Diplomarbeit

Real-time Communication Systems for Small Autonomous Robots

ausgeführt zum Zwecke der Erlangung des
akademischen Grades eines Diplom - Ingenieurs am

Institut für Computertechnik 384 der

Technischen Universität Wien

unter der Leitung von

O. Univ. - Prof. Dr. Dietmar Dietrich

und

Univ. Ass. Dr. Stefan Mahlknecht

Univ. Ass. Dr. Wilfried Elmenreich

als verantwortlich mitwirkende Assistenten durch

Stefan Krywult
Matr. - Nr. 9827121
Anton-Bosch-Gasse 4/Stg. 1/5, 1210 Wien

Wien, im Oktober 2006 .....

Abstract

Autonomous systems perform complex tasks to analyse and to react on their environment. One way to handle this complexity is distributing the functionality on several hardware modules. Even in small autonomous systems the predictability of the communication and the synchronization of all modules is vitally important.

This thesis provides a survey of five protocols that play a major role in the automotive industry and in the domain of real-time communication: CAN, LIN, Flexray, TTP/C, and TTP/A. The protocols are compared and analysed regarding their suitability for small autonomous systems.

Following the results of this investigation, TTP/A is ported to the hardware of the Tinyphoon robot, a research platform for small autonomous and distributed systems. On the basis of the outcome of the case study enhancements and adoptions of TTP/A are proposed. To address the diversity of hardware a general concept of making TTP/A more portable is elaborated.

Kurzfassung

Autonome Systeme analysieren ihr Umfeld und reagieren entsprechend darauf. Um die Komplexität dieser Funktionalität besser handhabbar zu machen, wird sie auf mehrere Hardwaremodule aufgeteilt. Auch für kleine autonome Systeme sind die Vorhersagbarkeit der Kommunikation zwischen diesen Modulen und die Synchronisierung des gesamten Systems von großer Bedeutung.

Diese Arbeit beschreibt die Protokolle CAN, LIN, Flexray, TTP/C und TTP/A, die in der Automobilindustrie und im Bereich der Echtzeitkommunikation eine wichtige Rolle spielen und vergleicht sie bezüglich ihrer Verwendbarkeit für kleine autonome Systeme.

Den Resultaten dieser Untersuchung entsprechend wird TTP/A auf die Hardware des Tinyphoon Roboters, eine Forschungsplattform für kleine autonome und verteilte Systeme, portiert. Basierend auf den Ergebnissen dieser Fallstudie werden Erweiterungen und Anpassungen für TTP/A vorgeschlagen. Um der Vielfältigkeit der Hardware in verteilten Systemen Rechnung zu tragen, wird ein Konzept erstellt, wie die aktuelle TTP/A Implementierung portabler gestaltet werden kann.

Contents

List of Figures

List of Tables

List of Listings

How wonderful it is
that nobody need wait a single moment
before starting to improve the world.

Anne Frank (1929 - 1945), 1952
Diary of a Young Girl

Chapter 1

Introduction

In many applications embedded systems take over even security and safety relevant tasks. Small integrated computer systems have been developed for controlling their environment (e. g., drive/fly by wire). The logical consequence is the development of completely autonomous systems. They explore their environment and cope with their tasks without a human user's action. Autonomous systems need to be able to collect relevant data of their environment with sensors, to make decisions based on that data and to influence the environment using actuators.

1.1 General Issue and Background

With the increasing computing power the complexity of embedded systems grows. Handling this complexity is a difficult issue that can be managed using distributed systems i. e., the system is subdivided in smaller parts that act jointly. The success and the quality of the collaboration highly depends on the communication system.

The communication system that connects the parts of an autonomous distributed system has to guarantee that the required pieces of information are delivered to the various subsystems with an almost constant delay and early enough, so that the autonomous system can react on changes of its environment in time i. e., real-time communication.

The case study and target platform of this work is the Tinyphoon research platform, a small robot for playing robot soccer in the Mirosot league. Because of its modular design a high performance real-time protocol is needed, which permits a communication suitable for such an autonomous system in a highly dynamic environment.

1.2 Problem Statements and Methodology

This work shows the applicability of a time-triggered approach as a solution to the problem of real-time communication in small real-time systems with special communication requirements.

The limited resources, the need of fast real-time communication and the variety of the involved platforms leads to high requirements:

- real-time: guaranteed transmission before a specified deadline
- performance: communication speed and efficiency
- integration: connection of the subsystems to the communication system
- availability: available in software and/or in hardware as chip or as intellectual properties
- portability: implementation for all platforms are available or source code can be ported easily
- resource-saving: low usage of Flash memory and Random Access Memory (RAM), small geometrical footprint and low power consumption
- licence: cost and conditions of licencing should allow the use in non-mass products with total system costs of less than Euro 2000.-

Two ways are chosen in this work to come up with a solution to this problem: Firstly, widely used popular protocols and decided real-time protocols are compared and analyzed. Secondly the open real-time protocol Time-Triggered Protocol – Class A (TTP/A) that has a small footprint and is designed to be implemented on off-the-shelf microcontrollers, is implemented keeping the source code portable. TTP/A is then used on a platform of the Tinyphoon project. This case study is expected to reveal, which features are missing, which features are not used at all and which issues have to be solved when making a real-time communication system portable to many different hardware platforms.

