Interface Design for Real-Time Smart Transducer Networks – Examining COSMIC, LIN, and TTP/A as Case Study

Wilfried Elmenreich
Institute of Computer Engineering
Technische Universität Wien
Austria
wil@vmars.tuwien.ac.at

Hubert Piontek
Dep. of Embedded Systems/RT Systems
Universität Ulm
Germany
hubert.piontek@uni-ulm.de

Jörg Kaiser
Institut für Verteilte Systeme
Universität Magdeburg
Germany
kaiser@ivs.cs.uni-magdeburg.de

Abstract *– This paper analyzes and discusses the interface models of the three real-time smart transducer networks COSMIC, LIN, and TTP/A.*

The COSMIC architecture follows a publish/subscribe model, where the producing smart devices broadcast their event data on the basis of a push paradigm. Subscribers receive data in form of a message-based interface.

LIN follows a strict pull principle where each message from a device node is requested by a respective message from a master. Applications have a message-based interface in order to receive and transmit data.

The nodes in a TTP/A network derive its sending instants from predefined instants in time. TTP/A maps communicated data into an Interface File System (IFS) that forms a distributed shared memory.

1 Introduction

The availability of cheap microcontrollers and network solutions has enabled distributed architectures with networked smart transducer devices. The hardware for a smart transducer consists of a physical sensor or actuator, a microcontroller or FPGA with on-chip memory and analog I/O, and a network interface. The software in the microcontroller contains transducer-specific routines, like de-noising, linearization, and feature extraction, and a communication protocol establishing a standardized interface to the smart transducer. This interface provides access to the transducer values, like sensor measurements or actuator set values, as well as configuration and management data (calibration data, error logs, etc.). In order to support an automatic (plug-and-play) or semi-automatic configuration, a smart transducer may also host an electronic description, i. e., a machine-readable datasheet describing its features and interfaces.

Real-time and bandwidth requirements make the design of an interface to a smart transducer a difficult task. This paper addresses the specific requirements and issues of interface design for smart transducers and examines three architectures for real-time smart transducer networks, i. e., the Cooperating SMart devices (COSMIC) middleware, the Local Interconnect Network (LIN), and the Time-Triggered Communication Protocol for SAE class A applications (TTP/A).

The paper is structured as follows: Section 2 states the basic concepts and requirements for smart transducer interface design. Section 3 describes communication model, interface design and node description approach for the COSMIC middleware. Accordingly, Section 4 and Section 5 examine the LIN and TTP/A approach. Section 6 elaborates common features and differences of the three approaches. The paper is concluded in Section 7.

2 Interface Concepts

A smart transducer interface can be decomposed into several sub-interfaces with different purposes and requirements [1]: The *real-time (RT) service interface* is required for transmitting transducer data such as measurements or set values. The *configuration and planning (CP) interface* provides access to protocol-specific functions like new node identification, obtaining electronic datasheets, and configuration of communication schedules. The CP interface is not time-critical. The *diagnostics and management (DM) interface* is used for accessing sensor and actuator-specific functions like monitoring, calibration, etc. The DM interface is not time-critical, however, some monitoring applications require timestamping.

In the following we discuss real-time requirements, flow control and interaction design patterns which are mostly relevant for the RT service. The DM interface has different requirements and should not interfere with the RT service.

2.1 Real-Time Requirements

There are different kinds of real-time requirements for a distributed system of smart transducers. As a common feature, there is always a *deadline* that specifies a point in time when a specific action has to be completed.

Performing some action locally with respect to real time, like generating a particular Pulse Width Modulation (PWM) signal or making a measurement every 100 ms is relatively easy to achieve if drift of the local clock source is sufficiently low to provide a useful time base.

Timestamping events can be used to temporally relate measurements to each other. In order to create timestamps with a global validity, a synchronized global time is required among the participating nodes. In most cases, clock synchronization has to be done periodically in order to compensate for the drift of the local clocks. Once a global time is established, timestamping does not pose real-time requirements on the communication system, since timestamped events can be locally stored.

Bounded maximum reaction time requires the communication system to deliver messages within a specified time interval. Standard feedback control algorithms also require low message jitter in order to work correctly.

Globally synchronized actions require the synchronized generation of action triggers in different nodes. This can be achieved by a multi-cast message or assigning actions to an instant on the globally synchronized time scale.

Moreover, a real-time requirement can be *hard*, i. e., deadlines must be held under all circumstances or *soft*, the system is still of use if deadlines are violated infrequently.

Many architectures implement a subset of the described features or provide different features with hard or soft real-time behavior. For example the LAAS architecture [2] for component-based mobile robots specifies local hard-real-time such as a locally closed control loop or the instrumentation of an ultrasonic sensor, while at higher levels, e. g., for globally synchronized actions it provides only soft real-time behavior.

2.2 Models of Flow Control

Communication between subsystems takes place in the *time domain* and the *value domain*. In the value domain, the message data is exchanged, while in the time domain *control information* is transmitted [3].

The communication partner that generates the control information influences the *temporal control flow* of the other communication partner(s). If a communication is controlled by the sender's request we speak of a *push* model, if communication is requested by the receiver, we speak of a *pull* model.

