Architectures

1. On Systems Design
2. Building Blocks
3. Time-Triggered System
4. COTS System

On Systems Design

1. Complexity
2. Requirements Engineering
3. Decomposition of a System

Complexity

In computer system design, the most important goal is controlling the complexity of the solution by introducing **structure**.

This introduction of a structure restricts the design phase and has a negative impact on the performance of the system. In the context of real-time systems, these performance penalties must be carefully evaluated.

In every project, there is an ongoing conflict between what is desired and what can be done within the technical and economic constraints.

A good **understanding** and documentation of these technical and economic **constraints** reduces the design space, and helps to avoid exploring unrealistic design alternatives.

Structuring

Two kinds of structuring a computer system can be distinguished to reduce the system complexity: horizontal and vertical structuring.

Horizontal structuring (or layering) is related to the process of stepwise abstraction (e.g. APIs). **Vertical structuring** is related to the process of partitioning a large system into a number of nearly independent subsystems.

While in central computer systems, layering is the only effective technique to handle complexity, the designer of a distributed computer system can take advantage of both techniques (cluster – partitioning, node – layering).

The major advantage of partitioning over layering is that the abstractions of partitioned systems also hold in case of failures. In layered systems, it is very difficult to define clean error-containment regions.

Legacy Systems

There are only very few large projects that can start “from scratch” with complete freedom in the design of the architecture and the selection of software and hardware.

Usually, developers are forced to “reuse” hardware and software from former projects. Most projects are extensions or **redesigns** of existing systems, the legacy systems.

Furthermore, there is a strong tendency to use “**COTS**” (Commercial off the Shelf) components to reduce both development time and the costs.

The integration of legacy systems is very challenging and requires clear **interfacing** or **encapsulation** of legacy parts.

Design Constraints

The **dependability** constraints of the application are often design drivers. A precise specification of the minimal dependability requirements helps to reduce the design space, and guides the designer in finding acceptable technical solutions.

The minimum **performance** criteria establishes a borderline between what constitutes success and what constitutes failure. A precise specification of the minimum performance, both in the value domain and in the temporal domain, is necessary for the design of a fault-tolerant system architecture that does not demand excessive resources.

A good understanding of the **economics** of the application domain is essential to achieve a proper system solution.

Composability

Large systems are often built by integrating a set of well- specified and tested subsystems.

The challenge is to link the subsystems such that their established properties at the subsystem level do hold on the system level.

An architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property once the property has been established at the subsystem level. Examples of such properties are **timeliness** or **testability**. In a composable architecture, the system properties follow from the subsystem properties.

The **communication system** has a central role in determining the composability of a distributed system.

Decomposition of a System

After the essential **requirements** have been captured and documented, the most crucial phase of the life cycle, the design of the system structure, is reached.

Stable intermediate forms are encapsulated by small and stable interfaces that restrict the interactions among the subsystems. In the context of distributed real-time systems, a **node** with autonomous temporal control can be considered a stable intermediate form. The specification of the interface between the nodes and the communication system, the CNI, is thus of critical importance.

The allocation of **functions to nodes** must be guided by the desire to build functional units (nodes) with a high inner connectivity and small external interfaces. It can be expected that there will be misfits, that some requirements cannot be accommodated in any sensible way.

Hierarchies of systems

A system shall keep high complexities inside of subsystems (clusters) with simple services at the interfaces of the subsystems. Clusters are composed in hierarchies to form the overall system.

Criteria for the formation subsystems are:

- locality: nodes shall be in proximity
- timing: nodes shall have similar pace
- fault-tolerance: nodes with different safety level shall be separated
- communication technology cannot be switched within subsystem

Subsystems are coupled via **gateways**.