The outcome of this work is relevant for the fast growing domain of high performance distributed real-time systems with low resource requirements e. g., in the automotive industry and small autonomous systems.

1.3 Outline of Thesis

This thesis is divided in two parts. The first one (chapter 2) explains the basic terms and concepts and discusses the advantages and disadvantages of various real-time communication systems (TTP/A, TTP/C and Flexray). Moreover, Controller Area Network (CAN) and Local Interconnect Network (LIN) are included within this discussion because of their importance in the automotive sector. The basic features of the USB protocol and some real-time Ethernet variants are also explained.

The second part gives an overview of the target platform and the communication requirements of its subsystems (chapter 3). Then the applicability of the protocols that are discussed in the first section, for small automotive distributed systems is analyzed (chapter 4). Finally, a design of a platform independent and enhanced version of the TTP/A protocol is proposed as a solution for the Tinyphoon robot and similar systems (chapter 5).

The thesis ends with a conclusion that sums up the contributions of this work and gives an outlook on future developments (chapter 6).

A man with a watch knows what time it is.
A man with two watches is never sure.

Segal's Law

Chapter 2

Real-time Protocols

This chapter explains basic concepts of real-time communication protocols, provides a survey of protocols that play a major role in the automotive industry. In contrast to LIN, Flexray, Time-Triggered Protocol – Class C (TTP/C), and TTP/A the CAN protocol has not been designed as real-time protocol but an extension exists that allows its use in a real-time environment. Each protocol is described for its own then main attributes of the protocols (e. g., performance, efficiency,...) are compared.

2.1 Concepts

In this section some basic terms are explained that are used in the following descriptions of the real-time communication protocols. Then the meaning of *real-time* and *fault-tolerance* is discussed.

2.1.1 Basic Terms

A *node* is an entity with a processor that runs an application. The application uses a communication protocol either in form of a software layer or in form of a piece of hardware. The *bus transceiver* converts logical signals in according voltage levels. In figure 2.1 the general structure of a node is shown. The *bus* is the medium (e. g., a kind of wire or fiber), which all nodes are connected to. The whole network, including the nodes an the bus, is called *cluster*. The data packet that is transported through the network is called *frame*. *Message* can be used as synonym for frame and is often used on a higher abstraction level.

The manner how the nodes are connected to each other is called *topology*. Common network topologies for real-time communication are the *bus topology*, the *star topology* and combinations of these two topologies. Star

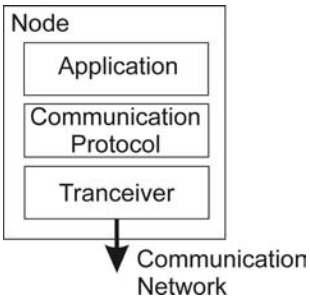
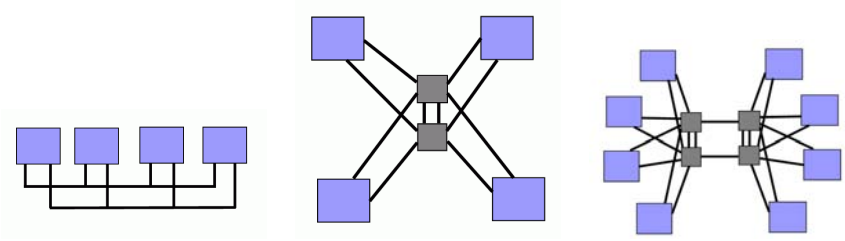


Figure 2.1: Structure of a Node

topologies can be passive (the branchings of the star are just connected at a central point) or active (the branchings of the star are connected to a central electronic device). In normal networks there is only one bus. To ensure that in case of a bus failure the communication can be continued, the bus can be replicated. In figure 2.2 three examples for redundant topologies are shown.



(a) Redundant Bus (b) Redundant Active Star (c) Redundant Mixed

Figure 2.2: Examples for Cluster Topologies

Babbling idiots are faulty nodes that transmit data on the bus constantly and detain other nodes from sending. *Bus guardians* can be used to avoid this if the communication schedule is defined a priori. The bus guardians can be situated on the nodes (local bus guardians) or in a central device like a star coupler (central bus guardian). The bus guardians check on the basis of the communication schedule, if a node is allowed to write on the bus at a given time and detain faulty nodes from monopolizing the bus.

If communication is triggered by the occurrence of a particular event, it is *event-triggered* (e. g., everytime the temperature of a room changes a frame is transmitted). In *time-triggered* communication networks only the progression of time determines when a frame is sent (e. g., the temperature of the room is transmitted every 30 seconds). The time-triggered approach has two major advantages: The communication is predictable (e. g., one frame every 30 seconds) and it supports a defined error detection latency

for omission failures (in our example a node is detected faulty, if no frame has been sent for 31 seconds). The major disadvantages of time-triggered communication is that in many cases bandwidth is wasted, because redundant information is transmitted (e. g., five frames are sent with the information "24°C", then with "25°C"; in an event-triggered system only two frames are needed instead of six). Moreover, events can be missed is the time-triggered communication is too slow (e. g., the room temperature increases from 24°C to 26°C within 30 seconds; the time-triggered system only sends two messages, the event-triggered three). However, it depends on the application whether this is a problem and whether it can be avoided by increasing the update rate of the time-triggered system.

