchor descriptor

The VMF anchor descriptor is located at a fixed address. This address must be well known by the RTOS. The VMF anchor descriptor contains several important values which are needed prior to be able to booting the RTOS and using the VMF functions.

```
typedef struct _VMF_ANCHOR_DESC
{
    UINT32 dwSignature;
    UINT32 dwFmwkAddrPhys;
    UINT32 dwOsId;
    UINT32 dwKernelMapEntries;
    VMF_KERNELMAP_ENTRY aKernelMapTable[VMF_KERNELMAP_MAX_ENTRY_COUNT];
} VMF_PACKED(1) VMF_ANCHOR_DESC;
```

Description

dwSignature	Validation signature (value VMF_ANCHOR_SIGNATURE)
dwFmwkAddrPhys	Physical address where the VMF binary image is located
dwOsId	OS ID (0 = first OS, 1 = second OS, …)
dwKernelMapEntries	Number of entries in the aKernelMapTable[] array
aKernelMapTable[]	Memory mapping information (see section 6.2)

6.4 Kernel memory mapping descriptor

The kernel memory mapping descriptor contains information about memory blocks which have to be mapped into the VMF memory space. The VMF functions require access to these memory areas.

```
typedef struct
{
    UINT32 dwPhysAddress;
    UINT32 dwSize;
    UINT32 dwType;
    UINT32 dwVirtAddress;
} VMF_PACKED(1) VMF_KERNELMAP_ENTRY;
```

Description

dwPhysAddress	Physical address of the memory area
dwSize	Size of the memory area in bytes
dwType	Memory type bit field. According to the memory type different MMU settings have to be used. Bit31: No execute (VMF_KERNELMAP_NOEXECUTE) Bit30: Read only (VMF_KERNELMAP_READONLY) Bit29: Uncached (VMF_KERNELMAP_UNCACHED) Bit28-8: Reserved Bit7-0: Type (VMF_KERNELMAP_TYPEMASK) The following type definitions exist: VMF_KERNELMAP_RTOSMEMORY → RTOS memory area VMF_KERNELMAP_FRAMEWORK → Framework binary image memory area VMF_KERNELMAP_INTERNALSHM – omitted → Internal shared memory area VMF_KERNELMAP_USERSHM → User shared memory area – superseded by VMF_KERNELMAP_SHM VMF_KERNELMAP_VNET → Virtual Network shared memory area – omitted VMF_KERNELMAP_INTERRUPT_PROCESSOR → Local Apic interrupt controller memory VMF_KERNELMAP_INTERRUPT_IOAPIC → I/O Apic interrupt controller memory VMF_KERNELMAP_PROCESSORBOOTCODE → Processor boot code memory area VMF_KERNELMAP_TIMESTAMP_HPET → High Performance Event Timer memory area VMF_KERNELMAP_SHM → Extended shared memory
dwVirtAddress	Virtual address where the memory is mapped to (has to be set by the RTOS)

6.5 Unavailable, unsupported functions

The following functions are described in this manual but currently either not available or not fully supported.

- vmfDeviceResourceDmaGet()
- vmfDeviceResourceInterruptGet()
- vmfDeviceResourcePortGet()
- vmfDeviceResourceMemoryGet()
- vmfConfigGetFirstValueA()
- vmfConfigGetNextValueA()
- vmfInByte()
- vmfOutByte()
- vmfInWord()
- vmfOutWord()
- vmfInDword()
- vmfOutDword()
- vmfEventGetState()
- vmfEventShowA()
- vmfEventShowW()
- vmfShmGetInfoW()
- vmfMessageBoxShowA()
- vmfMessageBoxShowW()
- vmfProcessorConfigShowA()
- vmfProcessorConfigShowW()
- vmfIoApicConfigShowA()
- vmfIoApicConfigShowW()
- vmfDeviceConfigShowA()
- vmfDeviceConfigShowW()
- vmfTimeSyncIsMaster()
- vmfTimeSyncDisable()
- vmfTimeSyncShowA()
- vmfTimeSyncShowW()
- vmfTimeSyncConvertTime()
- vmfTimzezoneSyncIsMaster()
- vmfTimzezoneSyncGetMaster()
- vmfTimzezoneSyncSetMaster()
- vmfTimzezoneSyncDisable()
- vmfTimezoneSyncGetMasterTimezoneA()
- vmfTimezoneSyncGetMasterTimezoneW()
- vmfTimezoneSyncGetOsTimezoneA()
- vmfTimezoneSyncGetOsTimezoneW()
- vmfTimezoneSyncSetOsTimezoneA()
- vmfTimezoneSyncSetOsTimezoneW()
- vmfTimezoneSyncShowA()
- vmfTimezoneSyncShowW()
- vmfOsGetInfoW()
- vmfConfigRegKeyOpenW()
- vmfConfigRegKeyEnumerateW()
- vmfConfigRegValueEnumerateW()

6.6 Core interface

6.6.1 vmfCorePreinit

This function has been renamed to “vmfCoreLoad”.

6.6.2 vmfCoreLoad

Initialize globally the VMF.

```
UINT32 vmfCoreLoad(
    VOID*       pvFmwkConfig,
    UINT32      dwFmwkConfigSize
);
```

Parameter

pvFmwkConfig

[in] Pointer to the configuration area.

dwFmwkConfigSize

[in] Size of the configuration area.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

– This function should be called once by the VMF-manager, typically from the RtosDrv under Windows.

6.6.3 vmfCoreUnload

De-Initialize globally the VMF.

```
UINT32 vmfCoreUnload(
    VOID
);
```

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

– This function should be called once by the VMF-manager, typically from the RtosDrv under Windows.

6.6.4 vmfCoreInit

Initialize the RTOS specific part of the VMF for the actual RTOS.

```
UINT32 vmfCoreInit (
    VMF_ANCHOR_DESC*          pAnchorDesc,
    VMF_KERNELMAP_ENTRY*      pKernelMapTable,
    INT32                     nEntries
);
```

Parameter

pAnchorDesc

[in] pointer to the anchor descriptor.

pKernelMapTable

[in] pointer to a buffer containing a kernel mapping table, filled with the virtual address information.

nEntries

[in] number of entries in the kernel mapping table.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

– This function should be call once by every RTOS, prior calling any other VMF function. The kernel mapping table should be based on the table placed in the anchor descriptor and the virtual address informations provided by the RTOS.

6.6.5 vmfCoreHwInit

Initialize the hardware specific part of the VMF for the actual RTOS and the actual processor.

```
UINT32 vmfCoreHwInit (
    VOID
);
```

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function should be called by every RTOS on each started processor (including the boot processor).
Spurious- and error- interrupt handler have to be initialized before calling this function.
Interrupts must remain closed on the processor until this function was called.

6.6.6 vmfCoreGetKernelMapTable

Get the kernel mapping table for the actual RTOS.
This function is deprecated, use the mapping table stored in the anchor descriptor instead (see chapter 6.1 and 6.4).

```
UINT32 vmfCoreGetKernelMapTable (
    VMF_ANCHOR_DESC*            pAnchorDesc,
    VMF_KERNELMAP_ENTRY*        pKernelMapTable,
    UINT32                      dwFreeEntries,
    UINT32*                     pdwUsedEntries
);
```

Parameter

pAnchorDesc
[in] pointer to the anchor descriptor.
pKernelMapTable
[in/out] pointer to a buffer receiving the kernel mapping table.
dFreeEntries
[in] number of entries which can be copied into the buffer.
pdwUsedEntries
[out] number of entries copied into the buffer.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.6.7 vmfCoreRelocate

Relocate the VMF.

```
UINT32 vmfCoreRelocate (
    VMF_KERNELMAP_ENTRY*        pKernelMapTable,
    INT32                       nEntries
);
```

Parameter

pKernelMapTable
[in] Pointer to the new kernel map table.
dwFmwkConfigSize
[in] Number of entries in the new kernel map table.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

– This function should be called by the VMF-manager, typically from the RtosDrv under Windows.

6.6.8 vmfCoreHandle

Callback routine for OS on certain events.

```
UINT32 vmfCoreHandle (
   UINT32      dwTypeToHandle,
   UINT32      dwParam
);
```

Parameter

dwTypeToHandle

[in] Type of the event to be handled:
- VMF_CORE_HANDLE_BSOD

dwParam

[in] Parameter depending on the type to be handled.
- VMF_CORE_HANDLE_BSOD: Unused

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function should be called by an OS being able to host other OS for handling bluescreens.
Additional events may be handled by this function in future.

6.7 OS switching

6.7.1 vmfTunnelSwitch

Execute a switch job in the tunnel context.

```
UINT32 vmfTunnelSwitch (
    UINT32      dwOsId,
    UINT32      dwJobId,
    UINT32      dwParam
);
```

Parameter

dwOsId

[in] ID of the targeted operating system.

dwJobId

[in] ID of the job to do with the targeted operating system.

dwParam

[in] parameter for the selected job.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

dwParam is used according the dwJobId as following:

dwJobId: VMF_TUNNEL_JOB_ID_BOOT	-> dwParam: EntryPoint
dwJobId: VMF_TUNNEL_JOB_ID_SWITCH	-> dwParam: Unused
dwJobId: VMF_TUNNEL_JOB_ID_RETURN	-> dwParam: Unused

Deprecated:
dwJobId: VMF_TUNNEL_JOB_ID_SWITCHWITHIRQ
Use vmfInterruptIsr (dwVector) instead.

6.8 Multiprocessor management

Synchronization notes:
The functions vmfProcessorStart and vmfProcessorStop use shared informations and their calls must be synchronized against themselves and against each other.

6.8.1 vmfProcessorGetMaskAll

Get mask of all RTOS processors.

```
UINT32 vmfProcessorGetMaskAll (
    UINT32*     pdwMaskAll,
);
```

Parameter

pwdMaskAll

[out] pointer to the variable receiving the processor mask.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.2 vmfProcessorGetMaskCurrent

Get mask of current processor.

```
UINT32 vmfProcessorGetMaskCurrent (
    UINT32*     pdwMaskCurrent,
);
```

Parameter

pwdMaskCurrent

[out] pointer to the variable receiving the processor mask.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.3 vmfProcessorGetMaskShared

Get mask of all rtos processors shared with an other OS.

```
UINT32 vmfProcessorGetMaskShared (
    UINT32*     pdwMaskShared,
);
```

Parameter

pwdMaskShared

[out] pointer to the variable receiving the processor mask.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.4 vmfProcessorStart

Starts the processor specified by the processor mask.

```
UINT32 vmfProcessorStart (
    UINT32      dwProcessorMask,
    UINT32      dwEntryPoint
);
```

Parameter

dwProcessorMask

[in] processor mask specifying the processor to start.

dwEntryPoint

[in] physical address of the entry point for the processor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.5 vmfProcessorStop

Stops the processor specified by processor mask.

```
UINT32 vmfProcessorStop (
    UINT32      dwProcessorMask
);
```

Parameter

dwProcessorMask

[in] processor mask specifying the processor to stop.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.6 vmfProcessorGetLocalApicId

Get local APIC ID of the specified processor.

```
UINT32 vmfProcessorGetLocalApicId (
    UINT32      dwProcessorMask,
    UINT32*     pdwLocalApicId
);
```

Parameter

dwProcessorMask

[in] processor mask specifying the processor to get the local APIC ID from.

pdwLocalApicId

[out] pointer to the variable receiving the local APIC ID.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.8.7 vmfProcessorGetCpuId

Get informations about the current processor.

```
UINT32 vmfProcessorGetCpuId (
    UINT32      aCpuInfo[4],
    UINT32      dwInfoType,
    UINT32      dwSelector
);
```

Parameter

aCpuInfo

[out] aCpuInfo contains the results where aCpuInfo[0] = EAX, aCpuInfo[1] = EBX, aCpuInfo[2] = ECX and aCpuInfo[3] = EDX.

dwInfoType

[in] EAX will be set to dwInfoType before executing cpuid.

dwSelector

[in] ECX will be set to dwSelector before executing cpuid

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Please read processor reference for detailed information about CPUID instruction.

6.8.8 vmfProcessorLock

This function locks the specified host processor(s) until vmfProcessorUnlock is called.

```
UINT32 vmfProcessorLock (
    UINT32              dwProcessorMask,
    VMF_HANDLE          *phLock
);
```

Parameter

dwProcessorMask

[in] The mask of the processor(s) to be locked or 0 to autoselect a processor.
Autoselect will lock all host processor(s).
The mask only allows host processors – any other value will return parameter error.

phLock

[out] Pointer to a handle which is required for calling vmfProcessorUnlock.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

To use this functionality COMM interrupt mode has to be enabled at least for host (Windows) side.

- RtosDrv has to be configured to use an interrupt:
- [Host] “CommInterruptMode” must be set to 1.

Please refer to “RtosVM-UserManual.pdf” chapter “OS communication” for details.

6.8.9 vmfProcessorUnlock

This function unlocks a processor previously locked using vmfProcessorLock.

```
UINT32 vmfProcessorUnlock (
   VMF_HANDLE       *phLock
);
```

Parameter

phLock

[in] Pointer to a handle received from call to vmfProcessorLock.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

See comment section of vmfProcessorLock for details.

6.9 Virtual Machine Device Management

The virtual machine device management is responsible for managing all RTOS devices.
Every RTOS device is represented using a unique device name. The name for a physical device is set according to the name the Windows Device Manager is using.
Currently the following device types exist:

- PCI and PCIe physical devices
- Other physical devices
  - ISA bus devices
  - legacy devices (e.g. serial COM interface)
- VMF defined virtual devices
  - The local APIC timer (device name = “RTOS Local Apic Timer”)
  - Virtual network driver (device name = “RTOS Vnet”)
  - Virtual timer 0 (device name = “RTOS Timer0”)
  - Virtual timer 1 (device name = “RTOS Timer1”)
- Additional user defined virtual devices
  If the RTOS needs to generate virtual interrupts a virtual device is required which generates such an interrupt.
  This is necessary for example in case Inter-Processor-Interrupts (IPIs) shall be generated to synchronize multi processor systems.

All physical devices and the VMF defined virtual devices will be automatically created. To enable a physical device the RtosPnp driver has to be installed to assign the physical device to the RTOS.
Additional user defined virtual devices have to be manually defined in the device.config file.
In the following example a virtual device (device index 5) representing an IPI is defined:
The section [InterruptSource#] defines the properties of one specific interrupt source (related to one device).
The section [InterruptTarget#] defines the target properties of one specific interrupt source (e.g. the which CPU cores will be interrupted.
In the following example the entry [Device5] defines the properties for a virtual device representing an IPI. The interrupt source and target properties for this virtual device are stored in the sections [InterruptSource5] and [InterruptTarget5].

```
[Device5]
    "RtosDeviceType"=dword:1                ; 01 = virtual device
    "RtosDeviceName"="RTOS IPI Device"      ; name of the virtual device
    "OsId"=dword:FFFFFFFE                   ; 0xFFFFFFFE = autodetect
    "InterruptSourceList"=multi_sz:"5"      ; refers to [InterruptSource5]

[InterruptSource5]
    "InterruptSourceType"=dword:3           ; 03 = IPI interrupt
    "InterruptTargetList"=multi_sz:"5"      ; refers to [InterruptTarget5]
    "InterruptId"=dword:FFFFFFFE            ; generate irq id automatically

[InterruptTarget5]
    "InterruptTargetAddressType"=dword:0    ; 0: target address = bit mask of
                                            ;    CPUs receiving this IPI
    "InterruptTargetAddress"=dword:6        ; Binary 0110 = processor 1 and 2
    "InterruptProcessorVector"=dword:FC     ; interrupt vector
```

6.9.1 Interrupt IDs

One single device (physical device as well as virtual devices) may require one or multiple interrupts. Every single interrupt is identified by a unique interrupt ID.
The interrupt IDs are determined automatically by the RTOS VM. It is not predictable which interrupt IDs will belong to which physical or virtual device.
The only assumption that can be made is that the interrupt IDs on a given system will not be changed by the RTOS VM if:

- The Windows interrupt resources are not changed (if no driver updates or hardware changes are made, Windows usually will not change interrupt resources)
- The RTOS VM physical and virtual devices are not changed

After a physical device is assigned to the RTOS the first time one cannot assume that existing interrupt IDs will not be changed.

6.9.2 vmfDeviceIsForRtosA

Determine whether the specified device and the device interrupt index is assigned to the RTOS or not.

```
UINT32 vmfDeviceIsForRtosA (
    CHAR*       szName,
    UINT32      dwInterruptIndex,
    BOOL*       pbIsForRtos
);
```

Parameter

szName

[in] ascii device name (as shown in the Windows Device Manager).

dwInterruptIndex

[in] interrupt index in case the device generates more than one interrupt.

pbIsForRtos

[out] TRUE if the device is for Rtos or FALSE if not.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.9.3 vmfDeviceIsForRtosW

Determine whether the specified device and the device interrupt index is assigned to the RTOS or not.

```
UINT32 vmfDeviceIsForRtosW (
    WCHAR*      wszName,
    UINT32      dwInterruptIndex,
    BOOL*       pbIsForRtos
);
```

Parameter

wszName

[in] unicode device name (as shown in the Windows Device Manager).

dwInterruptIndex

[in] interrupt index in case the device generates more than one interrupt.

pbIsForRtos

[out] TRUE if the device is for Rtos or FALSE if not.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.9.4 vmfDevicePciIsForRtos

Determine whether the specified PCI device is assigned to the RTOS or not.

```
UINT32 vmfDevicePciIsForRtos (
    INT32       nBus,
    INT32       nDevice,
    INT32       nFunction,
    UINT32      dwInterruptIndex
    BOOL*       pbIsForRtos
);
```

Parameter
nBus
[in] PCI bus number.
nDevice
[in] PCI device number.
nFunction
[in] PCI function number.
dwInterruptIndex
[in] interrupt index in case the device generates more than one interrupt.
pbIsForRtos
[out] TRUE if the device is for Rtos or FALSE if not.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.9.5 vmfDeviceIoIsForRtos
Determine whether the specified device is assigned to the RTOS or not.

```
UINT32 vmfDeviceIoIsForRtos (
    UINT32      dwIoPort
    BOOL*       pbIsForRtos
);
```

Parameter
dwIoPort
[in] IO port used to access the device.
pbIsForRtos
[out] TRUE if the device is for Rtos or FALSE if not.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.9.6 vmfDeviceInterruptIdFromNameA
Determine the interrupt id of the specified device.

```
UINT32 vmfDeviceInterruptIdFromNameA (
    CHAR*       szName,
    UINT32      dwInterruptIndex,
    UINT32*     pdwInterruptId
);
```

Parameter
szName
[in] ascii device name (as shown in the Windows Device Manager).
dwInterruptIndex
[in] interrupt index in case the device generates more than one interrupt.
pdwInterruptId
[out] interrupt id of the device.

Return
RTE_SUCCESS when a valid interrupt Id was returned or an error-code on failure.

Comment

–

6.9.7 vmfDeviceInterruptIdFromNameW

Determine the interrupt id of the specified device.

```
UINT32 vmfDeviceInterruptIdFromNameW (
    WCHAR*    wszName,
    UINT32    dwInterruptIndex,
    UINT32*   pdwInterruptId
);
```

Parameter

wszName
[in] unicode device name (as shown in the Windows Device Manager).

dwInterruptIndex
[in] interrupt index in case the device generates more than one interrupt.

pdwInterruptId
[out] interrupt id of the device.