For explanation, let us assume that two or more subsystems need to exchange data over a network. Further, without restrictions to generality, we assume message data to be transmitted from a *producer* to one or more *consumers*. Different from the very popular client-server communication pattern it is necessary to support a one-to-many or many-to-many communication pattern in smart transducer networks because in the common case, the sensor readings of a transducer is needed in more than one place in control applications. Therefore, all of considered networks support this property, however, in different ways. In order to transfer data between the subsystems, they must agree on the mechanism to use and the direction of the transfer.

Figure 1 a) shows the *push* method. The producer is empowered to generate and send its message spontaneously at any time and is therefore independent of the consumer. This loose coupling enables independence between the supplier and consumers of information [4], [5],

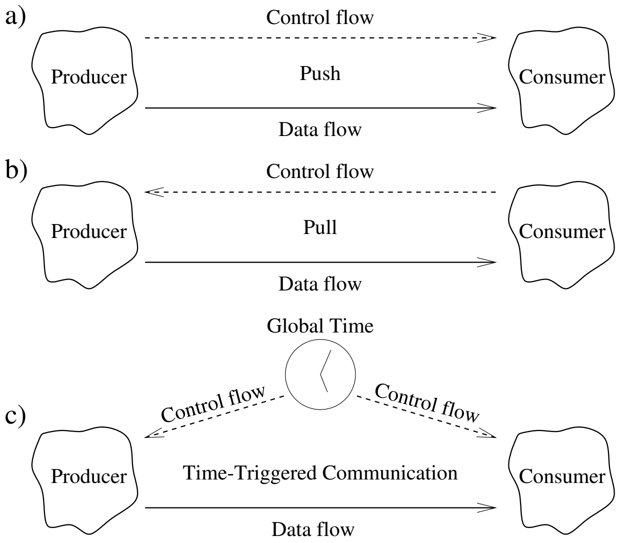


Figure 1. Push-, Pull-, and Time-Triggered Communication

but makes it difficult to enforce temporal predictability in a purely event driven model. Without the option to enforce further temporal constraints, the communication system and the receiving push consumer have to be prepared for data messages at any time, which may result in high resource costs and difficult scheduling. Popular "push" mechanisms are messages, interrupts, or writing to a memory element [6]. The push-style communication is the basic mechanism of event-triggered systems.

In the *pull* model depicted in Figure 1 b) the consumer governs the flow control. Whenever the consumer wants to access the message information, the producer has to respond to the consumer's request.. This facilitates the task for the pull consumer, but the pull supplier (producer) must be watchful for incoming data requests. Popular "pull" mechanisms are polling or reading from a memory element [6]. Pull-style communication is the basic mechanism of client-server systems.

Figure 1 c) depicts a communication model where the flow control is derived from an external trigger. This can be another physical system or the derivation of the triggers from the progress of physical time. In the latter case, the control signals are known *a priori*, which requires predefined scheduling and error detection in the control domain.

2.3 Interaction Design Patterns

We distinguish three basic interaction design patterns for network communication.

In a *master/slave relationship*, at any time one node is considered the *master* while the other nodes are considered to be *slaves*. The master is able to issue synchronization events or to start communications. All slave nodes depend on one master, while the master is independent of a particular slave.

In a *client/server relationship*, a *client* issues a request to a *server*, which has to answer the request. The client and the server are thus tightly coupled via a pull model.

In a *publish/subscribe relationship*, a *publisher* generates data using the push model. A number of nodes may subscribe to a particular publisher but there is no control flow from subscriber to publisher. Depending on the implementation, a publisher may broadcast its data immediately, transmit its data to an intermediary broker, or transmit its data via point-to-point connections to a list of subscribers. Thus, the number of subscribers may influence the time it takes for a publisher to publish its data.

An architecture may hide the communication model by implementing a distributed shared memory on top of the communication. This way, an application uses the same interface to access data locally or remote. However, a memory interface does not transport control data, e. g., in order to launch a particular task upon reception of an event. Such functionality can either be achieved via polling, but that requires an adequate poll frequency and comes with a noticeable overhead [7] or solved via additional features like interrupt generation after update of a specific data field.

2.4 Diagnostics and Management

While the RT interface provides only access to a limited data set consisting of the actual needed transducer data, for debugging or monitoring purposes, additional data about the operation of the transducer is of interest.

Transmission of these data typically is not time-critical, but must not interfere with the RT service leading to an unwanted probe-effect [8]. Furthermore, monitored RT data should either have time stamps or it must be transmitted before a (typically soft) deadline.

2.5 Configuration and Planning

Large smart transducer systems require support by automatic setup facilities in order to keep up with the complexity of setting parameters correctly. Therefore, smart transducer system should be supported by a tool architecture enabling a plug-and-play-like integration of new nodes.

We consider different configuration and planning scenarios:

In the *replacement scenario*, a broken node is to be exchanged by a new one of the same type. Therefore, the new node has to be detected and a backup of the configuration of the broken node has to be uploaded. However, specific parameters like calibration data will have to be created anew.

In the *initial set-up scenario*, a set of nodes is configured in order to execute a particular communication. This action requires a system specification, and, in most cases, a human operator to perform tasks that cannot be solved automatically.

In the *extension scenario*, a distributed application is extended by extra nodes in order to improve the performance and possibilities of the system. In this case the system managing the configuration must have knowledge how to integrate new transducers in order to upgrade the system. An example of such an approach is outlined in [9].

For the purpose of node identification and documentation, a node is assigned a machine-readable description describing the node's features. Example for such descriptions are the Transducer Electronic Datasheets of IEEE 1451.2 [10] or the Device Profiles in CANopen [11].