The *efficiency* of a communication protocol is the proportion of the durations of the transmission of the plain payload and of the complete data frame. The efficiency strongly depends on the actual application. In the following the best case efficiency of the protocols is calculated.

2.1.2 Real-Time

In a *real-time* systems the correctness of a result does not only depend on its value but also on the time[Kop97]. The error in the time domain can be as severe as an error in the value domain.

Every communication protocol delays the messages that it transports. Real-time protocols ensure that this delay is predictable and that the deadline for the arrival of the message is not missed. Moreover, by using a real-time protocol the jitter of that delay can be kept small. That means, that from one communication round to another the delay is almost constant. This is important for the implementation of feedback control algorithms, which are in general unable to handle varying delays.

If the clocks of all nodes in a cluster are synchronized, a *global time* is established. In a cluster with a global time a time stamp refers to the same moment in time on all nodes. The clocks of a cluster cannot be synchronized perfectly[Kop97]. The greatest difference between two clocks in a cluster is the *precision* of the cluster.

Three types of real-time systems can be distinguished based on the consequences when a deadline is missed[Kop97]:

Soft Real-Time In soft real-time systems the violation of a deadline causes only a degradation of the service that is provided by the system (e. g., voice-over-IP, video streaming, ...).

Firm Real-Time In a firm real-time system the violation of a deadline

prevents the system from working correctly but the malfunction has no drastic consequences (e.g. small robots)

Hard Real-Time In hard real-time systems the violation of a deadline causes a catastrophe (e. g., fly-by-wire system, nuclear plant automatization, break-by-wire, ...).

2.1.3 Fault Tolerance

A *fault-tolerant* system continues being fully operational even if faults occur. The type and number of faults that have to be tolerated are specified in the *fault hypothesis*. Faults that are not covered by the fault hypothesis, could cause a malfunction. To prevent the system from a steady malfunction a *never-give-up* strategy can be implemented. It tries to bring the system back to an operational state even after an unexpected fault.

2.2 Controller Area Network

The first CAN specification was released by the Robert Bosch GmbH in the year 1985. It describes the implementation of the first (physical) and the second (data link) layer of the Open System Interconnect (OSI) model. Currently various implementations for the application layer of CAN communication systems exist (e.g. CANopen, SDS, DeviceNet and CAL) and CAN has become an International Organization for Standardization (ISO) standard. This standard with the number 11898 is divided in four parts: 1. data link layer and physical signalling, 2. high-speed medium access unit, 3. low-speed, fault-tolerant, medium dependent interface, 4. time-triggered communication.[1]

2.2.1 Mode of Operation

CAN nodes do not have addresses, but Instead every message has an identifier. The nodes of a cluster have to know, which messages they are interested in and how to interpret them.

The medium access in a CAN cluster is controlled in a decentralized way. Data is encoded using a recessive and a dominant voltage level where a dominant level always prevails a recessive one. Every node of the cluster listens on the bus and verifies, that no other node is currently sending. Then the node starts the transmission of the message identifier field. After applying the appropriate voltage level (recessive or dominant) according to the bit being transmitted, the node checks the state of the bus. If it is equal

[1] available at www.iso.org

to the state aimed by the node the transmission of the bit is successful. If the node tries to transmit a recessive symbol but the bus stays at a dominant level, another node is transmitting a dominant Symbol simultaneously. That is the other node has started transmitting a message with a higher priority at the same time.

This technique of distributed arbitration is called Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA). If messages collide, the message with the identifier starting with the most dominant bits has the highest priority and is transmitted successfully. The transmission of all other messages is canceled. They are retransmitted as soon as the bus is idle again.

CAN defines four different types of frames. Remote frames are used to request data from another node, which responds with the according data frame with the same identifier. Data frames may also be sent spontaneously. Error and overflow frames are used to signal error conditions on the bus.

2.2.2 Additional Services

Beyond the normal data transportation CAN provides two additional services.

Acknowledgement

During the transmission of a frame a recessive bit is sent, the acknowledgement slot. This bit is overwritten with a dominant one by a receiver, if the frame has been received successfully. By checking the acknowledgement slot a sender can notice if its transmission has been received successfully by at least one node.

Error Signaling and Fault Confinement

All nodes perform tests to detect various errors (bit monitoring, bit stuffing check, frame check, acknowledgement and Cyclic Redundancy Check (CRC)). A detected error can be signaled to the other nodes by transmitting an error frame. Thus, all nodes of the clusters are informed about the error and may discard the received message to keep the data consistent within the cluster. The transmitters will try to resend the message as soon as the bus is available again.

The reception of an error frame and the results of the own error detection are used to maintain two counters, one for receiving and one for transmitting errors. The current value of these counters represents the quality of the

communication. Their values are increased in case of an error and decreased whenever communication is performed successfully.

Depending on these two values the CAN is one of the three states:

Error active The controller participates in the communication and sends error frames when it detects an error.

Error passive The controller participates in the communication and suspends communication for the time of one error frame when it detects an error.

Bus off The controller does not participate in the communication at all.