Gateways

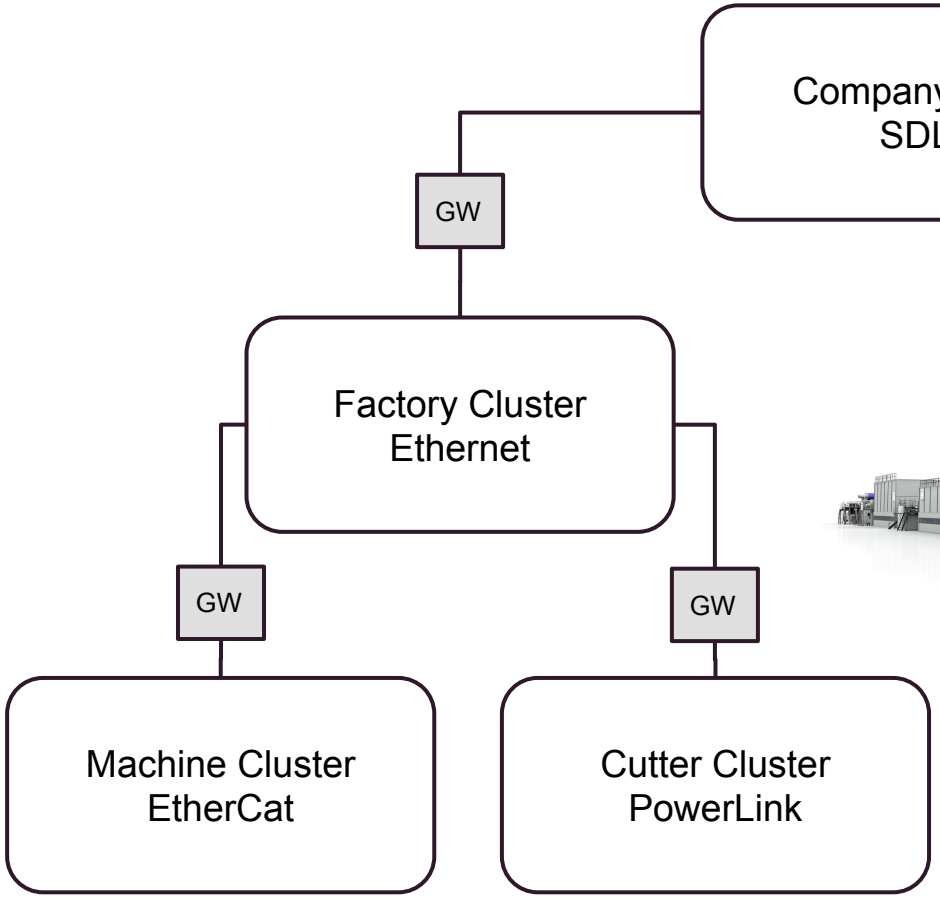
Important tasks of gateways result from the fact that they connect subsystems with different properties.

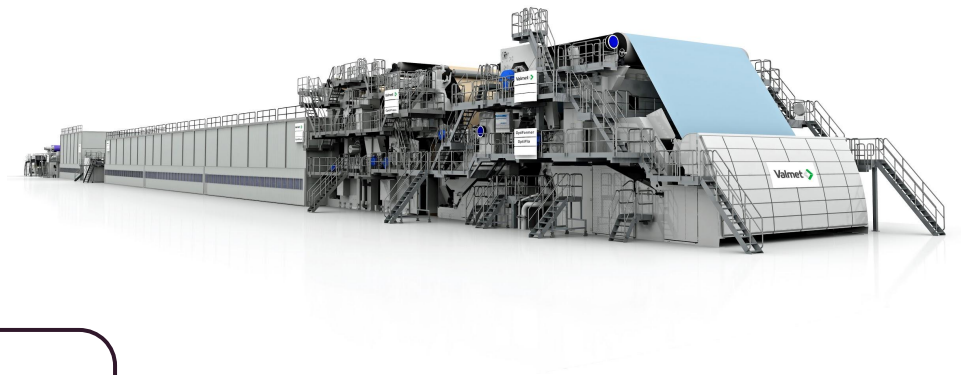
Typical tasks to consider are:

- pace: change pace of messages (aggregation, filtering, …)
- flow control: crossing flow control regions
- safety: support different safety levels on one node
- technologies: transfer data between different communication systems
- syntax: change data syntax

While fulfilling those tasks, a gateway still needs to work as a gateway in order to keep **interferences** between connected subsystems low.

Example: Paper mill





Requirements Engineering

The analysis and understanding of a large problem is never complete and there are always good arguments for asking more questions concerning the requirements before starting with the real design work.

A special problem arises from the fact that during the design process, new requirements have to be established in relation to the introduction of certain technologies (e.g. a selected CPU has an own set of requirements for safe operation) and design elements used (e.g. a fault-tolerant network has own requirements for correct use).

Research shows that many software triggered failures of systems result from **missing or incomplete requirements**. In order to avoid missing requirements use **requirement templates** and review requirements of **similar projects**.

