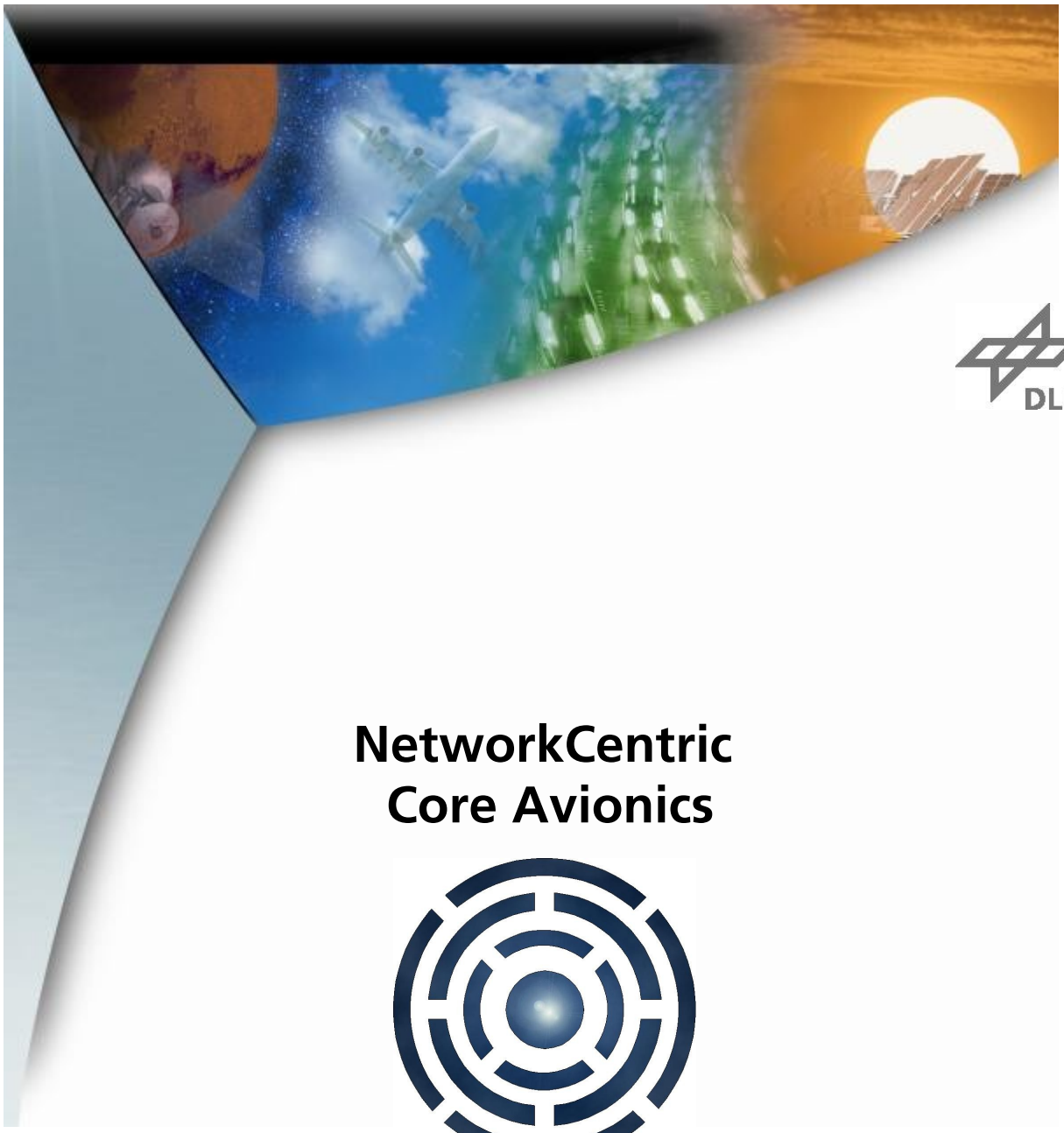


# NetworkCentric Core Avionics



# RODOS IO Concept

**Version:** 01
**Date:** 29.01.2011

# RODOS Input Output Concept

Sergio Montenegro
Univerity Würzburg
sergio.montenegro@uni-wuerzburg.de

The Input output concept for RODOS targets modularity and to implement the Building Blocks / (software) components paradigm. A RODOS system shall be implemented as a composition of simple components which are independent form each other. For each component only its interface have to be known in order to be able to compose the system.
Components can be for example user applications, basis services, gateways to different networks and IO Drivers. From outside there is no difference what is inside of the components. The IO subsystem shall not be an exception. Each device interface (typically a Device-Driver) shall be implemented as an software component which may be accessed by middleware communication. Components (including IO Drivers) are interconnected by the RODOS middleware which bridge over node boundaries and software and hardware boundaries. In this way, applications running in any node in the system may access devices attached to other nodes. Without any extra coding.

