ETAS

ETAS RTA Lightweight Hypervisor

User Manual

DRIVING | EMBEDDED EXCELLENCE

Copyright

Document LightweightHypervisorUserManual v1.1.0 R01 EN – 11.2020

Contents

Figures

Document History

Date	Summary	State
25 March 2016	First draft	Draft
15 March 2017	First complete version	Draft
04 April 2017	Updated after review	Released V1.0.0 R01
07 September 2017	Separated into core manual (this document) and target specific manual. Added support for multiple application cores in the configuration (see section 6.3.2). Added support for mapping application cores to physical cores in the configuration (see section 6.3.3). Added descriptions of `LWHVR_GlobalUnlockCallback()`, `LWHVR_GlobalRelockCallback()`, `LWHVR_CoreUnlockCallback()` and `LWHVR_CoreRelockCallback()`. Updated section 7.7 to include target specific error codes.	Draft
08 November 2017	Updated section 7.7 to include `LWHVR_ErrorStackInvalid` error code.	Draft
08 December 2017	Updated section 7.7 to include `LWHVR_ErrorInitializing` error code. Updated section 8.1 (`LWHVR_Init`) to explain that `LWHVR_Init()` must be called again before re-starting the RTA-LWHVR when it has been stopped with `LWHVR_Stop()`. Updated section 8.2 (`LWHVR_Start`) to mention re-starting individual application cores. Added note about `LWHVR_ErrorCallback()` being called with the `LWHVR_ErrorInitializing` error code to sections 8.5 (`LWHVR_StopVM`), 8.6 (`LWHVR_ShutdownVM`), 8.7 (`LWHVR_RestartVM`) and 8.8 (`LWHVR_RequestExtraTimeForVM`). Added restriction about APIs running without pre-emption to sections 8.5 (`LWHVR_StopVM`), 8.6 (`LWHVR_ShutdownVM`), 8.7 (`LWHVR_RestartVM`) and 8.8 (`LWHVR_RequestExtraTimeForVM`). Removed restriction about master software API functions being called before `LWHVR_AllHaveStarted()` returns `LWHVR_TRUE`. Update section 4.6 with the need to re-call `LWHVR_Init()`. Added section 4.7 about re-starting individual application cores.	Draft
30 January 2018	Fixed some spelling mistakes.	Draft
20 March 2018	Added section 5.8.8 and a note to section 8.6 about pseudo-interrupts being disabled and pending pseudo-interrupts being cleared when a VM is re-started.	Draft
8 October 2020	Updated ETAS address.	Draft
19 October 2020	Updated section 9.13 to note that	Draft

	`LWHVR_RequestExtraTimeForVM()` may be called by `LWHVR_ClockCallback()`.	
16 November 2020	Updated after review: fixed UK vs US spelling inconsistencies.	Released v1.1.0 R01

1 Introduction

This document is the user manual for the ETAS RTA Lightweight Hypervisor (RTA-LWHVR). The intended audience for this document is the integrator who builds the RTA-LWHVR into an ECU and the application developer who creates virtual machines (VMs) to be run under the control of the RTA-LWHVR.

This document describes the target independent features of the RTA-LWHVR. Please see the target specific user manual for your target, "LightweightHypervisorUserManual-<target>.pdf", for target specific details.

1.1 Safety Notice

WARNING!

The use and application of this product can be dangerous. It is critical that you carefully read and follow the instructions and warnings below and in all associated user manuals.

This ETAS product fulfills standard quality management requirements. If requirements of specific safety standards (e.g. IEC 61508, ISO 26262) need to be fulfilled, these requirements must be explicitly defined and ordered by the customer. Before use of the product, customer must verify the compliance with specific safety standards.

This ETAS product enables a user to influence or control the electronic systems in a vehicle or in a test-bench. THE PRODUCT IS SPECIFICALLY DESIGNED FOR THE EXCLUSIVE USE BY PERSONNEL WHO HAVE SPECIAL EXPERIENCE AND TRAINING.

Improper use or unskilled application of this ETAS product may alter the vehicle performance or system performance in a manner that results in death, serious personal injury or property damage.

- Do not use this ETAS product if you do not have the proper experience and training.
- Also, if a product issue develops, ETAS will prepare a Known Issue Report (KIR) and post it on the internet. The report includes information regarding the technical impact and status of the solution. Therefore you must check the KIR applicable to this ETAS product version and follow the relevant instructions prior to operation of the product.

  The Known Issue Report (KIR) can be found here: http://www.etas.com/kir
- Any data acquired through the use of this ETAS product must be verified for reliability, quality and accuracy prior to use or distribution. This applies both to calibration data and to measurements that are used as a basis for calibration work.
- When using this ETAS product with vehicle systems that influence vehicle behavior and can affect the safe operation of the vehicle, you must ensure that the vehicle can be transitioned to a safe condition if a malfunction or hazardous incident should occur.
- When using this ETAS product with test-bench systems that influence system behavior and can affect the safe operation of the system, you must ensure that the test-bench can be transitioned to a safe condition if a malfunction or hazardous incident should occur.
- All legal requirements, including regulations and statutes regarding motor vehicles and test-benches, must be strictly followed when using this product.
- It is recommended that in-vehicle use of the ETAS product be conducted on enclosed test tracks.
- Use of this ETAS product on a public road should not occur unless the specific calibration and settings have been previously tested and verified as safe.

DANGER!

IF YOU FAIL TO FOLLOW THESE INSTRUCTIONS, THERE MIGHT BE A RISK OF DEATH, SERIOUS INJURY OR PROPERTY DAMAGE.

THE ETAS GROUP OF COMPANIES AND THEIR REPRESENTATIVES, AGENTS AND AFFILIATED COMPANIES DENY ANY LIABILITY FOR THE FUNCTIONAL IMPAIRMENT OF ETAS PRODUCTS IN TERMS OF FITNESS, PERFORMANCE AND SAFETY IF NON-ETAS SOFTWARE OR MODEL COMPONENTS ARE USED WITH ETAS PRODUCTS OR DEPLOYED TO ACCESS ETAS PRODUCTS. ETAS PROVIDES NO WARRANTY OF MERCHANTABILITY OR FITNESS OF THE ETAS PRODUCTS IF NON-ETAS SOFTWARE OR MODEL COMPONENTS ARE USED WITH ETAS PRODUCTS OR DEPLOYED TO ACCESS ETAS PRODUCTS.

THE ETAS GROUP OF COMPANIES AND THEIR REPRESENTATIVES, AGENTS AND AFFILIATED COMPANIES SHALL NOT BE LIABLE FOR ANY DAMAGE OR INJURY CAUSED BY IMPROPER USE OF THIS PRODUCT. ETAS PROVIDES TRAINING REGARDING THE PROPER USE OF THIS PRODUCT.

1.2 Definitions and Abbreviations

API
: Application Programmer's Interface.

Application core
: A processor core that runs application software.

Application software
: Software that runs inside a virtual machine on an application core.

Integrator
: The person who integrates the RTA-LWHVR into an ECU system.

RTA-LWHVR
: ETAS RTA Lightweight Hypervisor

Master core
: The processor core that controls the ECU system and performs most of the I/O.

Master software
: All software, except for the RTA-LWHVR itself, that does not run inside VMs.

Pseudo-interrupt
: An asynchronous interrupt-like event injected into a VM.

Time-slicing
: A method of sharing a processor's time between multiple VMs.

Virtual Machine (VM)
: A container or "sandbox" that runs application software in a way that the application software cannot interfere with application software running in another virtual machine or the master software.

VM Status Block
: A block of memory that is used to communicate information between the RTA-LWHVR and a VM.

1.3 Conventions

The following typographical conventions are used in this document:

`OCI_CANTxMessage msg0 =`	Code snippets are presented on a gray background and in the Courier font. Meaning and usage of each command are explained by means of comments. The comments are enclosed by the usual syntax for comments.
Choose **File** → **Open**.	Menu commands are shown in boldface.
Click **OK**.	Buttons are shown in boldface.
Press <ENTER>.	Keyboard commands are shown in angled brackets.
The "Open File" dialog box is displayed.	Names of program windows, dialog boxes, fields, etc. are shown in quotation marks.
Select the file `setup.exe`	Text in drop-down lists on the screen, program code, as well as path- and file names are shown in the Courier font.
A *distribution* is always a one-dimensional table of sample points.	General emphasis and new terms are set in italics.

2 Installation

The contents of the RTA-LWHVR distribution are as follows:

`Hypervisor\`

This directory contains the source code for the embedded hypervisor.

`Configuration Generator\`

This directory contains the configuration generator tool executable.

`Documentation\`

This directory contains documentation.

There is no installation process beyond copying files to a Windows PC.

To use the RTA-LWHVR a `LWHVR_Configuration.h` file should be created – see section 6 – and then the source code compiled and linked into your system – see section 4.

3 ETAS RTA Lightweight Hypervisor Concepts

This chapter explains what the ETAS RTA Lightweight Hypervisor is and provides background information to help understand later chapters.

The ETAS RTA Lightweight Hypervisor (RTA-LWHVR) is used in ECUs with multicore processors where one core, the *master core*, runs software directly on the hardware – e.g. RTA-OS and other AUTOSAR software - and other, *application cores*, run independent applications contained inside virtual machines (VMs).

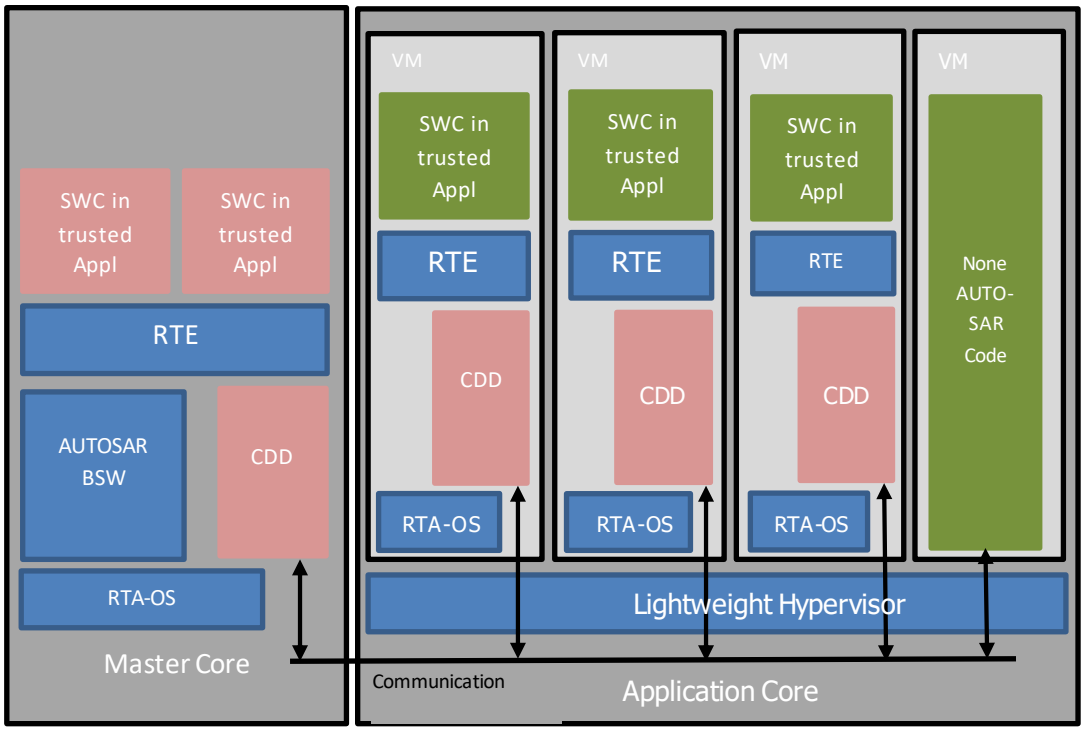


Figure 1: A RTA-LWHVR System

The software running on the master core is called the *master software*. The master software is responsible for all interaction with hardware. The master software contains device drivers and their interrupt handlers. VMs do not access hardware[1] or contain hardware interrupt handlers. When an application running in a VM needs to access a peripheral, the access is carried out by the master software on behalf of the VM.

A VM is a container for some software. A VM provides an execution environment that is almost the same as the environment provided by the application core if the RTA-LWHVR were not present. The RTA-LWHVR is agnostic to what the software is that runs inside a VM. For example, a VM could run hand written assembler, a simple C code program, or an AUTOSAR system with RTA-OS, an RTE and Application Software Components.

The software running inside a VM is isolated from the master software and other VMs except where explicit communication (via shared memory) occurs. Each VM is assigned a portion of the system's memory and the processor's memory protection unit (MPU) is used to ensure that a VM can only access the memory it has been assigned.

VMs execute code in one of the processor's less privileged modes (e.g. "user" mode) to ensure that the code in a VM cannot damage the master software, RTA-LWHVR or another VM.

[1] In fact a VM could directly access a hardware register if the register were memory mapped and the VM were given permission to access the register's memory address. However this would not be the normal way of using the RTA-LWHVR.

One application core may support multiple VMs. This is done by sharing the core between VMs by using *time-slicing*. A time-slice is a period of time for which a VM runs. The RTA-LWHVR scheduler runs a VM for a time-slice, then suspends the VM and runs another VM for a time-slice.

The master software and the RTA-LWHVR code are compiled and linked into a single memory image (e.g. elf, Intel hex or Motorola SRec). The software that will run in a VM is compiled and linked into a memory image that is separate from the master software and the other VMs. A complete system therefore has multiple separate memory images; one for the master software/RTA-LWHVR and one for each VM. These images may be merged into a single image (e.g. Intel hex or Motorola SRec) file and flashed to the ECU or may be flashed separately.

3.1 Time-Slicing

Each application core may support multiple VMs. This is done by sharing the core's time between the VMs using *time-slicing*. For each application core the RTA-LWHVR's configuration contains a *schedule table.* Each entry in a schedule table defines a *time-slice*. A time-slice definition specifies the length (duration) of the time-slice and either the identity of the VM that should run in the time-slice or an indication that the time-slice is spare and can be assigned to VMs dynamically.

The RTA-LWHVR contains a scheduler for each application core that is invoked by a clock-tick interrupt. Normally a scheduler decides on the order in which to run VMs by iterating though schedule table entries in the order they occur in the schedule table (returning to the first entry when it reaches the end of the table) and running the specified VMs for the specified durations, or idling if a time-slice is spare. However, limited dynamic scheduling of VMs is also support – please see section 8.8 and section 11.5.

When a scheduler runs a VM it does the following:

1. Loads the VM's execution context. The VM's execution context consists of the core's general-purpose registers, user-mode accessible system registers, program counter, and the MPU configuration.
2. Executes the VM's code.
3. Saves the VM's execution context.

3.2 Master Software and VM Separation

The master software/RTA-LWHVR and each VM is compiled and linked separately into its own separate memory image.

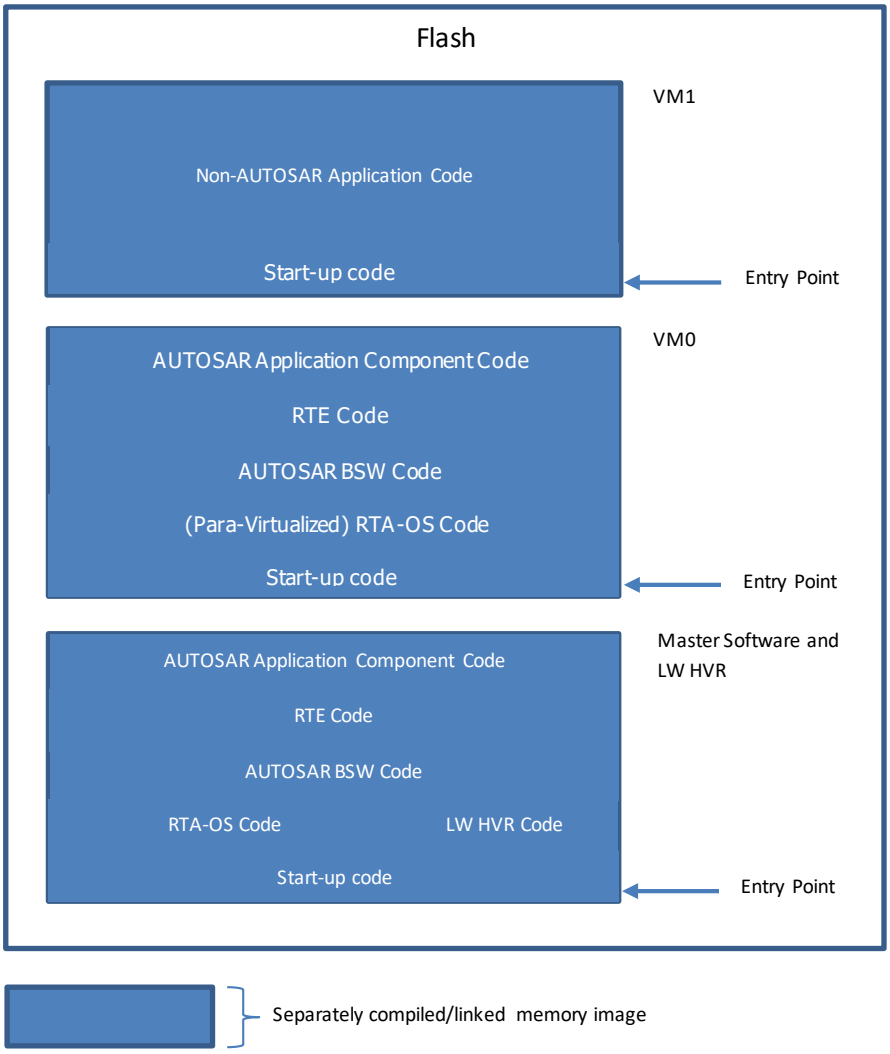


Figure 2: Separate Master Software and VM Memory Images

The master software and VMs may communicate via shared memory but they cannot call each other's functions or access each other's variables.

The RTA-LWHVR provides a VM with a virtual processor consisting of registers and a portion of the real processor's memory. The RTA-LWHVR is agnostic about what the code running in the VM does with this virtual processor. A combination of memory protection, time-slicing and running code in a less privileged mode ensures that a VM cannot damage the master software, RTA-LWHVR or another VM.

Figure 3 shows how the processor's MPU might be configured for a system with two VMs. Areas of Flash and RAM are reserved for the master software. Each VM also has private areas of flash and RAM reserved for it, and there is an area of RAM shared by the VMs. When VM0 is running the processor's MPU is configured so that VM0 can only access its private areas of flash and RAM, and the shared area of RAM. Likewise when VM1 is running the processor's MPU is configured so that VM1 can only access its private areas of flash and RAM, and the shared area of RAM.

