Julius-Maximillians-
UNIVERSITÄT
WÜRZBURG

Aerospace Information Technology

RODOS

Introduction

Version:	**2.0**
Date:	**Sept. 2019**
Author:	**S. Montenegro**

RODOS (Real Time Object Oriented Dependable Operating System) is a real-time operating system for embedded systems. In contrast to non real-time operating systems such as Windows or Linux, RODOS is designed to execute actions at a specific point in time. Naturally, no computer can meet an exact point in time but it is possible to fulfill a task in a defined time window. In order to determine a time window that can always be met, the software has to be predictable. That means that, for example, the scheduler which distributes the processor time has to be called in constant periods. Besides that, real-time systems have to consider the priority of each process. If a highly prioritized task is executed, no process of lower importance will be called by the scheduler. All these requirements are met in RODOS.

RODOS is designed for the use on systems that have to be highly dependable and fail-safe. In order to achieve this dependability, the philosophy of the development of RODOS is to keep it as simple as possible because complexity is the reason for most development faults. In other words:

> *“Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.” - Antoine de Saint-Exupery (1900 – 1944)*

This philosophy leads to important advantages for high dependable systems. One example is a higher execution speed because no processing time is wasted for unnecessary tasks. Another advantage is the extremely short boot period because there is only a small amount of software that has to be initialized. Consequently, the system can reboot very quickly after a malfunction has occurred. Because of the compactly written code there is only a small amount of memory needed which allows RODOS to be executed on a variety of microcontrollers.

RODOS is written in an object-oriented way in C++ so that it can be easily understood and modified by a large number of developers. Some small parts of RODOS are coded in C or assembler for hardware specific purposes only.

RODOS supports several hardware platforms such as microcontrollers with ARM7 architecture, STM32 Cortex-M3 or Atmel AVR32. Furthermore, RODOS can run as a guest on different host operating systems such as Linux, Windows and FreeRTOS.

1.1. Time management

RODOS has a global timer counter that measures the time from the start of the system in nanoseconds. The whole system uses this counter, for example for scheduling and execution of tasks at a specific point in time.

1.2. Threads

Threads in RODOS are user defined parts of software that contain logic to fulfil a specific purpose. Multiple threads can be executed simultaneously. For instance, one thread is reading the current sensor values while another one is sending commands to an actuator. These threads do not really run simultaneously, but they are executed pre-emptively. That means that the scheduler regularly interrupts the current running process and looks for another thread that has to be executed. In case that there is a thread with a priority that is at least as high as the one of the interrupted process, this thread will be executed until the next scheduler interruption. Because of short scheduler periods, it creates the impression that the threads are executed simultaneously.

Another possible interruption of a thread can be triggered by the thread itself. If the thread has finished its tasks, it can suspend for a defined amount of time. After that time, the scheduler calls the suspended thread which is very useful for actions that have to be executed periodically. While one thread is suspended, another thread can be executed and no CPU time is wasted for waiting.

For safe communication between local threads, the RODOS framework provides several synchronous and asynchronous possibilities, such as communication buffers and FiFos (fist in - fist out). The synchronization of threads can also be realized by the use of semaphores.