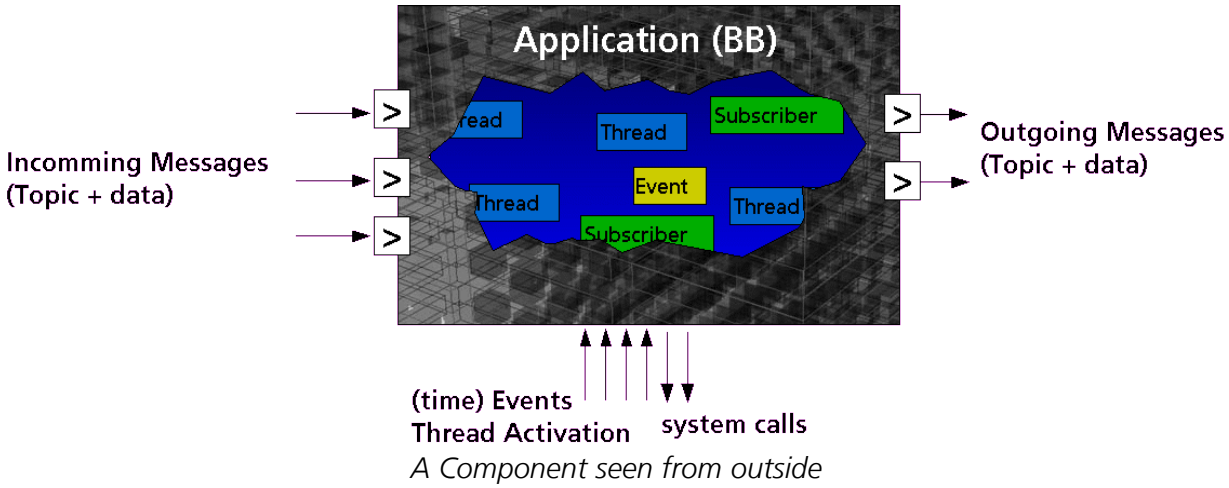
Please refer to the document "RODOS Real time kernel design for dependability"

## 1. The Theory and RODOS concepts review

### 1.1 What is a (software) component (including IO Drivers)

In engineering design, the idea of aggregating standardized components to create a complex system has allowed engineers to create better systems more easily. Components are described in a handbook, where each has a "data sheet" entry. Its data sheet describes what a component does, and equally important, it gives constraints that allow the system designer to decide if the component is "good enough" for the application.

A (software) component (or building block) is a unit with defined function, behaviour, interface (data, time), environment, required resources (cpu-IEPS (number of inscriptions to be executed each second), real time (time points to execute), memory, devices).