2.2.3 Packet Format

Four different frame formats are defined in the CAN specification:

Data frame Data frames are used for sending application data from one node to another. In figure 2.3 the structure of the CAN data frame is shown.

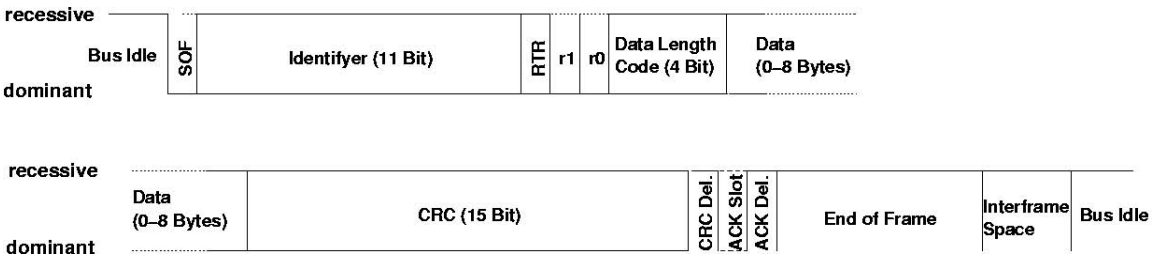


Figure 2.3: CAN Frame Format

SOF The start of frame field is a single dominant bit that marks the start of the data or remote frame.

Identifier The identifier is used for classifying the type of the message. This field is also used for prioritizing the frames using the CSMA/CA technique.

RTR The Remote Transmission Request (RTR) bit is used to distinguish data (RTS bit is dominant) and remote frames (RTR bit is recessive). This bit is also used for Carrier Sense Multiple Access/Collision Detection (CSMA/CD).

r1 and r0 The bits r0 and r1 are reserved. They have to be transmitted as dominant bits but are ignored by the receiver.

Data Length Code The data length code stores the length of the payload. It is encoded as standard binary number from zero to eight. The first bit is the Most Significant Bit (MSB) and dominant bits represent a binary zero and recessive bits a binary one.

Data The data field contains the payload and is transferred MSB first.

CRC The CRC code is 15 bits long and is generated from all fields described above using the generator polynomial shown in equation 2.1.

$$x^15 + x^14 + x^10 + x^8 + x^7 + x^4 + x^3 + 1 \tag{2.1}$$

CRC Delimiter This single recessive bit delimits the CRC.

Acknowledge Slot This single recessive bit is overwritten by the receiver, if a correct frame has been received.

Acknowledge Delimiter Another single recessive bit.

End of Frame The end of the frame is signaled by 7 recessive bits.

Remote frame This type of frames is used for polling other nodes. The structure of these frames is almost the same as the structure of the data frames. But remote frames do not have a payload, the Remote Transmission Request (RTR) bit is recessive and the data length code is interpreted as the length of the requested data.

Error frame Error frames are used for signaling errors. It consists of six to twelve dominant bits and eight recessive ones.

Overload frame These frames consist of six dominant and twelve recessive ones. They can only be distinguished from error frames because of the context they occur in.

Two data or remote frames are always divided by at least three recessive bits if no overload or error frame is transmitted.

Part B of the CAN specification introduces extended frames. These frames have a 29 bit long identifier field. The additional 18 bits of the identifier are prefixed with the substitute remote request bit (SRR, recessive) and the identifier extension bit (IDE, also recessive) and are filled in after the eleven identifier bits of the standard frame.

In this way the SRR bit replaces the RTR bit of the standard frame and the IDE bit the reserved bit r1. Standard and extended frames are distinguished by the IDE bit. Thus, both type of frames can be used within the same cluster.

2.2.4 Data Encoding

CAN uses the Non-Return to Zero (NRZ) encoding. A bit is signaled by applying a dominant voltage level to the bus or releasing it for the complete bit time.

On all fields of data or remote frames except the delimiters, the acknowledgment slot and the end of frame field *bit stuffing* is applied. Whenever five equal bits are transmitted consecutively a complementary bit is filled in.

Some error and overflows violate the bit stuffing rule and can thus be identified as bit stuffing errors.

2.2.5 Physical Layer

Multiple standards describe a physical layer for CAN . The most important ones are:[2]

ISO 11898-2 high speed This physical layer specification provides a communication speed of up to 1 Mbit and is the most common physical layer for CAN networks. Signals are transmitted differentially on two wires using -2 V and +7 V. Only bus topology is allowed. Each end of the bus has to be terminated with a 120Ω resistor.

ISO 11898-3 fault-tolerant This specification targets applications where fault tolerance is required. It needs no termination resistors and supports various topologies. During normal operation differential communication is used as in ISO 11898-2 at a speed of up to 125kbit/s. But communication can be performed even using only one of the two wires in a degraded mode with reduced communication speed. This fact makes this physical layer fault-tolerant. The ISO 11898-3 specification is mainly used in the automotive industry for car body electronics.

SAE J2411 single wire This specification has been designed for the control of comfort electronics in cars. Short single wire networks of almost any topology can be created. A maximum of 32 nodes can communicate in this network with a speed of 33.3 kbit/s. A high speed diagnostic mode allows speeds up to 83.3 kbit/s.