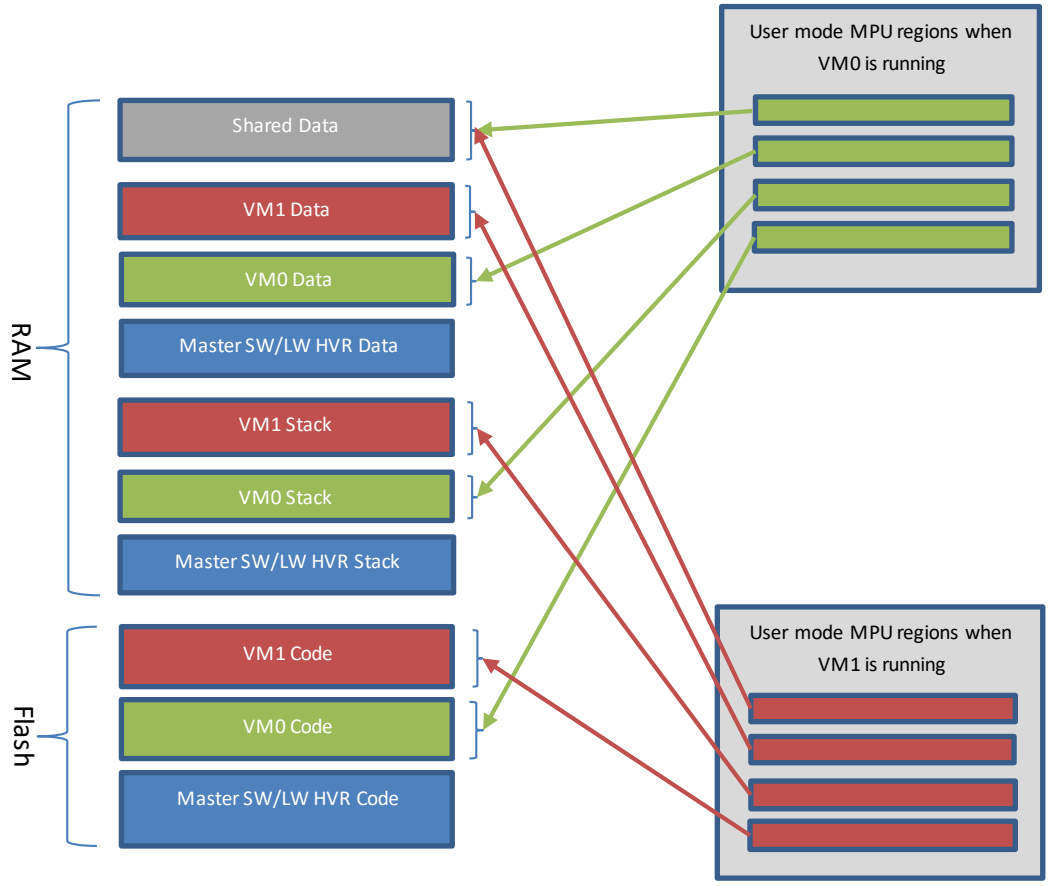


Figure 3: Virtual Machine Memory Regions

3.3 The Master Software Controls the System

The master software is in overall control of the system. When the processor starts, the master core will start executing code (e.g. from the reset vector) and then carry out a sequence of actions something like:

Setup the stack-pointer

Call the C start-up code.

Call `main()`.

Initialize the master core's hardware (peripherals).

Call `LWHVR_Init()`.

Start the application core(s).

Wait until the application cores have started the RTA-LWHVR by polling `LWHVR_HaveAllStarted()`.

Start RTA-OS.

When an application core is started it carries out a sequence of actions something like:

Setup the stack-pointer.

Call the C start-up code (not always required).

Call `main()`.

Initialize the application core's hardware (peripherals).

Call `LWHVR_Start()` to start the RTA-LWHVR.

Once this start-up sequence has completed the application cores will start running VMs according to their schedule tables. The master software can control the behaviour of the RTA-LWHVR and VMs by using the master software API – see sections 4.10 and 8.

3.4 Pseudo-Interrupts

Although VMs do not handle interrupts generated by hardware, VMs need to be able to handle asynchronous events such as timer ticks and shut down requests. This is done through the *pseudo-interrupt* mechanism. Pseudo-interrupts are similar to hardware interrupts except they are generated and controlled by the RTA-LWHVR software – see section 5.8).

3.5 APIs

The RTA-LWHVR provides two APIs, one to allow the master software to control the RTA-LWHVR and VMs, and one to allow VMs to request services from the RTA-LWHVR.

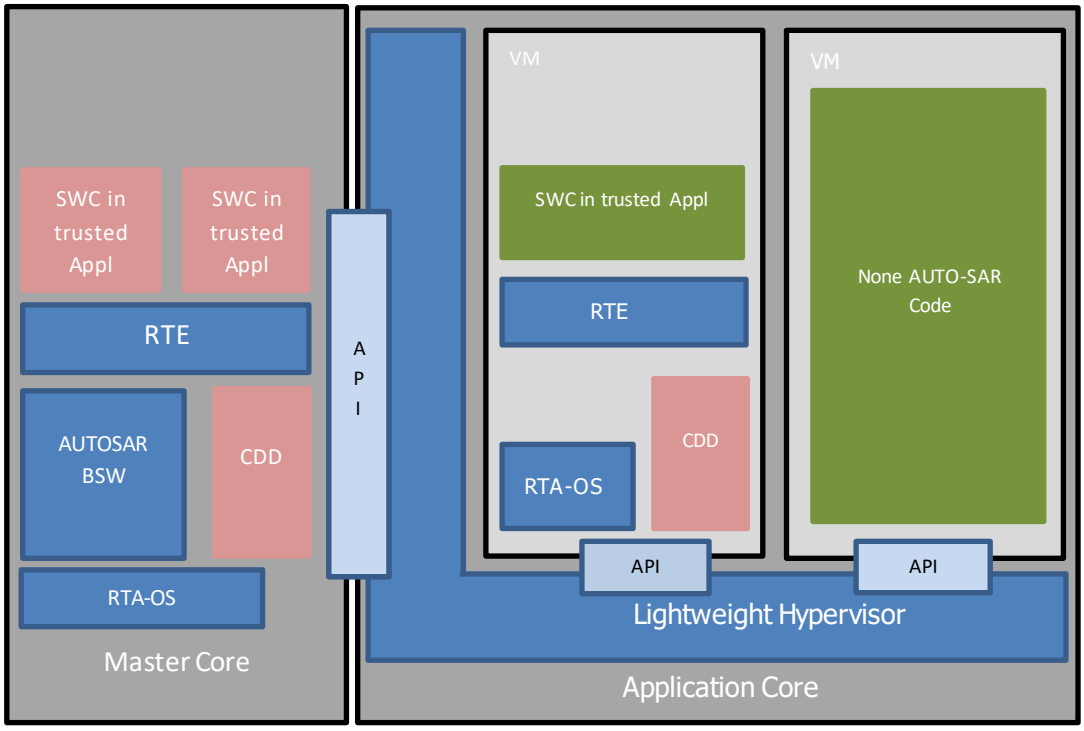


Figure 4: APIs in a RTA-LWHVR System

Since the master software and RTA-LWHVR are compiled and linked into a single memory image the API provided to the master software uses normal C function calls and call-backs – see chapters 8 and 9.

The VMs and the RTA-LWHVR are in different memory images however. The API provided to VMs therefore uses "trap" instructions to transfer control to the RTA-LWHVR via an interrupt – see chapter 11.

4 Master Software

The master software consists of

- The code that runs when the master core comes out of reset.
- C start-up code and libraries.
- Possibly an operating system that runs on the master core (e.g. RTA-OS).
- Any other software that must run on the master core - e.g. AUTOSAR Basic Software, an RTE and AUTOSAR Application Software Components.
- Some call-back functions used by the RTA-LWHVR (these run on the master core and application cores).
- Code that runs on the application cores to initialize the cores and start the RTA-LWHVR running on the cores.

Please see chapter 3 for information about how the master software fits into a RTA-LWHVR system. Chapter 8 provides details of the RTA-LWHVR API functions used by the master software, and chapter 9 describes the call-back functions used by the RTA-LWHVR.

Please note that the call-back functions must be implemented by the integrator who integrates the RTA-LWHVR into the ECU system.

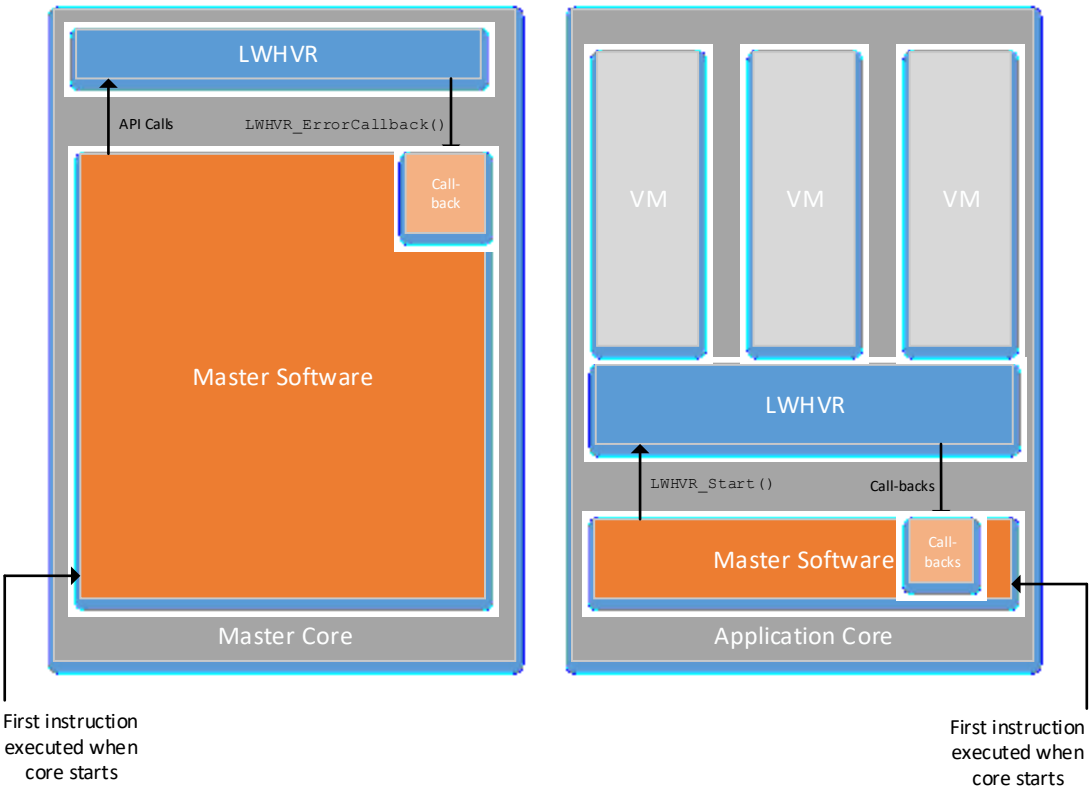


Figure 5: Master Software

4.1 Configuring RTA-OS

The master core does not have to run an operating system, however in most cases it will. This section assumes that the master core runs RTA-OS (that is a normal port of RTA-OS that directly manages the core's interrupts etc.) as part of the master software.

Although the RTA-LWHVR manages the application cores, RTA-OS must be aware of the application cores. To achieve this the RTA-OS configuration must contain the total number of cores under `OsInfo/OsNumberOfCores`, but OS applications must only be assigned to the master core.

For example, on a two core processor one might specify the total number of processors like:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>OsInfo</SHORT-NAME>
    <DEFINITION-REF DEST='ECUC-PARAM-CONF-CONTAINER-
DEF'>/ETAS_RTAOS/Os/OsOS</DEFINITION-REF>
    <PARAMETER-VALUES>
      <ECUC-NUMERICAL-PARAM-VALUE>
        <DEFINITION-REF DEST='ECUC-INTEGER-PARAM-
DEF'>/ETAS_RTAOS/Os/OsOS/OsNumberOfCores</DEFINITION-REF>
          <VALUE>2</VALUE>
      </ECUC-NUMERICAL-PARAM-VALUE>
    </PARAMETER-VALUES>
    …
```

And then assign OS applications to the master core (AUTOSAR core 0) like:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>Application0</SHORT-NAME>
    <DEFINITION-REF DEST='ECUC-PARAM-CONF-CONTAINER-
DEF'>/ETAS_RTAOS/Os/OsApplication</DEFINITION-REF>
      <PARAMETER-VALUES>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST='ECUC-BOOLEAN-PARAM-
DEF'>/ETAS_RTAOS/Os/OsApplication/OsTrusted</DEFINITION-REF>
            <VALUE>true</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST='ECUC-INTEGER-PARAM-
DEF'>/ETAS_RTAOS/Os/OsApplication/OsApplicationCoreAssignment</DEFINITION-
REF>
          <VALUE>0</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
        …
```

This results in the application cores being *non-AUTOSAR cores*.

4.2 Building the RTA-LWHVR with the Master Software

The RTA-LWHVR source code should be compiled and linked together with the components of the master software so that the RTA-LWHVR is part of the same memory image as the master software. The same compiler and linker options used to build the master software should be used to build the RTA-LWHVR. (Please see the RTA-LWHVR release notes for details of the toolchain and compiler options used for testing.)

4.2.1 Locating the RTA-LWHVR Sections

The sections containing RTA-LWHVR code and data must be located when the master software and RTA-LWHVR are linked. The section names are target specific.

4.2.2 Locating the RTA-LWHVR Vector Table

The RTA-LWHVR's interrupt vector tables used for application cores must also be located when the master software is linked. Details of the interrupt vector table are target specific.

4.3 Initialising the Master Core

The RTA-LWHVR does not carry out any hardware initialisation on master cores. Therefore, before any RTA-LWHVR APIs are called on a master core the hardware must be initialized. For example, processor and peripheral clocks must be configured, data/code caches must be configured, and interrupts may need to be initialized.

4.4 Setting up the C Execution Environment

On both master and application cores the RTA-LWHVR assumes that a valid C execution environment exists when any of its APIs are called. This includes a valid stack, initialized small data-area registers, and initialized data in initialized data sections. Typically, such setup would be done by calling the compiler's C start-up code.

4.5 Starting Application Cores and the RTA-LWHVR

The API function `LWHVR_Init()` **must** be called by the master software on the master core before any other RTA-LWHVR API is used.

Once `LWHVR_Init()` has been called, the master software running on the master core is responsible for starting the application cores that run the RTA-LWHVR. (Since the application cores are non-AUTOSAR cores they must be started with the RTA-OS function `StartNonAutosarCore()`.)

Once an application core is running it should carry out any target specific initialization needed and then call the API function `LWHVR_Start()`. `LWHVR_Start()` will configure the MPU and configure the interrupt controller to route the configured clock-tick interrupt to the RTA-LWHVR (including setting up the interrupt vector table for the application core). However, the RTA-LWHVR will not carry out any other hardware initialisation.

WARNING!

Whilst initializing the RTA-LWHVR on an application core, `LWHVR_Start()` will configure the interrupt controller to route the clock-tick interrupt to the RTA-LWHVR. To avoid contention when configuring the interrupt controller, once `LWHVR_Start()` has been called by any application core, no other software may change the configuration of the interrupt controller until the API function `LWHVR_AllHaveStarted()` returns `LWHVR_TRUE`.

If you are using RTA-OS the following rules **must** be followed:

- Application cores must **not** call any RTA-OS functions, including `Os_InitializeVectorTable()`.
- Application cores must not call `LWHVR_Start()` until after the master core has called `Os_InitializeVectorTable()`.

Once `LWHVR_Start()` has completed initialisation the RTA-LWHVR will start running time-slices as defined by the application core's schedule table. `LWHVR_Start()` will only return when the RTA-LWHVR has been stopped by calling the API `LWHVR_Stop()`.

WARNING!

It is recommended that the API function `LWHVR_HaveAllStarted()` be used by the master software to determine when the application cores have called `LWHVR_Start()` and completed sufficient initialization to allow other RTA-LWHVR APIs to be called without error.

A typical start-up sequence for a two core system would be:

```
OS_MAIN() {
  StatusType Status;

  if (OS_CORE_ID_MASTER == GetCoreID()) {
    /* Initialze the master core - calls Os_InitializeVectorTable(). */
    InitTargetMaster();
    LWHVR_Init();
    StartCore(OS_CORE_ID_0, &Status);
    StartNonAutosarCore(OS_CORE_ID_1, &Status);
    while (LWHVR_HaveAllStarted() != LWHVR_TRUE) { /* Idle. */ }
    StartOS(OSDEFAULTAPPMODE);
  }
  else {
    /* Initialize the application core -
     * does not call Os_InitializeVectorTable(). */
    InitTargetApplication();
    LWHVR_Start();
  }
}
```

4.6 Stopping and Re-starting the RTA-LWHVR

The RTA-LWHVR may be stopped using the RTA-LWHVR API `LWHVR_Stop()` – see section 8.4.

To re-start the RTA-LWHVR after it has been stopped with `LWHVR_Stop()` the following should be done:

1. The master software must wait until all application cores have returned from `LWHVR_Start()`.
2. Then, the master software must call `LWHVR_Init()` again.
3. Next, the applications cores should call `LWHVR_Start()` again.
4. Finally, master software should wait until `LWHVR_AllHaveStarted()` returns `LWHVR_TRUE` before using any RTA-LWHVR API functions.

4.7 Re-Starting the RTA-LWHVR on Individual Application Cores

There may be situations when it is necessary to re-start the RTA-LWHVR on one application core without stopping and re-starting the RTA-LWHVR on all cores as described in section 4.6. For example, a trap occurs (resulting in `LWHVR_UnexpInterruptHook` or `LWHVR_UnexpInterruptCallback()` being invoked) that can only be resolved by re-starting the application core. In this case the application core can call `LWHVR_Start()` again to re-start the RTA-LWHVR on itself without affecting other application cores.

For example, the following may take place:

1. `LWHVR_UnexpInterruptHook` is invoked due to an unrecoverable trap occurs.
2. `LWHVR_UnexpInterruptHook` forces a core reset.
3. The core re-starts.
4. The core sets up a C environment.

5. The core calls `LWHVR_Start()`.

WARNING!

`LWHVR_Start()` may only be used to re-restart the RTA-LWHR on an individual application core if the LWHVR's memory is not modified by resetting/re-starting the application core.

4.8 Clock-Tick Interrupt Source

The RTA-LWHVR scheduler is driven by a timer interrupt referred to as the *clock-tick interrupt*. The clock-tick interrupt source used for each application core must be specified in the configuration of the application core (see section 6.3.2) and the master software must provide two call-back functions to manage the clock-tick interrupt.

4.8.1 Clock-Tick Frequency and Ticks

The RTA-LWHVR does not care what at what frequency the clock-tick interrupt occurs. The RTA-LWHVR works entirely in *ticks* – a tick being the interval between two clock-tick interrupts.

4.8.2 Clock-Tick Initialization

The master software is responsible for initialising the peripheral that generates the clock-tick interrupt. The RTA-LWHVR will call the call-back function `LWHVR_StartTimerCallback()` on each application core to configure and start the clock-tick interrupt source for that application core.

4.8.3 Clock-Tick Reset

When a clock-tick interrupt occurs, the RTA-LWHVR will call the call-back function `LWHVR_ClockCallback()` to reset the clock-tick interrupt source so that it generates another interrupt in one tick.