Return

RTE_SUCCESS when a valid interrupt Id was returned or an error-code on failure.

Comment

–

6.9.8 vmfDeviceResourceDmaGet

Get DMA informations for the selected device.

```
UINT32 vmfDeviceResourceDmaGet (
    UINT32              dwDeviceId,
    UINT32              dwIndex,
    PVMF_DEVICE_DMA     pDma
);
```

Parameter

dwDeviceId
[in] ID of the device to get the informations from.

dwIndex
[in] index in the DMA information table.

pDma
[out] pointer to the buffer receiving the DMA informations.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

VMF_DEVICE_DMA.dwSize must be initialized with sizeof(VMF_DEVICE_DMA).

6.9.9 vmfDeviceResourceInterruptGet

```
UINT32 vmfDeviceResourceInterruptGet (
    UINT32                      dwDeviceId,
    UINT32                      dwIndex,
    PVMF_DEVICE_INTERRUPT       pInterrupt
);
```

Parameter

dwDeviceId
[in] ID of the device to get the informations from.

dwIndex
[in] index in the interrupt information table.

pInterrupt
[out] pointer to the buffer receiving the interrupt informations.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

VMF_DEVICE_INTERRUPT.dwSize must be initialized with sizeof(VMF_DEVICE_INTERRUPT).

6.9.10 vmfDeviceResourceInterruptSet

```
UINT32 vmfDeviceResourceInterruptSet (
    UINT32                      dwDeviceId,
    UINT32                      dwIndex,
    PVMF_DEVICE_INTERRUPT       pInterrupt
);
```

Parameter

dwDeviceId
[in] ID of the device to get the informations from.

dwIndex
[in] index in the interrupt information table.

pInterrupt
[in] pointer to the buffer containing the interrupt informations.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

VMF_DEVICE_INTERRUPT.dwSize must be initialized with sizeof(VMF_DEVICE_INTERRUPT).
This function can be used to update an interrupt vector. It can be called by HOST or by an ExclusiveCore RTOS. The interrupt to be modified must be disabled when calling this function.
It is recommended to call vmfDeviceResourceInterruptGet() first, modify the vector and then call vmfDeviceResourceInterruptSet().

6.9.11 vmfDeviceResourceIoPortGet

```
UINT32 vmfDeviceResourceIoPortGet (
    UINT32                      dwDeviceId,
    UINT32                      dwIndex,
    PVMF_DEVICE_IOPORT          pIoPort
```

);

Parameter

dwDeviceId
[in] ID of the device to get the informations from.
dwIndex
[in] index in the IO information table.
pIoPort
[out] pointer to the buffer receiving the IO informations.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

VMF_DEVICE_IOPORT.dwSize must be initialized with sizeof(VMF_DEVICE_IOPORT).

6.9.12 vmfDeviceResourceMemoryGet

```
UINT32 vmfDeviceResourceMemoryGet (
    UINT32                  dwDeviceId,
    UINT32                  dwIndex,
    PVMF_DEVICE_MEMORY      pMemory
);
```

Parameter

dwDeviceId
[in] ID of the device to get the informations from.

dwIndex
[in] index in the memory information table.

pMemory
[out] pointer to the buffer receiving the memory informations.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

VMF_DEVICE_MEMORY.dwSize must be initialized with sizeof(VMF_DEVICE_MEMORY).

6.9.13 vmfDeviceGetIdByPci

```
UINT32 vmfDeviceGetIdByPci (
    UINT32      dwBus,
    UINT32      dwDevice,
    UINT32      dwFunction,
    UINT32*     pdwDeviceId
);
```

Parameter

dwBus
[in] High word: PCI segment.
Low word: PCI bus number.

dwDevice
[in] PCI device number.

dwFunction
[in] PCI function number.

pdwDeviceId
[out] ID of the device – required for example to query resources.

Return

RTE_SUCCESS if the device is available for the calling OS and an error-code if not or on failure.

Comment

–

6.10 Interrupts

6.10.1 vmfInterruptLock

```
UINT32 vmfInterruptLock (
    VOID
);
```

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.2 vmfInterruptUnlock

```
UINT32  vmfInterruptUnlock (
    VOID
);
```

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.3 vmfInterruptIsApicUsed

```
UINT32 vmfInterruptIsApicUsed (
    BOOL*       pbIsApic
);
```

Parameter
pbIsApic
[out] TRUE if Apic is used, FALSE otherwise.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.4 vmfInterruptVectorToId

Converts an interrupt vector to an interrupt id.

```
UINT32 vmfInterruptVectorToId (
    UINT32      dwInterruptVector,
    UINT32*     pwdInterruptId
);
```

Parameter

dwInterruptVector
[in] interrupt vector.
pdwInterruptId
[out] interrupt id.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.10.5 vmfInterruptIdToVector

Converts an interrupt id to an interrupt vector.
The interrupt id is an internally defined value that should not be evaluated by the RTOS. The interrupt vector is the offset in the processor's vector table. Valid values for the vector are 0 to 254 (0xFE).

```
UINT32 vmfInterruptIdToVector (
    UINT32      dwInterruptId,
    UINT32*     pdwInterruptVector
);
```

Parameter

dwInterruptId
[in] interrupt id.
pdwInterruptVector
[out] interrupt vector.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.10.6 vmfInterruptEnable

```
UINT32 vmfInterruptEnable (
    UINT32      dwInterruptId,
    UINT32      dwProcessorMask,
    UINT32      dwFlags
);
```

Parameter

dwInterruptId
[in] interrupt id.
dwProcessorMask
[in] processor mask.
dwFlags

[in] flags (unused).

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.7 vmfInterruptDisable

```
UINT32 vmfInterruptDisable (
    UINT32      dwInterruptId,
    UINT32      dwProcessorMask,
    UINT32      dwFlags
);
```

Parameter
dwInterruptId
[in] interrupt id.
dwProcessorMask
[in] processor mask.
dwFlags
[in] flags (unused).

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.8 vmfInterruptGenerate

Generate (simulate) an interrupt.

```
UINT32 vmfInterruptGenerate (
    UINT32      dwInterruptId
    UINT32      dwProcessorMask,
    UINT32      dwFlags
);
```

Parameter
dwInterruptId
[in] interrupt id of the interrupt to be generated.
dwProcessorMask
[in] processor mask.
dwFlags
[in] flags (unused).

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
By APIC systems the interrupts are generated using the IPI mechanism.

6.10.9 vmfInterruptEoi

```
UINT32 vmfInterruptEoi (
    UINT32      dwInterruptId,
);
```

Parameter
dwInterruptId
[in] interrupt id.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.10.10 vmfInterruptIsr

Forward SharedCore interrupts.

```
UINT32 vmfInterruptIsr (
    UINT32      dwInterruptVector
);
```

Parameter
dwInterruptVector
[in] interrupt vector of the interrupt to be handled.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
This routine is currently used by Windows only.

6.10.11 How to generate an IPI

This example shows what steps are required to generate a virtual interrupt or an IPI.

6.10.11.1 Device configuration

To generate an IPI a virtual device has to be created.
The following virtual device configuration shows how this can be achieved. The name of the virtual device is “RTOS IPI Device”.
The section [Device#] defines the name and properties of the device. It refers to one or multiple interrupts the device can generate.
The section [InterruptSource#] defines the properties of one specific interrupt source (related to one device).
The section [InterruptTarget#] defines the target properties of one specific interrupt source (e.g. the which CPU cores will be interrupted.
In the following example the entry [Device5] defines the properties for a virtual device representing an IPI. The interrupt source and target properties for this virtual device are stored in the sections [InterruptSource5] and [InterruptTarget5].

```
[Device5]
    "RtosDeviceType"=dword:1                ; 01 = virtual device
    "RtosDeviceName"="RTOS IPI Device"      ; name of the virtual device
    "OsId"=dword:FFFFFFFE                   ; 0xFFFFFFFE = autodetect
    "InterruptSourceList"=multi_sz:"5"      ; refers to [InterruptSource5]

[InterruptSource5]
    "InterruptSourceType"=dword:3           ; 03 = IPI interrupt
    "InterruptTargetList"=multi_sz:"5"      ; refers to [InterruptTarget5]
    "InterruptId"=dword:FFFFFFFE           ; generate irq id automatically

[InterruptTarget5]
    "InterruptTargetAddressType"=dword:0    ; 0: target address = bit mask of
                                            ;    CPUs receiving this IPI
    "InterruptTargetAddress"=dword:6        ; Binary 0110 = processor 1 and 2
    "InterruptProcessorVector"=dword:FC     ; interrupt vector
```

6.10.11.2 Determine the interrupt ID

The following call will determine the interrupt ID:

```
vmfDeviceInterruptIdFromName(“RTOS IPI Device”, 0, &dwInterruptId);
```

6.10.11.3 Generate the IPI (not yet available, has to be done manually by programming the Local APIC)

The following call will generate the IPI:

```
vmfInterruptGenerate(dwInterruptId,…);
```

6.10.12 How to handle interrupts

This example shows what steps are required to handle interrupts.

6.10.12.1 Determine the interrupt ID

For PCI devices the interrupt id can be determined by reading the appropriate entry in the PCI configuration space.
For other devices (e.g. ISA bus devices, legacy devices like the serial interface) and for IPIs the interrupt id has to be determined by the following call:

```
vmfDeviceInterruptIdFromName("RTOS IPI Device", 0, &dwInterruptId);
```

6.10.12.2 Connect the interrupt service routine

In the first step the interrupt vector has to be determined.

```
vmfInterruptIdToVector(dwInterruptId, &dwInterruptVector)
```

Then the appropriate RTOS operating system function has to be called to connect the interrupt service routine with the interrupt vector.

```
intConnect(dwInterruptVector,pfInterruptHandler)
```

The interrupt has to be enabled:

```
vmfInterruptEnable(dwInterruptId,…)
```

When the interrupt is generated the interrupt handle will be called. The last step in the interrupt handler is to call the end-of-interrupt function:

```
vmfInterruptEoi(dwInterruptId)
```

6.11 Timer

The Virtual Machine Framework supports two virtual timers which are based on one single physical hardware timer (typically the Local Apic timer on APIC systems). The physical hardware timer shall not be directly used by the RTOS (the output frequency cannot be directly set, it will be set by the VMF to an unspecified value depending on the output frequency of the virtual timers and the maximum output frequency of the physical timer).
Note: the physical timer has to be enabled as soon as possible. In later versions of the VMF it may be internally used by the VMF itself.

Synchronization notes:
The timer functions use shared informations and their calls must be synchronized against themselves and against each other. The vmfTimerHwIsr function must not be synchronized.

6.11.1 Physical timer initialization

Prior to using the two virtual timers the following steps for the physical timer have to be made:

- Set the minimum and maximum timer output frequency (timer interrupt frequency). If these values are not set some default values will be used.
  – vmfTimerHwSetOutputFreqMin(dwFrequency)
  – vmfTimerHwSetOutputFreqMax(dwFrequency)
- A RTOS interrupt service routine for the physical timer has to be connected to the physical timer interrupt. This routine has to call the framework's physical timer interrupt service routine.
  – vmfTimerHwGetInterruptId(&dwVmfTimerHwInterruptId)
  – vmfInterruptIdToVector(dwVmfTimerHwInterruptId, &dwTimerHwVector)
  – rtosBspIntConnect (dwTimerHwVector, rtosBspTimerHwIsr, 0);
  – rtosBspTimerHwIsr():call vmfTimerHwIsr(dwVmfTimerHwInterruptId)
- The interrupt shall be enabled using the appropriate RTOS function
  – rtosBspIntEnablePIC(dwVmfTimerHwInterruptId)
- The physical timer has to be started
  – vmfTimerHwEnable()

6.11.2 Virtual timer operation

The following steps are required to use the virtual timer 0 - the usage of virtual timer 1 is identical:

- Set the minimum and maximum timer output frequency (timer interrupt frequency). If these values are not set some default values will be used.
  – vmfTimer0SetOutputFreqMin(dwFrequency)
  – vmfTimer0SetOutputFreqMax(dwFrequency)
- A RTOS interrupt service routine for the physical timer has to be connected to the physical timer interrupt. This routine has to call the framework's physical timer interrupt service routine.
  – vmfTimer0GetInterruptId(&dwVmfTimer0InterruptId)
  – vmfInterruptIdToVector(dwVmfTimer0InterruptId, &dwTimer0Vector)
  – rtosBspIntConnect (dwTimer0Vector, rtosTimer0Isr, 0);
- The timer output frequency has to be set to an appropriate value:
  – vmfTimer0SetOutputFreq(1000);
- The interrupt shall be enabled using the appropriate RTOS function
  – rtosBspIntEnablePIC(dwVmfTimer0InterruptId)
- The physical timer has to be started
  – vmfTimer0Enable()

6.11.3 Virtual timer output frequency adjustment

While the virtual timers are running it is possible to adjust the timer's output frequency "on-the-fly" without disabling the timer and setting a new output frequency.
As both virtual timers are based on the same physical timer these timers can neither be adjusted with arbitrary values nor be adjusted independently from each other.

The following example shall show the relationship between the virtual timers:
Basic timer settings:

- Physical timer input frequency = 1.000.000 Hz
- Virtual timer 0 output frequency = 1000 Hz (e.g. required for the RTOS timer)
- Virtual timer 1 output frequency = 10.000 Hz (e.g. required for high-speed controller)

As a result the physical timer's output frequency will be set to 10.000 Hz.
The initial count of the physical timer will be set to a value of 100.
After every physical timer interrupt one virtual timer 1 interrupt will be generated.
After 10 physical timer interrupts one virtual timer 0 interrupt will be generated.

The physical timer input period is 1 microsecond. If the initial count of the physical timer is changed by 1 the output period will be changed by 1 microsecond.

The period of the virtual timer 0 is 10 times higher than the physical timer's period.
The period of the virtual timer 0 therefore can only be adjusted by 10 microseconds.
The granularity of timer 0 is 10.

The period of the virtual timer 1 is identical to the physical timer's period.
The period of the virtual timer 1 therefore can be adjusted by 1 microsecond.
The granularity of timer 1 is 1.

6.11.4 Physical timer frequency

As described in chapter 6.8.3 the physical timer's frequency depends on the settings for the virtual timers.
Therefore the calling sequence should be as follows:

A) set the timer output frequency of the virtual timer 0, vmfTimer0SetOutputFreq
B) set the timer output frequency of the virtual timer 1, vmfTimer1SetOutputFreq
C) Enable the physical timer, vmfTimerHwEnable

As result the physical timer will immediately use the appropriate frequency according to the requirements of the two virtual timers.

Important: If only one single virtual timer is used, the frequency of the other virtual timer should be set to the same frequency!

6.11.5 vmfTimerHwIsr

Interrupt service routine for the physical timer.

```
UINT32 vmfTimerHwIsr(
    UINT32      dwInterruptId
);
```

Parameter

dwInterruptId

[in] interrupt ID of the generated interrupt.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

– This function should be assigned to the interrupt ID using the standard interrupt handling function of the RTOS before calling vmfTimerHwEnable().

6.11.6 vmfTimerHwGetInterruptId

Determine the interrupt ID of the physical timer.

```
UINT32 vmfTimerHwGetInterruptId(
    UINT32*     pdwInterruptId
);
```

Parameter

pdwInterruptId

[out] pointer to the variable receiving the imterrupt ID.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.7 vmfTimerHwEnable

Enable the physical timer.

```
UINT32 vmfTimerHwEnable(
);
```

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to be called as soon as possible after the initialization of the VMF. The physical timer may generate interrupts after this function is called.
The output frequency of the virtual timers should be set prior to enabling the physical timer (see also chapter 6.8.4).

6.11.8 vmfTimerHwDisable

Disable the physical timer.

UINT32 vmfTimerHwDisable(
);

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.9 vmfTimerHwGetInputFreq

Determine the input frequency of the physical timer.

UINT32 vmfTimerHwGetInputFreq(
UINT32* pdwInputFreqHz
);

Parameter
pdwInputFreqHz
[out] Physical timer input frequency in Hertz.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.10 vmfTimerHwGetInitialCount

Determine the initial counter value of the physical timer.

UINT32 vmfTimerHwGetInitialCount(
UINT32* pdwInitialCount
);

Parameter
pdwInitialCount
[out] Physical timer initial counter value.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.11 vmfTimerHwSetInitialCount

Sets the initial counter value of the physical timer.

UINT32 vmfTimerHwSetInitialCount(
UINT32 dwInitialCount,
);

Parameter

dwInitialCount
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.12 vmfTimerHwModifyInitialCount

Modify the hardware timer initial count value.

UINT32 vmfTimerXModifyInitialCount (
INT32 nDelta,
);

Parameter

nDelta
[in] this value will be added on top of the timer's current initial count value.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The first call to this function enables the controlled mode for the hardware timer. The controlled mode is disabled after a call with the delta parameter set to zero.
In the controlled mode, the initial counter is set every interrupt to a corrected value compensing the interrupt jitter. The function vmfTimerHwGetInitialCount returns always the currently loaded initial count.

6.11.13 vmfTimerHwGetCurrentCount

Determine the current counter value of the physical timer.