Requirements for DRS

The following list gives an incomplete example of relevant requirements for a distributed real-time system:

- Structure of all components (nodes, networks, gateways, IO)
- Interfaces (structure, timing, control mechanisms)
- Message and data elements (semantics, coding, size)
- Timing of actions (WCET, min period, max period)
- Resource requirements (memory, CPU, IO)
- Fault-Tolerance requirements (fault hypothesis)
- Performance requirements (load hypothesis)
- Failure modes of nodes
- Error detection coverage (numbers and mechanisms)

Project Standards

Beside the requirements, several other aspects have to be defined for a working project. The communication between the client and the designer, as well as within the design team, can only be consistent if all concerned parties agree to a common language:

- Information representation: e.g., m/inches
- Naming conventions, abbreviations, glossary
- Documentation: Well-structured project documents
- Development Tools: Selected and frozen

Result of Architecture Design

At the end of the architecture design phase, a document that describes the computer system architecture must exist. An architecture design document should contain at least

- the decomposition of the system into clusters
- the specification of data semantics and timing at the gateway interfaces (e. g., to legacy systems)
- the decomposition of each cluster into nodes
- the function of each node
- a high-level specification of the I/O-interfaces
- a precise specification of all messages
- a detailed plan of the bus schedules
- a description of data transformation algorithms at each node
- an analysis of the dependability requirements and how these requirements are addressed at the cluster level

Detailed Design and Implementation

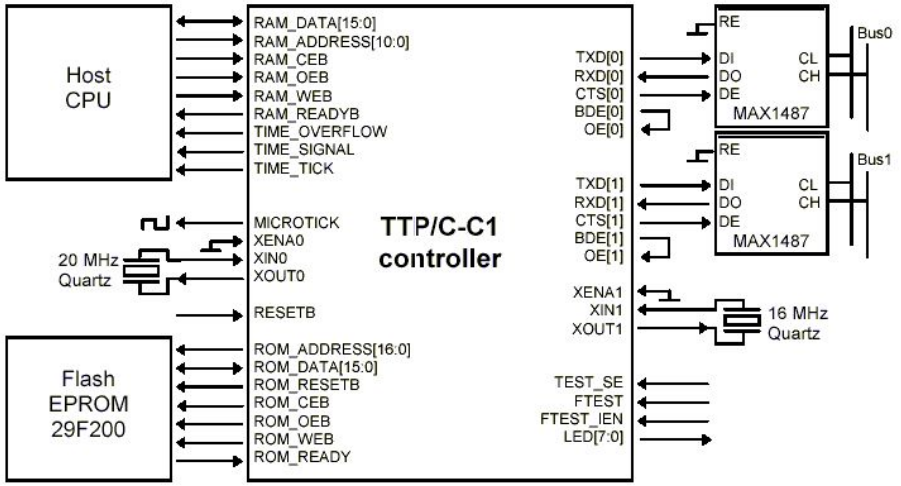
At the end of the architectural design phase, the message interfaces among the nodes within a cluster are established and stable. The design effort can be broken down into a set of loosely-related concurrent activities, each one focusing on the design, implementation, and testing of an individual node.

The activities include, e.g., the definition of I/O-interfaces, task development, and task scheduling.

A very important step in this phase is the evaluation of the WCET and schedulability of the task set.

Time-Triggered System

Example evaluation of TTP/C



Network TTP/C

		TTP/C
Bandwidth, MTU		2-5 Mbit/s twisted pair, 240B
Redundancy	***multi channel***	*multi channel* ✓
Failure mode	*depends on FT architecture*	*fail silent in time domain bus guardian*
Dependability values	*depends on FT architecture*	*?*
Flow control	***implicit flow control***	*implicit flow control* ✓
Protocol latency	***known, bound and low jitter***	*known, bound and low jitter* ✓

Property

required for hard real-time system

TTP/C

CNI TTP/C

		TTP/C
Bandwidth, MTU		?
Temporal firewall	***yes***	***yes***
Failure mode	*depends on FT model*	
State messages	***yes***	***yes***
Flow control	***yes***	***yes***
Permanence/action delay	***yes***	***no***
Idempotency	***yes***	***yes***
Mode	***multicast***	***multicast***