4.8.4 Ideal Clock-Tick Sources

The ideal clock-tick source is something like a programmable interval timer (PIT) that generates interrupts at a fixed frequency. This has the advantage of avoiding drift in VM time-slicing. The RTA-LWHVR will call `LWHVR_ClockCallback()` very quickly after the clock-tick interrupt arrives, but there will still be a short delay between the clock-tick interrupt arriving and `LWHVR_ClockCallback()` being called. This means that if a clock-tick interrupt source is used that generates another interrupt one tick after `LWHVR_ClockCallback()` is called, the ticking of the RTA-LWHVR scheduler will drift with respect to "wall-clock time". This type of timer may also introduce jitter into time-slicing because interrupts are disabled when the RTA-LWHVR is handling VM API calls.

WARNING!

Please see section 12 for more information about avoiding timing issues

4.9 RTA-LWHVR Stack Usage on Application Cores

When running on an application core the RTA-LWHVR will use stack. It is not possible to say how much stack will be needed as this depends on exactly how the RTA-LWHVR was compiled and how much stack is used by call-backs such as `LWHVR_ClockCallback()` or `LWHVR_VMErrorCallback()`. It is highly recommended that the integrator uses a technique such as stack colouring to work out the maximum amount of stack needed by the RTA-LWHVR and call-backs on an application core and allocates stack accordingly.

4.10 Controlling the RTA-LWHVR and VMs

The RTA-LWHVR provides API functions that the master software may call to control the behaviour of the RTA-LWHVR and VMs:

`LWHVR_Stop()` can be used to stop the whole RTA-LWHVR from running.

`LWHVR_ShutdownVM()` can be used to inject a shutdown interrupt into a VM.

`LWHVR_StopVM()` can be used to forcibly stop a VM.

`LWHVR_RestartVM()` can be used to re-start a VM that has been shut down, forcibly stopped or is in error.

`LWHVR_RequestExtraTimeForVM()` can be used to request that a VM be given extra execution time.

Please see chapter 8 for details.

The call-back functions `LWHVR_StoppedVMCallback()`, `LWHVR_StoppedVMCallback()` and `LWHVR_VMErrorCallback()` are called by the RTA-LWHVR to inform the master software about the state of VMs. Please see chapter 9 for details.

4.11 Communicating with and between VMs

Communication between the master software and VMs and between VMs should be done via shared memory. The RTA-LWHVR does not mandate any particular shared-memory protocol or implement any particular protocol. The RTA-LWHVR does provide API functions to allow memory to be copied atomically to support shared-memory communication. Please see 11.6 for details.

4.12 Restrictions on the Master Software

The master software (including call-backs) must not:

- Modify the configuration of the processor's interrupt controller that is associated with routing clock-tick interrupts to application cores.
- Modify an application core's MPU configuration.

5 Virtual Machines

This chapter provides details of virtual machines (VMs). Please see chapter 3 for information about how VMs fit into a RTA-LWHVR system, and chapter 11 for the API services that may be called by VMs.

5.1 What VMs Can and Cannot Do

In general a VM should be thought of in the same way as the software that would run on a processor core in the absence of a hypervisor. The RTA-LWHVR is agnostic to the operation of the software in a VM with the following restrictions:

- Memory regions that a VM can access are specified in the VM's configuration. If the VM tries to access memory outside of one of these regions, or access memory in one of these regions in a way that is not allowed (e.g. write to a region for which it only has read permission) then the VM will be in error (see section 5.6).
- If the VM tries to execute instructions that would allow it to threaten the integrity of the RTA-LWHVR or another VM then the VM will be in error. Generally this means that the VM cannot execute "privileged" instructions.
- If the VM tries to modify processor configuration that would allow it to threaten the integrity of the RTA-LWHVR or another VM then the VM will be in error. Generally this means that the VM cannot access "privileged" registers.

5.2 Scheduling VMs

The RTA-LWHVR configuration specifies a schedule table for each application core (see section 6.3.5). A schedule table is composed of one more entries. Each entry describes a time-slice. A time-slice identifies a VM that should be executed, or an indication that the time-slice is spare, and the duration of the time-slice it ticks.

The RTA-LWHVR contains a scheduler for each application core that is invoked by a clock-tick interrupt. Normally a scheduler decides on the order in which to run VMs by iterating though schedule table entries in the order they occur in the application core's schedule table (returning to the first entry when it reaches the end of the table) and running the specified VMs for the specified durations, or idling if a time-slice is spare. However, limited dynamic scheduling of VMs is also supported – please see section 8.8 and section 11.5.

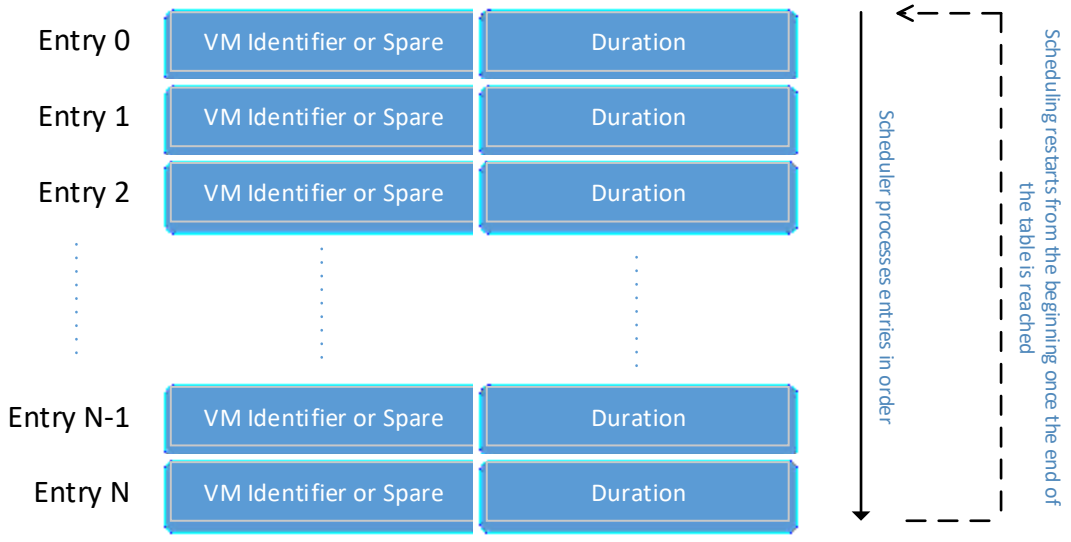


Figure 6: A Schedule Table

When a scheduler runs a VM it does the following:

1. Loads the VM's execution context. The VM's execution context consists of the core's general-purpose registers, user-mode accessible system registers, program counter, and the MPU configuration.
2. Executes the VM's code.
3. Saves the VM's execution context.

The RTA-LWHVR carries out scheduling separately for each application core.

5.2.1 Example of Scheduling

Consider the following example of a schedule table:

VM	Duration
VM0	1
VM1	1
Spare	1
VM2	2
VM1	1
VM3	1

The VMs will be executed as follows:

1. When the first clock-tick interrupt occurs VM0 will be executed for 1 clock-tick.
2. When the second clock-tick interrupt occurs VM1 will be executed for 1 clock-tick.
3. When the third clock-tick interrupt occurs the RTA-LWHVR will find the spare time-slice and will idle for 1 clock-tick.
4. When the fourth clock-tick interrupt occurs VM2 will be executed for 1 clock-tick.
5. When the fifth clock-tick interrupt occurs VM2 will be executed for another 1 clock-tick.
6. When the sixth clock-tick interrupt occurs VM1 will be executed again for 1 clock-tick.
7. When the seventh clock-tick interrupt occurs VM3 will be executed for 1 clock-tick.
8. When the eighth clock-tick interrupt occurs the scheduler will return to the start of the schedule table and VM0 will be executed for 1 clock-tick.

This is illustrated in Figure 7.

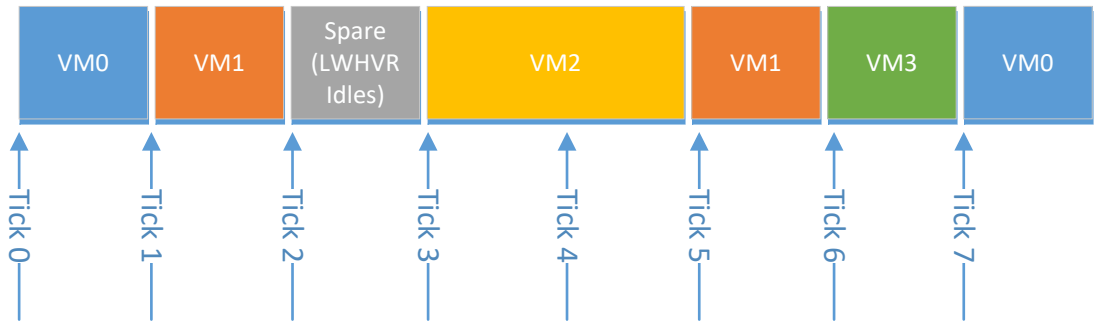


Figure 7: Simple Scheduling

5.3 VM Identifiers

A VM identifier is a small integer that corresponds to the VM's number in the configuration. That is, the VM described by macros `LWHVR_VM<N>_XXXX` in `LWHVR_Configuration.h` would have identifier `<N>` (see section 6.3.4).

For example, the VM described by macros `LWHVR_VM0_XXXX` would have identifier 0, the VM described by macros `LWHVR_VM1_XXXX` would have identifier 1, the VM described by macros `LWHVR_VM2_XXXX` would have identifier 2, and so on.

5.4 Building VMs

Each VM has a separate executable image and is compiled and linked separately from every other VM. The VM's image must contain all code that the VM needs; e.g. C start-up code and libraries (in general VMs will not share executable code - although this could be done with appropriate configuration of memory regions.).

The RTA-LWHVR configuration (see section 6) specifies each VM and the memory regions that code running in the VM may accesses. The VM must be linked so that its memory usage matches what is in its configuration.

5.5 Entry-Point

A VM's entry-point is the address in the VM's code where execution of the VM starts. Every time the RTA-LWHVR (re-) starts a VM it starts executing the code at the VM's entry-point.

The code at a VM's entry-point must set up the run-time environment for the code that runs in the VM. This includes:

- Setting up the stack.
- Setting up the heap.
- Copying initialization data from flash into RAM.

For C code this would normally be done by calling the compiler provided C start-up code.

A VM's entry-point must be declared in the VM's configuration.

5.6 VM Errors

If a VM tries to do something, that if allowed, would threaten the integrity of the RTA-LWHVR or another VM (e.g. accessing memory for which the VM does not have permission), or causes an unrecoverable error (e.g. calling a VM API service with invalid arguments) then the VM is said to be *in error*. When a VM is in error it stops running and will not be run again until the master software re-starts it. As soon as a VM goes into error the RTA-LWHVR calls the `LWHVR_VMErrorCallback()` function to inform the master software. To re-start the VM the master software should call the `LWHVR_RestartVM()` API function.

5.7 VM Status Block

Each VM has a *status block*. This is a collection of 32-bit fields in a region of memory that can be read and written by the VM. A VM status block is used to communicate information between the RTA-LWHVR and the VM. The fields of a VM status block fulfil the same role as special function or system registers do when code is not running in a VM. That is the fields contain information about enabled and pending pseudo-interrupts and the address of the instruction at which to resume execution after a pseudo-interrupt handler terminates.

The address of a VM's status block must be specified in the VM's configuration.

The type definition `LWHVR_VMStatusBlockType` type in `LWHVR_VMAPI.h` defines the layout of a VM status block. The fields are also documented in the subsections below.

5.7.1 ticksSinceStart (offset 0)

This field contains the number of clock-ticks that have elapsed since the VM was started. Note that code running in the VM may observe this field being incremented by more than 1 because this field continues to increment whilst the VM is not running in a time-slice. See also section 5.7.10.

The VM should treat this field as read-only. This field is set by the RTA-LWHVR at the start of each clock-tick during which the VM runs, but is never read by the RTA-LWHVR. Therefore if the VM does modify this field it will have no effect on the operation of the RTA-LWHVR.

5.7.2 ticksLeftInTimeslice (offset 4)

This fields contains the number of clock-ticks left in the current time-slice.

The VM should treat this field as read-only. This field is set by the RTA-LWHVR at the start of each clock-tick during which the VM runs, but is never read by the RTA-LWHVR. Therefore if the VM does modify this field it will have no effect on the operation of the RTA-LWHVR.

5.7.3 psIntEnabled (offset 8)

This field contains a bit-mask of pseudo-interrupts that are currently enabled. The bit for pseudo-interrupt number n is (1 << n).

This field may be read and written by the VM.

5.7.4 psIntPending (offset 12)

This field contains a bit-mask of pseudo-interrupts that are currently pending. The bit for pseudo-interrupt number n is (1 << n).

The VM must treat this field as read-only. This field **must not** be written by the VM otherwise pseudo-interrupts may be lost and not injected into the VM. However, such loss of pseudo-interrupts will only affect this VM, it will not affect other VMs or the RTA-LWHVR.

5.7.5 psIntResumeAddress (offset 16)

When a pseudo-interrupt is injected into a VM and the VM's pseudo-interrupt handler runs this field contains the address of the instruction where execution (in the VM) should resume when the pseudo-interrupt handler terminates (using the VM API service `LWHVR_VMAPI_RETURN_FROM_PS_INT`).

This field may be read and written by the VM.

5.7.6 psIntReason (offset 20)

When a pseudo-interrupt is injected into a VM and the VM's pseudo-interrupt handler runs this fields contains the number of the pseudo interrupt that has been injected. The numbers (reasons) of pseudo-interrupts normally used by the RTA-LWHVR can be found in `LWHVR_VMAPI.h`.

The VM should treat this field as read-only. This field is set by the RTA-LWHVR just before a pseudo-interrupt is injected into to the VM, but is never read by the RTA-LWHVR. Therefore if the VM does modify this field it will have no effect on the operation of the RTA-LWHVR.

5.7.7 psIntPreviousEnabled (offset 24)

When a pseudo-interrupt is injected into a VM and the VM's pseudo-interrupt handler runs the `psIntEnabled` field will have been set to zero to disable further pseudo-interrupts. The

`psIntPreviousEnabled` field will contain the value that was in `psIntEnabled` before it was set to zero. When the VM API service `LWHVR_VMAPI_RETURN_FROM_PS_INT` is used to terminate a pseudo-interrupt handler the RTA-LWHVR copies the value in `psIntPreviousEnabled` back into `psIntEnabled`.

This field may be read and written by the VM.

5.7.8 psIntRestoreRegister (offset 28)

This field is used to restore the value of a register that would otherwise be corrupted by making a `LWHVR_VMAPI_RETURN_FROM_PS_INT` VM API call. The details are target specific, please see section 11.2.

This field may be read and written by the VM.

5.7.9 psIntGenerateOnTick (offset 32)

This field contains a bit-mask of pseudo-interrupts that the RTA-LWHVR will make pending on each clock-tick. That is on each clock-tick `psIntPending = psIntPending | psInGenerateOnTick`.

This field may be read and written by the VM.

5.7.10 ticksWhileRunning (offset 36)

This field contains the number of ticks for which the VM has been running. Note that this field does not increment when the VM is not running in a times-slice. See also section 5.7.1.

The VM should treat this field as read-only. This field is incremented by the RTA-LWHVR at the start of each clock-tick during which the VM runs, but is never read by the RTA-LWHVR. Therefore if the VM does modify this field it will have no effect on the operation of the RTA-LWHVR.

5.8 Pseudo-Interrupts

VMs cannot handle real hardware generated interrupts (where the term "interrupts" includes traps and exceptions). To support ticking RTA-OS counters and passing asynchronous shutdown signals into VMs, the RTA-LWHVR can inject *pseudo-interrupts* into VMs.

Each VM has a pseudo-interrupt handler which is the code run to handle pseudo-interrupts injected into the VM. The address of a VM's pseudo-interrupt handler must be specified in the VM's configuration.

5.8.1 Pseudo-Interrupt Numbers and Priorities

There are 32 possible pseudo-interrupts, numbered 0 to 31. The higher the pseudo-interrupt's number the higher its priority.

5.8.2 Pending and Enabled Pseudo-Interrupts

Injection of pseudo-interrupts is controlled by the `psIntPending` and `psIntEnabled` fields in a VM's status block. For pseudo-interrupt number psIntNum, if bit (1 << psIntNum) is set in `psIntPending` then the pseudo-interrupt is *pending*, and if bit (1 << psIntNum) is set in `psIntEnabled` then the pseudo-interrupt is *enabled*.

5.8.3 Pseudo-Interrupt Injection

A pseudo-interrupt behaves much like a real hardware interrupt. When the RTA-LWHVR injects a pseudo-interrupt into a VM the following occurs (where `VMStatusBlock` is the VM status block for the VM and psIntNum is the number of the pseudo-interrupt):

1. VMStatusBlock.`psIntReason` = psIntNum
2. VMStatusBlock.p`sIntPreviousEnabled` = VMStatusBlock.`psIntEnabled`
3. VMStatusBlock.`psIntEnabled` = 0
4. VMStatusBlock.`psIntPending` = VMStatusBlock.`psIntPending` & ~(1 << psIntNum)
5. VMStatusBlock.p`sIntResumeAddress` = address of instruction in VM that was about to execute.
6. Execution starts at the address of the VM's pseudo-interrupt handler.

When the VM's pseudo-interrupt handler terminates by calling the VM API service `LWHVR_VMAPI_RETURN_FROM_PS_INT` the following occurs:

1. VMStatusBlock.p`sIntEnabled` = VMStatusBlock.`psIntPreviousEnabled`
2. Possibly restore a register from VMStatusBlock.p`sIntRestoreRegister`
3. Execution resumes at the address in VMStatusBlock.`psIntResumeAddress`

5.8.4 When Pseudo-Interrupts are injected

The RTA-LWHVR injects the highest priority pseudo-interrupt that is both pending and enabled into the currently running VM at the following times:

- When the RTA-LWHVR processes a clock-tick interrupt (this includes the clock-tick interrupt that results in the VM starting a time-slice, but does not include the clock-tick interrupt that results in the VM finishing a time-slice).
- When the VM calls the VM API service `LWHVR_VMAPI_RETURN_FROM_PS_INT`.
- When the VM calls the VM API service `LWHVR_VMAPI_SYNC_PS_INTS`.
- When the VM calls the VM API service `LWHVR_VMAPI_INJECT_PS_INT`.

5.8.5 Pseudo-Interrupts used by the RTA-LWHVR

The following pseudo-interrupts are used by the RTA-LWHVR. Interrupt number (reason) and pending/enabled mask constants can be found in the header file `LWHVR_VMAPI.h`.

Timer0

Number: `LWHVR_PS_INT_REASON_TIMER0` (3)

Pending/Enabled Mask: `LWHVR_PS_INT_MASK_TIMER0`

Description:

By convention pseudo-interrupt number 3 is used as a low priority timer. If the `LWHVR_PS_INT_MASK_TIMER0` bit is set in the `psInGenerateOnTick` field of the VM's status block then this pseudo-interrupt is made pending every time the RTA-LWHVR processes a clock-tick interrupt and the VM is running in a time-slice (this includes the clock-tick interrupt that results in the VM starting a time-slice, but does not include the clock-tick interrupt that results in the VM finishing a time-slice).