UINT32 vmfTimerHwGetCurrentCount(
UINT32* pdwCurrentCount
);

Parameter

pdwCurrentCount
[out] Physical timer current counter value.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.14 vmfTimerHwGetCurrentPosition

Determine the current position between the last and the next timer HW interrupt.

```
UINT32 vmfTimerHwGetCurrentPosition(
    UINT32*     pdwCountSinceInterrupt,
    UINT32*     pdwCountBeforeInterrupt
);
```

Parameter

pdwCountSinceInterrupt

[out] Physical timer tick count since last interrupt.

pdwCountBeforeInterrupt

[out] Physical timer tick count before next interrupt.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.15 vmfTimerHwGetOutputFreq

Get hardware timer output frequency.

```
UINT32 vmfTimerHwGetOutputFreq(
    UINT32*     pdwOutputFreqHz
);
```

Parameter

pdwOutputFreqHz

[out] Physical timer output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.16 vmfTimerHwSetOutputFreqMin

Set minimal hardware timer output frequency.

```
UINT32 vmfTimerHwSetOutputFreqMin(
    UINT32      dwFrequencyHz
);
```

Parameter

dwFrequencyHz

[in] the timer's minimal output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.17 vmfTimerHwGetOutputFreqMin

Get minimal hardware timer output frequency.

```
UINT32 vmfTimerHwGetOutputFreqMin(
    UINT32*     pdwOutputFreqMinHz
);
```

Parameter

pdwOutputFreqMinHz

[out] Physical timer min. frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.18 vmfTimerHwSetOutputFreqMax

Set maximal hardware timer output frequency.

```
UINT32 vmfTimerHwSetOutputFreqMax(
    UINT32      dwFrequencyHz
);
```

Parameter

dwFrequencyHz

[in] the timer's maximal output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.19 vmfTimerHwGetOutputFreqMax

Get maximal hardware timer output frequency.

```
UINT32 vmfTimerHwGetOutputFreqMax(
    UINT32*     pdwOutputFreqMaxHz
);
```

Parameter

pdwOutputFreqMaxHz

[out] Physical timer max. frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.20 vmfTimerVirtGetInterruptId

Determine the interrupt ID of the specified virtual timer.

```
UINT32 vmfTimerVirtGetInterruptId(
    INT32       nTimerVirtId,
    UINT32*     pdwInterruptId
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to get the interrupt ID from.

pdwInterruptId

[in] interrupt ID of the virtual timer.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.21 vmfTimerVirtSetOutputFreq

```
UINT32 vmfTimerVirtSetOutputFreq(
    INT32       nTimerVirtId,
    UINT32      dwFrequencyHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to set the output frequency.

dwFrequencyHz

[in] desired frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.22 vmfTimerVirtGetOutputFreq

```
UINT32 vmfTimerVirtGetOutputFreq(
    INT32       nTimerVirtId,
    UINT32*     pdwOutputFreqHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to set the output frequency.

pdwOutputFreqMaxHz

[out] desired max output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.23 vmfTimerVirtGetOutputFreqMin

```
UINT32 vmfTimerVirtGetOutputFreqMin(
    INT32       nTimerVirtId,
    UINT32*     pdwOutputFreqMinHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to get the min output frequency from.

pdwOutputFreqMinHz

[out] min output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.24 vmfTimerVirtGetOutputFreqMax

```
UINT32 vmfTimerVirtGetOutputFreqMax(
    INT32       nTimerVirtId,
    UINT32*     pdwOutputFreqMaxHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to get the maxvc output frequency from.

pdwOutputFreqMaxHz

[out] max output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.25 vmfTimerVirtSetOutputFreqMin

```
UINT32 vmfTimerVirtSetOutputFreqMin(
    INT32       nTimerVirtId,
    UINT32      dwFrequencyHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to set the min output frequency.

dwFrequencyHz

[int] desired min output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.26 vmfTimerVirtSetOutputFreqMax

```
UINT32 vmfTimerVirtSetOutputFreqMax(
    INT32       nTimerVirtId,
    UINT32      dwFrequencyHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to set the max output frequency.

dwFrequencyHz

[int] desired max output frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.27 vmfTimerVirtGetInputFreq

```
UINT32 vmfTimerVirtGetInputFreq(
    INT32       nTimerVirtId,
    UINT32*     pdwInputFreqHz
);
```

Parameter

nTimerVirtId

[in] ID of the virtual timer to get the input frequency from.

pdwInputFreqHz

[out] input frequency in Hertz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.28 vmfTimerVirtGetInitialCount

```
UINT32 vmfTimerVirtGetInitialCount(
    INT32       nTimerVirtId,
    UINT32*     pdwInitialCount
);
```

Parameter

nTimerVirtId
[in] ID of the virtual timer to get initial count from.

pdwInitialCount
[out] initial count.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.29 vmfTimerVirtGetCurrentCount

```
UINT32 vmfTimerVirtGetCurrentCount(
    INT32       nTimerVirtId,
    UINT32*     pdwCurrentCount
);
```

Parameter

nTimerVirtId
[in] ID of the virtual timer to get the current count from.

pdwCurrentCount
[out] current count

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.30 vmfTimerGetCpuFrequency

Gets Cpu Frequency

```
UINT32 vmfTimerGetCpuFrequency(
    UINT32*     pdwCpuFrequencyMhz
);
```

Parameter

pdwCpuFrequencyMhz
[out] Cpu frequency in MHz.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.31 vmfTimerVirtSetInterruptId

```
UINT32 vmfTimerVirtSetInterruptId(
    INT32       nTimerVirtId,
    UINT32      dwInterruptId
);
```

Parameter

nTimerVirtId
[in] ID of the virtual timer to set the interrupt ID.

dwInterruptId
[in] interrupt ID.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.32 vmfTimerVirtXxx
vmfTimer0Xxx
vmfTimer1Xxx

General virtual timer function call mechanism.

```
vmfTimerVirtXxx (
    UINT32      dwTimerId,
    …
);
```

Parameter

dwTimrId
[in] virtual timer ID. Currently two timers are supported (id = 0 or 1).

vmfTimerVirtXxx(0,…) ➔ vmfTimer0Xxx(…)
vmfTimerVirtXxx(1,…) ➔ vmfTimer1Xxx(…)

6.11.33 vmfTimerXGetInterruptId

Get interrupt ID of specified virtual timer.

```
UINT32 vmfTimerXGetInterruptId (
    UINT32      &pdwInterruptId
);
```

Parameter

pdwInterruptId
[out] Interrupt Id.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.34 vmfTimerXEnable

Enable the virtual timer.

UINT32 vmfTimerXEnable (
);

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.35 vmfTimerXDisable

Disable the virtual timer.

UINT32 vmfTimerXDisable (
);

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.36 vmfTimerXGetInputFreq

Determine the input frequency of the virtual timer.

UINT32 vmfTimerXGetInputFreq (
UINT32* pdwInputFreqHz,
);

Parameter
pdwInputFreqHz
[out] Input frequency (in Hz) of the timer.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
The timer's output frequency is determined by taking the timer's input frequency divided by the timer's initial count value: Fout = Fin / InitCount.

6.11.37 vmfTimerXSetOutputFreq

Set the virtual timer output frequency.

```
UINT32 vmfTimerXSetOutputFreq (
    UINT32      dwFrequencyHz,
);
```

Parameter

dwFrequencyHz

[in] the timer's output frequency in Hertz. Within one second *dwFrequency* timer interrupts will be generated.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Important: if only one virtual timer is used the output frequency of the second virtual timer should be set to the same value (see also chapter 6.8.4).

6.11.38 vmfTimerXGetOutputFreq

Determine the output frequency of the virtual timer.

```
UINT32 vmfTimerXGetOutputFreq (
    UINT32*     pdwOutputFreqHz,
);
```

Parameter

pdwOutputFreqHz

[out] Output frequency (in Hz) of the timer.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The timer's output frequency is determined by taking the timer's input frequency divided by the timer's initial count value: Fout = Fin / InitCount.

6.11.39 vmfTimerXSetOutputFreqMin

Set the min. virtual timer output frequency.

```
UINT32 vmfTimerXSetOutputFreqMin (
    UINT32      dwFrequencyHz
);
```

Parameter

dwFrequencyHz

[in] the timer's min. output frequency in Hertz. Within one second *dwFrequency* timer interrupts will be generated.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.40 vmfTimerXGetOutputFreqMin

Determine the min. output frequency of the virtual timer.

```
UINT32 vmfTimerXGetOutputFreqMin (
    UINT32*     pdwOutputFreqMinHz
);
```

Parameter

pdwOutputFreqMinHz

[out] Min. output frequency (in Hz) of the timer.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.41 vmfTimerXSetOutputFreqMax

Set the max. virtual timer output frequency.

```
UINT32 vmfTimerXSetOutputFreqMax (
    UINT32      dwFrequencyHz
);
```

Parameter

dwFrequencyHz

[in] the timer's max. output frequency in Hertz. Within one second *dwFrequency* timer interrupts will be generated.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.42 vmfTimerXGetOutputFreqMax

Determine the max. output frequency of the virtual timer.

```
UINT32 vmfTimerXGetOutputFreqMax (
    UINT32*     pdwOutputFreqMaxHz
);
```

Parameter

pdwOutputFreqMaxHz

[out] Max. output frequency (in Hz) of the timer.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.11.43 vmfTimerXGetInitialCount

Determine the initial counter value of the virtual timer.

```
UINT32 vmfTimerXGetInitialCount (
    UINT32*     pdwInitialCount
);
```

Parameter

pdwInitialCount

[out] Virtual timer initial counter value.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The timer's output frequency is determined by taking the timer's input frequency divided by the timer's initial count value: Fout = Fin / InitCount.

6.11.44 vmfTimerXGetCurrentCount

Determine the current counter value of the virtual timer.

```
UINT32 vmfTimerXGetCurrentCount (
    UINT32*     pdwCurrentCount
);
```

Parameter

pdwCurrentCount

[out] Virtual timer current counter value.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The virtual timer's counter will be loaded with the initial counter value. The initial counter value can be determined using the vmfTimerXGetInitialCount() function.
The will decrement the counter with the timer's input frequency. The timer's input frequency can be determined using the vmfTimerXGetInputFrequency() function.

6.11.45 vmfTimerXGetGranularity

Determine the virtual timer divider granularity.

```
UINT32 vmfTimerXGetGranularity (
    UINT32*     pdwGranularity
);
```

Parameter

pdwGranularity

[out] Virtual timer divider granularity.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The virtual timer's counter will be loaded with the initial counter value. While the timer is enabled the initial counter value can be modified by multiples of the timer's granularity value using the vmfTimerXModifyInitialCount() function.

6.11.46 vmfTimerXModifyInitialCount

Modify the virtual timer initial count value.

```
UINT32 vmfTimerXModifyInitialCount (
    INT32       nDelta
);
```

Parameter

nDelta

[in] this value will be multiplied with the timer divider granularity and added on top of the timer's current initial count value.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The virtual timer's counter will be loaded with the initial counter value. While the timer is enabled the initial counter value can be modified by multiples of the timer's granularity value using the vmfTimerXModifyInitialCount() function.
Both virtual timers are based on the same physical timer. Thus a modification of one of the timers will change the behaviour of the second as well. The timers cannot be adjusted independently.

6.11.47 vmfTimerXIntEnable

Enables virtual timer interrupt of the specified virtual timer.

UINT32 vmfTimerXIntEnable (
);

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.11.48 vmfTimerXIntDisable

Disables virtual timer interrupt of the specified virtual timer.

UINT32 vmfTimerXIntDisable (
);

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.12 Realtime Clock

The Virtual Machine Framework provides a Software RTC. This Soft-RTC will be initialized once when the RTOS is booted and then triggered by the physical hardware timer (see chapter 6.8). The accuracy of the Soft-RTC depends on the accuracy of the physical hardware timer. If better accuracy is required the RTOS Library's time and date synchronization has to be activated (see chapter 8).

Synchronization notes:
The realtime clock functions use shared informations and their calls must be synchronized against themselves and against each other.

6.12.1 vmfRtcGetTime

```
UINT32 vmfRtcGetTime (
    VMF_TIME* pTime
);
```

Parameter

pTime

[in,out] pointer to structure where new system time is stored.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The Soft-RTC internally uses a simple counter value that is incremented every time the physical timer generates an interrupt. On each call to vmfRtcGetTime() the real-time date is computed out of this counter value. This computation only works correctly if the time between two calls does not exceed one day (24 hours). Thus, the application has to call vmfRtcGetTime() at least one time per day.

6.12.2 vmfRtcSetTime

```
UINT32 vmfRtcSetTime (
    VMF_TIME* pTime
);
```

Parameter

pTime

[in] pointer to structure with new system time to set.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.12.3 vmfRtcSetAlarm

Not implemented at the moment.

```
UINT32 vmfRtcSetAlarm (
    VMF_TIME* pTime
);
```

Parameter
pTime
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.12.4 vmfRtcSetReference

Not implemented at the moment.

```
UINT32 vmfRtcSetReference (
    VMF_STAMPED_TIME*       pStampedTime
);
```

Parameter
pStampedTime
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.13 Configuration

6.13.1 vmfConfigGetFirstValueA

```
UINT32 vmfConfigGetFirstValueA (
    CHAR*       szKeyName,
    CHAR*       szValueName,
    UINT32*     pdwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

szKeyName

[in] (ascii) requested section in the RTOS configuration area.

szValueName

[in] (ascii) requested key in the RTOS configuration area.

pdwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.13.2 vmfConfigGetFirstValueW

```
UINT32 vmfConfigGetFirstValueW (
    WCHAR*      wszKeyName,
    WCHAR*      wszValueName,
    UINT32*     pdwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

wszKeyName

[in] (unicode) requested section in the RTOS configuration area.

wszValueName

[in] (unicode) requested key in the RTOS configuration area.

pdwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.13.3 vmfConfigGetNextValueA

```
UINT32 vmfConfigGetNextValueA (
    CHAR*       szKeyName,
    CHAR*       szValueName,
    UINT32*     pdwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

szKeyName

[in] (ascii) requested section in the RTOS configuration area.

szValueName

[in] (ascii) requested key in the RTOS configuration area.

pdwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.13.4 vmfConfigGetNextValueW

```
UINT32 vmfConfigGetNextValueW (
    WCHAR*      wszKeyName,
    WCHAR*      wszValueName,
    UINT32*     pdwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

wszKeyName

[in] (unicode) requested section in the RTOS configuration area.

wszValueName

[in] (unicode) requested key in the RTOS configuration area.

pdwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.13.5 vmfConfigQueryValueA

Determine the entry value of a requested section/key pair in the RTOS configuration area.

```
UINT32 vmfConfigQueryValueA (
    CHAR*       szKeyName,
    CHAR*       szValueName,
    UINT32      dwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

szKeyName

[in] (ascii) requested section in the RTOS configuration area.

szValueName

[in] (ascii) requested key in the RTOS configuration area.

dwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

{

Return

RTE_SUCCESS on success and an error-code on failure. Detailed error information can be found in the pAddData variable.

Comment

–

6.13.6 vmfConfigQueryValueW

Determine the entry value of a requested section/key pair in the RTOS configuration area.

```
UINT32 vmfConfigQueryValueW (
    WCHAR*      wszKeyName,
    WCHAR*      wszValueName,
    UINT32      dwType,
    UINT8*      pbyData,
    UINT32*     pdwSize,
    VMF_CONFIG_ADDDATA* pAddData
);
```

Parameter

wszKeyName

[in] (unicode) requested section in the RTOS configuration area.

wszValueName

[in] (unicode) requested key in the RTOS configuration area.

dwType

[in] data type, the following data types are defined
VMF_CONFIG_SZ_TYPE: zero terminated string
VMF_CONFIG_DWORD_TYPE: unsigned 32 bit integer value

pbyData

[out] pointer to buffer where the configuration value will be stored.

pdwSize

[in,out] the caller of this function has to store the size of the output buffer where pbyData in *pdwSize, the function will set the value of *pdwSize to the actual number of bytes copied into pbyData.

pAddData

[out] Additional configuration data error information.
pAddData->dwLineNr: line number where the error occurred
pAddData->dwErrorCode: error code:
REG_NO_ERROR – no error
REG_INVALID_VALUENAME – syntax error in configuration value name
REG_INVALID_KEYNAME – syntax error in configuration key name

{

Return

RTE_SUCCESS on success and an error-code on failure. Detailed error information can be found in the pAddData variable.

Comment

–

6.14 Virtual Network: Network Packet Library

The virtual network packet library provides functions to transfer network data packets between Windows and the RTOS.
An example implementation for VxWorks is provided in …\Examples\RtosVnet\VxWin.

Synchronization notes:
The virtual network functions use shared informations and their calls must be synchronized against themselves and against each other.

6.14.1 Operation modes

The virtual network packet library may either operate in interrupt mode or in polling mode.
Polling Mode:
For the receiver part this means that the network driver cyclically has to check if new data packets are received (no receive interrupts).
And for the sender part, after the network packet buffer is full the network driver has to check cyclically if new data packets can be sent again (no send interrupts).
Interrupt Mode:
An interrupt will be generated if newly received packets are available. When sending network packets it may happen that no send buffers are available. In that case an interrupt will be generated by the counter part when at least one single buffer is available again.

6.14.2 Configuration

Two configuration settings are common for all operating systems. The polling period (in timer ticks) and the MAC address of the virtual network adapter.
An RTOS may use an arbitrary configuration parameter but as a convention the following configuration parameters shall be used (located in the RTOS configuration file)

```
[Rtos]
   "VnetMacAddress"="AA:BB:CC:DD:EE:02"        ; MAC address of the vnet adapter
   "VnetPollPeriod"=dword:1                    ; polling period in timer ticks
                                               ; (when set to 0: operation mode = interrupt)
```

These parameters can be determined using the vmfConfigQueryValue() function.
If the polling period is set to a value of 0, the network packet library will operate in interrupt mode, otherwise it will operate in polling mode.

6.14.3 Initialization, De-Initialization

Prior to using any other vnet function the vmfVnetInit() routine has to be called.
When the driver shall be shut down, first vmfVnetDetach() has to be called to detach from the network. Finally a call to vmfVnetDeinit() has to be executed.

6.14.3.1 vmfVnetInit

Initialize the virtual network packet library.

```
UINT32 vmfVnetInit (
   VOID*                  pvCoreDev,
   VMF_VNET_INIT_PARMS*   poInitParms
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor where the device data will be stored. The vnet driver has to provide a pointer to zero-initialized memory area. The size of the memory area is VMF_VNET_DEVICE_SIZE.

poInitParms

[in] pointer to initialization parameters

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The virtual network packet library will be initialized by calling this function. This function has to be called prior to calling any other functions of the library.

VMF_VNET_INIT_PARMS
initialization parameters.

```
typedef struct _VMF_VNET_INIT_PARMS
{
    VOID*                          pvDrvContext;
    BOOL                           bMaster;
    VMF_VNET_PFN_GET_ACCESS        pfnGetAccess;
    VMF_VNET_PFN_RELEASE_ACCESS    pfnReleaseAccess;
    VMF_VNET_PFN_GET_SHM           pfnGetShm;
    VMF_VNET_PFN_COPY_BLOCK_TO_BUF pfnCopyBlockToBuf;
    VMF_VNET_PFN_COPY_BUF_TO_BLOCK pfnCopyBufToBlock;
} VMF_PACKED(1) VMF_VNET_INIT_PARMS;
```

Description

pvDrvContext
[in] Driver context. The network packet library does not evaluate this pointer. On several callback functions this pointer will be returned to the driver.

bMaster
[in] TRUE if the network peer is the virtual network master. As the Windows network peer is the network master all RTOS have to set this value to FALSE.

pfnGetAccess
[in] Pointer to a function that will be called to get access to shared resources. Typically this function will take a mutex.

pfnReleaseAccess
[in] Pointer to a function that will be called to release access to shared resources. Typically this function will release a mutex.

pfnCopyBlockToBuf
[in] Pointer to a function that copies an OS specific network packet data block (which contains data to send) into a plain memory buffer located in the virtual network's shared memory area.

pfnCopyBufToBlock
[in] Pointer to a function that copies network data from a plain memory buffer located in the virtual network's shared memory area (contains received data) into an OS specific network packet data block.

6.14.3.2 vmfVnetMpGetInterruptId

Determine the interrupt id of the virtual network adapter.

```
UINT32 vmfVnetMpGetInterruptId (
    VMF_HANDLE      hAdapter,
    UINT32*         pdwInterruptId,
);
```

Parameter

hAdapter

[in] adapter handle (0, if only peer-to-peer network is used).

pdwInterruptId

[out] interrupt ID.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the virtual network adapter is operating in interrupt mode, then this function will return the appropriate interrupt ID.

6.14.3.3 vmfVnetMpIntEnable

Enable interrupts.

```
UINT32 vmfVnetMpIntEnable (
    VMF_HANDLE      hAdapter
);
```

Parameter

hAdapter

[in] adapter handle (0, if only peer-to-peer network is used).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the virtual network adapter is operating in interrupt mode, then this function will enable interrupts.

6.14.3.4 vmfVnetMpIntDisable

Disable interrupts.