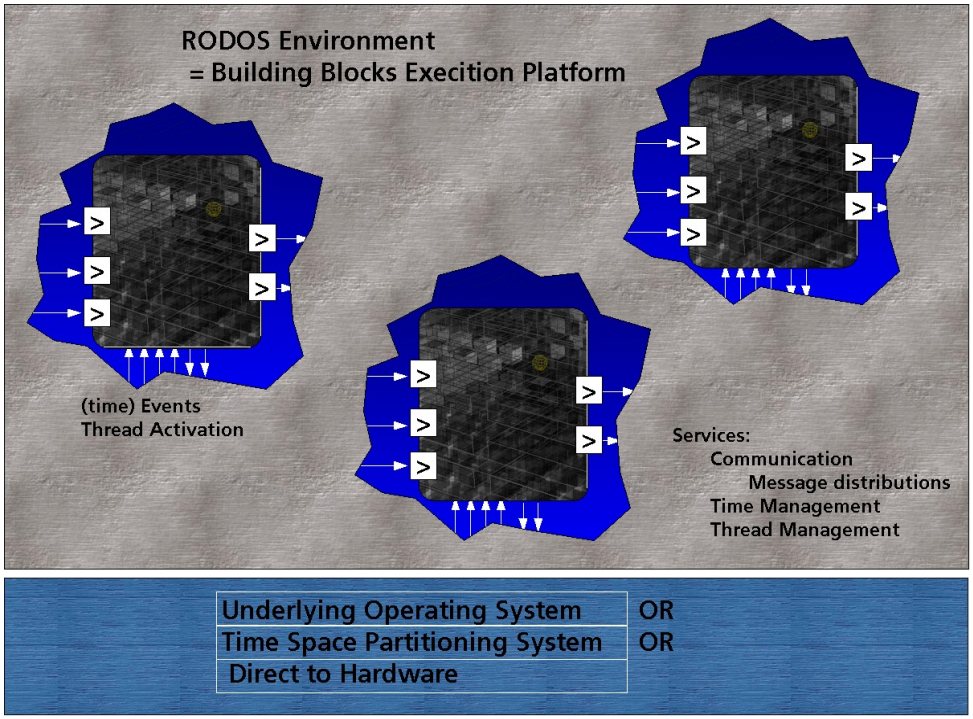



*A Component seen from outside*

A very important aspect of components and the corresponding building block execution platform is the capacity to build complex systems by composition: plugging (relative) simple components together. In such a way the final application will be a network of cooperating building blocks (or components). Our paradigm is to support software design methods for composability instead of just objects and connections.

## 1.2 The Building Blocks Execution Platform

Components provide services to other components and require some other services from other components to be able to execute. We compose complex systems by interconnecting single components via the services they provide and the ones they require. The composition does not need to know internal implementation properties of the components, just the interfaces (services: data and timings).