ISO 11992 point-to-point In this specification a physical layer is defined using daisy-chaining to connect the nodes. The networks must have a bus topology and must not exceed 40m of length. The communication

[2] http://www.can-cia.org/can/physical-layer/

	ISO 11898-2	**ISO 11898-3**	**SAE J2411**	**ISO 11992**
type	high speed	fault-tolerant	single wire	point-to-point
maximum speed	1 Mbit	125 kbit/s	33.3 kbit/s	125kbit/s
max. nodes	110	32	32	2
max. network length	6500m	500m		40 m
topology	bus			daisy-chain
fault-tolerant	no	yes	no	yes
connection	2 wires	2 wires	1 wire	unshielded twisted pair
voltage	-2V/+7V	-2V/+7V, supply: 5V		supply: 12V or 24V
termination	120 Ω	none		-
usage	standard applications	car body electronic	car comfort electronic	vehicle with trailer(s)

Table 2.1: CAN Physical Layers

speed is up to 125 kbit/s. The ISO 11992 standard has been developed to electrically connect vehicles with their trailers.

Many other physical layers are also used for CAN (e. g., optical physical layers), but those mentioned above are the most popular ones. Because of the need of a high bit rate in the field of application of this thesis, *CAN physical layer* refers to the ISO 11898-2 standard unless otherwise noted.

The length of the bus is limited by the communication speed, because of the bit-wise arbitration in CAN. It has to be ensured, that every CAN node reads the same bit value from the bus, if two nodes start sending simultaneously. Thus, the propagation delay between two arbitrary nodes must not exceed a certain value. This value depends on the configuration of the cluster, but is allowed to be at most $\frac{11}{8}$ of the length of a bit.

2.2.6 Real-Time Extensions

The basic CAN is not suitable for the communication in hard real-time systems. The Carrier Sense Multiple Access/Collision Detection (CSMA/CD) medium access strategy introduces a non-deterministic jitter of the latency. Only the message with the highest priority has a fixed, defined latency, if it can be presumed, that the system design inhibits a collision of two frames

with the highest priority. The jitter of the latency is undetermined even for messages with the second highest priority, because the whole capacity of the bus may be used for messages with the highest priority and starvation of nodes that try to send messages with a smaller priority, might occur.

Time-Triggered CAN (TT-CAN)[ISO04] (ISO standard 11898-4) solves this problem by introducing a time-triggered additional medium access strategy that is time-triggered. A time master node starts the communication cycle by transmitting a reference message. The falling edge of the start of frame bit of this message marks the start of the communication cycle, the basic cycle. The basic cycle is divided in time windows (slots) of various lengths. In exclusive time windows only one node is allowed to transmit a message. In arbitrating time windows all nodes are allowed to send and CSMA/CA is used for bus arbitration.

Every node has to know, in which slot it has to transmit or receive a certain message. The schedule of a cluster may have various basic cycles of the same structure, which are grouped to the system matrix. In figure 2.4 an example for a TT-CAN system matrix is shown. The lines are formed by the basic cycles. The rows are the time windows. Adjacent arbitrating time windows may be combined to a single large one.

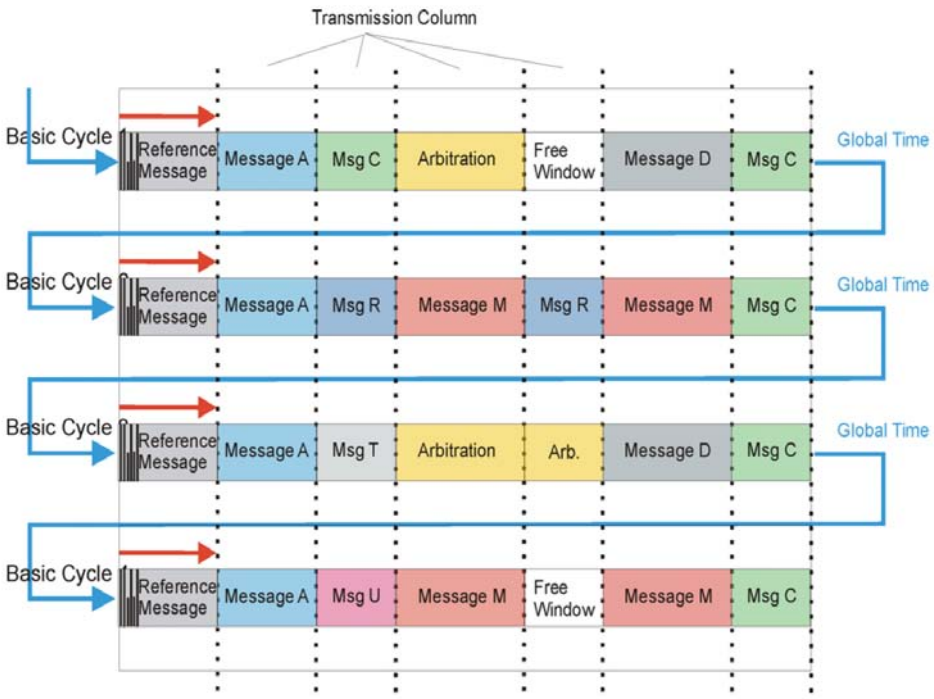


Figure 2.4: TT-CAN System Matrix[FMD+00]

Two levels of TT-CAN are defined. TT-CAN level one the time is measured in bit times of the CAN bus. Thus, the timeliness of the of the communication can be guaranteed, but no global time is established. Level 2 of TT-CAN addresses this feature. The time master includes its current time in the reference message. The other nodes measure the time between two reference messages and correct their clocks to match the values sent by the master. The time in TT-CAN level 2 clusters is measured in Network Time Unit (NTU). The duration of a NTU is in the same order as the duration of one bit time. A global time with a precision of one NTU can be established.