3 COSMIC

3.1 COSMIC communication abstractions

COSMIC is middleware which is designed for small embedded systems, supporting heterogeneous networks and cross network communication. It provides a publish/-subscribe abstraction over different addressing and routing mechanisms as well as it considers different latency properties of the underlying networks. COSMIC provides typed event messages (EM) identified by an event UID which identifies the content of a message rather than a source or destination address. Further, EMs have attributes which define a temporal validity of the EM. It should be noted that the term "event" does not refer to a specific synchrony class but just denotes a typed message. As indicated previously, in a real–time embedded environment the pure push model creates problems because the consumers of the information must be ready to receive and process this information at any time. This may lead to situations where some of the messages are lost. Therefore, COSMIC introduces the notion of event channels (EC) which allow specifying temporal constraints and delivery guarantees of individual communication channels explicitly. COSMIC supports three event channel classes: A hard real-time event channel (HRTEC) offers delivery guarantees based on a time-triggered scheme. EMs pushed to a soft real-time event channel (SRTEC) are scheduled according to the earliest deadline first (EDF) algorithm. The respective deadline is determined by the temporal validity information in the attribute field of the EM. Because soft real-time EMs which have already missed their transmission deadline may cause further deadline misses of other soft real-time EMs, they are discarded and the respective local application is notified. The application then can decide about re-sending in a lower real-time class or just skip it. Finally, a non real-time event channel (NRTEC) disseminates events that have no timeliness requirements.

ECs are established prior to communication allowing the middleware to reserve the necessary resources and perform the binding to the underlying mechanisms of the communication network.

The COSMIC architecture is not bound to a particular network. An implementation based on Controller Area Network (CAN) [12] is described in the next section.

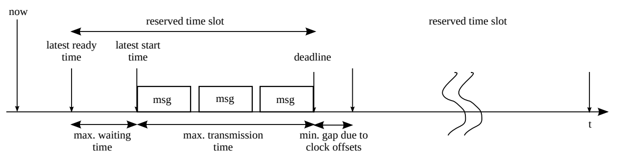


Figure 2. Structure of a time slot

3.2 COSMIC–on–CAN architecture

Implementing the event model requires to map the abstractions of that model (publisher, subscriber, event channel, event instance) to the elements provided by the infrastructure of the communication system. The respective functionality to perform these mapping in COSMIC is encapsulated in the Event Channel Handler which resides in every node. Given the constraints in bandwidth and in message length on CAN, the implementation of events and event channels has to exploit the underlying CAN mechanisms. To save the rare space in the message body [13] and to reduce the inherent CAN message overhead, significant information is also encoded via the CAN ID.

The implementation is based on the extended 29 bit CAN ID of the CAN 2.0 B specification. The 29-Bit CAN identifier (CAN-ID) is structured into three fields, i. e., an 8 bit priority field used to prioritize messages according to HRTEC, SRTEC, and NRTEC, a 7 bit node- ID ensuring unique identifiers and a 14-Bit event tag. The assignment of an event UID to an event tag is performed dynamically by the COSMIC middleware infrastructure. A description of this infrastructure and the respective binding protocol is described in [14].

3.3 Enforcing temporal constraints in COSMIC

HRTECs provide delivery guarantees and use reserved time slots in a Time Division Multiple Access (TDMA) scheme organized in periodic rounds. The intention of the reservation-based scheme is to avoid collisions by statically planning the transmission schedule. Hence, any conflict between HRTECs is avoided. COSMIC is implements clock synchronization based on the algorithm proposed in [15].

Because a CAN message cannot be preempted a non hard real-time message transmission may delay a hard real-time message by the maximum length of one CAN message in the worst case. Furthermore, transient transmission faults may increase the time needed to transmit a hard real-time message. Therefore, a hard real-time slot is extended according to Fig. 2. The protocol relies on the fault–handling mechanisms of the standard CAN which has an impact on the fault classes which we can handle. For a message with b bytes of data, the maximum length of the message including header and bit–stuffing is: $Length_{message} = 75 + \lfloor b \cdot 9.6 \rfloor$[1]. Under the assumption of f single transmission failures, the required minimum time–slot length is: $slot\ length = 2 \cdot t_{message} + (t_{message} + 18) \cdot f + 3 bittimes$. Assuming a single message failure of an 8 Byte message at 1 Mbit/sec (msg transmission time: 151μsec, fault detection and retransmission 18μsec) and a gap between the slots of 50μsec, approx. 1900 slots/sec can be allocated. If it is necessary to tolerate a permanent controller failure, this number drops down to an approximate number of 350 slots/sec. Compared to a maximum throughput of about 6500 maximum length messages per second, the number of possible HRT slots is low. However, these numbers refer to the number of *guaranteed* HRTECs not to the number of messages which actually can be sent. Unlike in pure time-triggered systems, the CAN priority mechanism can be used to transmit SRT or NRT messages in cases where a HRT message has been received a message successfully by all operational nodes[2]. Thus, time redundancy only costs bandwidth if faults really occur, which may be relatively rare compared to the overall traffic. The priority-based arbitration mechanism is also exploited to schedule SRTECs and NRTECs. HRT messages always reserve the highest priority. The relation between the priorities of HRT, SRT and NRT messages can be expressed by the relation: $P_{HRT} < P_{SRT} < P_{NRT}$ (a lower numerical value represents a higher priority). The assignment enforces that a message of a lower real–time class never will interfere with one of a higher class during bus arbitration. We assume the highest priority (0) for HRT messages and a small number of fixed low priorities for NRT messages. The remaining priority levels are available for scheduling SRT messages. They have to be mapped on a time scale to express the temporal distance of a deadline. The closer the deadline, the higher the priority. Mapping deadlines to priorities will cause the problem that static priorities cannot express the properties of a deadline, i.e. a point in time. A priority corresponding to a deadline can only reflect this deadline in a static set of messages. When time proceeds and new messages become ready, a fixed priority mechanism cannot implement the deadline order any more. It is necessary to increase the priorities of a message when time approaches the deadline, i.e. with decreasing laxity.