In fact any pseudo-interrupt can be made pending on a clock-tick by setting the pseudo-interrupt's bit in `psInGenerateOnTick`. Use of the `LWHVR_PS_INT_MASK_TIMER0` and `LWHVR_PS_INT_MASK_TIMER1` bits is just convention.

Timer 1

Number: `LWHVR_PS_INT_REASON_TIMER1` (7)

Pending/Enabled Mask: `LWHVR_PS_INT_MASK_TIMER1`

Description:

By convention pseudo-interrupt number 7 is used as a high priority timer. If the `LWHVR_PS_INT_MASK_TIMER1` bit is set in the `psInGenerateOnTick` field of the VM's status block then this pseudo-interrupt is made pending every time the RTA-LWHVR processes a clock-tick interrupt and the VM is running in a time-slice (this includes the clock-tick interrupt that results in the VM starting a time-slice, but does not include the clock-tick interrupt that results in the VM finishing a time-slice).

In fact any pseudo-interrupt can be made pending on a clock-tick by setting the pseudo-interrupt's bit in `psInGenerateOnTick`. Use of the `LWHVR_PS_INT_MASK_TIMER0` and `LWHVR_PS_INT_MASK_TIMER1` bits is just convention.

Shutdown

Number: `LWHVR_PS_INT_REASON_SHUTDOWN` (11)

Pending/Enabled Mask: `LWHVR_PS_INT_MASK_SHUTDOWN`

Description:

This is a signal that the VM should gracefully shutdown. It is made pending when the master software calls the `LWHVR_ShutdownVM()` API for the VM.

5.8.6 Pseudo-Interrupt Handlers

A pseudo-interrupt handler has a similar form to a normal interrupt handler. It should do something like:

1. Save necessary (i.e. volatile) processor registers.
2. Save VMStatusBlock.`psIntPreviousEnabled`.
3. Save VMStatusBlock.`psIntResumeAddress`.
4. Make a local copy of VMStatusBlock.`psIntReason`.
5. Enable pseudo-interrupts by writing to VMStatusBlock.`psIntEnabled`.
6. Use the local copy of VMStatusBlock.`psIntReason` to decide what to do.
7. Disable pseudo-interrupts by writing 0 to VMStatusBlock.`psIntEnabled`.
8. Restore VMStatusBlock.`psIntResumeAddress`.
9. Restore VMStatusBlock.`psIntPreviousEnabled`.
10. Write to VMStatusBlock.`psIntRestoreRegister` if required.
11. Restore processor registers.
12. Call the VM API service `LWHVR_VMAPI_RETURN_FROM_PS_INT`.

5.8.7 Responding to a Shutdown Pseudo-Interrupt

When the master software calls the API function `LWHVR_ShutdownVM()` for a VM the shutdown pseudo-interrupt will be become pending in the VM and will eventually be injected into the VM.

When this occurs the VM should gracefully stop processing and call the VM API `LWHVR_VMAPI_SHUTDOWN`.

5.8.8 Pseudo-Interrupts on VM (Re-) Start

When a VM first starts, or is re-started, the `psIntEnabled`, `psIntPending` and `psIntGenerateOnTick` fields of its VM status block are all set to 0. This means that when a VM is started, or re-started, all pseudo-interrupts are disabled, pending pseudo-interrupts are cleared, and the VM will not receive any simulated timer-tick interrupts (`psIntGenerateOnTick`).

Note: this also means that the effect of any calls of `LWHVR_ShutdownVM()` that are made after a VM has been forcibly stopped, shut down or has caused an error (`LWHVR_VMErrorCallback()` called), but before the VM has re-started will be lost.

6 Configuration

When the RTA-LWHVR is compiled it expects its configuration to be defined in a header file called `LWHVR_Configuration.h`. This header file may be generated by hand or by using the RTA-LWHVR configuration generator tool. The configuration generator tool is provided with the RTA-LWHVR configuration in an XML file and outputs the configuration in a `LWHVR_Configuration.h` header file.

IMPORTANT
The RTA-LWHVR configuration generator tool is only provided as a convenience utility. The integrator is responsible for ensuring that the `LWHVR_Configuration.h` header file generated by the RTA-LWHVR configuration generator tool is correct.

This chapter is in three parts. First it provides an introduction to the logical concepts in an RTA-LWHVR configuration. Second it describes how to run the RTA-LWHVR configuration generator tool. Finally it describes the contents of a `LWHVR_Configuration.h` header file and how it must be checked for correctness.

6.1 Concepts

The RTA-LWHVR configuration describes one or more application cores. The configuration contains the following information for each application core:

- The hardware core number. This is a target specific number that maps the application core in the configuration to a physical processor core.
- Clock-tick interrupt number. This is a target specific number that identifies the interrupt source that will be used for clock-tick interrupts.
- Size of the extra-time request queue. This is the number of entries in the queue used when the master software requests extra execution time for a VM (see section 8.8). 0 is a valid value.
- A schedule table for the VMs that run on the application core.

The configuration contains the following information for each VM:

- The name of the VM.
- The application core on which the VM runs. A VM only runs on a single application core. (In the XML consumed by the configuration generator tool the link between a VM and an application core is implied from the VM being in the application core's `VMs` container.)
- Entry-point address. This is the address in memory where execution of the VM will start. This address must be in memory that the VM has permission to execute.
- Pseudo-interrupt handler address. This is the address in memory of the VM's pseudo-interrupt handler. This address must be in memory that the VM has permission to execute.
- Status block address. This is the address in memory where the VM's status block will be located. This address must be in memory that the VM has permission to read and write.
- Optional target specific information.
- One or more memory regions. Each memory region describes an extent of memory that the VM has permission to access. The configuration of a memory region contains:
    - Start address.
    - End address.

- Access permissions: readable, writeable, executable.

There must be between 1 and 40 (inclusive) VMs.

The configuration for an application core's schedule table contains one or more schedule entries. Each entry describes a time-slice. When the RTA-LWHVR starts running on an application core it starts at the first entry in the core's schedule table, when the time-slice described by this entry has completed the RTA-LWHVR moves to the next entry in the schedule table, and so on. When the end of the schedule table is reached the RTA-LWHVR starts at the beginning of the schedule table again. Each schedule table entry contains the following information:

- The identity of the VM to be scheduled or an indicator that means this is a spare time-slice.
- The duration of the time-slice in clock-ticks. This must be greater than 0 and must 1 if the entry is spare.

There must be between 1 and 256 (inclusive) entries in a schedule table.

6.2 Running the Configuration Generator Tool

If the RTA-LWHVR configuration generator tool is used the RTA-LWHVR configuration must be provided in an XML file.

- The schema for this XML file can be found in LWHVRConfiguration.xsd.
- The documentation for the XML schema can be found in the file LWHVRConfiguration.html inside the ZIP archive LWHVRConfigurationXMLDocumentation.zip.
- A structural view of the XML schema can be found in LWHVRConfiguration.png.

The configuration generator tool is run as follows:

```
RTA-LWHVRConfigGenerator.exe <xml-name>
```

Where <xml-name> is the path to an XML file containing the RTA-LWHVR configuration. The tool will generate a `LWHVR_Configuration.h` header file in the current directory.

6.3 LWHVR_Configuration.h

The actual RTA-LWHVR configuration used when compiling the RTA-LWHVR is stored in a header file called `LWHVR_Configuration.h`. This header file is included by the other RTA-LWHVR source files. The `LWHVR_Configuration.h` header file specifies the RTA-LWHVR configuration using `#defines`. This section describes those `#defines` and provides guidance on how to ensure that the configuration is correct.

IMPORTANT

The safety requirements of the RTA-LWHVR assume that it is provided with a valid configuration. Therefore the integrator must ensure that the contents of `LWHVR_Configuration.h` are correct.

In this section text prefixed with Check: contains important checks that must be carried out to ensure the correctness of `LWHVR_Configuration.h`.

6.3.1 General

Check: when the RTA-LWHVR source is compiled, the compiler must not generate any diagnostic messages related to `LWHVR_Configuration.h` or the use of its contents.

Check: all numbers and addresses on the right-hand-side of `#define`s must be unsigned integers and be surrounded by brackets. For example, the number 101 would be specified as `(101U)` and the memory address 0x0400b000 would be specified as `(0x0400b000U)`.

6.3.2 Application Cores

Define `LWHVR_NUM_APPLICATION_CORES` to be the number of application cores to use.

For each application core the following values must be specified. In the following `<A>` is the application core number. This is a decimal integer in the range 0 to `LWHVR_NUM_APPLICATION_CORES` – 1 inclusive.

Define `LWHVR_CORE<A>_NUM_VMS` to be the number of VMs that run on the application core. This must be at least 1. The total number of VMs (across all application cores) must not exceed 40.

Define `LWHVR_CORE<A>_CLOCK_TICK_INT` to be the target specific number of the clock-tick interrupt source.

Define `LWHVR_CORE<A>_EXTRA_TIME_Q_SIZE` to be the size of the queue on the application core used for extra-time requests by the master software (see section 8.8). This number must be between 0 and 256 (inclusive).

Check: each application core must run at least one VM.

Check: there must not be more than 40 VMs across all application cores.

Check: the clock-tick interrupt source specified for an application core must correspond to the clock-tick interrupt source enabled when the RTA-LWHVR calls `LWHVER_StartTimerCallback()` on the application core.

6.3.3 Assigning Application Cores to Physical Processor Cores

Application cores must be assigned to physical processor (hardware) cores. To assign application core number `<A>` to physical core number `<P>` define `LWHVR_HARDWARE_CORE<P>` to be `LWHVR_CORE<A>`.

The number of physical cores, and hence the values of `<P>` is target specific.

Check: every application core must be assigned to a physical core.

Check: an application core must not be assigned to multiple physical cores.

Check: that an application core is assigned to a physical core that is present in your processor.

6.3.4 Configuring VMs

Each VM in the system is configured as described in the following sub-sections. In what follows `<N>` is the number of the VM being configured, where `<N>` is in the range 0 to <total number of VMs> – 1.

Symbolic Identifier

Define `LWHVR_<name>_ID <N>` to create a symbolic identifier for the VM. For example if you want VM number 0 to be called "Wombat" and VM number 1 to be called "Kangaroo" then you would define `LWHVR_Wombat_ID` to be `(0U)` and `LWHVR_Kangaroo_ID` to be `(1U)`. Symbolic identifiers are used to refer to VMs in the schedule table (see below) and can be used in API functions whenever a parameter has the type `LWHVR_VmIdType`.

Core Identifier

Assign the VM to an application core by defining `LWHVR_VM<N>_CORE` to be `LWVHR_CORE<A>`. This specifies that the VM runs on application core number `<A>`.

Check: every VM must be assigned to exactly one application core using a LWHVR_VM<N>_CORE definition.

Check: the value of <A> must be in the range 0 to LWHVR_NUM_APPLICATION_CORES-1 inclusive.

Entry-Points and Status Block

Define LWHVR_VM<N>_ENTRY_POINT to be the address of the VM's entry-point.

Define LWHVR_VM<N>_PS_INT_HANDLER to be the address of the VM's pseudo-interrupt handler.

Define LWHVR_VM<N>_STATUS_BLOCK to be the address of the VM's status block.

Check: the VM's entry-point must be in memory that the VM has permission to execute.

Check: the VM's pseudo-interrupt handler must be in memory that the VM has permission to execute.

Check: the VM's status block must be entirely contained within in memory that the VM has permission to read and write. The status block is 40 bytes long.

Memory Regions

A VM is only allowed to access memory inside the memory regions specified in the VM's configuration. *To prevent VMs from being able to damage other VMs, the RTA-LWHVR, or software that does not run in VMs, each VM must be assigned its own private subset of the processor's memory.*

Define LWHVR_VM<N>_NUM_MPU_REGIONS to be the number of memory regions that the VM may access. This must be between 1 and 12 (inclusive).

In what follows <P> is the number of the memory region being configured, where <P> is in the range 0 to LWHVR_VM<N>_NUM_MPU_REGIONS – 1. For each memory region:

Define LWHVR_VM<N>_MPU_REGION<P>_PERMS to be the permissions granted to the VM when accessing the memory region. The possible values are:

LWHVR_MPU_REGION<P>_EXEC – the memory in the region may be executed.

LWHVR_MPU_REGION<P>_RW – the memory in the region may be read and written.

LWHVR_MPU_REGION<P>_RDONLY – the memory in the region may only be read.

LWHVR_MPU_REGION<P>_WRONLY – the memory in the region may only be written.

Define LWHVR_VM<N>_MPU_REGION<P>_END to be the end address for the memory region.

Define LWHVR_VM<N>_MPU_REGION<P>_START to be the start address for the memory region.

Check: LWHVR_VM<N>_MPU_REGION<P>_PERMS is only defined to be one of LWHVR_MPU_REGION<P>_EXEC, LWHVR_MPU_REGION<P>_RW, LWHVR_MPU_REGION<P>_RDONLY or LWHVR_MPU_REGION<P>_WRONLY.

Check: a memory region's start address must be before its end address.

Check: unless a memory region is being used for communication between VMs, memory regions assigned to different VMs must not overlap.

Check: each VM must have at least one memory region.

Check: a VM must not have more than 12 memory regions.

Check: when defining the permissions for a memory region make sure that the <P> in LWHVR_VM<N>_MPU_REGION<P>_PERMS matches the <P> in

LWHVR_MPU_REGION<P>_EXEC, LWHVR_MPU_REGION<P>_RW, LWHVR_MPU_REGION<P>_RDONLY or LWHVR_MPU_REGION<P>_WRONLY.

Check: *unless a memory region is being used for communication between a VM and the master software, the memory region must not overlap with memory used by any software that does not run in a VM (this includes the master software and the RTA-LWHVR).*

6.3.5 Schedule Table

An application core's schedule table is used to specify in what order and for how long VMs are run. For each application core number <A> a schedule table must be specified as follows:

Define LWHVR_NUM_CORE<A>_SCHED_ENTRIES to be the number of entries in the schedule table. This must be between 1 and 256 (inclusive).

Each entry in the schedule table represents a time-slice. The entry specifies the length of the time-slice in ticks and either the VM that should run in the time-slice or an indication that the time-slice is spare and can be used to run a VM that requests extra time. In what follows <T> is the number of the schedule table entry, where <T> is in the range 0 to LWHVR_NUM_CORE<A>_SCHED_ENTRIES – 1.

For each schedule table entry:

Define LWHVR_CORE<A>_SCHED<T>_VM to be the symbolic identifier of the VM to run in the time-slice or LWHVR_SPARE if the time-slice is spare.

Define LWHVR_CORE<A>_SCHED<T>_DURATION to be the length of the time-slice in ticks as an unsigned integer. If the times-slice is spare then the duration must be 1. Otherwise the duration must be 1 or greater.

Check: there must be a schedule table for every application core.

Check: a schedule table must contain between 1 and 256 (inclusive) entries.

Check: the schedule table for an application core must only contain VMs that run on that application core. I.e. All VMs in the schedule table for application core number <A> must have defined LWHVR_VM<N>_CORE to be LWVHR_CORE<A>.

Check: a LWHVR_CORE<A>_SCHED<T>_VM definition must only specify the symbolic name of a VM or the spare time-slice indicator LWHVR_SPARE.

Check: every VM must appear in a schedule table.

Check: if a schedule table entry is spare then the duration must be 1.

Check: if a schedule table entry is not spare then the duration must be 1 or greater.

6.3.6 Example

In the following example configuration:

- There are two application cores.
- There are three VMs. The VMs are called "VmZero", "VmOne" and "VmTwo". The VMs have a region of memory that they share for communication. "VmZero" and "VmOne" run on application core 0 and "VmTwo" runs on application core 1.
- Application core 0 is assigned to physical core 2.
- Application core 1 is assigned to physical core 4.

```
/* The number of application cores. */
#define LWHVR_NUM_APPLICATION_CORES (2U)

/* Assign application cores to physical cores. */
#define LWHVR_HARDWARE_CORE2  LWHVR_CORE0
#define LWHVR_HARDWARE_CORE4  LWHVR_CORE1
```

```
/**** Application core number 0. ****/
#define LWHVR_CORE0_NUM_VMS            (2U)
#define LWHVR_CORE0_EXTRA_TIME_Q_SIZE  (2U)
#define LWHVR_CORE0_CLOCK_TICK_INT     (908U)

/**** Application core number 1. ****/
#define LWHVR_CORE1_NUM_VMS            (1U)
#define LWHVR_CORE1_EXTRA_TIME_Q_SIZE  (1U)
#define LWHVR_CORE1_CLOCK_TICK_INT     (912U)

/**** Virtual machine number 0 - called 'VmZero' ****/
#define LWHVR_VmZero_ID              (0U)

/* The VM runs on application core 0. */
#define LWHVR_VM0_CORE               LWHVR_CORE0

#define LWHVR_VM0_ENTRY_POINT        (0x01000000U)
#define LWHVR_VM0_PS_INT_HANDLER     (0x01000004U)
#define LWHVR_VM0_STATUS_BLOCK       (0x40010000U)

#define LWHVR_VM0_NUM_MPU_REGIONS    (5U)

/* Memory region for executable code in FLASH. */
#define LWHVR_VM0_MPU_REGION0_PERMS  LWHVR_MPU_REGION0_EXEC
#define LWHVR_VM0_MPU_REGION0_START  (0x01000000U)
#define LWHVR_VM0_MPU_REGION0_END    (0x010ffff7U)

/* Memory region for constants in FLASH. */
#define LWHVR_VM0_MPU_REGION1_PERMS  LWHVR_MPU_REGION1_RDONLY
#define LWHVR_VM0_MPU_REGION1_START  (0x01000000U)
#define LWHVR_VM0_MPU_REGION1_END    (0x010ffff7U)

/* Memory region for variable data in RAM. */
#define LWHVR_VM0_MPU_REGION2_PERMS  LWHVR_MPU_REGION2_RW
#define LWHVR_VM0_MPU_REGION2_START  (0x40010000U)
#define LWHVR_VM0_MPU_REGION2_END    (0x4001fff7U)

/* Memory region for stack in RAM. */
#define LWHVR_VM0_MPU_REGION3_PERMS  LWHVR_MPU_REGION3_RW
#define LWHVR_VM0_MPU_REGION3_START  (0x40020000U)
#define LWHVR_VM0_MPU_REGION3_END    (0x4002fff7U)

/* Memory region shared with other VM for communication. */
#define LWHVR_VM0_MPU_REGION4_PERMS  LWHVR_MPU_REGION4_RW
#define LWHVR_VM0_MPU_REGION4_START  (0x400f0000U)
#define LWHVR_VM0_MPU_REGION4_END    (0x400ffff7U)

/**** Virtual machine number 1 called 'VmOne' ****/
#define LWHVR_VmOne_ID               (1U)

/* The VM runs on application core 0. */
#define LWHVR_VM1_CORE               LWHVR_CORE0

#define LWHVR_VM1_ENTRY_POINT        (0x01100000U)
#define LWHVR_VM1_PS_INT_HANDLER     (0x01100004U)
#define LWHVR_VM1_STATUS_BLOCK       (0x40030000U)

#define LWHVR_VM1_NUM_MPU_REGIONS    (5U)

/* Memory region for executable code in FLASH. */
#define LWHVR_VM1_MPU_REGION0_PERMS  LWHVR_MPU_REGION0_EXEC
#define LWHVR_VM1_MPU_REGION0_START  (0x01100000U)
#define LWHVR_VM1_MPU_REGION0_END    (0x011ffff7U)

/* Memory region for constants in FLASH. */
#define LWHVR_VM1_MPU_REGION1_PERMS  LWHVR_MPU_REGION1_RDONLY
#define LWHVR_VM1_MPU_REGION1_START  (0x01100000U)
#define LWHVR_VM1_MPU_REGION1_END    (0x011ffff7U)
```

```
/* Memory region for variable data in RAM. */
#define LWHVR_VM1_MPU_REGION2_PERMS  LWHVR_MPU_REGION2_RW
#define LWHVR_VM1_MPU_REGION2_START  (0x40030000U)
#define LWHVR_VM1_MPU_REGION2_END    (0x4003fff7U)

/* Memory region for stack in RAM. */
#define LWHVR_VM1_MPU_REGION3_PERMS  LWHVR_MPU_REGION3_RW
#define LWHVR_VM1_MPU_REGION3_START  (0x40040000U)
#define LWHVR_VM1_MPU_REGION3_END    (0x4004fff7U)

/* Memory region shared with other VM for communication. */
#define LWHVR_VM1_MPU_REGION4_PERMS  LWHVR_MPU_REGION4_RW
#define LWHVR_VM1_MPU_REGION4_START  (0x400f0000U)
#define LWHVR_VM1_MPU_REGION4_END    (0x400ffff7U)

/**** Virtual machine number 2 called 'VmTwo' ****/
#define LWHVR_VmTwo_ID               (2U)

/* The VM runs on application core 1. */
#define LWHVR_VM2_CORE               LWHVR_CORE1

#define LWHVR_VM2_ENTRY_POINT        (0x01200000U)
#define LWHVR_VM2_PS_INT_HANDLER     (0x01200004U)
#define LWHVR_VM2_STATUS_BLOCK       (0x40050000U)

#define LWHVR_VM2_NUM_MPU_REGIONS    (5U)

/* Memory region for executable code in FLASH. */
#define LWHVR_VM2_MPU_REGION0_PERMS  LWHVR_MPU_REGION0_EXEC
#define LWHVR_VM2_MPU_REGION0_START  (0x01200000U)
#define LWHVR_VM2_MPU_REGION0_END    (0x012ffff7U)

/* Memory region for constants in FLASH. */
#define LWHVR_VM2_MPU_REGION1_PERMS  LWHVR_MPU_REGION1_RDONLY
#define LWHVR_VM2_MPU_REGION1_START  (0x01200000U)
#define LWHVR_VM2_MPU_REGION1_END    (0x012ffff7U)

/* Memory region for variable data in RAM. */
#define LWHVR_VM2_MPU_REGION2_PERMS  LWHVR_MPU_REGION2_RW
#define LWHVR_VM2_MPU_REGION2_START  (0x40050000U)
#define LWHVR_VM2_MPU_REGION2_END    (0x4005fff7U)

/* Memory region for stack in core local RAM. */
#define LWHVR_VM2_MPU_REGION3_PERMS  LWHVR_MPU_REGION3_RW
#define LWHVR_VM2_MPU_REGION3_START  (0x40060000U)
#define LWHVR_VM2_MPU_REGION3_END    (0x4006fff7U)

/* Memory region shared with other VM for communication. */
#define LWHVR_VM2_MPU_REGION4_PERMS  LWHVR_MPU_REGION4_RW
#define LWHVR_VM2_MPU_REGION4_START  (0x400f0000U)
#define LWHVR_VM2_MPU_REGION4_END    (0x400ffff7U)

/**** Application core 0 schedule table. ****/
#define LWHVR_NUM_CORE0_SCHED_ENTRIES  (6U)

#define LWHVR_CORE0_SCHED0_VM          LWHVR_VmZero_ID
#define LWHVR_CORE0_SCHED0_DURATION    (2U)

#define LWHVR_CORE0_SCHED1_VM          LWHVR_VmOne_ID
#define LWHVR_CORE0_SCHED1_DURATION    (2U)

#define LWHVR_CORE0_SCHED2_VM          LWHVR_SPARE
#define LWHVR_CORE0_SCHED2_DURATION    (1U)

#define LWHVR_CORE0_SCHED3_VM          LWHVR_VmZero_ID
#define LWHVR_CORE0_SCHED3_DURATION    (1U)

#define LWHVR_CORE0_SCHED4_VM          LWHVR_VmOne_ID
#define LWHVR_CORE0_SCHED4_DURATION    (1U)
```

```
#define LWHVR_CORE0_SCHED5_VM          LWHVR_SPARE
#define LWHVR_CORE0_SCHED5_DURATION    (1U)

/**** Application core 1 schedule table. ****/
#define LWHVR_NUM_CORE1_SCHED_ENTRIES  (1U)

#define LWHVR_CORE1_SCHED0_VM          LWHVR_VmTwo_ID
#define LWHVR_CORE1_SCHED0_DURATION    (2U)
```

7 Types and Constants

This chapter describes the types and constants used in the RTA-LWHVR APIs.

7.1 LWHVR_BooleanType

`LWHVR_BooleanType` is a target specific type used to represent a Boolean value. The constant `LWHVR_TRUE` is used to represent true and the constant `LWHVR_FALSE` is used to represent false.

7.2 LWHVR_UInt32Type

`LWHVR_UInt32Type` is a target specific type used to represent an unsigned 32-bit value.

7.3 LWHVR_RegisterType

`LWHVR_RegisterType` is a target specific type used to represent a processor register.

7.4 LWHVR_InterruptIdType

`LWHVR_InterruptIdType` is a target specific type used to represent an interrupt source.

7.5 LWHVR_MemoryCopyExtentType

`LWHVR_MemoryCopyExtentType` is used to describe a memory extent to be copied – see section 11.6.

7.6 LWHVR_VmIdType

`LWHVR_VmIdType` is used to represent a VM identifier. A VM identifier is a small integer that corresponds to the VM's number in the configuration. That is, the VM described by macros `LWHVR_VM<N>_XXXX` in `LWHVR_Configuration.h` would have identifier `<N>` (see section 6.3.4).

For example, the VM described by macros `LWHVR_VM0_XXXX` would have identifier 0, the VM described by macros `LWHVR_VM1_XXXX` would have identifier 1, the VM described by macros `LWHVR_VM2_XXXX` would have identifier 2, and so on.

7.7 LWHVR_ErrorType

`LWHVR_ErrorType` is used to represent an error cause. This is an enumeration with the following values:

`LWHVR_ErrorNone`
: No error.

`LWHVR_ErrorInvalidVmId`
: An invalid VM identifier has been specified.

`LWHVR_ErrorInvalidVmAPI`
: A VM has specified an invalid API service number when making a VM API call.

`LWHVR_ErrorInvalidPsInterrupt`
: A VM has specified an invalid pseudo-interrupt number when making a VM API call.

`LWHVR_ErrorMemoryPermission`

A VM has tried to access memory for which it does not have permission. Either the VM does not have permission to access the memory at all, or it has tried to access the memory in a way that does not match the VM's MPU configuration. E.g. the VM has tried to write to memory for which it only has read permission.

`LWHVR_ErrorMemoryAlignment`

A VM has made an unaligned memory access.

`LWHVR_ErrorInstruction`

A VM has executed an invalid instruction or an instruction that it is not allowed to execute (e.g. a privileged instruction).

`LWHVR_ErrorTooManyExtents`

A VM has called the atomic memory-copy VM API specifying too many memory extents to be copied.

`LWHVR_ErrorExtentTooLarge`

A VM has called the atomic memory-copy VM API specifying a memory extent that is too large.

`LWHVR_ErrorExtraTimeQueueFull`

The queue of VMs for which extra-time has been requested was full when the `LWHVR_RequestExtraTimeForVM()` API function was called.

`LWHVR_ErrorNotApplicationCore`

The `LWHVR_Start()` API function was called on a core that is not an application core.

`LWHVR_ErrorInitializing`

The `LWHVR_StopVM()`, `LWHVR_ShutdownVM()`, `LWHVR_RestartVM()` or `LWHVR_RequestExtraTimeForVM()` master software API function has been called for a VM on an application core that has not completed initialisation or has stopped running the LWHVR.

The following values are not used on all targets. See the target specific documentation for which values are used on your target.

`LWHVR_ErrorRegisterPermission`

A VM has tried to access a register for which it does not have permission.

`LWHVR_ErrorStackOverflow`

A VM has overflowed its stack.

`LWHVR_ErrorStackUnderflow`

A VM has underflowed its stack.

`LWHVR_ErrorStackInvalid`

The VM's stack is not in a valid state for the instruction being executed by the VM.

`LWHVR_ErrorMemoryAddress`

A VM has tried to execute an instruction whose operand is not a valid memory address.

`LWHVR_ErrorInstrFetch`

A VM has tried to execute an instruction that has caused a synchronous instruction fetch error.

`LWHVR_ErrorInstrFetchAsync`

A VM has tried to execute an instruction that has caused an asynchronous instruction fetch error.

`LWHVR_ErrorProgramMemory`

A VM has tried to execute an instruction that has caused a synchronous error in the program memory.

`LWHVR_ErrorProgramMemoryAsync`

A VM has tried to execute an instruction that has caused an asynchronous error in the program memory.

`LWHVR_ErrorDataAccess`

A VM has tried to access a memory location that has caused a synchronous data access error.

`LWHVR_ErrorDataAccessAsync`

The VM has tried to access a memory location that has caused an asynchronous data access error.

`LWHVR_ErrorDataMemory`

A VM has tried to access a memory location that has caused a synchronous data memory error.

`LWHVR_ErrorDataMemoryAsync`

A VM has tried to access a memory location that has caused an asynchronous data memory error.

`LWHVR_ErrorCoprocessor`

A co-processor has caused a synchronous error.

`LWHVR_ErrorCoprocessorAsync`

A co-processor has causes an asynchronous error.

`LWHVR_ErrorTemporal`

A synchronous temporal error has occurred.

`LWHVR_ErrorTemporalAsync`

An asynchronous temporal error has occurred.

`LWHVR_FatalInterruptDecode`

The RTA-LWHVR is not able to decode an interrupt/trap. This is a fatal error and if seen the RTA-LWHVR is in an invalid state.

`LWHVR_FatalMemoryRegionIndex`

The RTA-LWHVR has tried to set an invalid MPU region. This is a fatal error and if seen the RTA-LWHVR is in an invalid state.

`LWHVR_FatalMemoryRegionConfig`

The memory region configuration is invalid. This is a fatal error and if seen the RTA-LWHVR is in an invalid state.

7.8 LWHVR_VMStatusBlockType

`LWHVR_VMStatusBlockType` defines the layout of a VM's status block. See section 5.7.

8 Master Software API

The RTA-LWHVR provides an API for use by the master software. This chapter describes this API.

The master software API consists of a collection of C functions that can be called by the master software to control the RTA-LWHVR and VMs.

To make use of this API the master software should include the header file `LWHVR.h`.

Since the RTA-LWHVR is linked into the master software image, the API functions are called as normal C functions.

8.1 LWHVR_Init

Prototype:

```
void LWHVR_Init(void);
```

Purpose:

Called by the master software to carry out RTA-LWHVR initialisation that **must be done before** any other API call (including `LWHVR_Start()` and `LWHVR_AllHaveStarted()`) is used.

If the RTA-LWHVR is stopped by calling `LWHVR_Stop()` then this API must be called again before application cores call `LWHVR_Start()` to re-start the RTA-LWHVR.

Parameters:

None.

Returns:

Nothing.

Restrictions:

- → This function must not be called until integrator code has initialized the hardware (e.g. configured PLLs and clocks, enabled/disabled data-caches).
- → This function must not be called until a suitable environment has been created to run C code – e.g. by running the compiler C start-up code.
- → This function is not re-entrant. That is, once this function has been called on any core, it must not be called again (e.g. on a different core, in an interrupt handler or in a higher priority task) until the first call has returned.
- → This function must not be called whilst any application is running the RTA-LWHVR – i.e. the application core is executing in the `LWHVR_Start()` function.

8.2 LWHVR_Start

Prototype:

```
void LWHVR_Start(void);
```

Purpose:

Called by an application core to start the RTA-LWHVR running on that core.

This function may be called by an application core that is running the RTA-LWHVR to re-start the RTA-LWHVR on that application core without the RTA-LWHVR being stopped with `LWHVR_Stop()` – see section 4.7.

Parameters:

None.

Returns:

Nothing.

Restrictions:

→ This function must not be called until integrator code has initialized the hardware (e.g. configured PLLs and clocks, enabled/disabled data-caches).

→ This function must not be called until a suitable environment has been created to run C code – e.g. by running the compiler C start-up code.

→ This function must not be called until the master software has called `LWHVR_Init()`.

→ Whilst initialising the RTA-LWHVR on an application core, `LWHVR_Start()` will configure the interrupt controller to route the clock-tick interrupt to the RTA-LWHVR. To avoid contention when configuring the interrupt controller, once `LWHVR_Start()` has been called by any application core, no other software may change the configuration of the interrupt controller until `LWHVR_AllHaveStarted()` returns `LWHVR_TRUE`.

Notes:

1. This function only returns when the RTA-LWHVR has shut down.
2. If this function is called on a core that is not an application core then `LWVHR_ErrorCallback()` will be called with an error of `LWHVR_ErrorNotApplicationCore` and the RTA-LWHVR will not be started.

8.3 LWHVR_AllHaveStarted

Prototype:

```
LWHVR_BooleanType LWHVR_AllHaveStarted(void);
```

Purpose:

Called by the master software to determine if all application cores have called `LWHVR_Start()` and completed sufficient initialisation to allow other RTA-LWHVR APIs to be called.

Parameters:

None.

Returns:

`LWHVR_TRUE` if all application cores have called `LWHVR_Start()` and completed initialisation. `LWHVR_FALSE` otherwise.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

8.4 LWHVR_Stop

Prototype:

```
void LWHVR_Stop(void);
```

Purpose:

Called by the master software to stop the RTA-LWHVR running on all application cores. After this call has been made application cores will return from `LWHVR_Start()`.

The RTA-LWHVR will stop running on an application core at the start of the next scheduled time-slice. That is, just before the next time-slice in the schedule would be run, but after any VMs for which the master software has requested extra time have run.

No master software API functions may be called after `LWHVR_Stop()` has been called unless the RTA-LWHVR has been re-started.

Parameters:

None.

Returns:

Nothing.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

8.5 LWHVR_StopVM

Prototype:

```
void LWHVR_StopVM(LWHVR_VmIdType vmId);
```

Purpose:

Called by the master software to request that a VM be forcibly stopped.

The VM will be stopped at the start of its next time-slice.

Immediately after the VM has been stopped `LWHVR_StoppedVMCallback()` will be called.

Parameters:

`vmId`	The identifier of the VM to be stopped.

Returns:

Nothing.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

→ If the RTA-LWHVR may be re-started on an individual application core (see section 4.7) then this function must run to completion without being pre-empted by any other code – e.g. a higher priority task or ISR.

Notes:

1. If `vmId` is not valid then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInvalidVmId`.

2. If this function is called for a VM on an application core that has not yet called `LWHVR_Start()`, or where `LWHVR_Start()` is carrying out initialisation, or where the RTA-LWHVR has stopped running, then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInitializing`.

8.6 LWHVR_ShutdownVM

Prototype:

```
void LWHVR_ShutdownVM(LWHVR_VmIdType vmId);
```

Purpose:

Called by the master software to request that a VM be shut down.

The VM will have a shutdown pseudo-interrupt injected at the start of its next time-slice.

Immediately after the VM has called the VM API service `LWHVR_VMAPI_SHUTDOWN`, `LWHVR_ShutdownVMCallback()` will be called.

Parameters:

`vmId` The identifier of the VM to be shut down.

Returns:

Nothing.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

→ If the RTA-LWHVR may be re-started on an individual application core (see section 4.7) then this function must run to completion without being pre-empted by any other code – e.g. a higher priority task or ISR.

Notes:

1. If `vmId` is not valid then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInvalidVmId`.
2. If this function is called for a VM on an application core that has not yet called `LWHVR_Start()`, or where `LWHVR_Start()` is carrying out initialisation, or where the RTA-LWHVR has stopped running, then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInitializing`.
3. When a VM is re-started, the mask of pending pseudo-interrupts in its status block is set to 0. Therefore the effect of any calls of `LWHVR_ShutdownVM()` that are made after the VM has been forcibly stopped, shut down or has caused an error (`LWHVR_VMErrorCallback()` called), but before the VM has re-started will be lost.

8.7 LWHVR_RestartVM

Prototype:

```
void LWHVR_RestartVM(LWHVR_VmIdType vmId);
```

Purpose:

Called by the master software to request that a VM be re-started after being forcibly stopped, shut down, or after it has caused an error.

This API must only be used to re-start a VM during or after the `LWHVR_VMErrorCallback()`, `LWHVR_StoppedVMCallback()` or `LWHVR_ShutdownVMCallback()` call-back has been made for the VM.

This call has no effect if the VM is running.

Parameters:

`vmId` The identifier of the VM to be re-started.

Returns:

Nothing.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

→ If the RTA-LWHVR may be re-started on an individual application core (see section 4.7) then this function must run to completion without being pre-empted by any other code – e.g. a higher priority task or ISR.

Notes:

1. If `vmId` is not valid then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInvalidVmId`.
2. If this function is called for a VM on an application core that has not yet called `LWHVR_Start()`, or where `LWHVR_Start()` is carrying out initialisation, or where the RTA-LWHVR has stopped running, then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInitializing`.

8.8 LWHVR_RequestExtraTimeForVM

Prototype:

```
void LWHVR_RequestExtraTimeForVM(LWHVR_VmIdType vmId);
```

Purpose:

Called by the master software to request that a VM be executed for an extra clock-tick. The VM will be added to a high-priority FIFO queue of VMs requiring extra-time on the VM's application core. (Note that this is a different queue to the low-priority queue used for extra-time requests made by VMs themselves – see section 11.5.) *Execution of VMs in this extra-time queue pre-empts normal scheduling on the application core*. That is, on a clock-tick, if this extra-time queue is not empty, then rather than selecting which VM to execute based on the application core's schedule table, the RTA-LWHVR scheduler will remove the VM at the front of the extra-time queue and execute it.

A VM may be in a high-priority extra-time queue multiple times.

The size of an application core's high-priority extra-time queue is specified in the RTA-LWHVR configuration.