```
UINT32 vmfVnetMpIntDisable (
    VMF_HANDLE      hAdapter
);
```

Parameter

hAdapter

[in] adapter handle (0, if only peer-to-peer network is used).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the virtual network adapter is operating in interrupt mode, then this function will disable interrupts.

6.14.3.5 vmfVnetGetPacketSize

Determine the maximum size of a network data packet that can be stored in the virtual network's shared memory. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetGetPacketSize(
    VOID*       pvCoreDev,
    UINT32*     pdwPacketSize,
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.
pvBlock
[out] maximum packet size.

Return

RTE_SUCCESS on success and an error-code on failure.

6.14.3.6 vmfVnetAttach

Attach to the shared memory network. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetAttach (
    VOID*       pvCoreDev,
    BOOL*       pbAttached
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.
pbAttached
[out]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.3.7 vmfVnetDetach

Detach from the virtual network's shared memory area. After calling vmfVnetDetach() the peer is not available on the network. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetDetach (
    VOID*       pvCoreDev,
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

After calling this function network packets cannot be received or sent any more.

6.14.3.8 vmfVnetCheckAnchor

Check the existence and validity of the network anchor. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetCheckAnchor (
    VOID*       pvCoreDev,
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.3.9 vmfVnetDynAttach

Dynamic network attach. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetDynAttach (
    VOID*       pvCoreDev,
    BOOL*       pbAttached
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.
pbAttached
[out]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.3.10 vmfVnetIsAttached

Check if the network peer is attached. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetIsAttached (
    VOID*       pvCoreDev,
    BOOL*       pbAttached
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.
pbAttached
[out]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.3.11 vmfVnetDeinit

De-Initialize the virtual network packet library.

```
UINT32 vmfVnetDeinit (
    VOID*       pvCoreDev,
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

After calling this function no network packet library function (except vmfVnetInit) can be called any more.

6.14.4 Heartbeat

Every network peer has to cyclically call a heartbeat function to signal a life sign. At least once every 500 milliseconds this function shall be called

---

6.14.4.1 vmfVnetHandleHearts

De-Initialize the virtual network packet library.

```
UINT32 vmfVnetHandleHearts (
    VOID*          pvCoreDev,
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.5 Receive network packets

The network packet library provides the following functions for the network driver receiver part.

- vmfVnetPacketsReceived():determine if new network packets are available
- vmfVnetRcv(): copy one received network packet into a network data packet block area
- vmfVnetRxFlush(): discard all currently received network packets

---

6.14.5.1 vmfVnetPacketsReceived

Determine if new network packets are available.

```
UINT32 vmfVnetPacketsReceived (
    VOID*       pvCoreDev,
    BOOL*       pbPacketsReceived,
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor.

pbPacketsReceived

[out] pointer to flag signaling if new data are available (TRUE if available, FALSE if not)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.5.2 vmfVnetRcv

Copy the latest received network packet into a provided network data packet block area.

```
UINT32 vmfVnetRcv(
    VOID*       pvCoreDev,
    VOID*       pvBlock,
    INT32*      pnLength,
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor.

pvBlock

[out] pointer to network data packet block area where the received packet shall be stored.

pnLength

[out] length of the data packet (byte)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The callback function pfnCopyBufToBlock provided in vmfVnetInit() will be called by the network packet library to copy the data into the network data packet block area.

6.14.5.3 vmfVnetRxFlush

Discard all currently received network packets.

```
UINT32 vmfVnetRxFlush(
    VOID*        pvCoreDev,
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.5.4 vmfVnetGetNumberOfPackets

Determine number of packets that can be stored internally. Note: this function should not be called, it is deprecated.

```
UINT32 vmfVnetGetNumberOfPackets(
    VOID*        pvCoreDev,
    UINT32*      pdwNumberOfPackets
);
```

Parameter

pvCoreDev
[in] pointer to device descriptor.
pdwNumberOfPackets
[out]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.6 Send network packets

The network packet library provides the following functions for the network driver sender part.

- vmfVnetIsTxBlocked ():determine if new data can be sent (if a free send buffer available)
- vmfVnetSend(): send one single network data stored in a packet block area

---

6.14.6.1 vmfVnetIsTxBlocked

Determine if new network packets can be sent.

```
UINT32 vmfVnetIsTxBlocked (
    VOID*       pvCoreDev,
    BOOL*       pbIsTxBlocked,
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor.

pbIsTxBlocked

[out] pointer to flag signaling if sending data is blocked (TRUE if sending data is blocked, FALSE if sending data is possible)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.6.2 vmfVnetSend

Copy the latest received network packet into a provided network data packet block area.

```
UINT32 vmfVnetRcv(
    VOID*       pvCoreDev,
    VOID*       pvBlock,
);
```

Parameter

pvCoreDev

[in] pointer to device descriptor.

pvBlock

[out] pointer to network data packet block area which shall be sent.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The callback function pfnCopyBlockToBuf provided in vmfVnetInit() will be called by the network packet library to copy the data from the network data packet block area into the virtual network's shared memory.

6.14.6.3 vmfVnetMpSendFrame

```
UINT32 vmfVnetMpSendFrame(
    VMF_HANDLE      hAdapter,
    UINT8*          pbyFrame,
    UINT32          dwLength
);
```

Parameter

hAdapter
[in] adapter handle (0, if only peer-to-peer network is used).
pbyFrame
[in]
dwLength
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.6.4 vmfVnetMpSendFrameComplete

```
UINT32 vmfVnetMpSendFrameComplete(
    VMF_HANDLE      hAdapter,
);
```

Parameter

hAdapter
[in] adapter handle (0, if only peer-to-peer network is used).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.14.6.5 vmfVnetMpReceiveFrame

```
UINT32 vmfVnetMpReceiveFrame(
    VMF_HANDLE      hAdapter,
    UINT8*          pbyFrame,
    UINT32*         pdwLength
);
```

Parameter

hAdapter
[in] adapter handle (0, if only peer-to-peer network is used).
pbyFrame
[in]
dwLength
[out]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.14.6.6 vmfVnetMpReceiveFrameComplete

UINT32 vmfVnetMpReceiveFrameComplete(
VMF_HANDLE hAdapter,
);

Parameter
hAdapter
[in] adapter handle (0, if only peer-to-peer network is used).

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.14.6.7 vmfVnetMpInitA

UINT32 vmfVnetMpInitA(
CHAR* tszDeviceName,
VMF_HANDLE* phAdapter
);

Parameter
tszDeviceName
[in]
hAdapter
[in] adapter handle (0, if only peer-to-peer network is used).

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.14.6.8 vmfVnetMpInitW

```
UINT32 vmfVnetMpInitW(
    WCHAR*          tszDeviceName,
    VMF_HANDLE*     phAdapter
);
```

Parameter

tszDeviceName

[in]

hAdapter

[in] adapter handle (0, if only peer-to-peer network is used).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.15 I/O port access

6.15.1 vmfInByte

```
UINT8 vmfInByte (
    UINT32      dwIoPort
);
```

Parameter

dwIoPort
[in]

Return

Comment
–

6.15.2 vmfOutByte

```
UINT8 vmfOutByte (
    UINT32      dwIoPort,
    UINT8       byValue
);
```

Parameter

dwIoPort
[in]
byValue
[in]

Return

Comment
–

6.15.3 vmfInWord

```
UINT16 vmfInWord (
    UINT32      dwIoPort
);
```

Parameter

dwIoPort
[in]

Return

Comment
–

6.15.4 vmfOutWord

```
UINT16 vmfOutWord (
    UINT32      dwIoPort,
    UINT16      wValue
);
```

Parameter

dwIoPort
[in]

wValue
[in]

Return

Comment
–

6.15.5 vmfInDword

```
UINT32 vmfInDword (
    UINT32      dwIoPort
);
```

Parameter

dwIoPort
[in]

Return

Comment
–

6.15.6 vmfOutDword

```
UINT32 vmfInDword (
    UINT32      dwIoPort,
    UINT32      dwValue
);
```

Parameter

dwIoPort
[in]

dwValue
[in]

Return

Comment
–

6.16 Basic Shared Memory Areas (deprecated)

6.16.1 vmfShmGetUserShmAddr

Gets start address of Multi Purpose Shared Memory with ID 0 (formerly UserShm).
This function is deprecated. Use “Multi Purpose Shared Memory Areas” instead.

```
UINT32 vmfShmGetUserShmAddr (
    UINT32*     pdwShmAddr
);
```

Parameter

pdwhmAddr

[in] Shared memory start address.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.16.2 vmfShmGetUserShmSize

Gets size of Multi Purpose Shared Memory with ID 0 (formerly UserShm).
This function is deprecated. Use “Multi Purpose Shared Memory Areas” instead.

```
UINT32 vmfShmGetUserShmSize (
    UINT32*     pdwShmSize
);
```

Parameter

pdwShmSize

[in] Shared memory size.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.16.3 vmfShmGetInternalShmAddr

Gets start address of internal shared memory.
This function is deprecated.

```
UINT32 vmfShmGetInternalShmAddr (
    UINT32*     pdwInternalShmAddr
);
```

Parameter

pdwInternalShmAddr

[in] Internal shared memory start address.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.16.4 vmfShmGetInternalShmSize

Gets size of internal shared memory.
This function is deprecated.

```
UINT32 vmfShmGetInternalShmSize (
    UINT32*      pdwInternalShmSize
);
```

Parameter

pdwInternalShmSize
[in] Internal shared memory size.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.17 Multi Purpose Shared Memory Areas

Multi purpose shared memory areas offer more possibilities compared with the basic shared memory areas.
These areas are configured using the [SharedMemoryXx] sections in the configuration file.

[SharedMemory*MySharedMem*] ; Shared Memory area

- "Name" (string)
  Shared Memory Area name.
- “Description” (string)
  Shared Memory Area description.
- “Base” (dword)
  Physical base address. If set to a value of 0, then this shared memory area is allocated by Windows and available for access within Windows.
- “Alignment” (dword)
  If the “Base” parameter is set to 0 then this parameter determines the alignment of the memory area,
- “Size” (dword)
  Size of the shared memory area (in bytes).
- “Initialize” (dword)
  0 → don’t initialize the shared memory area
  1 → initialize the shared memory area with 0
  2 → initialize the shared memory area with the content of a file
  3 → initialize the shared memory area with 0 and then initialize with the content of a file
- “Save” (dword)
  0 → don’t save the shared memory content when the RTOS is stopped
  1 → store the shared memory content into a file when the RTOS is stopped
- “File” (string)
  If “Initialize” is set to a value of 2, then this parameter determines the path and filename of the file that shall be used to initialize the shared memory area. The file size must be less or equal than the shared memory size.
  If “Save” is set to a value of 1 the shared memory area is stored in the file determined by this parameter.
- “AccessDefault” (dword)
  Default access mode (bit mask) for all RTOS instances
  0x01 → shared memory is present if bit 0 is set
  0x02 → read only access if bit 1 is set
  0x04 → uncached access if bit 2 is set
  0x08 → code execution in shared memory is allowed if bit 4 is set
  If the parameter is not set a value of 1 is assumed (present, cached, read/write access, code execution not allowed).

[SharedMemory*MySharedMem*\AccessModes] ; RTOS specific access mode definitions

- "0" (dword)
  Specific access mode for the RTOS described in section [Rtos]. These values will override the default values set by parameter “AccessDefault”
- "1" (dword)
  Specific access mode for the RTOS described in section [Rtos1]. These values will override the default values set by parameter “AccessDefault”

The shared memory area must not overlap with the RTOS memory area(s). It must be part of the memory area reserved for the real-time part (parameters RteMemoryStartAddress and RteMemorySize in section [Upload]).
Note: The section [SharedMemory0] must not be enabled. It is reserved for future use!

6.17.1 vmfShmGetInfo

Determine configuration for a given shared memory area.

```
UINT32 vmfShmGetInfo(
   UINT32          dwShmId
   VMF_SHM_INFO*   pVmfShmInfo
);
```

Parameter

dwShmId

[in] Id of the shared memory area (0 = user shm, 1/2/3 = enumerated according to configuration file (first [SharedMemory\...] entry will be ID 1, second will be ID 1.

pVmfShmInfo

[out] Configuration info.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The value for dwShmId is interpreted as follows:
0 → first enumerated entry in [SharedMemory\...]
1 → second enumerated entry in [SharedMemory\...]
…

VMF_SHM_INFO

Shared Memory information descriptor.

```
typedef struct _VMF_SHM_INFO
{
    UINT32  dwSize;
    CHAR    tszName[VMF_SHM_MAX_NAME_SIZE];
    CHAR    tszDescription[VMF_SHM_MAX_DESCRIPTION_SIZE];
    CHAR    tszFilename[VMF_SHM_MAX_FILENAME_SIZE];
    UINT64  qwMemoryBasePhysical;
    UINT32  dwMemorySize;
    UINT32  dwInitialize;
    UINT32  dwSave;
    UINT32  dwAccessMode;
} VMF_PACKED(1) VMF_SHM_INFO;
```

Description

dwSize
: [in] Size of the VMF_SHM_INFO descriptor. This value has to be set by the caller.

tszName
: [out] Name of the shared memory area

tszDescription
: [out] Description of the shared memory area

tszFilename
: [out] File name

qwMemoryBasePhysical
: [out] Physical base address

qwMemorySize
: [out] Memory size in bytes

dwInitialize
: [out] Initialization setting (`VMF_SHM_INITMODE_NOTHING, VMF_SHM_INITMODE_ZERO or VMF_SHM_INITMODE_FILE`)

dwSave
: [out] Save mode (`VMF_SHM_SAVEMODE_NOTHING, VMF_SHM_SAVEMODE_FILE`)

dwAccessMode
: [out] Access mode

6.18 Shared Events

Synchronization notes:
The shared event functions use shared informations and their calls must be synchronized against themselves and against each other.
Starting with version 6.1.00.06 the formaly fixed total number of events (100) has changed to 500 per default and can be modified using config file entry [Vmf] "EventCount".
Please refer to “RtosVM-UserManual.pdf” chapter “Section [Vmf]” for details.

6.18.1 vmfEventCreateA

Creates a new event with the specified name or increments the event handle counter if an event with this name already exists.

```
UINT32 vmfEventCreateA (
    BOOL            bManualReset,
    BOOL            bInitialState,
    CHAR*           szName,
    BOOL            bUserEvent,
    VMF_HANDLE*     phEvent,
);
```

Parameter

bManualReset
[in] unused
bInitialState
[in] initial state of the OS event
szName
[in] ascii name of the event to be created
bUserEvent
[in] event will be user event or internal event
phEvent
[out] handle of the event created, or NULL on failure.

Return

RTE_SUCCESS	*on success*
RTE_BUSY_RECALL	*if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx	*on failure.*

Comment

–

6.18.2 vmfEventCreateW

Creates a new event with the specified name or increments the event handle counter if an event with this name already exists.

```
UINT32 vmfEventCreateW (
    BOOL            bManualReset,
    BOOL            bInitialState,
    WCHAR*          wszName,
    BOOL            bUserEvent,
    VMF_HANDLE*     phEvent,
);
```

Parameter

bManualReset
[in] unused

bInitialState
[in] initial state of the OS event
wszName
[in] unicode name of the event to be created
bUserEvent
[in] event will be user event or internal event
phEvent
[out] handle of the event created, or NULL on failure.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.3 vmfEventOpenA
Increments the event handle counter of the event with this name.

```
UINT32 vmfEventOpenA (
   CHAR*          szName,
   BOOL           bUserEvent,
   VMF_HANDLE*    phEvent
);
```

Parameter
szName
[in] ascii name of the event to be opened
bUserEvent
[in] event will be user event or internal event
phEvent
[out] handle of the event created, or NULL on failure.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.4 vmfEventOpenW
Increments the event handle counter of the event with this name.

```
UINT32 vmfEventOpenW (
   WCHAR*         wszName,
   BOOL           bUserEvent,
   VMF_HANDLE*    phEvent
);
```

Parameter
wszName
[in] unicode name of the event to be opened
bUserEvent
[in] event will be user event or internal event
phEvent
[out] handle of the event created, or NULL on failure.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.5 vmfEventSet

Sets the specified event to signalled state.

```
UINT32 vmfEventSet (
    VMF_HANDLE      hEvent,
    BOOL            bUserEvent,
);
```

Parameter
hEvent
[in] event handle.
bUserEvent
[in] event is user event or internal event

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.6 vmfEventClose

Closes the specified event and decrements the event handle counter. If the last handle is closed, the event is removed from the operating system.

```
UINT32 vmfEventClose (
    VMF_HANDLE      hEvent,
    BOOL            bUserEvent,
);
```

Parameter
hEvent
[in] event handle.
bUserEvent
[in] event is user event or internal event

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.7 vmfEventCommAttach

Has been renamed to “vmfEventReferenceAdd”.
A define replaces “vmfEventCommAttach” with “vmfEventReferenceAdd”.

6.18.8 vmfEventDelete

Has been renamed to “vmfEventReferenceRelease”.
A define replaces “vmfEventDelete” with “vmfEventReferenceRelease”.

6.18.9 vmfEventListGetLockFlagAddr

This function is deprecated. This function should not be used any more since locking is now internally handled. Any use of this function will result in the use of a less performant compatibility mode.

6.18.10 vmfEventGetLockFlagAddr

This function is deprecated. This function should not be used any more since locking is now internally handled. Any use of this function will result in the use of a less performant compatibility mode.

6.18.11 vmfEventGetNextJob

Searchs the event list for events with signalled state.

```
UINT32 vmfEventGetNextJob (
    UINT32*         pdwJob,
    VMF_HANDLE*     phEvent
);
```

Parameter

pdwJob

[out] Processing job:
DO_NOTHING
DO_EVENT_CREATE
DO_EVENT_SET
DO_EVENT_RESET
DO_EVENT_PULSE
DO_EVENT_DELETE

phEvent

[out] event handle of first event with signalled state

Return

RTE_SUCCESS	*on success*
RTE_BUSY_RECALL	*if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx	*on failure.*

Comment

–

6.18.12 vmfEventJobDone

.

```
UINT32 vmfEventJobDone (
    UINT32          dwJob,
    VMF_HANDLE      hEvent
);
```

Parameter

dwJob

[in] -

hEvent
[out] event handle.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.13 vmfEventGetNameA

```
UINT32 vmfEventGetNameA (
    VMF_HANDLE      hEvent,
    CHAR*           szEventName
);
```

Parameter
hEvent
[in] event handle.
szEventName
[out] event name.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.14 vmfEventGetNameW

```
UINT32 vmfEventGetNameW (
    VMF_HANDLE      hEvent,
    WCHAR*          szEventName
);
```

Parameter
hEvent
[in] event handle.
wszEventName
[out] event name.

Return
RTE_SUCCESS *on success*
RTE_BUSY_RECALL *if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx *on failure.*

Comment
–

6.18.15 vmfEventGetState

UINT32 vmfEventGetState (
VMF_HANDLE hEvent,
UINT32* pdwEventState
);

Parameter
hEvent
[in] event handle.
pdwEventState
[out] state of the event.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.18.16 vmfEventReferenceAdd
Increments the event handle counter.

UINT32 vmfEventReferenceAdd (
VMF_HANDLE hEvent
);

Parameter
hEvent
[in] event handle.

Return
RTE_SUCCESS on success
RTE_BUSY_RECALL if function needs to be recalled after a sleep(1)
RTE_ERROR_xxx on failure.

Comment
Since the event has to be locked the function can only be called between *"vmfEventGetNextJob"* and *"vmfEventJobDone"*.

6.18.17 vmfEventReferenceRelease
Decrements the event handle counter. If the last handle is closed, the event descriptor is removed.

UINT32 vmfEventDelete (
VMF_HANDLE hEvent
);

Parameter
hEvent
[in] event handle.

Return
RTE_SUCCESS on success
RTE_BUSY_RECALL if function needs to be recalled after a sleep(1)
RTE_ERROR_xxx on failure.

Comment
Since the event has to be locked the function can only be called between *"vmfEventGetNextJob"* and *"vmfEventJobDone"*.

6.18.18 vmfEventShowA

Show routine of the event handling.