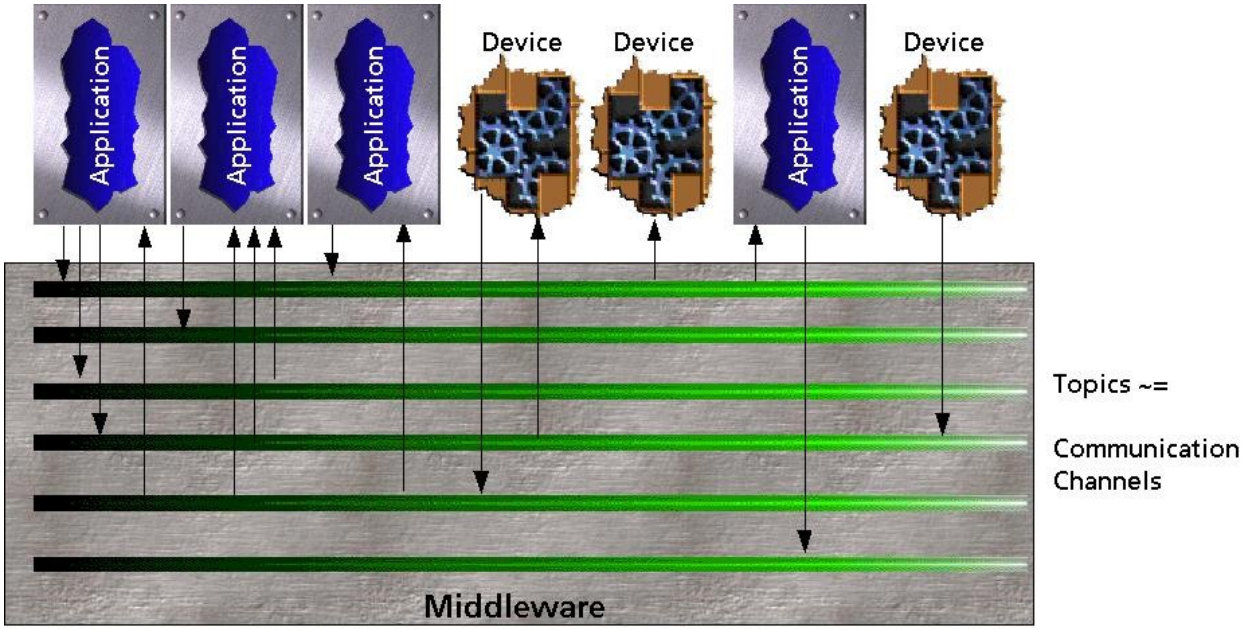

*BB Execution platform*

We use the same communication protocol for software components, for the network and for (our own) hardware devices. All communications in the system are based on the publisher/subscriber protocol (in software and in hardware). Publishers make messages public under a given topic. Subscribers (zero, one or more) to a given topic get all messages which are published under this topic. To establish a transfer path, both the publisher and the subscriber must share the same topic. A topic is represented by a topic ID and a data type.

The services are distributed in the (software/hardware) network from producers (publishers) to consumers (subscribers). This does not depend upon if the services are produced by software components or by hardware components. The same applies to the consumer of services.

This gives us very high flexibility and users do not have to differentiate between local/remote communication and between any combination of software/hardware/device communication.

Communication relationships can be very dynamic. Units may disappear or appear, tasks may be migrated, activated or deactivated at any time. The position of applications can even change (migration) at runtime, without requiring any explicit reaction of the other involved applications. There are no fixed communication paths. Each data transfer is resolved just in time using the registered communication topics.




*Topics in the Middleware*

Software applications may access an arbitrary number of topics, both as subscriber and as publisher. Devices may be attached to a computing node. In this case the device interface (IO) shall be encapsulated into a component which communicates by publisher/subscriber methods attached to the local middleware.

# 2. Creating Components (Including IO Drivers)

In RODOS there is **no** thing like a central element (a main program) which has to instantiate and reference all other (or a tree root of) elements which shall be linked and used. In RODOS, at linking time, the user says which components (eg. Applications, Drivers, Services) shall be included in the boot image (or executable .elf program). Usually components have only references to the middleware and to RODOS services and **no** other linkable connections to the world outside its walls and no other component may access (or know) internal objects of it.

RODOS provides classes like Initiator, Application, Thread, Event, Topic, Subscriber, etc. Which shall be declared as static elements. The constructors of objects from these classes (or their subclasses) register themselves automatically (no extra action from user is required) in the corresponding lists. RODOS implements a discovery protocol for each of these classes. After the underlying system has called all constructors (from such static objects) and they have registered themselves, RODOS goes through all these lists and interconnects elements into its network. After this, applications may use the application discovery protocol to see if other application were loaded or not, in the running boot image. But normally they do not need to know it.

We recommend that each component (including IO Drivers) shall have **one** object from a subclass of Application of from Initiator (preferable application). Your Application::init() or Initiator::init() shall initialize your data structures and in the case of IO drivers it shall initialize the corresponding hardware (registers).

Now please refer to the Middleware tutorials.

Define the data types (classes) and topics to communicate with other components, as well as to get data as to provide data. It could be, that topics and classes for your interface are already defined by other components to which you shall communicate.

Normally each component shall provide at least one subscriber to accept requests or data (maybe there could be components which do not need any inputs form other components, they do not need subscribers).

To distribute data you produce, publish it using a predefined topic.
If you shall propagate interrupts or events, use the Event::propagate() method.

If you need it, your component may have one or more threads.