The time master is obviously a single point of failure. Therefore, the time master can be replicated in a TT-CAN cluster. If no traffic is received a potential time master starts sending reference frames with a certain priority. If it receives a reference frame with a lower priority, it transmits a reference frame at the start of the next basic cycle. The other time master loses the arbitration because of the CSMA/CA algorithm. Thus, conflicts between potential time masters that start sending coinstantaneously are resolved and the potential time master with the highest priority succeeds. When an expected reference frame is not sent within a certain time-out, potential time masters start transmitting reference frames.

Though TT-CAN can be build in software using a CAN controller, the hardware of this controller at least has to be equipped with an additional circuitry for time-stamping incoming messages[HMFH00].

2.2.7 Efficiency

The maximum length of the payload of a CAN frame is eight bytes (= 64 bits = $T_{payload\_max}$). The overhead caused by wrapping the payload in a frame ($T_{framing}$) is calculated according to equation 2.7.

$$n_{header} = n_{sof} + n_{id} + n_{rtr} + n_{reserved} + n_{lengthcode} = \tag{2.2}$$
$$= 1 + 11 + 1 + 2 + 4 = 19 \tag{2.3}$$
$$n_{trailer} = n_{crc} + n_{crcdel} + n_{ackslot} + n_{ackdel} + n_{eof} = \tag{2.4}$$
$$= 15 + 1 + 1 + 1 + 7 = 25 \tag{2.5}$$
$$T_{framing} = (n_{header} + n_{trailer}) \cdot T_{bit} = \tag{2.6}$$
$$= (19 + 25) \cdot T_{bit} = 44 \cdot T_{bit} \tag{2.7}$$

Another overhead (T_{coding}) is added because bit stuffing is performed and the bus has to be released for at least three bit times before a new frame is

started. The amount of bits that are stuffed in the frame depends on the content of the frame and the identifier. Thus, an upper and a lower bound are calculated for the coding overhead.

$$T_{coding\_max} = n_{stuffbits\_max} \cdot T_{bit} + T_{interframe} = \tag{2.8}$$

$$= \frac{n_{header} + n_{maxpayload} + n_{crc}}{5} \cdot T_{bit} + 3 \cdot T_{bit} = \tag{2.9}$$

$$= (\frac{19 + 64 + 15}{5} + 3) \cdot T_{bit} = (\frac{98}{5} + 3) \cdot T_{bit} = 22 \cdot T_{bit} \tag{2.10}$$

$$T_{coding\_min} = (0 + 3) \cdot T_{bit} = 3 \cdot T_{bit} \tag{2.11}$$

$$efficiency_{max} = \frac{T_{payload\_max}}{T_{payload\_max} + T_{framing} + T_{coding\_min}} = \tag{2.12}$$

$$= \frac{64 \cdot T_{bit}}{(64 + 44 + 3) \cdot T_{bit}} = \frac{64 \cdot T_{bit}}{111 \cdot T_{bit}} = 0.5766 \tag{2.13}$$

$$efficiency'_{max} = \frac{T_{payload\_max}}{T_{payload\_max} + T_{framing} + T_{coding\_max}} = \tag{2.14}$$

$$= \frac{64 \cdot T_{bit}}{(64 + 44 + 22) \cdot T_{bit}} = \frac{64 \cdot T_{bit}}{130 \cdot T_{bit}} \cdot T_{bit} = 0.4923 \tag{2.15}$$

Depending on content and the identifier of the frames from zero to 19 bits are stuffed in the frame. The maximum efficiency of CAN is 57.66% if no bit stuffing occurs. The TT-CAN protocol is less efficient because an additional overhead is caused by embedding the communication in time slots.

2.2.8 Availability

CAN controllers are available in many different versions: as Intellectual Property (IP) (e.g. net list), as stand-alone controller or integrated in various microcontrollers.

Due to this variety and the low costs of these hardware implementations, software implementations of CAN are not common.

Manufactures of CAN enabled Integrated Circuits (IC) have to license CAN from the Robert Bosch GmbH.

2.3 Local Interconnect Network

The Local Interconnect Network (LIN) is developed by the LIN consortium[3]. This protocol has been designed to satisfy the needs of the automotive industry for cheap and simple communication protocol for multiplexing various signals on a single bus. LIN can be used for real-time communication, but it does not offer a global-time or fault-tolerance features. The protocol can be implemented in hardware and in software even on low performance off-the-shelf microcontrollers using a standard Universal Asynchronous Receiver and Transmitter (UART). This description of the LIN protocol is based on the LIN specification 2.0 issued in 2003[LIN03].