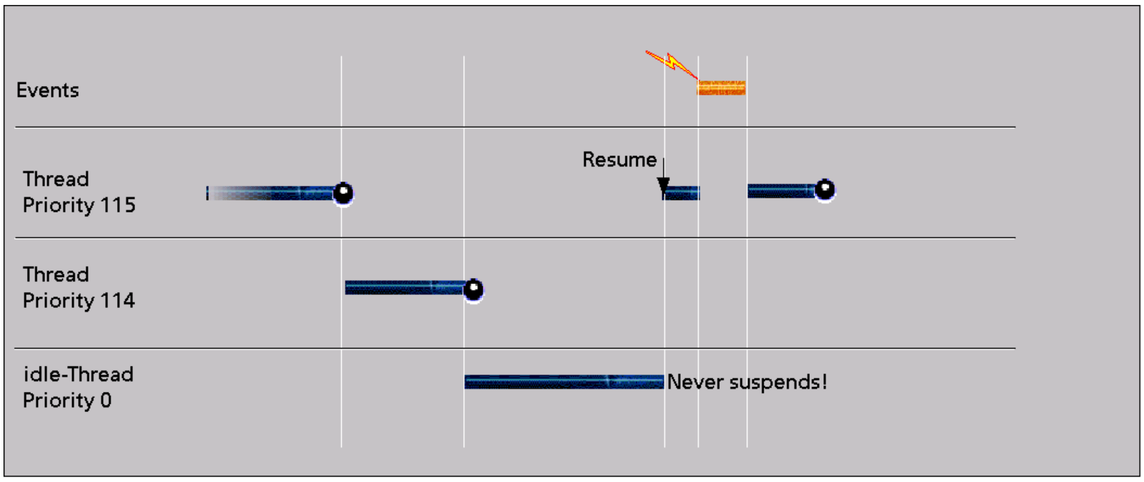

Illustration 1: Preemptive threads and time events

1.3. Time Events

It is possible to define time events in RODOS. These events are executed at a defined point in time or a defined period, or they are triggered by an external interrupt. Time events are not designed to perform extensive tasks, but to handle an event briefly and to initiate further steps if necessary.

1.4. Applications

An application encloses threads and events. Consequently, the complexity of the system is reduced because it seems that the system consists of a few applications only. The complex threads and events are hidden inside. Furthermore, unused applications can be disabled.

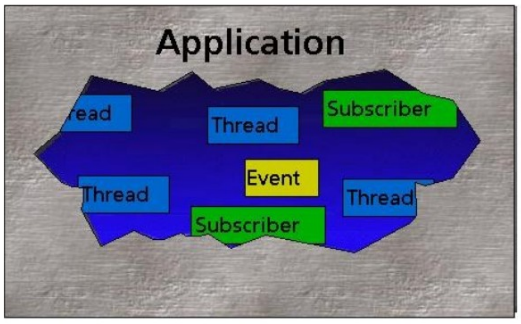

Illustration 2: Threads and events hidden in an application

1.5. Middleware

RODOS uses a middleware for communication between local threads and threads on distributed RODOS systems. This middleware follows the publisher–subscriber pattern. The interface between the publisher and the subscriber is called “Topic”. A thread publishes information using a topic, for example the current temperature. Every time the publisher provides new temperature data, the thread that subscribes to this topic will receive the data. However, there can also be multiple publishers and subscribers. The advantage of this middleware is that all publishers and subscribers can work independently without any knowledge about the other. Consequently, publishers can easily be replaced in case of a malfunction. The subscribers do not notice the replacement.

Different RODOS systems can communicate via gateways. These gateways use different protocols like UDP and UART to transfer the middleware messages to different RODOS nodes. A subscriber cannot determine from which node the subscribed information origins.