Any component may be loaded or not. As long as the interfaces are the same, it is possible to load different versions of a service (implemented in components) without modifying the rest of the system. For example, an IO driver can access directly the hardware, or if the hardware is not available this service can be simulated by another component, or the interface may be completely ignored if no loaded component provide the corresponding services.

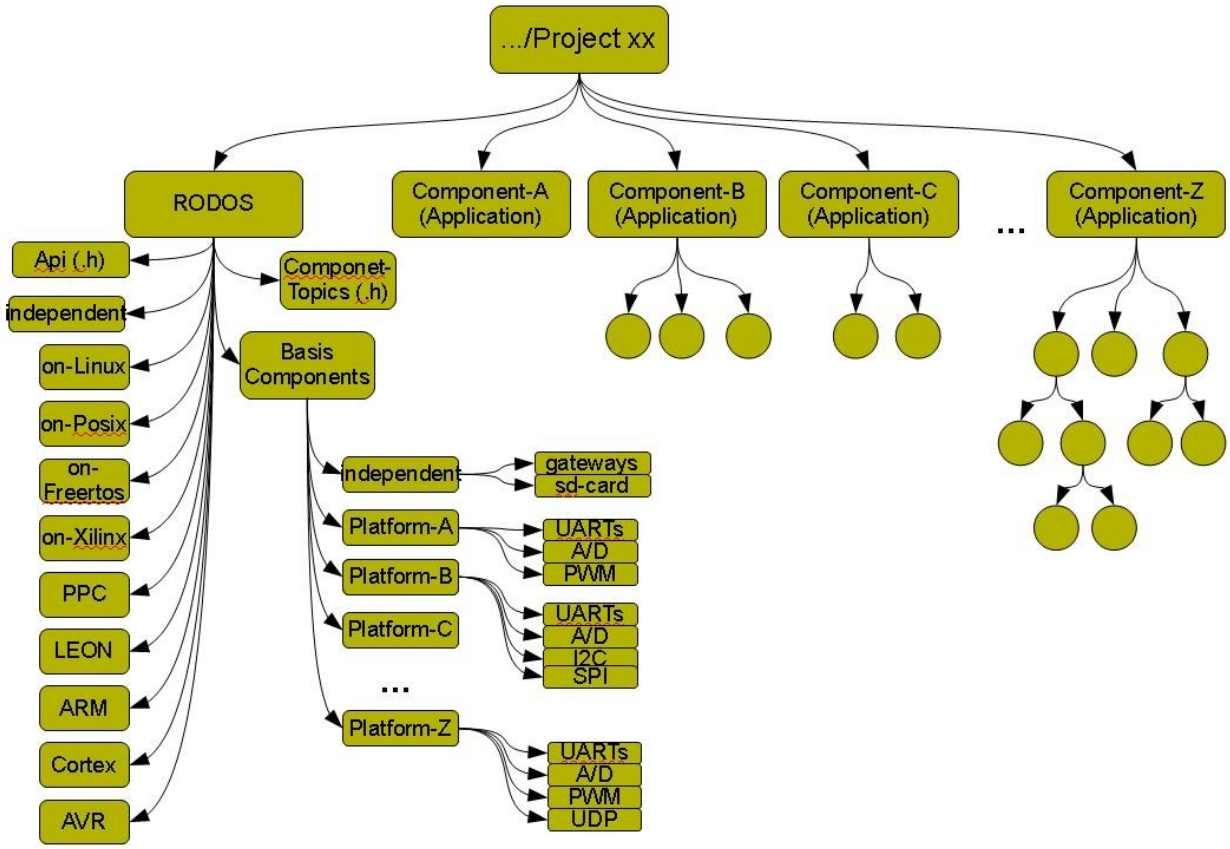
IO Devices attached to any node in the network may be accessed by any components in any node, because the IO Driver is a component like any other.

# 3. File structure and boot images

We support X86, PPC, LEON, ARM, Cortex and AVR CPUS. But for each one of these families there are many different configurations of Periphery. For example for Cortex, we know about 20 different Chips with different periphery and they are not compatible, only the CPU core (Cortex) is almost the same. RODOS provides some Basic components, very few are hardware independent, all other have to be implemented for each micro-controller architecture (and for each CPU Type).