3.4 Device Descriptions

COSMIC devices are described in an XML-based language called CODES (COSMIC embedded DEvice Specification) [18]. The descriptions are structured into three parts. Part one, *General Information* contains the node's name, its type, its manufacturer, its unique identifier, its networking facilities, its supported event channel types, recycling information, and a clear text description of the device. This part also contains version information about the component. The second part contains all *event definitions*, i. e., the description of all events produced or consumed by the device.

[1]The factor 9.6 is because of the bit stuffing mechanism

[2]There are situations of inconsistent replicas and even inconsistent omissions (according to [16], inconsistent omissions occur with a probability in the order of 10^{-9}). Kaiser and Livani [17] describe a transparent mechanism to handle these situations.

For each event is described by a plain text tag and a unique identifier. The event's definition includes a list of attributes giving non-functional details about the event, e. g., the event's expiration time. Whenever an event is disseminated, it is sent as a compact message. This message's data structure is specified in the event definition. For each field in the data structure, its name, data type and byte order are included in the description. This information can be used by tools to automatically create decoder for a compact message. Fields representing a measurement are annotated by the corresponding physical dimension in a machine-readable format. Non-measurement fields are described by lists or state machines. Each field may contain also attributes, e. g., the valid data range. The last part contains the declaration of all *event channels* and their properties. Each event channel definition contains the subject UID linking it to the respective event definition, the class of event channel, the direction of the event channel as seen locally, and again a list of attributes, e. g., the channel's period.

While the greater part of the description contains static information, some elements are not suitable for integration into a static description document, e.g. the period of an event channel, which certainly will vary depending on the application. To overcome this problem, parameterization was introduced. Any non–static element can be marked as a parameter in the static description. The element's actual value is then defined and stored external to the static description. Parameters are stored in path–value–pairs, similar to well–known name–value–pairs. Instead of naming the parameter, it's XPath expression within the static description acts as the identifier. A scheme for mapping this structure down to a binary parameter storage scheme suitable for small devices exists. Whenever the description is used, the parameters are included beforehand. The query service (see below) is a suitable place that will handle this inclusion in running systems. Having each parameter's path expression eases the integration into the description document.

CODES descriptions play a central role for COSMIC components. The life of a COSMIC component starts with the description document created during the component's design phase. It is used during the following implementation phase to generate parts of the component's code [18]. Black–box tests of the component can be assisted, e.g. tests for timing behavior or testing the compliance of disseminated events with their description in terms of the data structure, value ranges, or precision. The descriptions are further useful throughout the component's life–cycle: During the integration phase into a larger system, a number of compatibility checks can be performed automatically. Schedules for the HRT communication can be derived from the respective set of descriptions. While the component is in use, the ready availability of its description forms the basis for dynamic use of formerly unknown components. Currently, this requires a priori knowledge or the interaction with a user. In the long run, the integration of semantic web technology is planned to enable true autonomous dynamic cooperation of components. Whenever a system is in need of maintenance, the availability of the descriptions is beneficial, too. They provide a quick overview of the system, i.e. what components are available, and how they are configured.

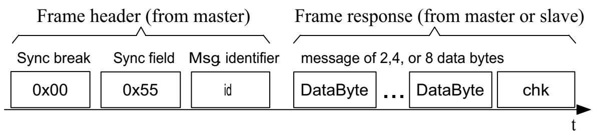
The descriptions are stored within the devices themselves. They can be retrieved and queried at run–time: On system start–up, and whenever a new component is added to a system, an automatic configuration is necessary for the component to be able to participate in communication. During this configuration, the components' descriptions and parameters are uploaded to the node running the event channel broker. This node also runs a query service which makes the descriptions accessible from outside the system. The parameters are included in the static part of the description, yielding a single document describing the current configuration of the components. Requests to the query service are given as XSLT transformations [19]. The transformations are applied to the CODES descriptions on the node running the query service, thus enabling even rather low–power nodes to make use of the query service. XSLT transformations represent a suitable technology not only for the query service, but throughout the different application areas of the CODES descriptions. They are e.g. also used for code generation.

4 LIN

4.1 System Architecture

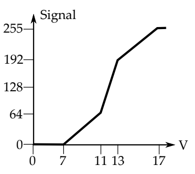
Each message in LIN is encapsulated in a single message cycle. The message cycle is initiated by the master and contains two parts, the frame header sent by the master and the frame response, which encompasses the actual message and a checksum field. The frame header contains a sync brake (allowing the slave to recognize the beginning of a new message), a sync field with a regular bit pattern for clock synchronization and an identifier field defining the content type and length of the frame response message. The identifier is encoded by 6 bit and 2 bits for protection. Figure 3 depicts the frame layout of a LIN message cycle.