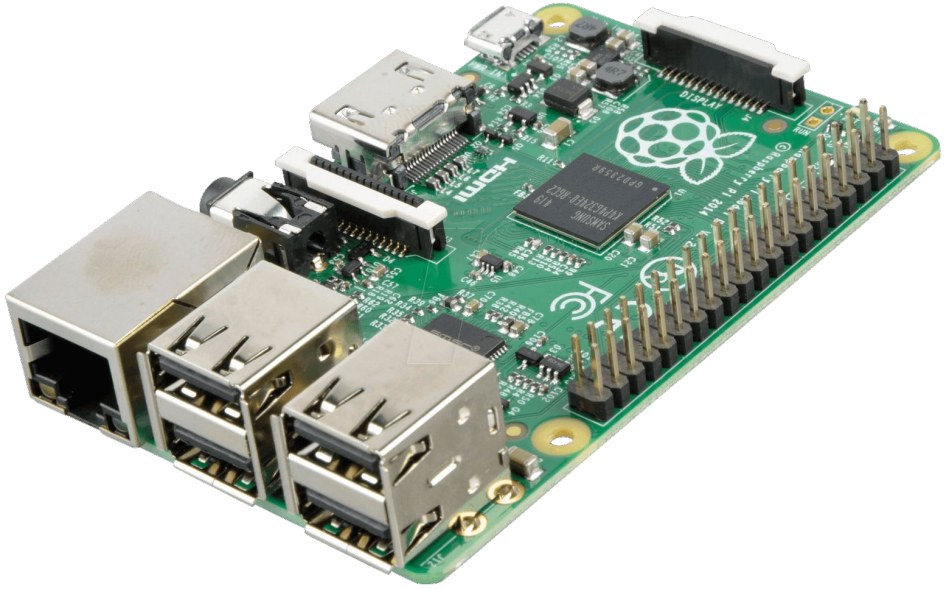
OS/CNI TTP/C

		TTP/C
Membership service	yes, no	***yes, bound delay***
Message ordering	yes, no	***yes***
Agreement protocols	yes, no	*no*
Atomic multicast	yes, no	*atomic broadcast partly implemented*
Data access	no protection semaphore non blocking write	***non blocking write***



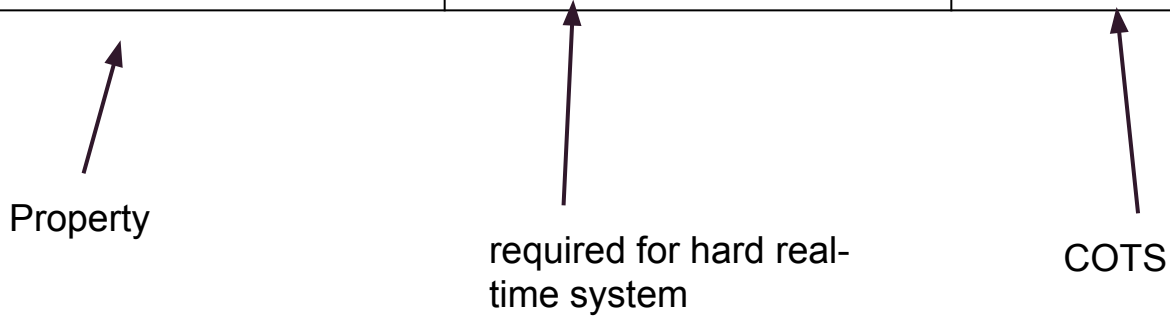
COTS System

Example evaluation of a COTS system with UDP/IP on Linux board



Network COTS

		COTS
Bandwidth, MTU		MBit - GBit
Redundancy	***multi channel***	*single channel*
Failure mode	*depends on FT architecture*	*no fail-silence*
Dependability values	*depends on FT architecture*	
Flow control	***implicit flow control***	*no flow control*
Protocol latency	***known, bound and low jitter***	*bound but high jitter*



CNI COTS

		COTS
Bandwidth, MTU		
Temporal firewall	***yes***	*no*
Failure mode	*depends on FT model*	*fail-inconsistent*
State messages	***yes***	*yes*
Flow control	***yes***	*no*
Permanence/action delay	***yes***	*no*
Idempotency	***yes***	*no*
Mode	***multicast***	*multicast*

OS/CNI COTS

		COTS
Membership service	yes, no	*no*
Message ordering	yes, no	*no*
Agreement protocols	yes, no	*no*
Atomic multicast	yes, no	*no*
Data access	no protection semaphore non blocking write	*semaphore*

Conclusion

The missing service elements have to be implemented on-top of the COTS system.