```
UINT32 vmfEventShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter

pfnPrintA

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in] not specified or 0: only active events are printed out.
otherwise : all event descriptors are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.18.19 vmfEventShowW

Show routine of the event handling.

```
UINT32 vmfEventShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter

pfnPrintW

[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);

nMode

[in] not specified or 0: only active events are printed out.
otherwise : all event descriptors are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.19 Message Box

6.19.1 vmfMessageBoxExW

prepares data for a message Box.

```
UINT32 vmfMessageBoxExW (
    WCHAR*          wszText,
    WCHAR*          wszCaption,
    UINT32          dwType,
    UINT32*         pdwUserReturnCode,
    VMF_PFN_SLEEP   pfnSleep
);
```

Parameter

wszText
[in] unicode string to be displayed in the message box

wszCaption
[in] unicode string to be displayed as title of the message box.

dwType
[in] type of appearance:
VMF_MESSAGEBOX_OK
VMF_MESSAGEBOX_OKCANCEL
VMF_MESSAGEBOX_ABORTRETRYIGNORE
VMF_MESSAGEBOX_YESNOCANCEL
VMF_MESSAGEBOX_YESNO
VMF_MESSAGEBOX_RETRYCANCEL

pdwUserReturnCode
[out] Return code from user.

pfnSleep
[in] function pointer to sleep function: INT32 *function* (INT32 nMilliSecs)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.19.2 vmfMessageBoxW

Same as vmfMessageBoxExW with pfnSleep = NULL.

6.19.3 vmfMessageBoxExA

Ascii version of function vmfMessageBoxExW.

6.19.4 vmfMessageBoxA

Ascii version of function vmfMessageBoxW.

6.19.5 vmfMessageBoxGetW

Query VMF for new data for displaying a message box.

UINT32 vmfMessageBoxGetW (
VMF_MESSAGE_BOX_W* pVmfMessageBoxW
);

Parameter

pVmfMessageBoxW
[out] pointer to a caller allocated structure for receiving message box data

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Deprecated – please use vmfMessageBoxExGetW() instead.

6.19.6 vmfMessageBoxGetA

Ascii version of function vmfMessageBoxGetW.

6.19.7 vmfMessageBoxExGetW

Query VMF for new data for displaying a message box.

UINT32 vmfMessageBoxGetW (
VMF_MESSAGE_BOX_EX* pVmfMessageBoxEx
);

Parameter

pVmfMessageBoxEx
[out] pointer to a caller allocated structure for receiving message box data

Return

RTE_SUCCESS on success
RTE_ERROR_NODATAEXCHANGEBUFFER if pVmfMessageBoxEx->dwDataSize is to small to receive all data
RTE_ERROR_... on failure.

Comment

This function should be called by the VMF managing OS, typically from the RtosControl under Windows.
The caller

- allocates a buffer with size = sizeof(VMF_MESSAGE_BOX_EX) + DesiredNumberOfBytesForStrings
- assigns the buffer to a variable of type VMF_MESSAGE_BOX_EX*
- initializes →dwSize = sizeof(VMF_MESSAGE_BOX_EX)
- initializes →dwDataSize = DesiredNumberOfBytesForStrings
- calls vmfMessageBoxGetW()
- if returned RTE_ERROR_NODATAEXCHANGEBUFFER:
  caller increases allocation and recalls the function
- if returned RTE_SUCCESS:
  →bdwMsgValid is true if a new message was available. In this case
  →aData[wCaptionOffset] contains the caption and →aData[wMessageOffset] is the text string.

See RtosLibMessageBox.c for an example implementation.

6.19.8 vmfMessageBoxExGetA

Ascii version of function vmfMessageBoxExGetW.

6.19.9 vmfMessageBoxShowA

Show routine of the message box.

```
UINT32 vmfMessageBoxShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter

pfnPrintA

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.19.10 vmfMessageBoxShowW

Show routine of the message box.

```
UINT32 vmfMessageBoxShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter

pfnPrintW

[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.20 Configuration Show Routines

6.20.1 vmfRtosConfigShowA

Show routine of the rtos configuration.

```
UINT32 vmfRtosConfigShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter

pfnPrintA

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.20.2 vmfRtosConfigShowW

Show routine of the rtos configuration.

```
UINT32 vmfRtosConfigShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter

pfnPrintW

[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.20.3 vmfProcessorConfigShowA

Show routine of the processor configuration.

```
UINT32 vmfProcessorConfigShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter

pfnPrintA
[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);
nMode
[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.20.4 vmfProcessorConfigShowW
Show routine of the processor configuration.

```
UINT32 vmfProcessorConfigShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter
pfnPrintW
[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);
nMode
[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.20.5 vmfIoApicConfigShowA
Show routine of the io-apic configuration.

```
UINT32 vmfIoApicConfigShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter
pfnPrintA
[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);
nMode
[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.20.6 vmfIoApicConfigShowW

Show routine of the io-apic configuration.

```
UINT32 vmIoApicConfigShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter

pfnPrintW

[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.20.7 vmfDeviceConfigShowA

Show routine of the device configuration.

```
UINT32 vmfDeviceConfigShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode,
);
```

Parameter

pfnPrintA

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.20.8 vmfDeviceConfigShowW

Show routine of the device configuration.

```
UINT32 vmfDeviceConfigShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode,
);
```

Parameter

pfnPrintW

[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat, ...);

nMode

[in] not specified or 0: only main information are printed out.
otherwise : additional details are printed out.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.21 Virtual I/O

The virtual I/O feature of the VMF provides a mechanism to transfer stream data between the participants. Currently only a single channel is supported. Virtual I/O data are transferred between the primary OS (POS) which usually is Windows and the secondary OS (SOS) which usually is the RTOS.

Synchronization notes:
The virtual IO functions use shared informations and their calls must be synchronized against themselves and against each other.

6.21.1 vmfVioRead

Read data from virtual I/O channel.

```
UINT32 vmfVioRead (
    UINT32          dwChanId,
    BOOL            bIsPOS,
    UINT8*          pbyData,
    UINT32          dwMaxNumBytesToRead,
    UINT32*         pdwNumBytesRead
);
```

Parameter

dwChanId
[in] Channel ID (0, 1, …)
bIsPOS
[in] Set to TRUE if called by the primary OS, FALSE if called by the secondary OS
pbyData
[out] pointer to buffer where data shall be stored
dwMaxNumBytesToRead
[in] size of the data buffer
pdwNumBytesRead
[out] number of actually read bytes (0 if no bytes are available)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

In case there is no data available a call to this function immediately returns.

6.21.2 vmfVioWrite

Write data to virtual I/O channel.

```
UINT32 vmfVioWrite (
    UINT32          dwChanId,
    BOOL            bIsPOS,
    UINT8*          pbyData,
    UINT32          dwNumBytes,
    UINT32*         pdwNumBytesWritten
);
```

Parameter

dwChanId
[in] Channel ID (0, 1, …)

bIsPOS
[in] Set to TRUE if called by the primary OS, FALSE if called by the secondary OS

pbyData
[in] pointer to buffer where data is stored

dwNumBytes
[in] number of bytes to write

pdwNumBytesWritten
[out] number of bytes actually written (0 if no bytes are written)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

In case no data can be written a call to this function immediately returns.

6.22 Time stamping

6.22.1 vmfTimeStampGetFrequency

```
UINT32 vmfTimeStampGetFrequency (
    UINT32      dwSourceId,
    UINT64*     pqwFrequency
);
```

Parameter

dwSourceId

[in] Time stamp source; should be **VMF_TIMESTAMP_SOURCE_ID_AUTO**

pqwFrequency

[out] Frequency in Hz

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The timestamp source is determined automatically. On default the processor's timestamp counter will be used. The automatic determination can be overridden by a config file entry:
[Rtos\Vmf] "TimeStampSourceAuto"=dword:1 ; value see VMF_TIMESTAMP_SOURCE_ID_xxx

6.22.2 vmfTimeStampGetValue

```
UINT32 vmfTimeStampGetValue (
    UINT32      dwSourceId,
    UINT64*     pqwValue
);
```

Parameter

dwSourceId

[in] Time stamp source; should be **VMF_TIMESTAMP_SOURCE_ID_AUTO**

pqwValue

[out] Time stamp value

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The timestamp source is determined automatically. On default the processor's timestamp counter will be used. The automatic determination can be overridden by a config file entry:
[Rtos\Vmf] "TimeStampSourceAuto"=dword:1 ; value see VMF_TIMESTAMP_SOURCE_ID_xxx
Please call vmfTimestampGetMaxValue to determine when the timestamp will wrap.

6.22.3 vmfTimeStampGetMaxValue

This function returns the max value before the timestamp wraps

```
UINT32 vmfTimeStampGetMaxValue (
    UINT32      dwSourceId,
    UINT64*     pqwMaxValue
);
```

Parameter

dwSourceId

[in] Time stamp source; should be **VMF_TIMESTAMP_SOURCE_ID_AUTO**

pqwMaxValue

[out] Time stamp max value before wrap

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The timestamp source is determined automatically. On default the processor's timestamp counter will be used. The automatic determination can be overridden by a config file entry:
[Rtos\Vmf] "TimeStampSourceAuto"=dword:1 ; value see VMF_TIMESTAMP_SOURCE_ID_xxx

6.23 Performance Measurement

Synchronization notes:
The functions vmfPerformanceStart and vmfPerformanceStop use shared informations and their calls must be synchronized against themselves and against each other.

6.23.1 vmf PerformanceStart

UINT32 vmfPerformanceStart (
UINT32 dwPerformanceId
);

Parameter
dwPerformanceId
[in] Performance measurement to start

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
- see vmfPerformanceGetData

6.23.2 vmfPerformanceReset

UINT32 vmfPerformanceReset (
UINT32 dwPerformanceId
);

Parameter
dwPerformanceId
[in] Performance measurement to reset

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
- see vmfPerformanceGetData

6.23.3 vmfPerformanceStop

UINT32 vmfPerformanceStop (
UINT32 dwPerformanceId
);

Parameter
dwPerformanceId
[in] Performance measurement to stop

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
- see vmfPerformanceGetData

6.23.4 vmfPerformanceGetData

```
UINT32 vmfPerformanceGetData (
    UINT32                  dwPerformanceId,
    UINT32                  dwOsId,
    UINT32                  dwProcessorMask,
    VMF_PERFORMANCE_DATA*   pPerformanceData,
);
```

Parameter

dwPerformanceId

[in] Performance measurement to get the data from

dwOsId

[in] OS to get the data from. VMF_OSID_THISOS can be used to query the data for the calling OS.

dwProcessorMask

[in] Processor to get the data from

pPerformanceData

[out] Pointer to the **VMF_PERFORMANCE_DATA** structure to write to.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

- Following performance IDs can be used:
  - **VMF_PERFORMANCE_ID_TUNNEL**:
  qwCount member of the **VMF_PERFORMANCE_DATA** structure returns the number of time the system switch into the selected OS since the measurement was started.
  qwMin member of the **VMF_ PERFORMANCE_DATA** structure returns the minimum time spent in the selected OS expressed in increments since the measurement was started.
  qwMax member of the **VMF_ PERFORMANCE_DATA** structure returns the maximum time spent in the selected OS expressed in increments since the measurement was started.
  qwSum member of the **VMF_ PERFORMANCE_DATA** structure returns the total time spent in the selected OS expressed in increments since the measurement was started.
  dwAvg member of the **VMF_ PERFORMANCE_DATA** structure returns the average time spent in the selected OS since the measurement was started.
  qwConverter member of the **VMF_ PERFORMANCE_DATA** structure returns the frequency in Hz of the increments.

6.24 Time Synchronisation

Synchronization notes:
The time synchronization functions use shared informations and their calls must be synchronized against themselves and against each other.

6.24.1 vmfTimeSyncIsMaster

```
UINT32 vmfTimeSyncIsMaster (
    BOOL*       pbIsMaster
);
```

Parameter
pbIsMaster
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.2 vmfTimeSyncGetMaster

```
UINT32 vmfTimeSyncGetMaster (
    UINT32*     pdwMasterOsId
);
```

Parameter
pdwMasterOsId
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.3 vmfTimeSyncSetMaster

```
UINT32 vmfTimeSyncSetMaster (
    UINT32      dwMasterOsId
);
```

Parameter
dwMasterOsId
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.4 vmfTimeSyncDisable

```
UINT32 vmfTimeSyncDisable (
    VOID
);
```

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.5 vmfTimeSyncGetMasterTime

```
UINT32 vmfTimeSyncGetMasterTime (
    VMF_STAMPED_TIME*      pVmfStampedTime,
    UINT32                 dwTimeIndex
);
```

Parameter
pVmfStampedTime
[out]
dwTimeIndex
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.6 vmfTimeSyncGetMasterTime2

```
UINT32 vmfTimeSyncGetMasterTime2 (
    VMF_STAMPED_TIME2*     pVmfStampedTime
);
```

Parameter
pVmfStampedTime
[out]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.7 vmfTimeSyncGetOsTime

```
UINT32 vmfTimeSyncGetOsTime (
    VMF_STAMPED_TIME*     pVmfStampedTime,
    UINT32                dwOsId,
    UINT32                dwTimeIndex
);
```

Parameter

pVmfStampedTime
[out]

dwOsId
[in] VMF_OSID_THISOS can be used to query the time for the calling OS.

dwTimeIndex
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.8 vmfTimeSyncGetOsTime2

```
UINT32 vmfTimeSyncGetOsTime2 (
    VMF_STAMPED_TIME2*    pVmfStampedTime,
    UINT32                dwOsId
);
```

Parameter

pVmfStampedTime
[out]

dwOsId
[in] VMF_OSID_THISOS can be used to query the time for the calling OS.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.9 vmfTimeSyncSetTime

```
UINT32 vmfTimeSyncSetTime (
    VMF_STAMPED_TIME*       pVmfStampedTime,
    UINT32                  dwTimeIndex
);
```

Parameter

pVmfStampedTime
[in]

dwTimeIndex
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.10 vmfTimeSyncSetTime2

```
UINT32 vmfTimeSyncSetTime (
    VMF_STAMPED_TIME2*      pVmfStampedTime,
);
```

Parameter

pVmfStampedTime
[in]

dwTimeIndex
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.11 vmfTimeSyncShowA

```
UINT32 vmfTimeSyncShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode
);
```

Parameter

pfnPrintA
[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.12 vmfTimeSyncShowW

```
UINT32 vmfTimeSyncShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode
);
```

Parameter

pfnPrintA
[in] function pointer to a unicode print function: INT32 *function*(const WCHAR *wszFormat,

nMode
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.13 vmfTimeSyncConvertTime

```
UINT32 vmfTimeSyncConvertTime (
    VMF_TIME*       pVmfTime,
    UINT16          wYearValue
);
```

Parameter
pVmfTime
[in]
wYearValue
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.14 vmfTimezoneSyncIsMaster

```
UINT32 vmfTimezoneSyncIsMaster (
    BOOL*       pbIsMaster
);
```

Parameter
pbIsMaster
[out]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.15 vmfTimezoneSyncGetMaster

```
UINT32 vmfTimezoneSyncGetMaster (
    UINT32*     pdwMasterOsId
);
```

Parameter
pdwMasterOsId
[out]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.16 vmfTimezoneSyncSetMaster

```
UINT32 vmfTimezoneSyncSetMaster (
    UINT32      dwMasterOsId
);
```

Parameter
dwMasterOsId
[in]

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.17 vmfTimezoneSyncDisable

```
UINT32 vmfTimezoneSyncDisable (
    VOID
);
```

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.18 vmfTimezoneSyncGetOsTimezoneA

```
UINT32 vmfTimezoneSyncGetOsTimezoneA (
    VMF_TIMEZONE_A*         pVmfTimezoneA,
    UINT32                  dwOsId
);
```

Parameter
pVmfTimezoneA
[in]
dwOsId
[in] VMF_OSID_THISOS can be used to query the timezone for the calling OS.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.24.19 vmfTimezoneSyncGetOsTimezoneW

```
UINT32 vmfTimezoneSyncGetOsTimezoneW (
    VMF_TIMEZONE_W*         pVmfTimezoneA,
    UINT32                  dwOsId
);
```

Parameter

pVmfTimezoneW
[in]

dwOsId
[in] VMF_OSID_THISOS can be used to query the timezone for the calling OS.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.20 vmfTimezoneSyncSetTimezoneA

```
UINT32 vmfTimezoneSyncSetTimezoneA (
    VMF_TIMEZONE_A*         pVmfTimezoneA
);
```

Parameter

pVmfTimezoneA
[in]

dwOsId
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.21 vmfTimezoneSyncSetTimezoneW

```
UINT32 vmfTimezoneSyncSetTimezoneW (
    VMF_TIMEZONE_A*         pVmfTimezoneW
);
```

Parameter

pVmfTimezoneW
[in]

dwOsId
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.22 vmfTimezoneSyncShowA

```
UINT32 vmfTimezoneSyncShowA (
    VMF_PFN_PRINT_A         pfnPrintA,
    INT32                   nMode
);
```

Parameter

pfnPrintA

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.24.23 vmfTimezoneSyncShowW

```
UINT32 vmfTimezoneSyncShowW (
    VMF_PFN_PRINT_W         pfnPrintW,
    INT32                   nMode
);
```

Parameter

pfnPrintW

[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

nMode

[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.25 Comm Task

6.25.1 vmfCommSignalStarted

```
UINT32 vmfCommSignalStarted (
    VOID
);
```

Parameter
–

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.25.2 vmfCommIsStarted

```
UINT32 vmfCommIsStarted (
    UINT32      dwOsId,
    BOOL*       pbCommIsStarted
);
```

Parameter
dwOsId
[in] ID of the OS to check if the communication sub-system is started.
VMF_OSID_THISOS can be used to query the state of the calling OS.
pbCommIsStarted
[out] TRUE if the communication sub-system of the OS is started.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.25.3 vmfCommGetLockFlagAddr

This function is deprecated. This function should not be used any more since locking is now internally handled. Any use of this function will result in the use of a less performant compatibility mode.

6.25.4 vmfCommCreateProcess

Not implemented at the moment.