The frame response contains up to 8 data bytes and a checksum byte. Since an addressed slave does not know *a priori* when it has to send a message, the response time of a slave is specified within a time window of 140% of the nominal length of the response frame. This gives the node some time to react on the master's message request, for example to perform a measurement on demand, but introduces a noticeable message jitter for the frame response.



Frame header (from master)
Frame response (from master or slave)
Sync break
Sync field
Msg. identifier
message of 2,4, or 8 data bytes
0x00
0x55
id
DataByte
...
DataByte
chk
t

Figure 3. LIN frame format



```
V_battery {
    logical_value, 0, "under voltage";
    physical_value, 1, 63, 0.0625, 7.0, "Volt";
    physical_value, 64, 191, 0.0104, 11.0, "Volt";
    physical_value, 192, 253, 0.0625, 13.0, "Volt";
    logical_value, 254, "over voltage";
    logical_value, 255, "invalid";
}
```

Figure 4. Example for a LIN signal definition

The interaction between master and slave is a plain pull mechanism, since the slaves only react on the frame header from the master. It is the master's task to issue the respective frame headers for each message according to a scheduling table. From a data-centric perspective, the communication is defined by messages that are subscribed by particular slaves for reception or transmission. The configuration of the network must ensure that each message has exactly one producer.

In 2003, LIN was enhanced by extra features leading to the LIN 2.0 specification. New features introduced in LIN 2.0 are an enhanced checksum, sporadic and event-triggered communication frames, improved network management (status, diagnostics) according to ISO 14230-3 / ISO 14229-1 standards, automatic baud rate detection, standardized LIN product ID for each node, and an updated configuration language to reflect the changes.

In addition to the unconditional frames (frames sent whenever scheduled according to the schedule table) provided by LIN 1.3, LIN 2.0 introduces *event-triggered frames* and *sporadic frames*.

Similar to unconditional frames, event-triggered frames begin with the master task transmitting a frame header. However, corresponding slave tasks only transmit their frame response if the corresponding signal has changed since the last transmission. Unlike unconditional frames, multiple slave tasks can provide the frame response to a single event-triggered frame, assuming that not all signals have actually changed. In the case of two or more slave tasks writing the same frame response, the master node has to detect the collision and resolve it by sequentially polling (i.e., sending unconditional frames) the involved slave nodes. Event-triggered frames were introduced to improve the handling of rare-event data changes by reducing the bus traffic overhead involved with sequential polling.

Sporadic frames follow a similar approach. They use a reserved slot in the scheduling table, however, the master task only generates a frame header when necessary, i. e., when involved signals have changed their values. As this single slot is usually shared by multiple sporadic frames (assuming that not all of them are sent simultaneously), conflicts can occur. These conflicts are resolved using a priority-based approach: frames with higher priority overrule those with lower priority.

In addition to signal-bearing messages, LIN 2.0 provides *diagnostic messages*. These messages use 2 reserved identifiers (0x3c, 0x3d). Diagnostic messages use a new format in their frame response called *PDU* (Packet Data Unit). There are two different PDU types: *requests* (issued by the client node) and *responses* (issued by the server node).

The LIN 2.0 configuration mode is used to set up LIN 2.0 slave nodes in a cluster. Configuration requests use SID values between 0xb0 and 0xb4. There is a set of mandatory requests that all LIN 2.0 nodes have to implement as well as a set of optional requests. Mandatory requests are:

- Assign Frame Identifier: This request can be used to set a valid (protected) identifier for the specified frame.
- Read By Identifier: This request can be used to obtain supplier identity and other properties from the addressed slave node.

Optional requests are:

- Assign NAD: Assigns a new address to the specified node. Can be used to resolve address conflicts.
- Conditional Change NAD: Allows master node to detect unknown slave nodes.
- Data Dump: Supplier specific (should be used with care).

4.2 Device Descriptions

Each LIN 2.0 [20] node is accompanied by a *node capability file* (NCF). The NCF contains:

- The node's name.
- General compatibility properties, e.g. the supported protocol version, bit rates, and the LIN product identification. This unique number is also stored in the microcontroller's ROM and links the actual device with its NCF. It consists of three parts: supplier ID (assigned to each supplier by the LIN Consortium), function ID (assigned to each node by supplier), and variant field (modified whenever the product is changed but its function is unaltered)
- Diagnostic properties, e.g. the minimum time between a master request frame and the following slave response frame.
- Frame definitions. All frames that are published or subscribed by the node are declared. The declaration includes the name of the frame, its direction, the message ID to be used, and the length of the frame in bytes. Optionally, the minimum period and the maximum period can be specified. Each frame may carry a number of signals. Therefore, the frame's declaration also includes the associated signals' definitions. Each signal has a name, and the following properties associated with it: **Init value** specifies the value used from power on until the first message from the publisher arrives. **Size** specifies the signal's size in bits. **Offset** specifies the position within the frame. **Encoding** specifies the signal's rep-

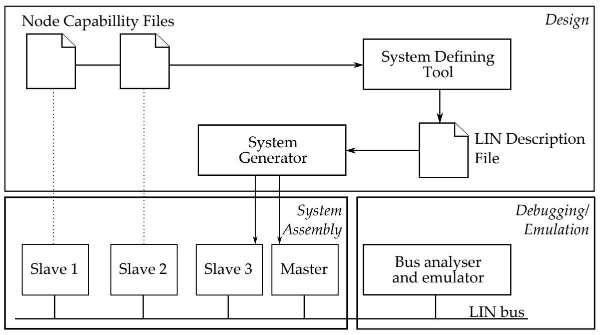


Figure 5. Development phases in LIN

resentation. The presentation may be given as a combination of the four choices *logical value*, *physical value*, *BCD value*, or *ASCII value*. Declarations of physical values include a valid value range (minimum and maximum), a scaling factor, and an offset. Optionally, this can be accompanied by a textual description, mostly to document the value's physical unit. An example is given in figure 4.