There are software libraries and hardware extensions available to get nearer to the goal of building a distributed hard real-time system from COTS components. However, those solutions are usually tailored for highly available server clusters and not for hard real-time systems.

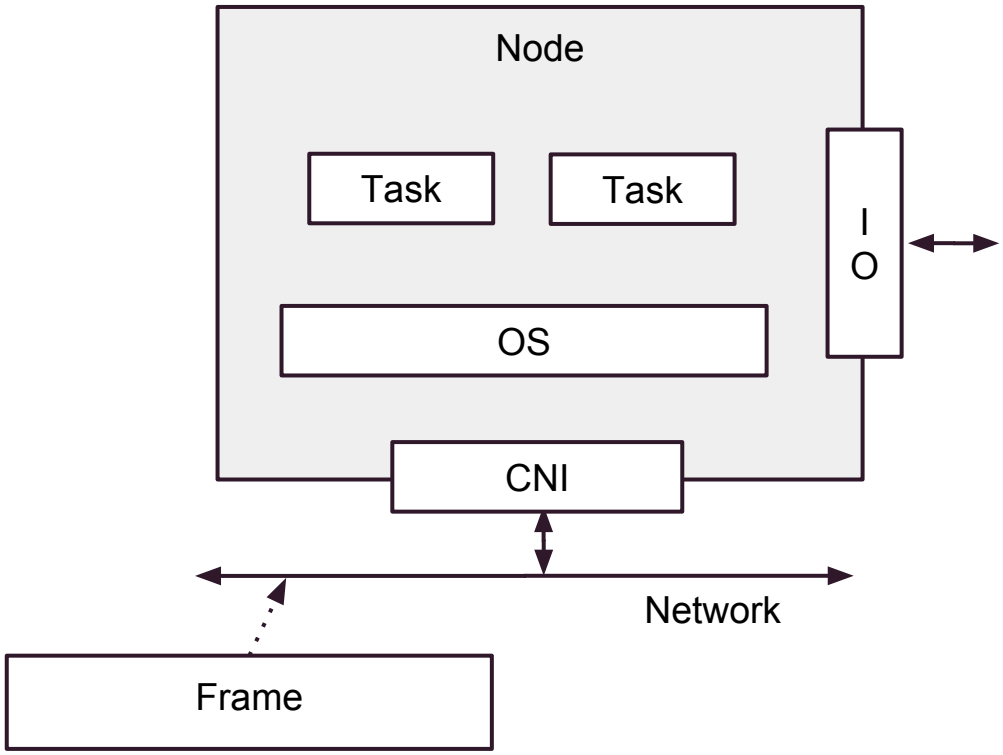
Sources

[1] Real-Time Systems: Design Principles for Distributed Embedded Applications, Hermann Kopetz, 1997, Springer Verlag

Building Blocks

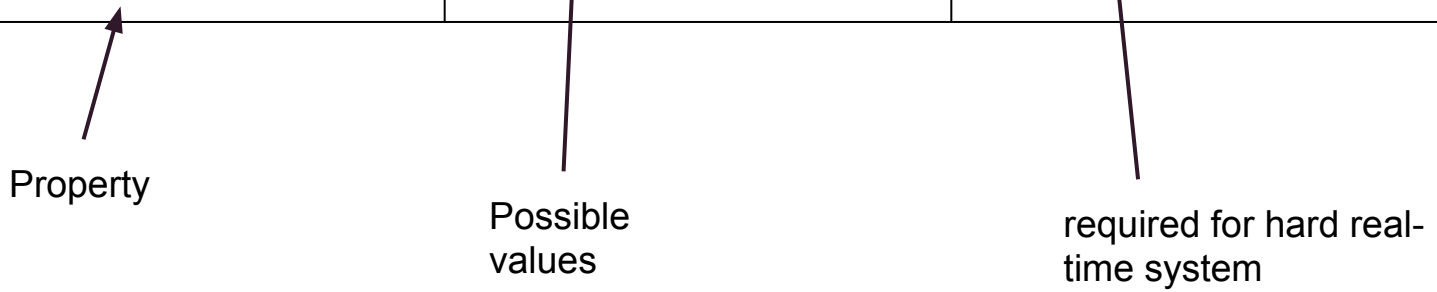
1. Node
2. Network
3. CNI
4. OS
5. Tasks
6. Frame
7. Data elements
8. IO

Overview



Node

CPU, RAM, ROM		
Failure mode	fail-silent, fail-inconsistent	*depends on FT architecture*
Dependability values	FIT, 99,999% availability	*depends on safety level*
Error detection coverage	%	*depends on safety level*
IO request limitation	yes no	***yes***



Communication System

Different alternatives are available for the realization of a communication service

- single channel systems, or
- multiple channel systems.