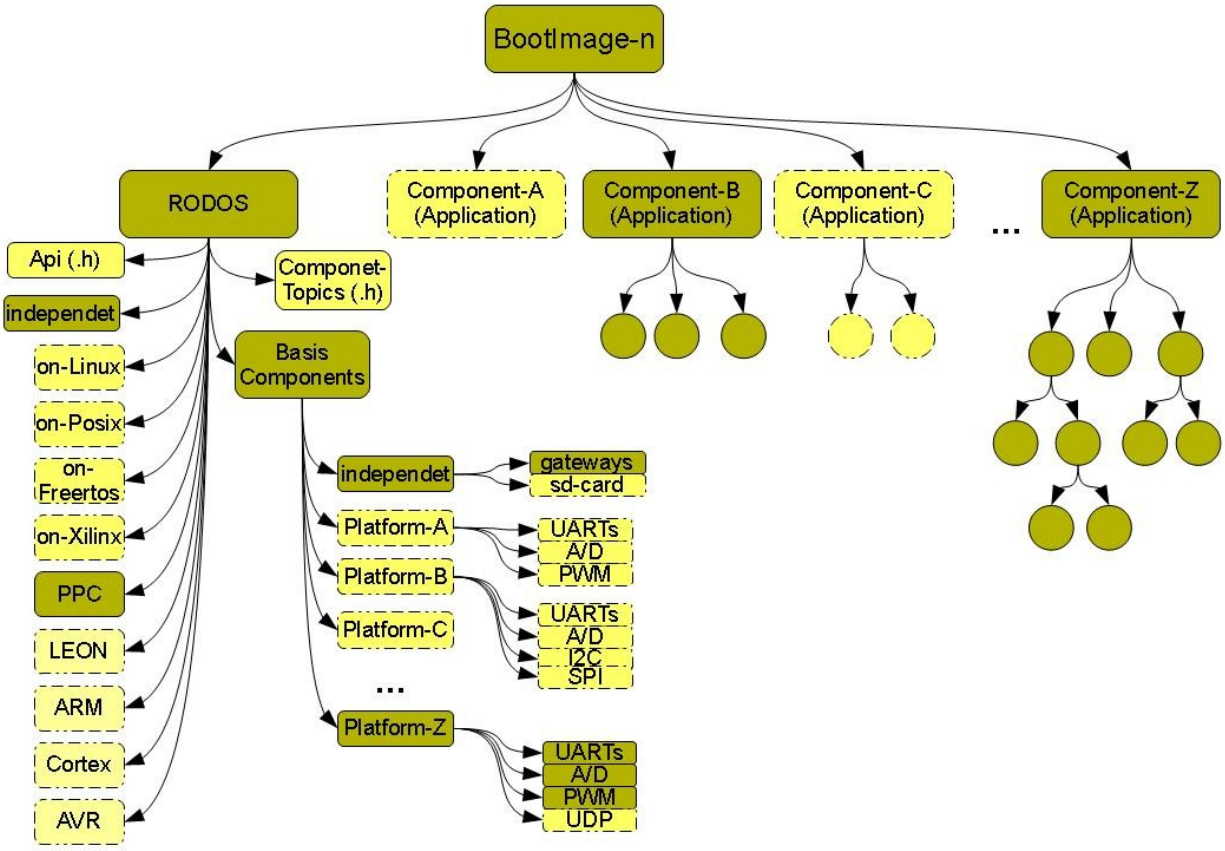
We recommend a directory for each Microcontroler type (Chip) and there please write the implementation of each IO Driver (eg, A/D, PWM, UARTs, I2c, SPI, etc) in a single .cpp file. To have

many and common .h files is not a problem. Please instantiate the topic Object inside of the file which handles this topics. This will link the driver automatically when any application uses the associated topic.




*recommended File Structure*

When linking we can select which applications shall be used and which RODOS implementation.




*Structure of a Boot Image*