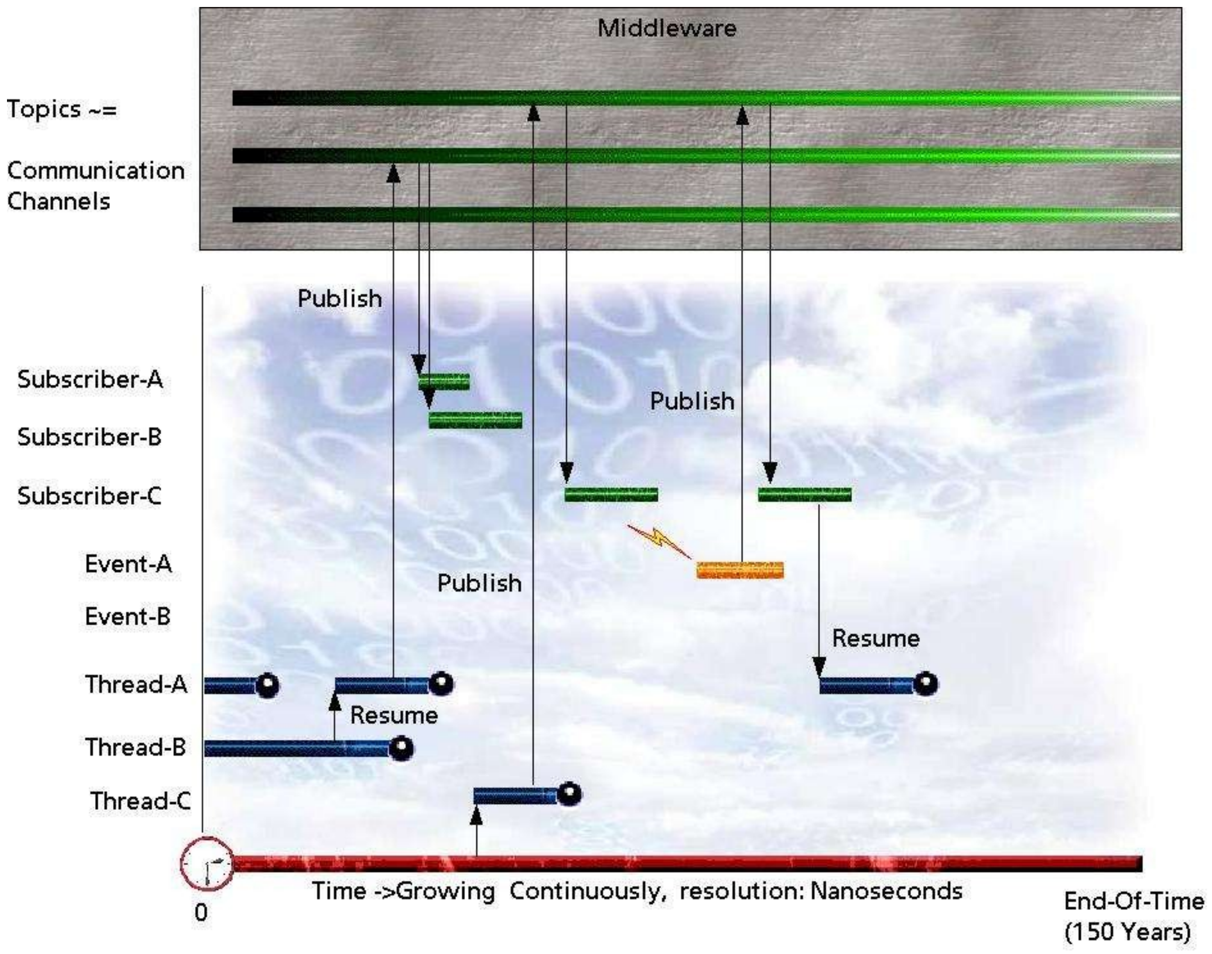

Illustration 3: Threads communicating via the middleware

1.6. Hardware abstraction layer (HAL)

In order to keep the use of protocols such as UART and I²C simple, RODOS has a hardware abstraction layer. This layer provides functions for several I/O interfaces so that sending and receiving data, for example via UART, is very simple. Furthermore, it is standardized, i.e. its function calls, for instance sending data via UART, are identical for all hardware platforms.

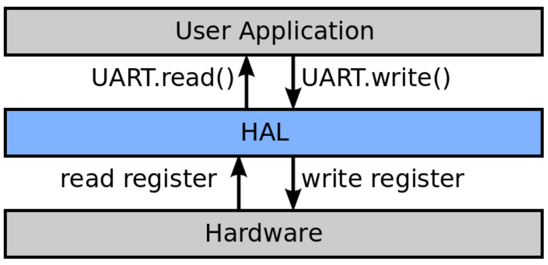

Illustration 4: Hardware abstraction layer