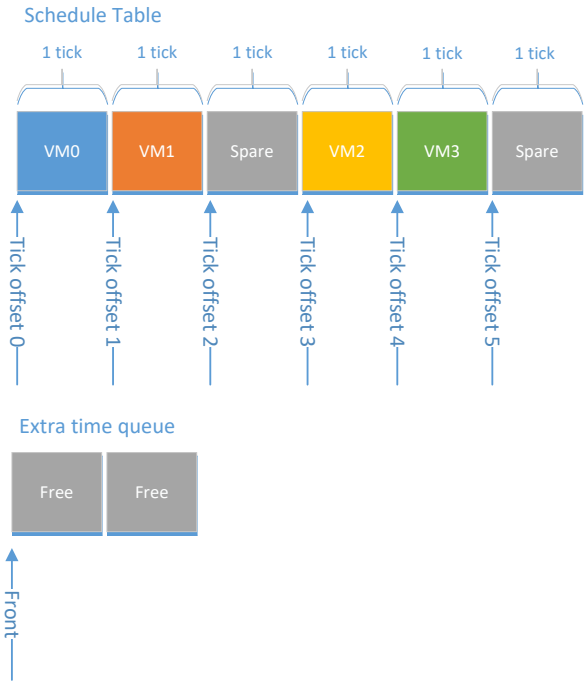
When an application core starts, the number of free entries in its high-priority extra-time queue is set to the size of the queue. When this API adds a VM to the queue it decrements by one the number of free entries in the queue. A VM can only be added to the queue when there is at least one free entry (otherwise `LWHVR_ErrorCallback()` is called and the VM is not added to the queue).

When the scheduler removes a VM from an application core's high-priority extra-time queue in order to execute it, the scheduler does not increment the number of free entries in the queue. Instead, when the scheduler encounters a spare time-slice in the application core's

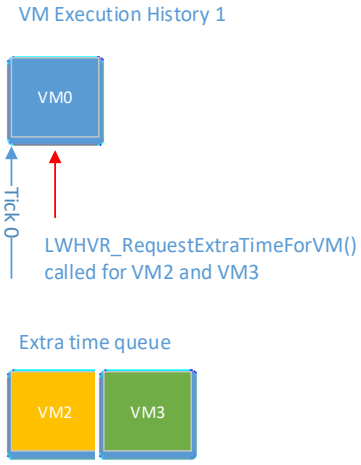
schedule table, and the number of free entries is less than the queue size, the scheduler skips the spare time-slice and increments by one the number of free entries in the queue. (Note that a skipped spare time-slice is not available for running a VM that has requested extra time for itself.)

In this way, we only allow the master software to use spare time that exists in the schedule, but in advance of when the spare time occurs in the schedule. The size of the queue limits how much spare time the master software can "borrow" from the future.

For example, consider the following RTA-LWHVR configuration. The schedule table has 6 entries, two of which are spare time. The extra-time queue has two entries.

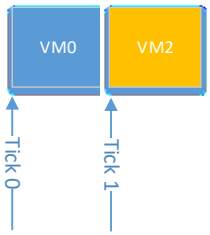


Assume that while VM0 is executing during clock-tick 0 `LWHVR_RequestExtraTimeForVM()` is called for VM2 and then VM3. VM2 and VM3 would be added to the extra-time queue.



At clock-tick 1 the RTA-LWHVR scheduler would notice that VM2 is at the front of the extra-time queue and execute it instead of looking at the next entry in the schedule table.

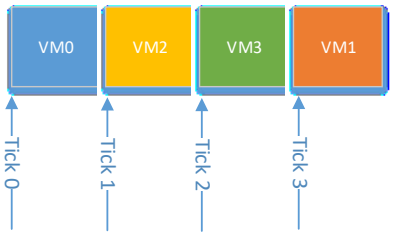
VM Execution History 2



Extra time queue

At clock-tick 2 the scheduler would notice that that VM3 is at the front of the extra-time queue and execute it instead of looking at the next entry in the schedule table. At clock-tick 3 the scheduler would return to executing VMs in schedule order and would execute VM1 (at clock-tick offset 1 in the schedule).

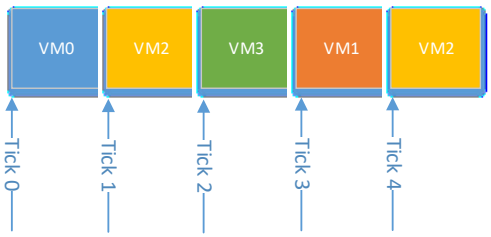
VM Execution History 3



Extra time queue

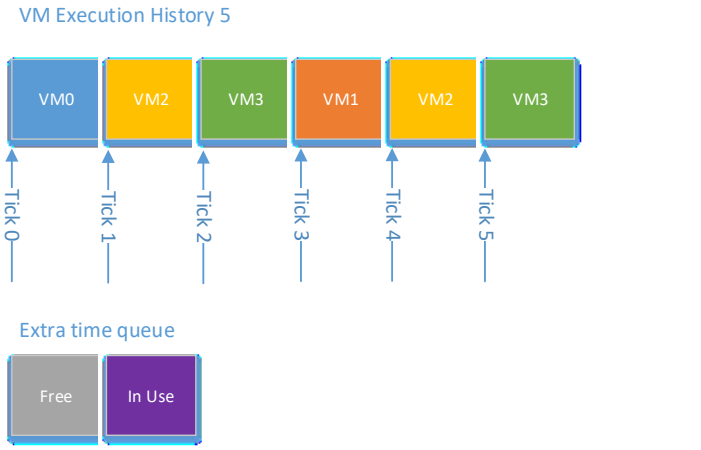
At clock-tick 4 the scheduler would find a spare time-slice in the schedule (at clock-tick offset 2 in the schedule). Therefore it would free up an entry in the extra-time queue and skip the spare time-slice. The scheduler would then find VM2 in the schedule (at clock-tick offset 3 in the schedule) and execute it.

VM Execution History 4

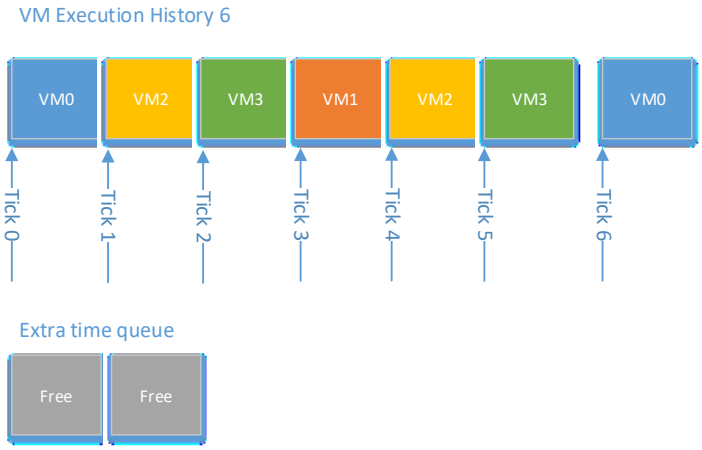


Extra time queue

At clock-tick 5 the scheduler would find VM3 in the schedule (at clock-tick offset 4 in the schedule) and execute it.



At clock-tick 6 the scheduler would find a spare time-slice in the schedule (at clock-tick offset 5 in the schedule). Therefore it would free up an entry in the extra-time queue and skip the spare time-slice. The scheduler would now be at the end of the schedule so it would return to the start of the schedule and execute VM0.



Parameters:

`vmId` The identifier of the VM for which extra-time is being requested.

Returns:

Nothing.

Restrictions:

→ This function must not be called until the master software has called `LWHVR_Init()`.

→ This function is not re-entrant. That is, once this function has been called on any core, it must not be called again (e.g. on a different core, in an interrupt handler or in a higher priority task) until the first call has returned.

→ If the RTA-LWHVR may be re-started on an individual application core (see section 4.7) then this function must run to completion without being pre-empted by any other code – e.g. a higher priority task or ISR.

Notes:

1. If `vmId` is not valid then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInvalidVmId`.

2. If the extra-time queue is full then `LWVHR_ErrorCallback()` will be called with an error of `LWHVR_ErrorExtraTimeQueueFull`.
3. If this function is called for a VM on an application core that has not yet called `LWHVR_Start()`, or where `LWHVR_Start()` is carrying out initialisation, or where the RTA-LWHVR has stopped running, then `LWHVR_ErrorCallback()` will be called with an error of `LWHVR_ErrorInitializing`.

9 Master Software Call-back Functions

This chapter describes the call-back functions that the master software provides to support the RTA-LWHVR and receive status information from the RTA-LWHVR.

Since the RTA-LWHVR is linked into the master software image, the call-back functions are called as normal C functions (except for `LWHVR_UnexpInterruptHook`).

Unless stated otherwise, except for `LWHVR_ErrorCallback()`, when these call-back functions are called:

- The call-back function will be executed by an application core.
- The same stack will be in use as when `LWHVR_Start()` was called on the calling application core.
- The small-data area register values will be the same as when `LWHVR_Start()` was called on the calling application core.
- Volatile registers will have been saved by the caller (the call-back must preserve any non-volatile registers required by the compiler/EABI).
- Interrupts will disabled. The call-back **must not** enable interrupts.

9.1 LWHVR_StartTimerCallback

Prototype:

```
void LWHVR_StartTimerCallback(void);
```

Purpose:

Called by the RTA-LWHVR to configure and start the clock-tick interrupt source on the application core on which the call-back is called – see section 4.8.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.2 LWHVR_ClockCallback

Prototype:

```
void LWHVR_ClockCallback(void);
```

Purpose:

Called by the RTA-LWHVR during the clock-tick interrupt handler to acknowledge and re-enable the clock-tick interrupt source, on the application core on which the call-back is called, to generate another interrupt on the next clock-tick – see section 4.8.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.3 LWHVR_UnexpInterruptCallback

Prototype:

```
void LWHVR_UnexpInterruptCallback(LWHVR_InterruptIdType cause);
```

Purpose:

Called by the RTA-LWHVR when an unexpected interrupt occurs on an application core. If the integrator can deal with the unexpected interrupt then this function may return normally to resume execution of the RTA-LWHVR.

See the target specific documentation for the situations in which this call-back will be called.

Parameters:

`cause`	The target specific cause of the interrupt.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.4 LWHVR_UnexpInterruptHook

Prototype:

```
void LWHVR_UnexpInterruptHook(LWHVR_InterruptIdType cause);
```

Purpose:

Jumped to when an unexpected interrupt occurs on an application core. If the integrator can deal with the unexpected interrupt then they may execute code that resumes execution after the interrupt (e.g. execute an rfi instruction).

This is **not** a normal C function, it is jumped to directly from the interrupt vector, therefore:

- The stack is undefined.
- Small-data area registers are undefined.
- If this hook will resume execution it must preserve any registers it uses.

See the target specific documentation for the situations in which this hook will be invoked.

Parameters:

`cause`	The target specific cause of the interrupt.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.5 LWHVR_ErrorCallback

Prototype:

```
void LWHVR_ErrorCallback(LWHVR_ErrorType error);
```

Purpose:

Called by the RTA-LWHVR when the master software calls an API function in an invalid way. This function is called directly by the master software API function. Therefore it executes on the same core and in the same environment as the caller of the API function.

Parameters:

`Error`	The error that has occurred.

Returns:

Nothing.

Error Reasons:

The master software may be notified of the following errors via this call-back:

```
LWHVR_ErrorInvalidVmId
LWHVR_ErrorExtraTimeQueueFull
LWHVR_ErrorNotApplicationCore
LWHVR_ErrorInitializing
```

In case of the above error the call-back may return.

See the target specific documentation for any additional situations in which this call-back may be called.

9.6 LWHVR_StoppedVMCallback

Prototype:

```
void LWHVR_StoppedVMCallback(LWHVR_VmIdType vmId);
```

Purpose:

Called by the RTA-LWHVR when a VM has stopped as the result of a previous `LWHVR_StopVM()` API call.

Parameters:

`vmId`	The identifier of the VM that has stopped.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.7 LWHVR_ShutdownVMCallback

Prototype:

```
void LWHVR_ShutdownVMCallback(LWHVR_VmIdType vmId);
```

Purpose:

Called by the RTA-LWHVR when a VM has shut itself down by calling the VM API `LWHVR_VMAPI_SHUTDOWN`.

Parameters:

`vmId`	The identifier of the VM that has shutdown.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.8 LWHVR_VMErrorCallback

Prototype:

```
void LWHVR_VMErrorCallback(
    LWHVR_VmIdType   vmId,
    LWHVR_ErrorType  error,
    LWHVR_UInt32Type data);
```

Purpose:

Called by the RTA-LWHVR when a VM goes into error (see section 5.6).

When this call-back is called the VM will have been forcibly stopped and will no-longer be scheduled. To re-start the VM use the `LWHVR_RestartVM()` API.

Parameters:

`vmId`	The identifier of the VM that is in error.
`error`	The error the VM caused.
`data`	Target and error specific.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

Error Reasons:

The master software may be notified of the following errors via this call-back:

```
LWHVR_ErrorInvalidVmAPI
LWHVR_ErrorInvalidPsInterrupt
LWHVR_ErrorMemoryPermission
LWHVR_ErrorMemoryAlignment
LWHVR_ErrorInstruction
LWHVR_ErrorToManyExtents
LWHVR_ErrorExtentTooLarge
```

See the target specific documentation for any additional situations in which this call-back may be called.

9.9 LWHVR_GlobalUnlockCallback

Prototype:

```
void LWHVR_GlobalUnlockCallback(void);
```

Purpose:

Called to enable access to configuration registers that require unlocking globally.

An application core will always call `LWHVR_GlobalUnlockCallback()` shortly followed by `LWHVR_GlobalRelockCallback()`. On any one application core calls to `LWHVR_GlobalUnlockCallback()` and `LWHVR_GlobalRelockCallback()` will never be nested - i.e. we always see:

```
LWHVR_GlobalUnlockCallback() // Application core X
LWHVR_GlobalRelockCallback() // Application core X

LWHVR_GlobalUnlockCallback() // Application core X
LWHVR_GlobalRelockCallback() // Application core X
```

and NEVER:

```
LWHVR_GlobalUnlockCallback() // Application core X
LWHVR_GlobalUnlockCallback() // Application core X

LWHVR_GlobalRelockCallback() // Application core X
LWHVR_GlobalRelockCallback() // Application core X
```

However, since application cores run in parallel you may see nested calls. E.g. you might see the following:

```
LWHVR_GlobalUnlockCallback() // Application core X
LWHVR_GlobalUnlockCallback() // Application core Y

LWHVR_GlobalRelockCallback() // Application core X
LWHVR_GlobalRelockCallback() // Application core Y
```

Therefore, if the unlocking operation uses a resource shared between cores it will be necessary to use some sort of lock to a stop a new call of `LWHVR_GlobalUnlockCallback()` from running until a previous call of `LWHVR_GlobalUnlockCallback()` has been completed with a call of `LWHVR_GlobalRelockCallback()`. E.g.

```
void LWHVR_GlobalUnlockCallback(void)
{
    <claim spin-lock>
    <unlock access to configuration registers>
}

void LWHVR_GlobalRelockCallback(void)
{
    <relock access to configuration registers>
    <release spin-lock>
}
```

See the target specific documentation for which configuration registers must be unlocked by this function.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.10 LWHVR_GlobalRelockCallback

Prototype:

```
void LWHVR_GlobalRelockCallback(void);
```

Purpose:

Called to disable access to configuration registers that require unlocking globally.

See section 9.9 for details.

See the target specific documentation for which configuration registers must be relocked by this function.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.11 LWHVR_CoreUnlockCallback

Prototype:

```
void LWHVR_CoreUnlockCallback(void);
```

Purpose:

Called to enable access to configuration registers that require unlocking for the calling application core.

An application core will always call `LWHVR_CoreUnlockCallback()` shortly followed by `LWHVR_CoreRelockCallback()`. On any one application core calls to `LWHVR_CoreUnlockCallback()` and `LWHVR_CoreRelockCallback()` will never be nested - i.e. we always see:

```
LWHVR_CoreUnlockCallback() // Application core X
LWHVR_CoreRelockCallback() // Application core X

LWHVR_CoreUnlockCallback() // Application core X
LWHVR_CoreRelockCallback() // Application core X
```

and NEVER:

```
LWHVR_CoreUnlockCallback() // Application core X
LWHVR_CoreUnlockCallback() // Application core X

LWHVR_CoreRelockCallback() // Application core X
LWHVR_CoreRelockCallback() // Application core X
```

However, since application cores run in parallel you may see nested calls. E.g. you might see the following:

```
LWHVR_CoreUnlockCallback() // Application core X
LWHVR_CoreUnlockCallback() // Application core Y

LWHVR_CoreRelockCallback() // Application core X
LWHVR_CoreRelockCallback() // Application core Y
```

Therefore, if the unlocking operation uses a resource shared between cores it will be necessary to use some sort of lock to a stop a new call of `LWHVR_CoreUnlockCallback()` from running until a previous call of `LWHVR_CoreUnlockCallback()` has been completed with a call of `LWHVR_CoreRelockCallback()`. E.g.

```
void LWHVR_CoreUnlockCallback(void)
{
    <claim spin-lock>
    <unlock access to configuration registers>
}
```

```
void LWHVR_CoreRelockCallback(void)
{
    <relock access to configuration registers>
    <release spin-lock>
}
```

See the target specific documentation for which configuration registers must be unlocked by this function.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.12 LWHVR_CoreRelockCallback

Prototype:

```
void LWHVR_CoreRelockCallback(void);
```

Purpose:

Called to disable access to configuration registers that require unlocking for the calling application core.

See section 9.11 for details.

See the target specific documentation for which configuration registers must be relocked by this function.

Parameters:

None.

Returns:

Nothing.

Restrictions:

See sections 9.13 and 9.14.

9.13 Restrictions on all Call-back Functions

The following restrictions apply to all call-back functions.

- Apart from the exceptions listed below, call-back functions must not call functions in the master software API.

  The exceptions to this rule are:

  - `LWHVR_StoppedVMCallback()`, `LWHVR_ShutdownVMCallback()` and `LWHVR_VMErrorCallback()` may call `LWHVR_RestartVM()`.
  - `LWHVR_ClockCallback()` may call `LWHVR_RequestExtraTimeForVM()`.

See also section 4.12.

9.14 Restrictions on Call-back Functions that run on an Application Core

The following restrictions apply to call-back functions running on application cores - that is all call-backs except for `LWHVR_ErrorCallback()` when it results from the master core calling `LWHVR_Start()`, `LWHVR_StopVM()`, `LWHVR_ShutdownVM()`, `LWHVR_RestartVM()` or `LWHVR_RequestExtraTimeForVM()`:

- No traps or interrupts may be generated (other than the clock-tick interrupt managed by `LWHVR_StartTimerCallback()` and `LWHVR_ClockCallback()`). If the call-back does generate a trap or interrupt then the `LWHVR_UnexpInterruptCallback()` call-back will be called or the `LWHVR_UnexpInterruptHook` hook will be invoked depending on the type of trap or interrupt.
- Since the call-back executes with interrupts disabled, scheduling of VMs is stalled during its execution. Therefore call-backs must execute as quickly as possible. Please see section 12 for more information about the effects of call-back function execution on scheduling.
- Call-backs must not enable interrupts or change the processor's interrupt priority level.

10 Configuration Variables

This chapter describes global configuration variables that may be used by the master software to affect the behaviour of the RTA-LWHVR.