```
UINT32 vmfCommCreateProcess (
    UINT32              dwOsId,
    CHAR*               szFilename,
    CHAR*               szCmdLine,
    VMF_PFN_SLEEP       pfnSleep
);
```

Parameter

dwOsId

[in] Not implemented at the moment.

szFilename

[in] Not implemented at the moment.

szCmdLine

[in] Not implemented at the moment.

pfnSleep

[in] Not implemented at the moment.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.25.5 vmfCommConfigGet

This function returns the configuration information required for Comm task.

```
UINT32 vmfCommConfigGet (
    UINT32              dwOsId,
    PVMF_COMM_CONFIG pCommConfig
);
```

Parameter

dwOsId

[in] The OS id whose config should be queried. VMF_OSID_THISOS can be used to query the configuration for the calling OS.

pCommConfig

[out] Comm configuration information

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

pCommConfig→dwSize must be set to sizeof(VMF_COMM_CONFIG) before the function is called.

6.26 Os management

Synchronization notes:
The OS management functions use shared informations and their calls must be synchronized against themselves and against each other.

6.26.1 vmfOsStart

Not implemented at the moment.

```
UINT32 vmfOsStart (
    UINT32                  dwOsId,
    VMF_OS_START_PARAMS*    pParams,
    UINT8*                  pbyAsyncJobContext
);
```

Parameter

dwOsId
[in] ID of the OS to start

pParams
[in]

pbyAsyncJobContext
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.26.2 vmfOsStop

Not implemented at the moment.

```
UINT32 vmfOsStop (
    UINT32                  dwOsId,
    VMF_OS_START_PARAMS*    pParams,
    UINT8*                  pbyAsyncJobContext
);
```

Parameter

dwOsId
[in]

pParams
[in]

pbyAsyncJobContext
[in]

Return

RTE_SUCCESS	*on success*
RTE_BUSY_RECALL	*if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx	*on failure.*

Comment

–

6.26.3 vmfOsSuspend

Not implemented at the moment.

```
UINT32 vmfOsSuspend (
    UINT32                  dwOsId,
    VMF_OS_START_PARAMS*    pParams,
    UINT8*                  pbyAsyncJobContext
);
```

Parameter

dwOsId
[in]

pParams
[in]

pbyAsyncJobContext
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.26.4 vmfOsResume

Not implemented at the moment.

```
UINT32 vmfOsResume (
    UINT32                  dwOsId,
    VMF_OS_START_PARAMS*    pParams,
    UINT8*                  pbyAsyncJobContext
);
```

Parameter

dwOsId
[in]

pParams
[in]

pbyAsyncJobContext
[in]

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.26.5 vmfOsNotificationModeSet

```
UINT32 vmfOsNotificationModeSet (
    UINT32                                  dwNotificationId,
    UINT32                                  dwMode,
    VMF_OS_NOTIFICATION_MODE_PARAMS*        pParams
);
```

Parameter

dwNotificationId

[in] ID of the notification to modify.

dwMode

[in] new mode for the selected notification ID.

pParams

[in] specific parameters.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.26.6 vmfOsGetIdCurrent

```
UINT32 vmfOsGetIdCurrent (
    UINT32*     pdwOsId
);
```

Parameter

pdwOsId

[out] ID of the current OS.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.26.7 vmfOsGetInfoA

UINT32 vmfOsGetInfoA (
UINT32 dwOsId,
VMF_OS_INFO_A* pInfo
);

Parameter
dwOsId
[in] ID of the OS to get the informations from. VMF_OSID_THISOS can be used to query the information from the calling OS.
pInfo
[out] ascii informations about the OS.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
pInfo→dwSize must be set to sizeof(VMF_OS_INFO_A) before the function is called.

6.26.8 vmfOsGetInfoW

UINT32 vmfOsGetInfoW (
UINT32 dwOsId,
VMF_OS_INFO_W* pInfo
);

Parameter
dwOsId
[in] ID of the OS to get the informations from. VMF_OSID_THISOS can be used to query the information from the calling OS.
pInfo
[out] unicode informations about the OS.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
pInfo→dwSize must be set to sizeof(VMF_OS_INFO_W) before the function is called.

6.26.9 vmfOsIdle
Signalize the idle state of the current OS on the current processor.
UINT32 vmfOsIdle (VOID);

Parameter
-

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
This function should be called when the OS goes into his idle state on the current processor. On a shared processor, this function switchs to the Os with the next lower priority.
If RTOS uses a lowest priority task for calling vmfOsIdle() it should not run any other task on the same or a lower priority because it might never get CPU time.

6.26.10 vmfOsReboot

Signalize that the calling OS requires to reboot now.
UINT32 vmfOsReboot (VOID);

Parameter
-

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
This function should be called when the OS decides to reboot.
This function is currently implemented to stop the OS – but a further version may reboot the OS automatically.

6.26.11 vmfOsShutdown

Signalize that the calling OS did shut down.
UINT32 vmfOsShutdown (VOID);

Parameter
-

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
This function should be called when the OS did shut down itself.

6.27 Config Registry

Starting with Build 4.1.00.41 / 4.5.00.41 / 5.0.00.41 VMF config registry functions support well-known sections:

- VMF_CONFIGREG_HKEY_VMF
  →Section [Vmf]
- VMF_CONFIGREG_HKEY_OS_CURRENT
  →Section [Host], [Rtos], [Rtos1],… depending on the calling OS!

This allows using the same code on several OS while each call automatically results in the OS depending section.

Attention to compatibility:
The new well known keys are not supported by versions previous to 4.1.00.41 / 4.5.00.41 / 5.0.00.41.
Using the well known keys in such a version might result in unpredictable behavior.
An application can check if the VMF version supports the well known keys like this:

```
// Compatibility:
// check for VMF_CONFIGREG_HKEY_OS_CURRENT support...
// using with VMF version prior to 4.1.00.41 / 4.5.00.41 / 5.0.00.41 might cause crash.
dwValueSize = sizeof( dwTmpVal );
dwRes = vmfConfigQueryValueA( VMF_TEXT(VMF_CONFIGREG_KEY_VMF),
VMF_TEXT(VMF_CONFIGREG_VALUE_VMF_VERSION),
                    VMF_CONFIG_DWORD_TYPE, (UINT8*)&dwTmpVal,
                    &dwValueSize, &AdditionalData );
if( (RTE_SUCCESS == dwRes) && (0x011B0E00 <= dwTmpVal) )
{
   dwRes = vmfConfigRegKeyOpenA( VMF_CONFIGREG_HKEY_OS_CURRENT,
                                 szRegSubKey, &hKey );
   if( RTE_SUCCESS == dwRes )
   {
      ...
      vmfConfigRegKeyClose( hKey );
      hKey = NULL;
   }
}
```

6.27.1 vmfConfigRegKeyOpenA

```
UINT32 vmfConfigRegKeyOpenA (
   VMF_HANDLE      hConfigRegRoot,
   CHAR*           szKey,
   VMF_HANDLE*     phConfigRegKey
);
```

Parameter

hConfigRegRoot
[in] Handle to already open key where szKey can be found or NULL to start with root.

szKey
[in] Name of key to be open.

phConfigRegKey
[out] Handle to the open key.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.2 vmfConfigRegKeyOpenW

```
UINT32 vmfConfigRegKeyOpenW (
    VMF_HANDLE      hConfigRegRoot,
    WCHAR*          wszKey,
    VMF_HANDLE*     phConfigRegKey
);
```

Parameter
hConfigRegRoot
[in] Handle to already open key where wszKey can be found or NULL to start with root.

wszKey
[in] Name of key to be open.

phConfigRegKey
[out] Handle to the open key.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.3 vmfConfigRegKeyClose

```
UINT32 vmfConfigRegKeyClose (
    VMF_HANDLE      hConfigRegKey
);
```

Parameter
hConfigRegKey
[in] Handle to be closed.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.4 vmfConfigRegKeyEnumerateA

```
UINT32 vmfConfigRegKeyEnumerateA (
    VMF_HANDLE      hConfigRegKey,
    UINT32          dwSubkeyIndex,
    CHAR*           szName,
    UINT32*         pdwNameLength,
    UINT32*         pdwFlags
);
```

Parameter
hConfigRegKey
[in] Handle to already open key where szName can be found or NULL to start with root.

dwSubkeyIndex
[in] Index for the key to be queried.

szName
[out] Name of the key at the queried index.

pdwNameLength
[in/out] Length of the szName buffer / Length of returned key name.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_KEY_FLAG_xxx might be returned.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.5 vmfConfigRegKeyEnumerateW

```
UINT32 vmfConfigRegKeyEnumerateW (
    VMF_HANDLE      hConfigRegKey,
    UINT32          dwSubkeyIndex,
    WCHAR*          wszName,
    UINT32*         pdwNameLength,
    UINT32*         pdwFlags
);
```

Parameter
hConfigRegKey
[in] Handle to already open key where wszName can be found or NULL to start with root.

dwSubkeyIndex
[in] Index for the key to be queried.

wszName
[out] Name of the key at the queried index.

pdwNameLength
[in/out] Length of the szName buffer / Length of returned key name.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_KEY_FLAG_xxx might be returned.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.6 vmfConfigRegValueQueryA

```
UINT32 vmfConfigRegValueQueryA (
    VMF_HANDLE      hConfigRegKey,
    CHAR*           szValueName,
    UINT32*         pdwFlags,
```

```
    UINT32*         pdwType,
    UINT8*          pbyData,
    UINT32*         pdwDataSize
);
```

Parameter

hConfigRegKey
[in] Handle to already open key where szValueName can be found or NULL to start with root.

szValueName
[in] Name of the value to be queried.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_VALUE_FLAG_xxx might be returned.

pdwType
[out] optional - if not NULL a value of VMF_CONFIGREG_TYPE_xxx will be returned.

pbyData
[out] Buffer to return the value data.

pdwDataSize
[in/out] Length of the data buffer available / Length of data buffer used.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.27.7 vmfConfigRegValueQueryW

```
UINT32 vmfConfigRegValueQueryW (
    VMF_HANDLE      hConfigRegKey,
    WCHAR*          wszValueName,
    UINT32*         pdwFlags,
    UINT32*         pdwType,
    UINT8*          pbyData,
    UINT32*         pdwDataSize
);
```

Parameter

hConfigRegKey
[in] Handle to already open key where wszValueName can be found or NULL to start with root.

wszValueName
[in] Name of the value to be queried.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_VALUE_FLAG_xxx might be returned.

pdwType
[out] optional - if not NULL a value of VMF_CONFIGREG_TYPE_xxx will be returned.

pbyData
[out] Buffer to return the value data.

pdwDataSize
[in/out] Length of the data buffer available / Length of data buffer used.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.8 vmfConfigRegValueEnumerateA

```
UINT32 vmfConfigRegValueEnumerateA (
    VMF_HANDLE      hConfigRegKey,
    UINT32          dwValueIndex,
    CHAR*           szName,
    UINT32*         pdwNameLength,
    UINT32*         pdwFlags,
    UINT32*         pdwType,
    UINT8*          pbyData,
    UINT32*         pdwDataSize
);
```

Parameter
hConfigRegKey
[in] Handle to already open key where wszValueName can be found or NULL to start with root.

dwValueIndex
[in] Index for the value to be queried.

szName
[out] Name of the value at the queried index.

pdwNameLength
[in/out] Length of the szName buffer / Length of returned value name.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_VALUE_FLAG_xxx might be returned.

pdwType
[out] optional - if not NULL a value of VMF_CONFIGREG_TYPE_xxx will be returned.

pbyData
[out] optional - if not NULL buffer to return the value data.

pbyDataSize
[in/out] optional - if not NULL length of the data buffer available / Length of data buffer used.

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
–

6.27.9 vmfConfigRegValueEnumerateW

```
UINT32 vmfConfigRegValueEnumerateW (
```

```
    VMF_HANDLE      hConfigRegKey,
    UINT32          dwValueIndex,
    WCHAR*          wszName,
    UINT32*         pdwNameLength,
    UINT32*         pdwFlags,
    UINT32*         pdwType,
    UINT8*          pbyData,
    UINT32*         pdwDataSize
);
```

Parameter

hConfigRegKey
[in] Handle to already open key where wszValueName can be found or NULL to start with root.

dwValueIndex
[in] Index for the value to be queried.

wszName
[out] Name of the value at the queried index.

pdwNameLength
[in/out] Length of the szName buffer / Length of returned value name.

pdwFlags
[out] optional - if not NULL a combination of VMF_CONFIGREG_VALUE_FLAG_xxx might be returned.

pdwType
[out] optional - if not NULL a value of VMF_CONFIGREG_TYPE_xxx will be returned.

pbyData
[out] optional - if not NULL buffer to return the value data.

pbyDataSize
[in/out] optional - if not NULL length of the data buffer available / Length of data buffer used.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

6.28 Miscellaneous

6.28.1 vmfIdGetByNameA

```
UINT32 vmfIdGetByNameA (
    CHAR*                       szName,
    UINT32                      dwIdType,
    UINT32*                     pdwId,
);
```

Parameter

szName
[in] Name of the ID

dwIdType
[in] Type of the ID

pdwId
[out] ID number

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

See vmfIdGetByNameW

6.28.2 vmfIdGetByNameW

```
UINT32 vmfIdGetByNameW (
    WCHAR*                      wszName,
    UINT32                      dwIdType,
    UINT32*                     pdwId,
);
```

Parameter

wszName
[in] Name of the ID

dwIdType
[in] Type of the ID

pdwId
[out] ID number

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

- Following ID Types are supported:

  - **VMF_ID_OS**
  - **VMF_ID_DEVICE**
  - **VMF_ID_SHM**

This function has to be used to get the ID to a known ID name. Usually only the Name of an OS, Device, or shared memory is known. Never use a ID which is not calculated with the function vmfIdGetByName().

6.28.3 vmfIdGetNameA

```
UINT32 vmfIdGetNameA (
    UINT32          dwIdType,
    UINT32          dwId,
    CHAR*           szName,
    UINT32*         *pdwNameLen,
);
```

Parameter

dwIdType
[in] Type of the ID

dwId
[in] ID number

szName
[in] Name of the ID

pdwNameLen
[in/out] Length of name in characters

Return

RTE_SUCCESS on success and an error-code on failure
*If RTE_ERROR_NODATAEXCHANGEBUFFER is returned * pdwNameLenth contains the required length.*

Comment

See vmfIdGetNameW

6.28.4 vmfIdGetNameW

```
UINT32 vmfIdGetNameW (
    UINT32          dwIdType,
    UINT32          dwId,
    WCHAR*          wszName,
    UINT32*         *pdwNameLen,
);
```

Parameter

dwIdType
[in] Type of the ID

dwId
[in] ID number

wszName
[in] Name of the ID

pdwNameLen
[in/out] Length of name in characters

Return
RTE_SUCCESS on success and an error-code on failure.
*If RTE_ERROR_NODATAEXCHANGEBUFFER is returned * pdwNameLenth contains the required length.*

Comment
- Following ID Types are supported:

  - **VMF_ID_DEVICE**
  - **VMF_ID_INTERRUPT**

This function can be used to get the name for an ID.

6.28.5 vmfIdShowA

```
UINT32 vmfIdShow (
    VMF_PFN_PRINT_A     pfnPrintA,
    UINT32              dwIdType,
    UINT32              dwId,
    INT32               nMode
);
```

Parameter
pfnPrintA
[in] function pointer to a ascii print function: INT32 *function*(const CHAR *szFormat, ...);

dwIdType
[in] Type of the ID

dwId
[in] ID number

nMode
[in] display mode

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
See vmfIdShowW

6.28.6 vmfIdShowW

```
UINT32 vmfIdShow (
    VMF_PFN_PRINT_W     pfnPrintW,
    UINT32              dwIdType,
    UINT32              dwId,
    INT32               nMode
);
```

Parameter
pfnPrintW
[in] function pointer to a unicode print function: INT32 *function*(const WHAR *wszFormat, ...);

dwIdType
[in] Type of the ID

dwId
[in] ID number

nMode
[in] display mode

Return
RTE_SUCCESS on success and an error-code on failure.

Comment
- Following ID Types are supported:

- **VMF_ID_OS**
- **VMF_ID_DEVICE**
- **VMF_ID_VNET**
- **VMF_ID_SHM**
- **VMF_ID_IOAPIC**
- **VMF_ID_PROCESSOR**
- **VMF_ID_COMM**
- **VMF_ID_EVENT**
- **VMF_ID_INTERRUPT**
- **VMF_ID_CORE**

This function can be used to show detailed information about the current VMF configuration. This can be used during development or in error cases. It should not be used to parse for any required information because the output may change in format and information without notice.

It can also be called directly from the Uploader:
RtosUpload.exe -IdShow "dwIdType,dwId,nMode"

6.28.7 vmfAsyncJobIsDone