2.3.1 Mode of Operation

LIN is a master-slave protocol. In every cluster a single master polls the slaves for a certain information according to its internal schedule. The polling process is implemented by splitting the LIN frame. The header is sent by the master. It contains unique mark for the frame start and an identifier that informs the nodes about the length and the meaning of the data. The actual data is then sent in the response field by the slaves, which is responsible for sending this type of messages. Every LIN master node also incorporates a client implementation, which is used for sending and receiving data as well as for error detection.

Beyond 60 identifiers (0 – 59) for data frames special identifiers exist for diagnostic frames (60 and 61) and user defined frames (62). The identifier 63 is reserved for future use. The payload has a maximum length of eight bytes and is divided in signals. A signal is a value of a certain meaning and a certain length. LIN frames with the same identifier always contain the same signals at the same position.

Each node has to be configured to read the frames, it is interested in, from the bus, and to respond to headers of frames it has to transmit. The complete schedule of the communication is only known by the master and is processed cyclically.

The LIN specification defines three modes a frame can be transmitted:

Unconditional frames These frames are standard way of transmitting frames. When the transmission of a certain frame is scheduled, the master sends the header and the according node completes the frame with its data and a checksum.

Event-triggered frames To reduce the average bandwidth requirements for rarely changing signals, event-triggered frames have been intro-

[3] `http://www.linlin-subbus.org`

duced. Multiple nodes may react on the same identifier. The slaves only respond on the header if they have something to transmit (e. g., an event has occurred). If none of the slaves responds, the master knows that no event happened. If one of the slaves replies, the frame is received correctly. However, another slave also may have started a transmission and stopped sending as it noticed the ongoing bus activity. Therefore, the master has to schedule another event triggered frame to ensure that there are no other events pending. If more than one slave responds on the header, the frame is corrupted, the master detects a checksum error and has to poll all slaves using unconditional frames. The answers of the slaves contain the identifier of the related unconditional frame. Thus, it can be distinguish, which event occurred.

Sporadic frames Sporadic frames can be scheduled by the master in free slots of the predefined, fixed communication schedule, whenever needed. Usually the slave implementation of the master node transmits the data of this frame.

2.3.2 Additional Services

Beyond the standard communication features, LIN offers means for diagnostic communication and status management is able wake-up a sleeping cluster.

Diagnostic Communication

The LIN consortium has defined a special diagnostic transport layer. Diagnostic communication is performed using special frames, master request frames (identifier 60) and slave response frames (identifier 61).

The nodes of a LIN cluster have a node address for diagnostic communication. Slaves can be addressed and messages up to 4095 bytes can be exchanged. The diagnostic transport layer fragments and defragments the messages accordingly, so they can be transported using the diagnostic frames that carry at most eight bytes. Thus, the same diagnostic protocols can be used in the LIN cluster and on a higher level communication systems that the cluster might be connected to.

Status management

Every node has to report its state with at least one bit in one of the unconditional frames, which it transmits. This bit is set, if an error occurred during sending or receiving the response fields. The bit is cleared after its

transmission. The master processes these bits an deduces the state of the nodes.

But also the slave nodes use this information and the information about successful data transfers to appraise their current status.

Cluster Wake-up

The master can put the cluster in sleep mode by sending a master request frame with the invalid node address zero. Once the cluster is asleep every node can wake it up by keeping the bus in the dominant state for $250\mu s$ to $5ms$. After the end of the wake-up signal all salves are ready for participating in the communication within $100ms$. The master starts executing its communication schedule not longer than $150ms$ after this signal.

2.3.3 Frame Format

Figure 2.5 shows the structure of a LIN frame.

Figure 2.5: Structure of a LIN Frame[LIN03]

Header The master starts a new frame by sending the header.

Break symbol The break symbol consists of at least 13 dominant bits, followed by a single recessive one. It is a unique symbol that signals the start of a new frame and that can not occur within a regular frame.

Synch byte The synch byte is a single byte with the value 0x55. It forms a pattern of alternating bit values that can be used by the slaves to synchronize their bit clock to the bit timing of the master.

Protected identifier The protected identifier consists of a six bit long identifier (0 – 63) and two parity bits that are calculated as shown in equation 2.17. $\oplus$ is the operator for the exclusive-or

operation and $ID[i]$ is the bit number i of the ID.

$$Bit6 = ID[0] \oplus ID[1] \oplus ID[2] \oplus ID[4] \tag{2.16}$$
$$Bit7 = \neg(ID[1] \oplus ID[3] \oplus ID[4] \oplus ID[5]) \tag{2.17}$$

Response A slave responds to the header and completes the frame by transmitting the response field.

Data The data field contains at least one at most eight bytes of data. Signals with a length of more than one byte are transmitted in little-endian order.

Checksum In the LIN specification two types of checksums are defined. The classic checksum is calculated using the bytes of the data field only. The calculation of the extended checksum also includes the protected identifier. Both types of checksums are calculated in the same way: All bytes are added in a eight bit register. In case of an overflow, the carry bit is also added. The inverted result of this calculation is then used as checksum.

2.3.4 Data Encoding

LIN uses standard UART frames with NRZ encoding. The frames have one start, one stop bit and eight data bits that are sent Least Significant Bit (LSB) first.