10.1 LWHVR_CoreConfigWord

Declaration:

```
extern LWHVR_RegisterType LWHVR_CoreConfigWord;
```

Purpose:

This variable may be used to control the configuration of an application core whilst VMs are running. Its value is target specific.

`LWHVR_CoreConfigWord` will have a default value that will allow correct operation of the RTA-LWHVR. If `LWHVR_CoreConfigWord` needs to be changed, it must be changed before `LWHVR_Init()` is called and it must not be changed again after `LWHVR_Init()` has been called.

11 VM API

VMs are able to call a small number of API services from the RTA-LWHVR. This chapter describes this VM API.

Typically calling a VM API service will involve the VM executing a trap instruction to transfer control to the RTA-LWHVR. (Each VM is in a separate memory image and is not linked together with the master software, so a VM cannot call a C function in the RTA-LWHVR in the normal way.)

Each API service has a number used to identify the service being called and may have additional arguments. How the service number and arguments are specified is target specific. The header file `LWHVR_VMAPI.h` contains constants for the service numbers. The constant names are used here to refer to the API services.

If an invalid service number is used the calling VM will be in error (see section 5.6) and `LWHVR_VMErrorCallback()` will be called with the error `LWHVR_ErrorInvalidVmAPI`.

11.1 LWHVR_VMAPI_SYNC_PS_INTS

Service Number: 0

Purpose:

This API service is used to inject pending and enabled pseudo-interrupts into the calling VM. Pending and enabled pseudo-interrupts are automatically injected into the running VM when the RTA-LWHVR processes a clock-tick interrupt or when a VM makes a `LWHVR_VMAPI_RETURN_FROM_PS_INT` API service call to return from a pseudo-interrupt handler.

VMs will only normally need to use this API if they disable interrupts by clearing bits in their status block's `psIntEnabled` field and then later re-enable the interrupts. To avoid the overhead of an unnecessary call to the RTA-LWHVR, a VM should only use this call if it has pending and enabled pseudo-interrupts (i.e. the bitwize-and of the VM status block `psIntPending` and `psIntEnabled` fields is not 0).

For example, assume that the VM status block can be accessed with the token `VMStatusBlock`:

```
/* Disable pseudo-interrupts. */
LWHVR_UInt32Type prevEnabled = VMStatusBlock.psIntEnabled;
VMStatusBlock.psIntEnabled = 0U;

/* Code that must not be interrupted. */

/* Re-enable pseudo-interrupts. */
VMStatusBlock.psIntEnabled = prevEnabled;

/* Inject any pending and enabled pseudo-interrupts. */
if ((VMStatusBlock.psIntPending & VMStatusBlock.psIntEnabled)!= 0U)
{
    VMSynchronisePseudoInterrupts();
}
```

11.2 LWHVR_VMAPI_RETURN_FROM_PS_INT

Service Number: 1

Purpose:

This API service is used to terminate execution of a pseudo-interrupt handler (see section 5.8.6) and resume execution of the interrupted code.

This API service does not take any arguments, however it is affected by the contents of the calling VM status block (see section 5.7). When this API call is made:

1. The VM's status block `psIntEnabled` field is set to the value of the VM status block `psIntPreviousEnabled` field.
2. A target specific register may be restored from the VM status block `psIntRestoreRegister` field.
3. Execution is resumed at the address stored in the VM status block `psIntResumeAddress` field.

Parameters:

None.

11.3 LWHVR_VMAPI_INJECT_PS_INT

Service Number: 2

Purpose:

This API allows the calling VM to explicitly inject a pseudo-interrupt into itself. That is:

1. The VM specifies the number of a pseudo-interrupt to be injected (0-31) in the first parameter.
2. The RTA-LWHVR sets the corresponding bit (1 << pseudo-interrupt number) in the VM status block `psIntPending` field.
3. If the pseudo-interrupt is also enabled in the VM status block `psIntEnabled` field then the pseudo interrupt is immediately injected into the calling VM.

Parameters:

First the number of the pseudo-interrupt to be injected.

Errors:

Calling this API with an invalid argument will result in the VM being in error and `LWHVR_VMErrorCallback()` being called with the following error:

`LWHVR_ErrorInvalidPsInterrupt`

Notes:

This API service is not intended for normal use. Its intended purpose is to support testing of RTA-OS. However it is documented here as it may have some other unforeseen applications.

11.4 LWHVR_VMAPI_SHUTDOWN

Service Number: 3

Purpose:

This API service is used by a VM when it wishes to stop executing; either in response to a shutdown pseudo-interrupt or for some other reason. A call to this API service does return.

After the VM has made this call it is marked as not runnable and will not be scheduled by the RTA-LWHVR. The only way that the VM will run again is if the master software calls `LWHVR_RestartVM()` for the VM.

Parameters:

None.

11.5 LWHVR_VMAPI_REQUEST_EXTRA_TIME

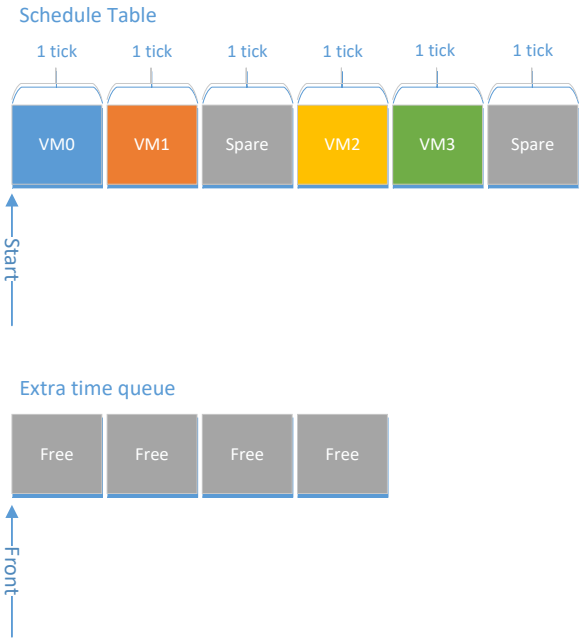
Service Number: 4

Purpose:

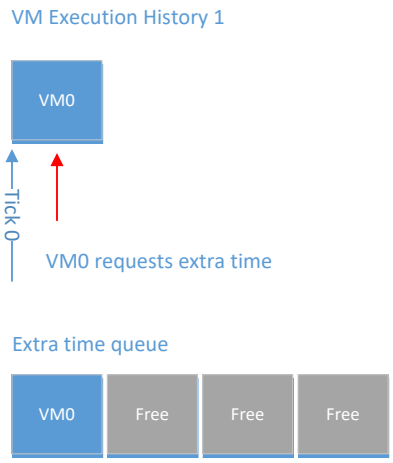
This API service is used by a VM to request extra execution time. Normally a VM runs in the time-slices assigned to it in a schedule table (see section 6.3.5). If a VM calls this API service it is added to a low-priority FIFO queue of VMs that have requested extra time on the calling VM's application core. (Note that this is a different queue to the high-priority queue used for extra-time requests made by the master software – see section 8.8.) If the RTA-LWHVR's scheduler encounters a spare time-slice entry in an application core's schedule table (that has not been skipped due to extra-time requests made by the master software – see section 8.8) it will remove the VM at the front of the application core's extra-time queue and then run it in the spare time-slice.

A VM may only be in a low-priority extra-time queue once. This API service may be called multiple times, but only those calls made when the VM is not already in a extra-time queue will have any effect. This means that spare time-slices are shared out fairly between requesting VMs. However, please note that spare time-slices are also consumed when the master software requests extra execution time for a VM.

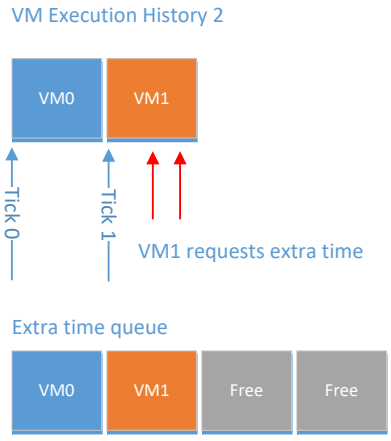
For example, consider the following schedule table that has 6 entries, two of which are spare time.



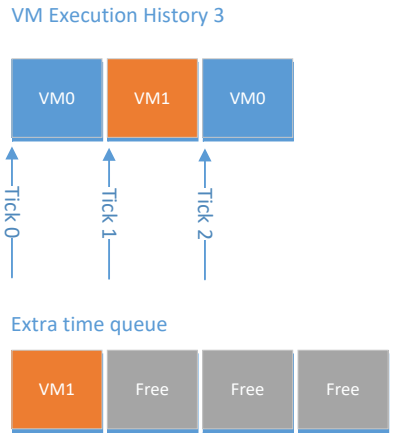
At clock-tick 0 the RTA-LWHVR would select the first entry in the schedule table, and therefore execute VM0. Assume that while VM0 is executing it requests extra-time. VM0 would be added to the extra-time queue.



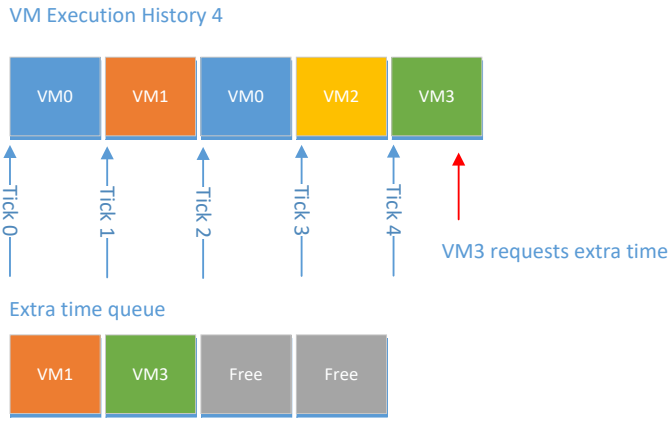
At clock-tick 1 the RTA-LWHVR would select the second entry in the schedule table, and therefore execute VM1. Assume that while VM1 is executing it requests extra-time twice. VM1 would be added to the extra-time queue but only once.



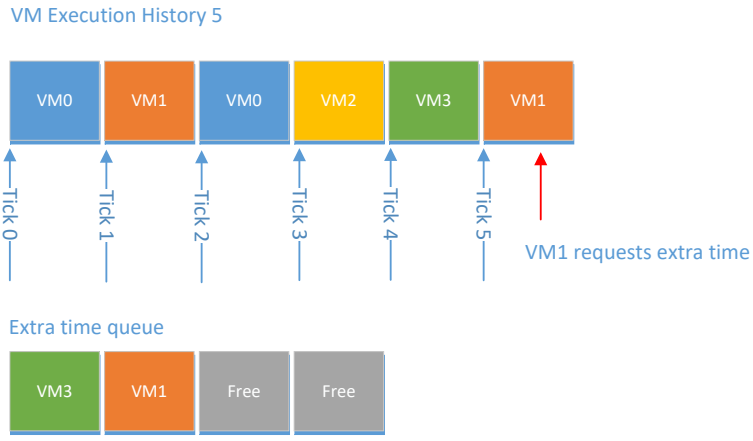
At clock-tick 2 the RTA-LWHVR would discover that the third entry in the schedule is spare. Therefore it would remove VM0 from the front of the extra-time queue and execute it.



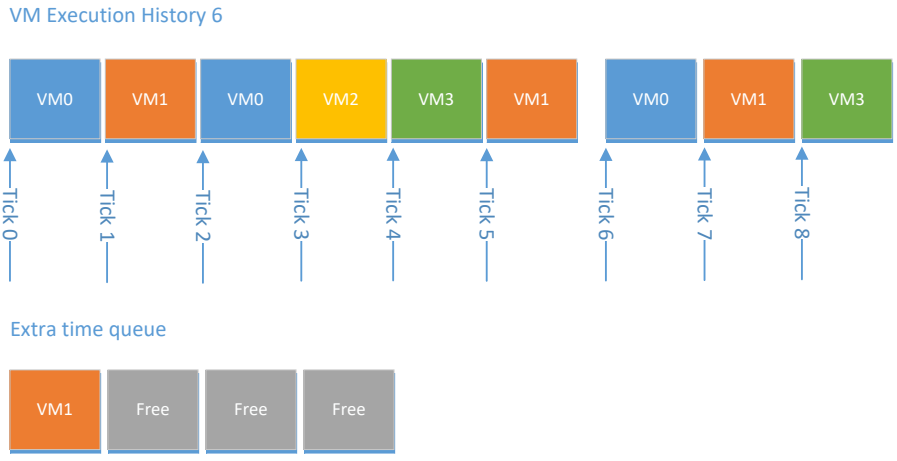
At clock-tick 3 the RTA-LWHVR would select the fourth entry in the schedule table, and therefore execute VM2. Then at clock-tick 4 the RTA-LWHVR would select the fifth entry in the schedule table, and therefore execute VM3. Assume that while VM3 is executing it requests extra-time. VM3 would be added to the extra-time queue.



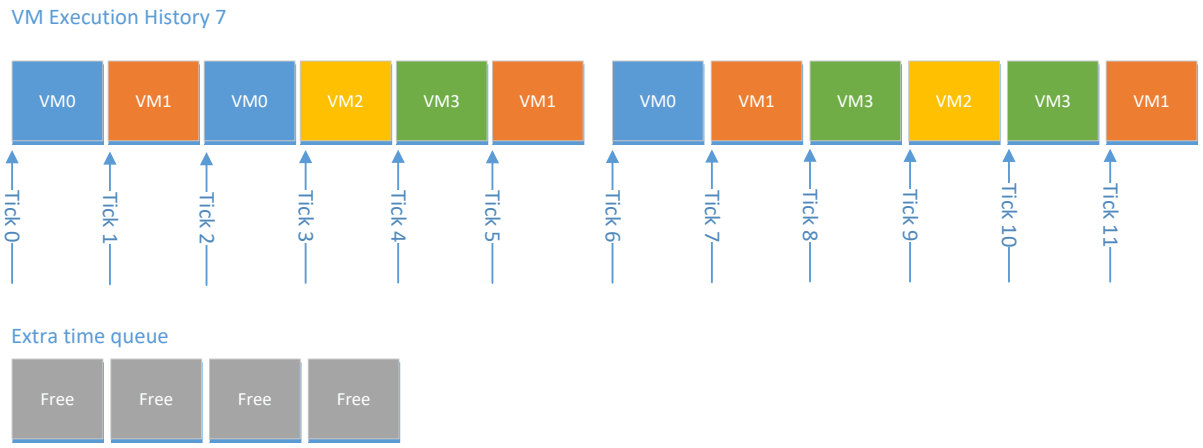
At clock-tick 5 the RTA-LWHVR would discover that the sixth entry in the schedule is spare. Therefore it would remove VM1 from the front of the extra-time queue and execute it. Assume that while VM1 is executing it requests extra-time. VM1 would be added to the extra-time queue.



At clock-tick 6 the RTA-LWHVR would return to the start of the schedule and therefore execute VM0 again and then execute VM1 again at clock-tick 7. At clock-tick 8 the LHWVR would again find that the third schedule table entry is spare, so it would remove VM3 from the front of the extra-time queue and execute it.



At clock-ticks 9 and 10 the RTA-LWHVR would execute VM2 and VM3 again. At clock-tick 11 LHWVR would again find that the sixth schedule table entry is spare, so it would remove VM1 from the front of the extra-time queue and execute it.



Parameters:

None.

11.6 LWHVR_VMAPI_ATOMIC_MEMORY_COPY

Service Number: 5

Purpose:

This API service is used to copy memory in a way that cannot be interrupted by time-slicing.

The memory to be copied is specified using an array of `LWHVR_MemoryCopyExtentType` structures. Each structure specifies an extent of memory to be copied as follows:

`LWHVR_MemoryCopyExtentType.fromAddress`

The source address.

`LWHVR_MemoryCopyExtentType.toAddress`

The destination address.

`LWHVR_MemoryCopyExtentType.size`

The number of bytes to copy from source to destination.

To stop a VM from "stealing" time from another VM by disabling time-slicing for an extended period of time the number and size of extents is limited as follows:

- By default the maximum size of a memory extent that may be copied is 256 bytes. To override this default value define the C macro `LWHVR_MAX_COPY_EXTENT_SIZE` to be the maximum extent size.
- By default the maximum number of memory extents that may be copied is 8. To override this default value define the C macro `LWHVR_MAX_NUM_COPY_EXTENTS` to be the maximum number of extents.

This API service can only be used to copy from memory that the VM's configuration allows it to read to memory that the VM's configuration allows it to write. Otherwise a `LWHVR_ErrorMemoryPermission` error will occur. Likewise the array of `LWHVR_MemoryCopyExtentType` structures must be in memory the VM is allowed to read.

Parameters:

First	The address of an array of `LWHVR_MemoryCopyExtentType` structures specifying the memory extents to be copied.
Second	The number of memory extents to be copied.

Errors:

Calling this API with invalid arguments will result in the VM being in error and `LWHVR_VMErrorCallback()` being called with one of the following errors:

- `LWHVR_ErrorMemoryPermission`
- `LWHVR_ErrorToManyExtents`
- `LWHVR_ErrorExtentTooLarge`

12 Avoiding Schedule Timing Issues

This chapter discusses two inter-related issues that can affect the scheduling of VMs: schedule drift and interrupt blocking by VMs.

12.1 Schedule Drift

Consider a RTA-LWHVR configuration that contains a schedule table that is L ticks long (e.g. a schedule table with a 1 tick-time slice, followed by a 2 tick-time slice, followed by a 1 tick time-slice would be 4 ticks long).

Assume that a tick is T micro-seconds long.

Let t be the current time in micro-seconds.

Assume that the first clock-tick arrives at time t = 0.

- The schedule does not *drift* if at the start of every cycle of the schedule t % (L * T) = 0.
- The schedule does *drift* if at the start of any cycle of the schedule t % (L * T) ≠ 0.

Consider the following case where the schedule has a 1 tick time-slice for VM0 followed by a 1 tick time-slice for VM1.