- Status management: This section specifies which published signals are to be monitored by the master in order to check if the slave is operating as expected.
- The free text section allows the inclusion of any help text, or more detailed, user–readable description.

The node capability file is a text file. the syntax is simple and similar to C. Properties are assigned using `name = value;` pairs. Subelements are grouped together using curly braces, equivalent to blocks in C.

LIN clusters are configured during the design stage using the *LIN Configuration Language*. This language is used to create a *LIN description file* (LDF). The LDF describes the complete LIN network. The development of a LIN cluster is partitioned into three phases (see figure 5). During the *design phase*, individual NCFs are combined to create the LDF. This process is called *System Definition*. For nodes to be newly created, NCFs can be created either manually or via the help of a development tool. From the LDF, communication schedules, and low–level drivers for all nodes in the cluster can be generated (*System Generation*). Based on the LDF, the LIN cluster can be emulated and debugged during the *Debugging and Node Emulation* phase. In the *System Assembly* phase, the final system is assembled physically, and put to service.

In addition to the LIN configuration language and LDF, which are the most important tools to design a LIN cluster, the LIN specification defines a (mandatory) interface to software device drivers written in C. Also, many tools exist that can parse a LDF and generate driver modules by themselves. The LIN C API provides a signal based interaction between the application and the LIN core (core API).

5 TTP/A

5.1 Communication System Architecture

The information transfer between a smart transducer and its communication partners is achieved by sharing information that is contained in an internal interface file system (IFS), which is situated in each smart transducer. The IFS provides a unique address scheme for transducer data, configuration data, self-describing information, and internal state reports of a smart transducer [1]. It also serves as decoupling element, providing a push interface for producers writing to the IFS and a pull interface for consumers reading from the IFS. Each transducer can contain up to 64 files in its IFS. An IFS file is an indexed array of up to 256 records. A record has a fixed length of four bytes. Every record of an IFS file has a unique hierarchical address (which also serves as the global name of the record) consisting of the concatenation of the cluster name, the logical node name, the file name, and the record name.

A time-triggered sensor bus will perform a periodical time-triggered communication by sending data from IFS addresses to the fieldbus and writing received data to IFS addresses at predefined points in time. Thus, the IFS is the source and sink for all communication activities. Furthermore, the IFS acts as a temporal firewall that decouples the local transducer application from the communication activities.

Communication is organized into rounds consisting of several TDMA slots. A slot is the unit for transmission of one byte of data. Data bytes are transmitted in a standard UART format. Each communication round is started by the master with a so-called fireworks byte. The fireworks byte defines the type of the round and is a reference signal for clock synchronization. The protocol supports eight different firework bytes encoded in a message of one byte using a redundant bit code supporting error detection.

Generally, there are two types of rounds:

Multipartner round: This round consists of a configuration-dependent number of slots and an assigned sender node for each slot. The configuration of a round is defined in a data structure called "RODL" (ROund Descriptor List). The RODL defines which node transmits in a certain slot, the operation in each individual slot, and the receiving nodes of a slot. RODLs must be configured in the slave nodes prior to the execution of the corresponding multipartner round. An example for a multipartner round is depicted in Figure 6.

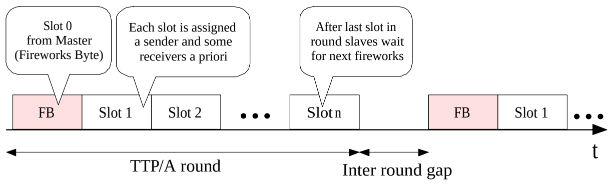

Slot 0 from Master (Fireworks Byte)
Each slot is assigned a sender and some receivers a priori
After last slot in round slaves wait for next fireworks
FB
Slot 1
Slot 2
Slot n
FB
Slot 1
TTP/A round
Inter round gap
t

Figure 6. TTP/A Multipartner Round

Master/slave round: A master/slave round is a special round with a fixed layout that establishes a connection between the master and a particular slave for accessing data of the node's IFS, e. g., the RODL information. In a master/slave round the master addresses a data record using a hierarchical IFS address and specifies an action like reading of, writing on, or executing that record.

The multipartner (MP) round establishes a real-time communication service with predefined access patterns. Master/slave (MS) rounds are scheduled periodically between multipartner rounds, whereas the most commonly used scheduling scheme consists of MP rounds alternating with MS rounds. The MS rounds allow maintenance and monitoring activities during system operation without a probe effect. The MS rounds enable random access to the IFS of all nodes, which is required for establishing two conceptual interfaces to each node, a *configuration and planning* (CP) interface and a *diagnosis and management* DM interface. These interfaces are used by remote tools to configure node and cluster properties and to obtain internal information from nodes for diagnosis.

5.2 Smart Transducer Descriptions

For a uniform representation of all system aspects, an XML-based format is used [21]. A *smart transducer descriptions* (STD) describe the node properties.