```
UINT32 vmfAsyncJobIsDone (
    UINT8*      pbyAsyncJobContext,
    BOOL*       pbIsDone
);
```

Parameter
pbyAsyncJobContext
[in] pointer to a asynchronous context structure.

pbIsDone
[in] TRUE if the asynchronous job is done.

Return
RTE_SUCCESS	*on success*
RTE_BUSY_RECALL	*if function needs to be recalled after a sleep(1)*
RTE_ERROR_xxx	*on failure.*

Comment
–

6.29 Privileged functions
Some of the VMF functions require specific privileges to be successfully executed.

6.29.1 Functions requesting IO privilege level

The following functions need IOPL=3, if they are called with CPL=3
(they are using assembler commands like pushfl, cli, popfl,…):

vmfOsStart
vmfOsStop
vmfInterruptGenerate
vmfProcessorStart
vmfProcessorStop
vmfTimerVirtGetCurrentCount
vmfTimerVirtSetOutputFreq

6.29.2 Functions requesting privileged execution level

The following functions need CPL=0 (due to access to CR registers and using assembler commands like lgdt, lidt, ltr,…):

vmfOsStart
vmfOsStop
vmfTunnelSwitch
vmfTimeStampGetValue if CR4.TSD set (rdtsc)

7 VMF Interface serialization

On some operating systems it is necessary to serialize VMF functions.
If for example on Windows or Windows CE a VMF function shall be called in user mode (Ring 3) the call has to be serialized first. In a second step the serialized data have to be transferred to a kernel driver (using some kind of IoControl call). Then the kernel driver has to de-serialize the data and call the appropriate VMF functions. The results then will have to be transferred back to the user mode application.
For this purpose VmfWin contains a portable serialization module. Most of the serialization can be handled in a generic way, only a few simple functions have to be implemented in a separate operating system specific adaptation layer.

The following files are shipped:
…\VmfInterfaceSerializing\UserMode\vmfCall.c
→ portable serialization module for the user mode
…\VmfInterfaceSerializing\UserMode\vmfCallOs.h
→ function prototypes for the operating system adaptation layer (user mode).
…\VmfInterfaceSerializing\UserMode\WinCE\rteOs.h
→ example Windows CE VMF adaptation layer header (user mode)
…\VmfInterfaceSerializing\UserMode\WinCE\vmfCallCE.c
→ example Windows CE adaptation layer (user mode)
…\VmfInterfaceSerializing\UserMode\WinXP\rteOs.h
→ example Windows VMF adaptation layer header (user mode)
…\VmfInterfaceSerializing\UserMode\WinXP\vmfCallXP.c
→ example Windows adaptation layer (user mode)
…\VmfInterfaceSerializing\KernelMode\vmfCallDispatch.c
→ portable de-serialization module for the kernel mode
…\VmfInterfaceSerializing\KernelMode\WinCE\rteOs.h
→ example Windows CE VMF adaptation layer header (kernel mode)
…\VmfInterfaceSerializing\KernelMode\WinCE\vmfCallDispatchCE.c
→ example Windows CE adaptation layer (kernel mode)
…\VmfInterfaceSerializing\KernelMode\WinXP\rteOs.h
→ example Windows VMF adaptation layer header (kernel mode)
…\VmfInterfaceSerializing\KernelMode\WinXP\vmfCallDispatchXP.c
→ example Windows adaptation layer (kernel mode)

To port this serialization module the following steps have to be done:

– adjust the user mode adaptation layer. The most important function is the vmfCall() function which has to transfer the serialized data between user mode and kernel mode. The macro VMFCALL_ABSOLUTE has to be set in rteOs.h!
– create a user mode library which contains the following files: vmfCall.c + vmfCallOS.c (e.g. vmfCallCE.c). This library will then provide vmfXxx() functions available as a user mode library.
– Create the counter part of the vmfCall() function in the kernel driver. Here the data transferred by vmfCall() will be received and the generic vmfCallDispatch() function (which actually executes the appropriate VMF routine) has to be called.
– Adjust the kernel mode adaptation layer.
  The macro VMFCALL_RELATIVE has to be set in rteOs.h!
  The following functions have to be implemented:
  - vmfIsReadyForCallsOs() This function will have to return a value of TRUE when the system is ready to execute VMF functions.
  - vmfCallPrintfInit() This function configures a buffer for a vmfCallPrintf callback. Returns RTE_SUCCESS, RTE_ERROR_NOT_IMPL or RTE_ERROR_xxx.
  - vmfCallPrintfDeinit() This function removes a buffer configuration for a vmfCallPrintf callback. Returns RTE_SUCCESS, RTE_ERROR_NOT_IMPL or RTE_ERROR_xxx.
  - vmfCallPrintfA() This is the ASCII printf callback function. It will use the buffer configured with a call to vmfCallPrintfInit. Returns RTE_SUCCESS, RTE_ERROR_NOT_IMPL or RTE_ERROR_xxx.
  - vmfCallPrintfW() This is the UNICODE printf callback function. It will use the buffer configured with a call to vmfCallPrintfInit. Returns RTE_SUCCESS, RTE_ERROR_NOT_IMPL or RTE_ERROR_xxx.
– Include the following files in your kernel driver or RTOS BSP: vmfCallDispatch.c + vmfCallDispatchOS.c (e.g. vmfCallDispatchCE.c). These functions will then act as a backend for the user mode library.

8 The RTOS Library

The RTOS library provides higher level communication services for synchronizing Windows with the RTOS or to exchange data between the operating systems. The RTOS library is based on VMF functions which provide the basic communication functionality.
This library is provided in source code as it has to be ported to the RTOS. To make this job easier most of the source code is written OS independently. The OS adaptation is done in a separate OS adaptation layer. VmfWin provides example

8.1 Porting the RTOS Library

Most of the RTOS library is generic source code which should be easily portable to any operating system. Only a small adaptation layer has to be adjusted to port the library.
The generic source files are located in …\RtosLib
The example adaptation layers are located in

a) VxWorks
…\RtosLib\VxWorks (one adaptation layer RtosLibOs.c for kernel mode and one RtosLibOsRtp.c for real-time processes)
b) Windows CE
…\RtosLib\WinCE (RtosLibOs.c).
c) Windows
…\RtosLib\WinXP (RtosLibOs.c).

To port the RTOS library the following steps have to be done:

- Adjust the adaptation layer located in rteOs.h and RtosLibOs.h and create required adapting function in RtosLibOs.c.
- Create a user mode library which contains all generic source files and the adaptation layer files. This library will then provide the Rtos Library functions available as a user mode library.

8.1.1 RTOS Library OS adaptation layer

The following routines are required by the RTOS library. These functions have to be ported to the RTOS. The header file for the adaptation layer is located in …\RtosLib\RtosLibOs.h.
The generic part of the RTOS library uses macros for OS specific function calls. These macros have to be defined at RtosLibOs.h.

For most generic functions a compatible os function can be used:

```
#define RTOSLIB_OS_MALLOC( SIZE ) malloc( SIZE )
```

```
If no compatible os function exists a convert function can be used:
#define RTOSLIB_OS_TIMESYNC_TIME_SET( TIME, RES ) OSTimeSyncTimeSet( TIME, RES )
UINT32 OSTimeSyncTimeSet( const VMF_STAMPED_TIME* pVmfStampedTime, const UINT32
                          dwReserved )
{
       …
       SetLocalTime(…);
       …
};
```

It is also possible to leave some not required functions blank:

```
#define RTOSLIB_OS_INITDONE() RTE_SUCCESS
```

8.1.1.1 RTOSLIB_OS_INIT

RTOS library Initialization function. This function is called once before other library functions can be called.

UINT32 RTOSLIB_OS_INIT(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.2 RTOSLIB_OS_DEINIT

RTOS library De-Initialization function. This function is called to terminate the RTOS library operation.

UINT32 RTOSLIB_OS_DEINIT(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.3 RTOSLIB_OS_INITDONE

RTOS library initialization done function.

UINT32 RTOSLIB_OS_INITDONE(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

RtosLibInit at last calls this function.
Except Windows every other OS should call RtosLibOnVmfLoadEx() from within OSRtosLibOnInitDone because VMF is already started.

8.1.1.4 RTOSLIB_OS_DEINITSTART

RTOS library de-initialization start function. RtosLibDeinit at first calls this function.

UINT32 RTOSLIB_OS_DEINITSTART(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Except Windows every other OS should call RtosLibOnVmfUnloadEx() from within OSRtosLibOnDeinitStart().

8.1.1.5 RTOSLIB_OS_ON_VMFLOAD

RTOS library on vmf load function. RtosLibOnVmfLoadEx calls this function.

UINT32 RTOSLIB_OS_ON_VMFLOAD(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

An OS now can query VMF for information. RtosLib uses this functionality to initialize all VMF dependent information.

8.1.1.6 RTOSLIB_OS_ON_VMFUNLOAD

RTOS library on vmf unload function. RtosLibOnVmfUnloadEx calls this function.

UINT32 RTOSLIB_OS_ON_VMFUNLOAD(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

An OS now should release VMF dependend information. RtosLib uses this functionality to de-initialize and release all VMF dependent information.

8.1.1.7 RTOSLIB_OS_SYSTEM_EVENT_SET

RTOS System Event set function.

UINT32 RTOSLIB_OS_SYSTEM_EVENT_SET(
UINT32 dwEventId
);

Parameter

dwEventId

[in] Event to be signaled. This can be one of the OS_RTOSLIB_SYSTEMEVENT_xxx events.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is currently used by Windows to signal other RtosLib instances load or unload of the VMF. It is currently not required for other OS's.

8.1.1.8 RTOSLIB_OS_APISYNC_INIT

Initialize the API synchronization function. This function is called once before other library functions can be called.

UINT32 RTOSLIB_OS_APISYNC_INIT(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The API synchronization function is intended for usage in Windows only. Porting this function to a new RTOS will usually leave these functions empty.
Windows API synchronization:
Prior to stop the whole system it has to be assured that no application is calling or entering one of the Rtos Library functions. This is handled by the RtosLib API synchronization mechanism.
Every time an RtosLib function is called a mutex will be taken inside. Thus a maximum number of RTOSLIB_APISYNC_MAX_USER threads may concurrently call RtosLib functions.
If the system shall be stopped the maximum count of mutex will be taken, then it can be assured that no other thread is calling any RtosLib function.

8.1.1.9 RTOSLIB_OS_APISYNC_DEINIT

De-Initialize the API synchronization function.

UINT32 RTOSLIB_OS_APISYNC_DEINIT(VOID)

Parameter

–

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is usually empty on the RTOS side.

8.1.1.10 RTOSLIB_OS_APISYNC_ENTER

Enter API synchronization. This function is called at the beginning of an RtosLib function. It assures that a maximum number of threads are concurrently executing RtosLib functions.

```
UINT32 RTOSLIB_OS_APISYNC_ENTER(
    VMF_HANDLE*                 phLock,
    BOOL                        bGlobalLock
);
```

Parameter

phLock

[out] synchronization object to be returned in the corresponding OSRtosLibApiSyncExit function

bGlobalLock

[in] TRUE if further calls to RtosLib functions shall be globally locked.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The bGlobalLock flag must only be set in Windows prior to stopping the RTOS operation.

8.1.1.11 RTOSLIB_OS_APISYNC_EXIT

Exit API synchronization. This function is called at the end of an RtosLib function.

```
UINT32 RTOSLIB_OS_APISYNC_EXIT(
    VMF_HANDLE*                 phLock
);
```

Parameter

phLock

[in] synchronization object returned in the corresponding OSRtosLibApiSyncEnter function

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.12 RTOSLIB_OS_APISYNC_STARTSTOPENTER

Enter start/stop API synchronization. This function is called at the beginning of a start or stop function.

```
UINT32 RTOSLIB_OS_APISYNC_STARTSTOPENTER(
    VMF_HANDLE*                 phLock
);
```

Parameter

phLock

[out] synchronization object to be returned in the corresponding OSRtosLibApiSyncStartStopExit function

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is only used by Windows to prevent multiple start/stop calls executed at the same time.

8.1.1.13 RTOSLIB_OS_APISYNC_STARTSTOPEXIT

Exit start/stop API synchronization. This function is called at the end of a start or stop function.

```
UINT32 RTOSLIB_OS_APISYNC_STARTSTOPEXIT(
    VMF_HANDLE*                 phLock
);
```

Parameter

phLock

[in] synchronization object returned in the corresponding OSRtosLibApiSyncStartStopEnter function

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is only used by Windows to prevent multiple start/stop calls executed at the same time.

8.1.1.14 RTOSLIB_OS_CALLOC

Allocate zero initialized memory.

```
VOID* RTOSLIB_OS_CALLOC(
    UINT32      dwNum,
    UINT32      dwSize,
);
```

Parameter

dwNum

[in] Number of memory blocks

dwSize

[in] Size of one memory block

Return

Pointer to the memory or NULL in case of an error.

Comment

–

8.1.1.15 RTOSLIB_OS_MALLOC

Allocate memory.

```
VOID* RTOSLIB_OS_MALLOC(
    UINT32      dwSize,
);
```

Parameter

dwSize

[in] Size of memory

Return

Pointer to the memory or NULL in case of an error.

Comment

–

8.1.1.16 RTOSLIB_OS_FREE

Free memory previously allocated with OScalloc() or OSmalloc().

```
VOID RTOSLIB_OS_FREE(
    VOID*               pMemory)
```

Parameter

pMemory

[in] Pointer to memory

Return

–

Comment

–

8.1.1.17 RTOSLIB_OS_MEMCPY

Copy memory.

```
VOID* RTOSLIB_OS_MEMCPY(
    VOID*           pDest,
    const VOID*     pSrc,
    UINT32          dwLen)
```

Parameter

pDest

[in] Pointer to destination memory

pSrc

[in] Pointer to source memory

dwLen

[in] number of bytes to copy

Return

Value of pDest.

Comment

–

8.1.1.18 RTOSLIB_OS_SLEEP

Delay execution.

```
VOID RTOSLIB_OS_SLEEP(
    UINT32          dwMillisec)
```

Parameter

dwMillisec

[in] delay time in milliseconds.

Return

–

Comment

–

8.1.1.19 RTOSLIB_OS_STRCMPA

String compare (ASCII version).

INT32 RTOSLIB_OS_STRCMPA(
const CHAR* szStr1,
const CHAR* szStr2)

Parameter

szStr1
[in] First string to compare
szStr2
[in] Second string to compare

Return

Compatible with the ANSI strcmp function.

Comment

–

8.1.1.20 RTOSLIB_OS_STRCMPW

String compare (wide char version).

INT32 RTOSLIB_OS_STRCMPW(
const WCHAR* wszStr1,
const WCHAR* wszStr2)

Parameter

wszStr1
[in] First string to compare
wszStr2
[in] Second string to compare

Return

Compatible with the ANSI strcmp function.

Comment

–

8.1.1.21 RTOSLIB_OS_STRNCPYA

String copy (ASCII version).

```
CHAR* RTOSLIB_OS_STRNCPYA(
    CHAR*           szDest,
    const CHAR*     szSrc,
    UINT32          dwLen)
```

Parameter

szDest
[out] Destination string

szSrc
[in] Source string

dwLen
[in] Maximum number of bytes to copy

Return

szDest on success, NULL on error.

Comment

–

8.1.1.22 RTOSLIB_OS_STRNCPYW

String copy (wide char version).

```
WCHAR* RTOSLIB_OS_STRNCPYW(
    WCHAR*          wszDest,
    const WCHAR*    wszSrc,
    UINT32          dwLen)
```

Parameter

wszDest
[out] Destination string

wszSrc
[in] Source string

dwLen
[in] Maximum number of bytes to copy

Return

wszDest on success, NULL on error.

Comment

–

8.1.1.23 RTOSLIB_OS_SHM_MAP

Shared memory mapping. This function has to provide access to the shared memory area(s) of the RTOS Virtual Machine.

UINT32 RTOSLIB_OS_SHM_MAP(
PRTOSLIBSHM_DESC pRtosShmDesc,
UINT32 dwRequestedSize,
UINT32 dwOffset)

Parameter

pRtosShmDesc
[in,out] Shared Memory Descriptor (see below)
dwRequestedSize
[in] Number of bytes to map
dwOffset
[in] Offset where the mapping shall begin

Return

Granted size (number of bytes actually mapped).

Comment

The granted size may be smaller than the requested size.

RTOSLIBSHM_DESC
Shared Memory Descriptor

```
typedef struct _RTOSLIBSHM_DESC
{
    UINT64      qwPhysAddr;
    UINT32      dwPhysSize;
    UINT32      dwAccessMode;
    UINT32      dwMappedSize;
    UINT32      dwMappedOffset;
    VOID*       lpMappedPtr;
    UINT32      dwCookie;
    UINT32      dwMappedSizeOsInternal;
    UINT32      dwMappedOffsetOsInternal;
    VOID*       lpMappedPtrOsInternal;
    VMF_HANDLE  hMutex;
    BOOL        bMapped;
} RTOSLIBSHM_DESC;
```

Description
qwPhysAddr
[in] Physical address
dwPhysSize
[in] Size of the shared memory area.
dwAccessMode
[in] Access mode bitmask. Contains VMF_SHM_ACCESSMODE_xxx values to differentiate between read only / read write, uncached / cached and executable / not executable memory.

dwMappedSize
[out] Size of the mapped memory area.
dwMappedOffset
[out] Offset where the mapping begins.
lpMappedPtr
[out] Pointer to the mapped memory area.

dwCookie

[out] Optional value. In further calls to the adaptation layer this value will be preserved. The adaptation layer may store internal data here.

dwMappedSizeOsInternal

[out] adaptation layer internal value for the mapped size. If the effectively mapped size is different from the requested size to map this information may be stored here. The value stored in dwMappedSizeOsInternal is not used by the generic RtosLib source code. This may be the case if only a multiple of the page size can be mapped and only a small amount is requested to be mapped.

dwMappedOffsetOsInternal

[out] adaptation layer internal value for the mapped offset. If the effectively mapped offset is different from the requested offset to map this information may be stored here. The value stored in dwMappedOffsetOsInternal is not used by the generic RtosLib source code. This may be the case if only page aligned offsets can be mapped and the requested offset is not page aligned.

lpMappedPtrOsInternal

[out] adaptation layer internal value for the mapped pointer. If the effectively mapped pointer is different from the value returned in lpMappedPtr this information may be stored here. The value stored in lpMappedPtrOsInternal is not used by the generic RtosLib source code. This may be the case if only page aligned offsets can be mapped and the requested offset is not page aligned.

hMutex

[] Mutex handle (only for internal use in the generic RtosLib source code).

bMapped

[in] TRUE if a part or the whole shared memory area is mapped.

8.1.1.24 RTOSLIB_OS_SHM_UNMAP

Shared memory mapping. This function has to release access to the shared memory area(s) of the RTOS Virtual Machine.

UINT32 RTOSLIB_OS_SHM_UNMAP(
PRTOSLIBSHM_DESC pRtosShmDesc)

Parameter

pRtosShmDesc

[in,out] Shared Memory Descriptor (see below)

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.25 RTOSLIB_OS_EVENT_CREATEA

Create a named event (ASCII version).

```
UINT32 RTOSLIB_OS_EVENT_CREATEA(
    BOOL            bInitialState,
    const CHAR*     szName,
    VMF_HANDLE*     phEvent)
```

Parameter

bInitialState
[in] TRUE if the event's state shall be signaled after creating the event object.
szName
[in] Event name
phEvent
[out] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the event already exists a handle to the existing event object will be returned.

8.1.1.26 RTOSLIB_OS_EVENT_CREATEW

Create a named event (wide char version).

```
UINT32 RTOSLIB_OS_EVENT_CREATEW(
    BOOL            bInitialState,
    const WCHAR*    wszName,
    VMF_HANDLE*     phEvent)
```

Parameter

bInitialState
[in] TRUE if the event's state shall be signaled after creating the event object.
wszName
[in] Event name
phEvent
[out] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the event already exists a handle to the existing event object will be returned.

8.1.1.27 RTOSLIB_OS_EVENT_OPENA

Open an already existing named event (ASCII version).

UINT32 RTOSLIB_OS_EVENT_OPENA(
const CHAR* szName,
VMF_HANDLE* phEvent)

Parameter

szName
[in] Event name
phEvent
[out] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.28 RTOSLIB_OS_EVENT_OPENW

Open an already existing named event (wide char version).

UINT32 RTOSLIB_OS_EVENT_OPENW(
const WCHAR* wszName,
VMF_HANDLE* phEvent)

Parameter

wszName
[in] Event name
phEvent
[out] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.29 RTOSLIB_OS_EVENT_CLOSE

Close the handle to an event.

UINT32 RTOSLIB_OS_EVENT_CLOSE(
VMF_HANDLE* phEvent)

Parameter

phEvent
[in] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If all handles to the event object are closed the event has to be deleted.

8.1.1.30 RTOSLIB_OS_EVENT_SET

Set the event to the signaled state.

UINT32 RTOSLIB_OS_EVENT_SET(
VMF_HANDLE* phEvent)

Parameter

phEvent
[in] Event handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

The highest priority thread waiting for the event will unblock and resume operation. The event will then be automatically set into the non-signaled state.

8.1.1.31 RTOSLIB_OS_EVENT_WAIT

Wait until the event is signaled.

UINT32 RTOSLIB_OS_EVENT_WAIT(
VMF_HANDLE* phEvent
UINT32 dwTimeout)

Parameter

phEvent
[in] Event handle
dwTimeout
[in] Timeout in milliseconds

Return

RTE_SUCCESS on success and an error-code on failure (RTE_ERROR_TIMEOUT on timeout).

Comment

Only the highest priority thread waiting for the event will unblock and resume operation. The event will then be automatically set into the non-signaled state.

8.1.1.32 RTOSLIB_OS_EVENT_SHOW

Show internal event descriptor information (for debugging purposes only).