Communication reliability can be increased by

- message retransmission in case of a failure
- replicate messages to mask loss of messages
- replicate channels to tolerate loss of a channel

The reliability of the communication system in relation to the reliability of the nodes plays an important role for error handling.

Network

		required for hard real-time system
Bandwidth, MTU	kB/s, Bytes	
Redundancy	single channel multi channel	***multi channel***
Failure mode	fail silent, fail-inconsistent	*depends on FT architecture*
Dependability values	failure rate	*depends on FT architecture*
Flow control	explicit flow control implicit flow control no flow control	***implicit flow control***
Protocol latency	bound, unbound	***known and bound***

Communication Network Interface

The purpose of the real-time communication system is to transport messages from the CNI of the sender node to the CNI of the receiver node within a predictable time interval, with a small latency, and with high reliability.

Furthermore, messages must not be corrupted, faults must be handled correctly, and protocol logic and structure of the communication network are hidden behind the CNI.

CNI

		required for hard real-time system
Bandwidth, MTU		
Temporal firewall	yes, no	***yes***
Failure mode	fail-silent fail-inconsistent	*depends on FT model*
State messages	yes, no	***yes***
Flow control	yes, no	***yes***
Permanence/action delay	yes, no	***yes***
Idempotency	yes, no	***yes***
Mode	point-to-point multicast broadcast	***multicast***

Control Strategy

The decision when a message must be sent can reside either within the sphere of control of the host computer (external control) or within the sphere of control of the communication system (autonomous control).

In case of external control, a “send” command in the host computer causes the transfer and a control signal (interrupt) initiates a “receive” command in the receiving host computer.

In case of autonomous control, the communication system decides autonomously when to send the next message, and when to deliver the message at the CNI of the receiver.

Autonomous control is normally time-triggered.

If the control of the communication system is autonomous, no control signals must cross the CNI. In this case the CNI is used strictly for data sharing.

OS

		required for hard real-time system
Scheduling	static scheduling table priority driven no control	***static scheduling table***
Clock synchronization	yes, no	***yes***
Control flow monitor	yes, no	*yes*
Watchdog	yes, no	*yes*
WCAO	known, unknown	***known and bound***
Memory protection	yes, no	*yes*
Time protection	yes, no	*yes*
h-State exchange	yes, no	***yes***
Reintegration	yes, no	***yes***

OS

		required for hard real-time system
interprocess communication	yes no	***yes***
synchronization of tasks	no semaphores priority inheritance priority ceiling	***priority ceiling***
Interrupt rate monitoring	yes, no	***yes***
Double execution of tasks	yes, no	*yes*
Monitored exception handling		*yes*
Mode changes	yes, no	*yes*
Stage management	yes, no	*yes*
Redundancy management	yes, no	*yes*

OS/CNI

		required for hard real-time system
Membership service	yes, no	***yes, bound delay***
Message ordering	yes, no	***yes, bound delay***
Agreement protocols	yes, no	*yes, bound delay*
Atomic multicast	yes, no	***yes, bound delay***
Data access	no protection semaphore non blocking write	***non blocking write***

Tasks

		required for hard real-time system
Periodic	yes, no	***yes***
WCET	known, unknown	***yes***
Resource requirements	CPU, RAM, ROM	
Trigger	time-triggered event-triggered	***time-triggered***
Phase sensitive scheduling	yes, no	*yes*
h-state recovery	yes, no	***yes***

Frame

		required for hard real-time system
error detection code	yes, no	***yes***
MTU size	Bytes	
multiplexing	yes, no	
sender ID	yes, no	***yes***
slot ID	yes, no	***yes***
transmission time	yes, no	***yes***

IO

		required for hard real-time system
transformation	raw values calibrated values agreed values	*yes*
sampling rate	kHz	kHz
memory element	yes, no	*yes*
timestamp	yes, no	*yes*
interrupt	raw rate protected	***rate protected***