There are two types of STDs: *Static STDs* describe the node properties of a particular field device family. Static STDs contain node properties that are fixed at node creation time and act as a documentation of the nodes' features. In contrast, *Dynamic STDs* describe properties of individual nodes, as they are used in a particular application.

Instead of storing the STDs directly on a smart transducer, the node contains only a unique identifier consisting of a series and a serial number, whereas the serial number identifies the node type and the serial number differentiates instances of the same node type. This unique identifier is used to access the node's datasheet on an external server. Thus, node implementations keep a small footprint, while the size of the descriptions is not significantly limited.

5.3 Cluster Configuration Description

The cluster configuration description (CCD) contains descriptions of all relevant properties of a fieldbus cluster. It acts as the central structure for holding the meta-information of a cluster. With help of a software tool capable of accessing the devices in a smart transducer network it is possible to configure a cluster with the information stored in the CCD. A CCD consists of the following parts:

- *Cluster description meta information:* This block holds information on the cluster description itself, such as the maintainer, name of the description file, or the version of the CCD format itself.
- *Communication configuration information:* This information includes round sequence lists as well as round descriptor lists, which represent the detailed specification of the communication behavior of the cluster. Other properties important for communication include the UART specification and minimum/maximum signal run times.
- *Cluster node information:* This block contains information on the nodes in a cluster. These nodes are represented either by a list of dynamic STDs or by references to static STDs.

6 Discussion

Table 1 lists the main features of the three transducer networks with respect to the concepts describe in Section 2. All three approaches provide a real-time service with hard real-time message guarantees, but use different interaction design patterns. COSMIC comes with a publish-subscribe approach where nodes publish their data using the push principle. LIN is a master-slave network where each message is activated by the master. TTP/A uses a master-slave configuration in order to establish a common time base and then follows a predefined communication schedule based on the physical progression of time.

The basic scheduling mechanisms for hard real-time messages by using a static TDMA scheme is the same in all three approaches. The mechanisms for other data is different – TTP/A provides a polling mechanism via master-slave rounds, LIN 2.0 introduced event messages. COSMIC is the most flexible by providing EDF-scheduled soft real-time messages as well as non real-time messages. However, a full implementation of the SRTC requires substantial software because of the dynamic priorities and the more complex handling of discarded messages. Therefore, it has so far only be implemented on more powerful hardware under RT-Linux. Additionally, COSMIC relies on synchronized clocks while LIN and TTP/A require less effort for the proper protocol operation.

The advantages of COSMIC's publish-subscribe are a loose coupling between producer and consumer which facilitates the configuration of a network. The type of the channel to which EM of a certain type is pushed is defined by the publisher. The subscription and the respective guarantees for delivery at the subscriber side, however, may be of the same or a lower real-time class. This enables reception of a critical hard real-time message also for applications which do not need the respective delivery guarantees, e.g. a navigation task which uses critical messages from an obstacle avoidance system.

The pull principle in LIN makes a node's implementation very simple, but causes an overhead on the network due to the frequent message requests from the master. Moreover, since the LIN slaves do not know the time of a request *a priori*, it becomes difficult to time a measurement adequately or to synchronize measurements.

The time-triggered approach of TTP/A comes with

Table 1. Feature comparison

	LIN	COSMIC	TTP/A
Criticality Levels	HRT, SRT	HRT, SRT	HRT
Flow control model	pull	push	TT, pull
Interaction pattern	master/slave	publish/ subscribe	TT, master/slave
Bounded transmission time	yes	yes	yes
Global Time	no	yes	yes
Synchronized actions	no	no	yes
Middleware abstraction	messages	event messages and channels	IFS
Device Descriptions Language	LIN-specific	XML	XML

high efficiency, predictability, and the possibility to synchronize actions. However, the configuration effort of a TTP/A network is higher than for a LIN device or COSMIC devices not requiring stringent real-time guarantees. For example, a TTP/A node has to be configured with the correct schedule before it can participate in the RT communication. In contrast, a LIN node or a COSMIC node might be reused in another application without reconfiguration of the node. Anyway, all three approaches depend on an adequate tool support.

The IFS concept of TTP/A is an abstraction mechanism that hides the time-triggered messages from the application. The IFS implements a distributed shared memory that provides a simple interface for applications. Therefore, TTP/A applications are not triggered by the reception of a message, which allows for a separation of communication and computation.

LIN is designed to serve as sub-bus in automobiles and is therefore specified in a very rigid way towards use in a specific end product. This makes the LIN architecture, though the approach is resource efficient and interesting, less suitable for applications which require a higher degree of cooperation between the nodes and also the rather constraint LIN message format restricts larger sensor-actuator systems. Also the LIN device description is rather focussed on the specific LIN application area.

In contrast, COSMIC and TTP/A specify several high-level features, while leaving details of physical and data link layer up to the implementer. The XML-based datasheets of COSMIC and TTP/A are easily extendable in order to support future extensions.

The mechanisms of the three approaches are different, which makes them incompatible in the first place. In order to achieve interoperability between heterogeneous networks, an adequate interface system, whereas the mechanisms of COSMIC and TTP/A are candidates rather than LIN. COSMIC provides a versatile message interface that abstracts over the underlying communication protocol. On the other hand, the IFS approach of TTP/A allows to abstract over the communication by establishing a distributed shared memory. The IFS comes with the main advantage of being easily adapted to a different protocol, however for convenient application development, tools supporting the set up of the distributed communication schedules are required. Thus, it is up to the application developer if a message-based interface (COSMIC) or a memory-based interface (TTP/A / IFS) is preferred.