```
UINT32 RTOSLIB_OS_EVENT_SHOW(
    INT32               nMode)
```

Parameter

nMode

[in] 0: show descriptor information of event descriptors which are in use
1: show descriptor information of all event descriptors

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.33 RTOSLIB_OS_INTERLOCKEDCOMPAREEXCHANGEEX

Performs an atomic compare-and-exchange operation on the specified values. The function compares two specified 32-bit values and exchanges with another 32-bit value based on the outcome of the comparison.

```
UINT32 RTOSLIB_OS_INTERLOCKEDCOMPAREEXCHANGEEX(
    UINT32 voilatile*  pdwDest,
    UINT32             dwExchg,
    UINT32             dwComparand,
    UINT32*            pdwInitial,
    const PRTOSLIBSHM_DESC pShmDesc
);
```

Parameter

pdwDest

[in,out] A pointer to the destination value.

dwExchg

[in] The exchange value.

dwComparand

[in] The value to be compared to pdwDest.

pdwInitial

[out] Destination value found at start of compare.

pShmDesc

[in] If not NULL then pdwDest points into the shared memory described by pShmDesc. If NULL then pdwDest points into the VMF.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

A return value of RTE_SUCCESS doesn't mean, that the value was exchanged. It just means that the call was executed and initial state of the variable is returned in pdwInitial.
If the return value is RTE_SUCCESS and dwInitial is equal to dwComparand then the value has been exchanged.

8.1.1.34 RTOSLIB_OS_MUTEX_CREATEA

Create a named mutex (ASCII version).

UINT32 RTOSLIB_OS_MUTEX_CREATEA(
const CHAR* szName,
VMF_HANDLE* phEvent)

Parameter

szName
[in] Mutex name
phEvent
[out] Mutex handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the mutex already exists an handle to the existing mutex object will be returned.

8.1.1.35 RTOSLIB_OS_MUTEX_CREATEW

Create a named mutex (wide char version).

UINT32 RTOSLIB_OS_MUTEX_CREATEW(
const WCHAR* wszName,
VMF_HANDLE* phEvent)

Parameter

wszName
[in] Mutex name
phEvent
[out] Mutex handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If the mutex already exists an handle to the existing mutex object will be returned.

8.1.1.36 RTOSLIB_OS_MUTEX_CLOSE

Close the handle to a mutex.

UINT32 RTOSLIB_OS_MUTEX_CLOSE(
VMF_HANDLE hMutex)

Parameter

hMutex
[in] Mutex handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

If all handles to the mutex object are closed the mutex has to be deleted.

8.1.1.37 RTOSLIB_OS_MUTEX_TAKE

Take access to the resource which is synchronized by the mutex object.

UINT32 RTOSLIB_OS_MUTEX_TAKE(
VMF_HANDLE hMutex)

Parameter

hMutex
[in] Mutex handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.38 RTOSLIB_OS_MUTEX_RELEASE

Release access to the resource which is synchronized by the mutex object.

UINT32 RTOSLIB_OS_MUTEX_RELEASE(
VMF_HANDLE hMutex)

Parameter

hMutex
[in] Mutex handle

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

–

8.1.1.39 RTOSLIB_OS_COMMTASK_START

Create and start the inter-OS communication task.

UINT32 RTOSLIB_OS_COMMTASK_START(
PRTOSLIBCOMM_DESC pRtosCommDesc)

Parameter

pRtosCommDesc
[in,out] RTOS communication descriptor, see below.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to create and start a new task which is responsible for inter-OS communication services (e.g. event or mutex handling). The task itself is part of the generic RtosLib sourcecode, its entrypoint is defined as follows: `UINT32 RtosLibCommTask(RTOSLIBCOMM_DESC* pRtosCommDesc)`

RTOSLIBCOMM_DESC
Communication Descriptor

```
typedef struct _RTOSLIBCOMM_DESC
{
    BOOL        bInitialized;
    BOOL        bCommTaskRunning;
    BOOL        bCommTaskStop;    /* set by start / stop function */
    VMF_HANDLE  hTaskId;
    VMF_HANDLE  hMutex;
    RTOSLIBLIST HandleList;
    VMF_HANDLE  hRtosLibApiSync;
} RTOSLIBCOMM_DESC;
```

Description
bInitialized
[] Used by the generic RtosLib source code.
bCommTaskRunning
[] Used by the generic RtosLib source code.
bCommTaskStop
[] Used by the generic RtosLib source code.
hTaskId
[out] Handle of the created task/thread.
hMutex
[] Used by the generic RtosLib source code.
HandleList
[] Used by the generic RtosLib source code.
hRtosLibApiSync
[] Used by the generic RtosLib source code.

8.1.1.40 RTOSLIB_OS_COMMTASK_STOP

Wait until the communication task has terminated.

UINT32 RTOSLIB_OS_COMMTASK_STOP(
PRTOSLIBCOMM_DESC pRtosCommDesc)

Parameter

pRtosCommDesc
[in,out] RTOS communication descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Prior to this call the RtosLib initiated a shutdown of the CommTask. The adaptation layer should wait here until the task is finally gone. If after some timeout it is still running an error message should be generated and the task may be killed.

8.1.1.41 RTOSLIB_OS_MSGBOX_TASK_START

Create and start the message box helper task.

UINT32 RTOSLIB_OS_MSGBOX_TASK_START(
PRTOSLIBMSGBOX_DESC pRtosMsgBoxDesc)

Parameter

pRtosMsgBoxDesc
[in,out] RTOS message box descriptor, see below.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to create and start a new task which is responsible for message box processing (e.g. wait for a new message box to be shown). The task itself is part of the generic RtosLib sourcecode, its entrypoint is defined as follows:

```
UINT32 RtosLibMsgBoxTask(RTOSLIBMSGBOX_DESC* pRtosMsgBoxDesc)
```

Note: Currently this function has to return RTE_ERROR on any RTOS. This function is intended only for the Windows part which is responsible to show a message box sent by the RTOS.

RTOSLIBMSGBOX_DESC
Message Box Descriptor

```
typedef struct _RTOSLIBMSGBOX_DESC
{
    BOOL        bInitialized;
    BOOL        bMsgBoxTaskRunning;
    BOOL        bMsgBoxTaskStop;
    VMF_HANDLE  hTaskId;
    VMF_HANDLE  hRtosLibApiSync;
} RTOSLIBMSGBOX_DESC;
```

Description
bInitialized
[] Used by the generic RtosLib source code.
bMsgBoxTaskRunning
[] Used by the generic RtosLib source code.
bMsgBoxTaskStop
[] Used by the generic RtosLib source code.
hTaskId
[out] Handle of the created task/thread.
hRtosLibApiSync
[] Used by the generic RtosLib source code.

8.1.1.42 RTOSLIB_OS_MSGBOX_TASK_STOP

Wait until the message box task has terminated.

UINT32 RTOSLIB_OS_MSGBOX_TASK_STOP(
PRTOSMSGBOX_DESC pRtosMsgBoxDesc)

Parameter

pRtosMsgBoxDesc
[in,out] RTOS message box descriptor.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Prior to this call the RtosLib initiated a shutdown of the MsgBoxTask. The adaptation layer should wait here until the task is finally gone. If after some timeout it is still running an error message should be generated and the task may be killed.
Note: Currently this function has to return RTE_ERROR on any RTOS. This function is intended only for the Windows part which is responsible to show a message box sent by the RTOS.

8.1.1.43 RTOSLIB_OS_MSGBOX_DISPLAY

Wait until the message box task has terminated.

```
UINT32 RTOSLIB_OS_MSGBOX_DISPLAY(
    const VMF_MESSAGE_BOX*      pVmfMsgBox,
    UINT32*                     pdwVmfUserReturnCode
);
```

Parameter

pVmfMsgBox

[in] VMF message box descriptor, see below

pdwVmfuserReturnCode

[out] VMF message box user return code representing the user's reaction on the message box. The following return codes are defined:

VMF_MESSAGEBOX_IDOK	→ OK button
VMF_MESSAGEBOX_IDCANCEL	→ CANCEL button
VMF_MESSAGEBOX_IDABORT	→ ABORT button
VMF_MESSAGEBOX_IDRETRY	→ RETRY button
VMF_MESSAGEBOX_IDIGNORE	→ IGNORE button
VMF_MESSAGEBOX_IDYES	→ YES button
VMF_MESSAGEBOX_IDNO	→ NO button

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Note: Currently this function has to return RTE_ERROR on any RTOS. This function is intended only for the Windows part which is responsible to show a message box sent by the RTOS.

VMF_MESSAGE_BOX

VMF message box descriptor

```
typedef struct _VMF_MESSAGE_BOX_A
{
    CHAR    tszTitle[VMF_MESSAGEBOX_MAX_TITLE_SIZE];
    CHAR    tszMessage[VMF_MESSAGEBOX_MAX_TEXT_SIZE];
    UINT32  bdwMsgValid;
    UINT32  dwFlags;
} VMF_PACKED(1) VMF_MESSAGE_BOX_A;
```

Description

tszTitle[]

[in] Message Box title.

tszMessage[]

[in] Message Box message.

bdwMsgValid

[] Used by the generic RtosLib source code.

dwFlags

[in] Message Box flags (currently hard coded flags for Windows).

8.1.1.44 RTOSLIB_OS_NOTIFICATION

Notification hook function. This function is called prior to event processing by the application.

UINT32 RTOSLIB_OS_NOTIFICATION(
UINT32 dwNotificationId)

Parameter

dwNotificationId

[in] Notification identifier. The following notification identifiers may be generated:

RTOS_NOTIFICATION_ID_BSOD	→ Windows BSOD
RTOS_NOTIFICATION_ID_SUSPEND	→ Windows Suspend
RTOS_NOTIFICATION_ID_RESUME	→ Windows Resume
RTOS_NOTIFICATION_ID_STOP	→ Uploader requests RTOS Stop

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is called on specific events. It is only intended for RTOS internal management. For example, if Windows is going to suspend (RTOS_NOTIFICATION_ID_SUSPEND) the RTOS has to shut down all its CPU cores and then to call the vmfSuspend() function. If the RTOS supports power management, these notification hooks may be used to inject the appropriate power events.
The notification hook is called directly after the notification is received (and prior to signaling the event to the application).
Note: RTOS_NOTIFICATION_ID_SUSPEND and RTOS_NOTIFICATION_ID_RESUME are currently not supported.

8.1.1.45 RTOSLIB_OS_NOTIFICATION_DONE

Notification done hook function. This function is called after event processing by the application.

UINT32 RTOSLIB_OS_NOTIFICATION_DONE(
UINT32 dwNotificationId)

Parameter

dwNotificationId

[in] Notification identifier. The following notification identifiers may be generated:

RTOS_NOTIFICATION_ID_BSOD	→ Windows BSOD
RTOS_NOTIFICATION_ID_SUSPEND	→ Windows Suspend
RTOS_NOTIFICATION_ID_RESUME	→ Windows Resume
RTOS_NOTIFICATION_ID_STOP	→ Uploader requests RTOS Stop

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

8.1.1.46 RTOSLIB_OS_TIMESYNC_TASK_START

Create and start the time synchronization helper task.

UINT32 RTOSLIB_OS_TIMESYNC_TASK_START(
PRTOSLIBTIMESYNC_DESC pRtosTimeSyncDesc)

Parameter

pRtosTimeSyncDesc
[in,out] Time synchronization descriptor, see below.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to create and start a new task which is responsible for time synchronization (time, date, timezone). The task itself is part of the generic RtosLib sourcecode, its entrypoint is defined as follows: `UINT32 RtosLibTimeSyncTask( RTOSLIBTIMESYNC_DESC* pRtosTimeSyncDesc)`

RTOSLIBTIMESYNC_DESC
Communication Descriptor

```
typedef struct _RTOSLIBTIMESYNC_DESC
{
    BOOL        bInitialized;
    BOOL        bTimeSyncTaskRunning;
    BOOL        bTimeSyncTaskStop;
    VMF_HANDLE  hTaskId;
    VMF_HANDLE  hRtosLibApiSync;
} RTOSLIBTIMESYNC_DESC;
```

Description
bInitialized
[] Used by the generic RtosLib source code.
bTimeSyncTaskRunning
[] Used by the generic RtosLib source code.
bTimeSyncTaskStop
[] Used by the generic RtosLib source code.
hTaskId
[out] Handle of the created task/thread.
hRtosLibApiSync
[] Used by the generic RtosLib source code.

8.1.1.47 RTOSLIB_OS_TIMESYNC_TASK_STOP

Wait until the time synchronization task has terminated.

UINT32 RTOSLIB_OS_TIMESYNC_TASK_STOP(
PRTOSLIBTIMESYNC_DESC pRtosTimeSyncDesc)

Parameter

pRtosTimeSyncDesc
[in,out] Time synchronization descriptor, see below.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

Prior to this call the RtosLib initiated a shutdown of the TimeSyncTask. The adaptation layer should wait here until the task is finally gone. If after some timeout it is still running an error message should be generated and the task may be killed.

8.1.1.48 RTOSLIB_OS_TIMESYNC_TIME_GET

Get and convert the current time and date.

UINT32 RTOSLIB_OS_TIMESYNC_TIME_GET(
VMF_STAMPED_TIME* pVmfStampedTime
const UINT32 dwReserved)

Parameter

pVmfStampedTime
[out] Time and date, see below.
dwReserved
[in] Currently always set to 0.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to determine the current time and date and store it into the VMF_STAMPED_TIME descriptor.

VMF_STAMPED_TIME
Time and Date descriptor

```
typedef struct _VMF_TIME
{
    UINT16 wYear;
    UINT16 wMonth;
    UINT16 wDayOfWeek;
    UINT16 wDay;
    UINT16 wHour;
    UINT16 wMinute;
    UINT16 wSecond;
    UINT16 wReserved;
    UINT32 dwNanosecond;
} VMF_PACKED(4) VMF_TIME;

typedef struct _VMF_STAMPED_TIME
{
    VMF_TIME    Time;
    UINT64      qwTimestamp;
} VMF_PACKED(4) VMF_STAMPED_TIME;
```

Description
VMF_TIME.wYear
Current year:
absolute: 1601 .. 30827
relative: 0
VMF_TIME.wMonth
Current month (January = 1, December = 12)
VMF_TIME.wDayOfWeek
Current day of the week (0 = Sunday, 6 = Saturday, 7 = unknown)
VMF_TIME.wDay
Current day:
absolute: year!=0, day in month (1 - 31)
relative: year=0, n-th occurrence of DayOfWeek in month (1 - 5; 5 equals last!)
VMF_TIME.wHour
Current hour (0..23)
VMF_TIME.wMinute
Current minute (0..59)
VMF_TIME.wSecond
Current second (0..59)

VMF_TIME.dwNanosecond
Current nanosecond (0..999999999)
VMF_STAMPED_TIME.Time
Current time (see above)
VMF_TIME.qwTimestamp
RTOS local Timestamp when the time and date was taken

8.1.1.49 RTOSLIB_OS_TIMESYNC_TIME_SET

Set the current time and date.

UINT32 RTOSLIB_OS_TIMESYNC_TIME_SET(
VMF_STAMPED_TIME* pVmfStampedTime
const UINT32 dwReserved)

Parameter

pVmfStampedTime
[in] Time and date.
dwReserved
[in] Currently always set to 0.

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to set the time and date stored in the VMF_STAMPED_TIME descriptor.

8.1.1.50 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_GETA

Get the current time zone (ASCII version).

UINT32 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_GETA(
VMF_TIMEZONE_A* pVmfTimezone)

Parameter

pVmfTimezone

[out] Timezone descriptor (all descriptor members are output values!).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to determine the current timezone and store it into the VMF_TIMEZONE_A timezone descriptor.

VMF_TIMEZONE_A
Timezone descriptor

```
typedef struct _VMF_TIMEZONE_A
{
    VMF_TIME    LocalTime;
    BOOL        bDaylightTime;
    BOOL        bAutoAdjustDSTime;
    INT32       nTimezoneBias;
    CHAR        tszStandardName[VMF_TIMEZONE_NAME_MAX_LENGTH];
    VMF_TIME    StandardDate;
    INT32       nStandardBias;
    CHAR        tszDaylightName[VMF_TIMEZONE_NAME_MAX_LENGTH];
    VMF_TIME    DaylightDate;
    INT32       nDaylightBias;
} VMF_PACKED(4) VMF_TIMEZONE_A;
```

Description

System information

LocalTime
Current local time/date (when the timezone information was determined)

bDaylightTime
TRUE if currently daylight saving is on (local time/date (LocalTime) is within given daylight period, i.e. between StandardDate and DaylightDate)

bAutoAdjustDSTime
TRUE if this OS automatically adjusts daylight saving

System information

nTimezoneBias
Timezone bias relative to UTC in minutes (e.g. +60 for Germany/Berlin)

tszStandardName
Timezone name for standard time (OS specific), when daylight saving is off

StandardDate
Transition date to standard time (date when daylight saving will be turned off)

nStandardBias
additional bias used when daylight saving time is off (0 in most timezones)

tszDaylightName
Timezone name for daylight saving time (OS specific) , when daylight saving is on

DaylightDate
Transition date to daylight saving time (date when daylight saving will be turned on)

nDaylightBias
additional bias used when daylight saving time is on (-60 in most timezones)

8.1.1.51 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_GETW

Get the current time zone (wide char version).

UINT32 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_GETW(
VMF_TIMEZONE_W* pVmfTimezone)

Parameter

pVmfTimezone
[out] Timezone descriptor (all descriptor members are output values!).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function is the wide char version of OSTimezoneSyncTimezoneGetA.

8.1.1.52 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_SETA

Set the current time zone (ASCII version).

UINT32 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_SETA(
VMF_TIMEZONE_A* pVmfTimezone)

Parameter

pVmfTimezone
[in] Timezone descriptor (all descriptor members are input values!).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to set the current timezone.

8.1.1.53 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_SETW

Set the current time zone (wide char version).

UINT32 RTOSLIB_OS_TIMEZONESYNC_TIMEZONE_SETW(
VMF_TIMEZONE_W* pVmfTimezone)

Parameter

pVmfTimezone
[in] Timezone descriptor (all descriptor members are input values!).

Return

RTE_SUCCESS on success and an error-code on failure.

Comment

This function has to set the current timezone.

8.2 RTOS Library – Application Layer API

The API is described in the RTOS VM User Manual.

8.3 RTOS Library example applications

Shipped with VmfWin are some example applications for VxWorks and Windows CE which show how to use the RTOS library in a user application. The example applications may serve as a starting point.
The following examples are extracted from the ACONTIS CeWin product (Windows CE + Windows)
…\Examples\RtosLib\CeWin\WinCE
→ Windows CE example applications
…\Examples\RtosLib\CeWin\Windows
→ Windows example applications
The following examples are extracted from the ACONTIS VxWin product (VxWorks + Windows)
…\Examples\RtosLib\VxWin\VxWorks
→ VxWorks example applications
…\Examples\RtosLib\VxWin\Windows
→ Windows example applications