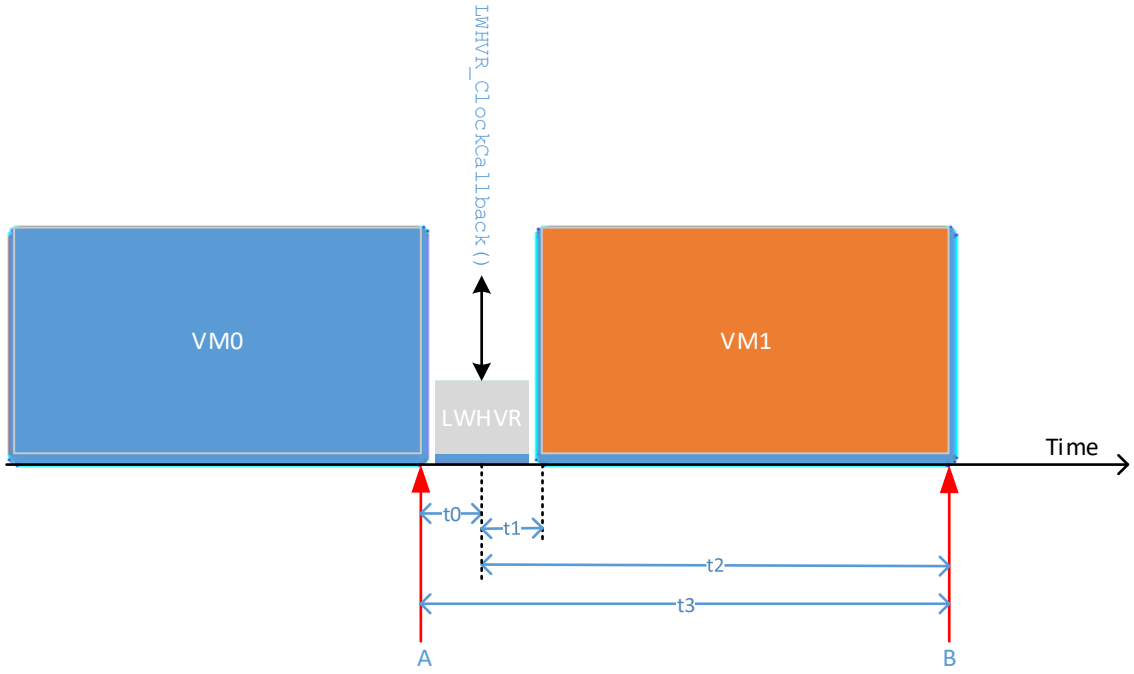


Figure 8: Schedule Drift

At time A a clock-tick interrupt arrives and control passes from VM0 to the RTA-LWHVR. The RTA-LWHVR processes the interrupt during time interval t0 and then calls the `LWHVR_ClockCallback()` call-back function to re-arm the clock-tick interrupt source. After `LWHVR_ClockCallback()` returns the RTA-LWHVR carries out some more processing during time interval t1[2] and then VM1 starts execution. Sometime later at time B another clock-tick interrupt arrives and the RTA-LWHVR runs again – and presumably selects another VM to execute.

[2] Note that the length of t1 will depend on how many memory regions VM1 has and thus how long it takes the RTA-LWHVR to program the processor's MPU.

The effect of the delay t0 depends on the kind of timer used to generate clock-tick interrupts:

a) If the clock-tick interrupt source uses a timer that generates an interrupt at a fixed frequency (e.g. a programmable interval timer) then the schedule will not drift because two timer-ticks will always occur at an interval of t3 irrespective of the time t0 before `LWHVR_ClockCallback()` is called.

b) If the clock-tick interrupt source uses a timer that generates an interrupt at a fixed interval after being (re-)armed (such as a counter with a compare register) then care must be taken to avoid schedule drift - if it is required that the schedule does not drift. If a tick length of t3 is required then when `LWHVR_ClockCallback()` is called the interrupt source must be set to generate an interrupt after time interval t2. t0, and hence t2, can be calculated in `LWHVR_ClockCallback()`. For example, if an interrupt source that uses a counter and a compare register is used, t0 is the difference between the value of the compare register and the current value of the counter when `LWHVR_ClockCallback()` is called (i.e. the amount that the counter has advanced beyond the value in the compare register). Note however, the value of t0 may vary due to interrupt blocking by VMs - see below – so t2 may not be fixed but may need to be calculated dynamically.

12.2 Interrupt Blocking by VMs

When the RTA-LWHVR executes a service call for a VM (see section 11), or a VM enters an error state, interrupts are blocked for the duration of the service call or error processing. This means that handling of a clock-tick interrupt will be delayed until the end of the service call or error processing. Consider the following case where VM0 either makes a service call or causes an error just before the end of its time-slice.

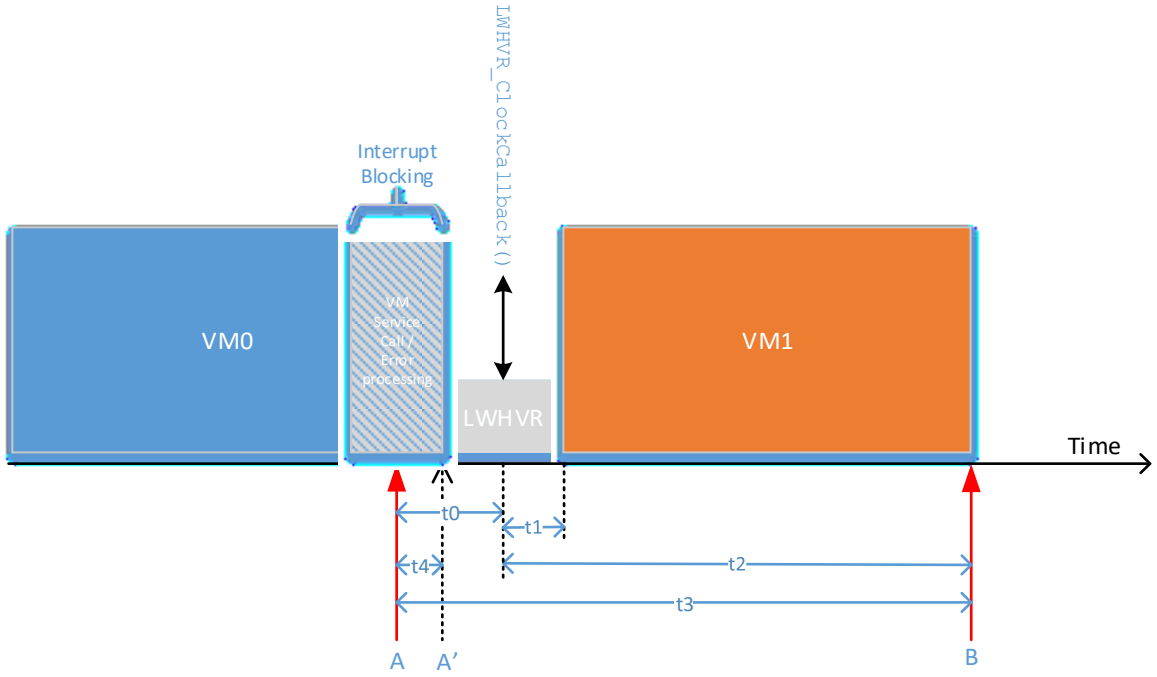


Figure 9: Interrupt Blocking by VMs

At time A a clock tick-interrupt arrives, but control does not pass to the RTA-LWHVR immediately because the RTA-LWHVR is processing a service call or handing an error and interrupts are disabled. The handling of the interrupt is delayed for the interval t4 to time A'. The effect of this interrupt blocking depends on how timer-ticks are generated:

a) If a fixed frequency interrupt source (e.g. a programmable interval timer) is used then the amount of time for which VM1 actually executes in the next time-slice will be shortened by t4. We shall refer to this effect as *time-slice shortening*.

b) If an interrupt source is used that generates an interrupt at a fixed interval after being (re-)armed (such as a counter with a compare register) then there is more choice over what happens:

- If the interrupt source is re-armed with an interval t2 that is calculated dynamically by measuring t0 then the effect is the same as using a fixed frequency timer – i.e. the amount of time for which VM1 actually executes in the next time-slice will be shortened by t4.
- If the timer is re-armed with a fixed interval t2 calculated by measuring t0 in the absence of interrupt blocking then the next time-slice will not be shortened, but the schedule will drift by t4.

Which of these behaviours is most appropriate depends on the application.

12.3 Dealing with Interrupt Blocking

As described in section 12.2 interrupt blocking by VMs will potentially lead to time-slice shortening or schedule drift. Which of these is more acceptable depends on the application. Either the schedule must be constructed so that the system is robust against time-slice shortening or robust against schedule drift.

To construct a robust schedule it may be necessary to know the worst-case time for which interrupts can be blocked and hence the largest potential interval t4 in Figure 10.

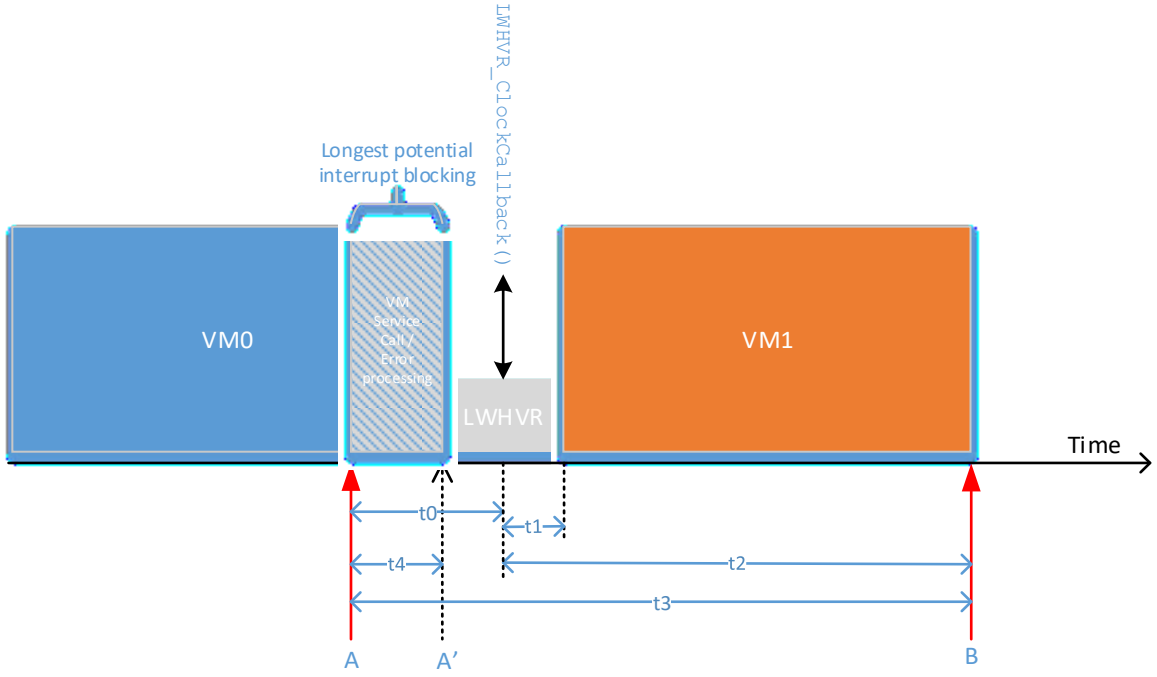


Figure 10: Worst-case Interrupt Blocking

Unfortunately the worst-case blocking time depends on how the RTA-LWHVR has been compiled and the execution time of the call-back functions `LWHVR_StoppedVMCallback()`, `LWHVR_ShutdownVMCallback()` and `LWHVR_VMErrorCallback()`. This section provides guidance on how the worst-case blocking time can be measured.

12.3.1 VM API Service Calls

The duration of a VM API service call can be measured as follows:

A. Create a RTA-LWHVR configuration that:
   - Contains a single VM.
   - Contains a schedule table that contains one single-tick time-slice that executes the single VM.
   - Allows the VM to read a high precision time source – e.g. configure the VM's memory so that it can read a high precision timer.

B. In the VM use code something like:
   1. Loop until the `ticksWhileRunning` field of the VM's status block changes value. (We now know that we are at the start of a time-slice.)
   2. Read the high precision time source and store the value read in `tS`.
   3. Call the VM service.
   4. Read the high precision time source and store the value read in `tE`.
   5. The duration of the service call is `tE` – `tS`.

The above must be done for every VM service call (see section 11). The longest duration measured is the worst-case interrupt blocking due to VM service calls.

Note: at step 1 above the code waits until it is at the start of a time-slice. This is done to avoid a clock-tick interrupt occurring during the VM service call and being handled as soon as the service call returns but before the timer is read at step 4. (Of course, if you have a very short tick length this may still happen.)

When carrying out such measurement the following are important:

1. It must be arranged that the call-back function `LWHVR_ShutdownVMCallback()` executes for its worst-case execution time.
2. When calling the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service the maximum number of extents must be used and the extents must be of maximum size.
3. If the system contains memory of different speeds, then when calling the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service the list of memory extents and the memory extents to be copied must be in the slowest memory that can be copied by the VM.
4. When calling the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service the memory extents should not be aligned on a 4-byte boundary (since unaligned copying is usually slower).
5. When the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service determines if a VM has permission to read or write memory it checks memory regions in the order they appear in the VM's configuration (e.g. `LWHVR_VM0_MPU_REGION0_XXX`, then `LWHVR_VM0_MPU_REGION1_XXX`, then `LWHVR_VM0_MPU_REGION2_XXX` etc.) until either it finds a region that allows the read/write or it exhausts the regions. Therefore to be sure that the worst-case duration of the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service is measured, the list of memory extents and the memory extents must be in the memory region that appears last in the VM's configuration. (The number of memory regions a VM will have in its configuration will depend on the system. Carry out measurement using the maximum number of regions that VMs in your system will have.)

Note that the maximum duration of the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service can be changed by modifying the maximum number and size of extents that can be copied. See section 11.6.

12.3.2 Error Handling

There are two ways that a VM can enter an error state and thus require error handling by the RTA-LWHVR:

a) The VM attempts to carry out a privileged operation or access memory without permission. This leads to the processor generating a trap that causes the RTA-LWHVR to start error handling immediately.

b) The VM makes a service call with an invalid argument. This is detected by the RTA-LWHVR, which then starts error handling.

Measuring the worst-case interrupt blocking due to error handling is similar to measuring the worst-case interrupt blocking due to VM service calls except that after the VM has triggered an error, control does not return to the VM; instead the RTA-LWHVR executes in an "idle context" for the remainder of the time-slice. When executing in the idle context the RTA-LWHVR loops calling the macro `LWHVR_IDLE()`. Normally this macro expands to a sequence of no-operation instructions. To measure error-handling time we measure the time between triggering the error and the `LWHVR_IDLE()` macro running.

Error handling duration can be measured as follows:

A. Create a RTA-LWHVR configuration that:
   - Contains a single VM.
   - Contains a schedule table that contains one single-tick time-slice that executes the single VM.
   - Allows the VM to read a high precision time source – e.g. configure the VM's memory so that it can read a high precision timer.

B. Create variable `tS` that is in memory that can be read and written by the master software and the VM.

C. Create variable `errorTriggered` in memory that can be read and written by the master software and the VM. Initialize `errorTriggered` to 0 in the master software before the RTA-LWHVR is started.

D. In the VM use code something like:
   1. Loop until the `ticksWhileRunning` field of the VM's status block changes value. (We now know that we are at the start of a time-slice.)
   2. Set `errorTriggered` to 1.
   3. Read the high precision source and store the value read in `tS`.
   4. Trigger the error.

E. Override the `LWHVR_IDLE()` macro. This is defined in `LWHVR_TargetDefinitions.h` and is surrounded by `#ifndef LWHR_IDLE` so it can be overridden either by modifying `LWHVR_TargetDefinitions.h` or by using a compiler option (usually –D) to provide a definition of `LWHVR_IDLE()`.

   The override version of `LWHR_IDLE()` should do the following:

   If `errorTriggered` is 1 then:

   Read the high precision time source and store the value read in `tE`.

   The duration of the error handling is `tE` – `tS`.

   Set `errorTriggered` back to 0.

The `errorTiggered` flag is needed because the idle context may execute at times other than immediately after error handing so we need to be able to distinguish the case when an error has occurred.

The following errors must be triggered and the error handling duration measured using the above method. The longest error handling duration measured is the worst-case interrupt blocking due to error handling.

- Access memory to which the VM does not have access permission (`((int *) 0) = 0;` will usually trigger such an error).
- Execute a privileged instruction.
- Call the VM API service with an invalid service number. It does not matter what number is used as long as it is not a valid service number listed in `LWHVR_VMAPI.h`. E.g. use service number 99.
- Call the `LWHVR_VMAPI_INJECT_PS_INT` service with a pseudo-interrupt number greater than 31.
- Call the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service with too many extents.
- Call the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` with an extent list that is in memory that the VM does not have permission to read.
- Call the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` with the maximum number of extents. All extents but the last must be valid. The last extend must be too large.
- Call the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` with the maximum number of extents. All extents but the last must be valid. The last extent must have a `toAddress` field that addresses memory to which the VM does not have permission to write.

When carrying out such measurement the following are important:

1. It must be arranged that the call-back function `LWHVR_VMErrorCallback()` executes for its worst-case execution time.
2. When the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service determines if a VM has permission to read or write memory it examines memory regions in the order they appear in the VM's configuration (e.g. `LWHVR_VM0_MPU_REGION0_XXX`, then `LWHVR_VM0_MPU_REGION1_XXX`, then `LWHVR_VM0_MPU_REGION2_XXX` etc.) until either it finds a region that allows the read/write or it exhausts the regions. Therefore to be sure that the worst-case error handling duration resulting from the `LWHVR_VMAPI_ATOMIC_MEMORY_COPY` service is measured, the list of extents and the valid extents must be in the memory region that appears last in the VM's configuration. (The number of memory regions a VM will have in its configuration will depend on the system. Carry out measurement using the maximum number of regions that VMs in your system will have.)

12.3.3 Stopping a VM

When a VM is forcibly stopped (see section 8.5) the RTA-LWHVR calls the call-back function `LWHVR_StoppedVMCallback()` at the start of the time-slice when the stopped VM would next have run. Since it is called at the start of a time-slice, as long as `LWHVR_StoppedVMCallback()` executes in less than 1 tick its execution should not result in time-slice shortening or schedule drift. However, the time for which `LWHVR_StoppedVMCallback()` can result in interrupt blocking can be determined as follows:

A. Create a RTA-LWHVR configuration that:
   - Contains a single VM.
   - Contains a schedule table that contains one single-tick time-slice that executes the single VM.

B. Initialize variable `stopTriggered` to 0 in the master software before the RTA-LWHVR is started.

C. In the master software call `LWHVR_StopVM()` to forcibly stop the VM.

D. At the start of `LWHVR_StoppedVMCallback()` use code that does:
   1. Read a high precision time source and store the value read in tS.
   2. Set `stopTriggerred` to 1.

E. Override the `LWHVR_IDLE()` macro. This is defined in `LWHVR_TargetDefinitions.h` and is surrounded by `#ifndef LWHR_IDLE` so it can be overridden either by modifying `LWHVR_TargetDefinitions.h` or by using a compiler option (usually –D) to provide a definition of `LWHVR_IDLE()`.

   The override version of `LWHR_IDLE()` should do the following:

   If `stopTriggered` is 1 then:

   Read the high precision time source and store the value read in `tE`.

   The duration of the forcible stop handling is `tE – tS`.

   Set `stopTriggered` back to 0.

   The `stopTiggered` variable is needed because the idle context may execute at times other than immediately after a VM has been forcibly stopped so we need to be able to distinguish the case when a forcible stop has occurred.

13 ETAS Contact Addresses

ETAS HQ

ETAS GmbH
Borsigstraße 24 Phone: +49 711 3423-0
70469 Stuttgart Fax: +49 711 3423-2106
Germany WWW: www.etas.com

ETAS Subsidiaries and Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries WWW: www.etas.com/en/contact.php
ETAS technical support WWW: www.etas.com/en/hotlines.php