7 Conclusion

The contributions of this paper are threefold: Firstly we have elaborated a set of requirements for different kinds of real-time constraints for a distributed system of smart transducers.

Secondly, we have presented and analyzed the concepts of three different smart transducer interface implementation approaches. Each approach has its specific focus concerning an application area. LIN is the protocol with the lowest hardware and cost requirements, however several design decisions restrict its use to an isolated sub-bus for automotive body electronics or simple control systems in industrial automation. LIN is supported by mature tools from automotive suppliers. TTP/A has a similar resource footprint as LIN but firstly substantially benefits from the strict time-triggered communication scheme and secondly provides a convenient distributed shared memory programming model where consistency problems are solved by the synchrony of the communication system. Parameters such as communication speed can be adapted in a rather flexible way depending on the physical network. This makes TTP/A an interesting choice for all kind of low-cost embedded time-triggered applications with real-time requirements. Additionally, the IFS is standardized by OMG in the Smart Transducer Interface Standard [22]. COSMIC provides flexible real-time support and will integrate well into distributed applications with a publish-subscribe communication scheme. The main objective of COSMIC was interoperability between networks with different real-time properties. Thus, a higher overhead in the nodes may be needed. COSMIC and TTP/A come with different configuration support providing similar features

A third contribution of the paper is the discussion of device description. We think that this is an important issue because it firstly underlines the hardware/software (and probably mechanical) nature of a smart transducer and the intrinsically component-based system structure and secondly is indispensable in a complex reliable control system. Presently, device descriptions are mainly used during system configuration to avoid faults from manual set-up. The LIN NFC and also LDF exactly meet these requirements. Device description of TTP/A and COSMIC go beyond the needs of configuration and also are intended for dynamic use. This can range from diagnostic purposes to dynamic device discovery and use during operation.

Although being quite different, we think that it will be possible to establish methods and tools that operate on a meta-level and can be used to configure an application using different underlying fieldbus systems. In order to achieve this, a generic interface model for transducer data has to be found. The Interface File System (IFS) presented with TTP/A seems to be a promising approach for forming a generalized interface, since it is relatively easy to convert transducer data onto an IFS. We will further investigate ways to provide coexistence and cooperation between the different network and programming models.

Acknowledgments

This work was supported in part by the ARTIST2 Network of Excellence on Embedded Systems Design under contract No. IST-004527 and by the Austrian FWF project TTCAR under contract No. P18060-N04. We would like to thank Christian Paukovits, Stefan Pitzek, Gernot Klingler, Christian El-Salloum and Andreas Pfandler for constructive comments on an earlier version of this paper.

References

[1] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, March 2001.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337, April 1998.

[3] A. Krüger. *Interface Design for Time-Triggered Real-Time System Architectures*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, April 1997.

[4] K. Mori. Autonomous decentralized systems: Concepts, data field architectures, and future trends. In *International Conference on Autonomous Decentralized Systems (ISADS93)*, 1993.

[5] J. Kaiser and M. Mock. Implementing the real–time publisher/subscriber model on the controller area network (CAN). In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 172–181, Saint Malo, France, May 1999.

[6] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1999.

[7] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, April-June 1997.

[8] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[9] S. Pitzek and W. Elmenreich. Plug-and-play: Bridging the semantic gap between application and transducers. In *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA05)*, volume 1, pages 799–806, Catania, Italy, September 2005.

[10] Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1451.2-1997, Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Micro-processor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*, 1997.

[11] CAN in Automation e.V. CANopen - Communication profile for industrial systems. available at http://www.can-cia.de/downloads/.

[12] Robert Bosch GmbH. CAN specification version 2.0, September 1991.

[13] L.-B. Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro*, 22(4):28–36, 2002.

[14] J. Kaiser and C. Brudna. A publisher/subscriber architecture supporting interoperability of the CAN–bus and the internet. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS 2002), Västerås, Sweden*, 2002.

[15] M. Gergeleit and H. Streich. Implementing a distributed high–resolution real–time clock using the CAN–bus. In *1st International CAN Conference*, 1994.

[16] J. Ruffino, P. Verissimo, C. Almeida, and L. Rodrigues. Fault–tolerant broadcasts in CAN. In *Proceedings FTCS–28, Munich, Germany*, 1998.

[17] J. Kaiser and M. A. Livani. Achieving fault–tolerant ordered broadcasts in CAN. In *Proceedings of the Third European Dependable Computing Conference (EDCC–3), Prague*, September 1999.

[18] J. Kaiser and H. Piontek. CODES: Supporting the development process in a publish/subscribe system. In *Proceedings of the fourth Workshop on Intelligent Solutions in Embedded Systems WISES 06*, 2006.

[19] M. Kay, Ed. W3C XSL transformations (XSLT) version 2.0. http://www.w3.org/TR/xslt20, June 2006.

[20] Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc. Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification v2.0, 2003.

[21] S. Pitzek and W. Elmenreich. Configuration and management of a real-time smart transducer network. In *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, pages 407–414, Lisbon, Portugal, September 2003.

[22] Object Management Group (OMG). *Smart Transducers Interface V1.0*, January 2003. Specification available at http://doc.omg.org/formal/2003-01-01 as document ptc/2002-10-02.