There is only one exception. The break field consists of at least of 13 consecutive dominant bits. This pattern is not a valid UART frame.

Every two bytes of a frame may be separated with an additional space. The total length of the space must be smaller than 40% of the total frame length.

Two frames are separated by an inter frame space.

2.3.5 Physical Layer

LIN uses an improved version of the physical layer defined in the ISO standard 9141[4]. The bidirectional bus consists of a single wire and supports a data rate of up to 20 kbit/s. The LIN bus forms a wired AND gate. Every LIN transmitter has a built-in pull-up resistor and can pull the bus down to ground level with a transistor. The recessive state ($\geq$0.6V) signals a logical one and the dominant state ($\leq$0.4V) signals a logical zero.

The maximum length of the LIN bus is 40 meters and up to 16 LIN nodes may be connected.

[4]available at `http://www.iso.org`

2.3.6 Host Interface

The LIN specification defines a set of C-functions as interface to the host application. This set includes functions for initialization, for manipulating and querying the predefined signals, for managing the schedule in the master node and for controlling the bus interface. A configuration and a diagnostic Application Programming Interface (API) are also available.

Beyond the programming interface the LIN specification specifies a file formats that store all communication-relevant settings, e. g., signals, frames and the master schedule (LIN description file) as well as a description of LIN nodes (node capability file). These file formats ensure the correct collaboration of development tools of various companies.

2.3.7 Efficiency

The maximum efficiency of a LIN frame can be calculated from the transferred bytes of data and the minimal slot length that is needed to schedule this message.

T_{header} is the amount of time for transmitting the header that has always the same length. T_{data_max} is the time needed for transferring the payload data of the maximum length of eight bytes. $T_{response\_max}$ includes T_{data_max} and the time needed to transmit the checksum. Header and response transmission time are added to get the minimum time T_{frame_max} for transmitting a LIN frame with a payload of maximum length.

$$T_{header} = (T_{break} + T_{synch} + T_{id}) \cdot T_{bit} = \tag{2.18}$$
$$= (14 + 10 + 10) \cdot T_{bit} = 34 \cdot T_{bit} \tag{2.19}$$
$$T_{data\_max} = N_{data\_max} \cdot 10 \cdot T_{bit} = \tag{2.20}$$
$$= 8 \cdot 10 \cdot T_{bit} = 80 \cdot T_{bit} \tag{2.21}$$
$$T_{response\_max} = T_{data\_max} + T_{crc} = \tag{2.22}$$
$$= 80 \cdot T_{bit} + 10 \cdot T_{bit} = 90 \cdot T_{bit} \tag{2.23}$$
$$T_{frame\_max} = (T_{header} + T_{response\_max}) \cdot T_{bit} = \tag{2.24}$$
$$= (34 + 90) \cdot T_{bit} = 124 \cdot T_{bit} \tag{2.25}$$

According to the LIN specification, the minimum length of a frame slot is 40% longer than the minimum length of the packet that is sent in the slot.

$$T_{frame\_slot} = T_{frame\_max} \cdot 1.4 = \tag{2.26}$$
$$= 124 \cdot T_{bit} \cdot 1.4 = 173.6 \cdot T_{bit} \tag{2.27}$$

The efficiency of LIN is calculated by opposing the minimum time for a frame slot (T_{frame_slot}) and the time for sending the actual payload (T_{data_max}).

$$efficiency_{max} = \frac{T_{data\_max}}{T_{frame\_slot}} = \tag{2.28}$$

$$= \frac{80 \cdot T_{bit}}{173.6 \cdot T_{bit}} = \frac{80}{173.6} = 0.4608 \tag{2.29}$$

The upper bound for the efficiency of the LIN protocol is thus 46.08%.

2.3.8 Availability

LIN is often implemented in software, but there are also hardware implementations and LIN IPs.

Only members may use intellectual properties of the LIN consortium. To become an associated member an admission fee of $ 10,000.– and an annual fee has to be paid. Members do not have to pay any license fees for their products that use LIN. The specification is openly available for download on the website of the consortium[5].

2.4 Flexray

Flexray is an upcoming standard for real-time communication in the automotive industry. It is developed by the Flexray consortium[6]. This section is based on the Flexray Protocol Specification[Fle05b] and the Flexray Physical Layer Specification[Fle05a].

2.4.1 Mode of Operation

The Flexray protocol is executed in communication cycles. Each cycle consists of a static segment, a dynamic segment, a symbol window and a network idle time. The length of each segment is specified in macroticks. If no dynamic segment or the symbol is not used the length of the respective segment can be set to zero. The static and dynamic segments are divided into time slots. The clocks of all nodes are synchronized. Therefore, every node knows the number of the current slot and when a new slot starts. An action point is defined by specifying the number of macroticks since the start of a slot. The transmitting node starts sending at the according action point.

[5] `http://www.lin-subbus.org/frontend/stylesheets/request_doc.htm`, visited on 2006-10-06

[6] http://www.flexray.org

Every frame has an identifier that specifies the number of the slot, which it has to be sent in. For every channel each identifier must not be used more than once within one communication cycle.

Figure 2.6 shows the structure of a Flexray communication cycle.

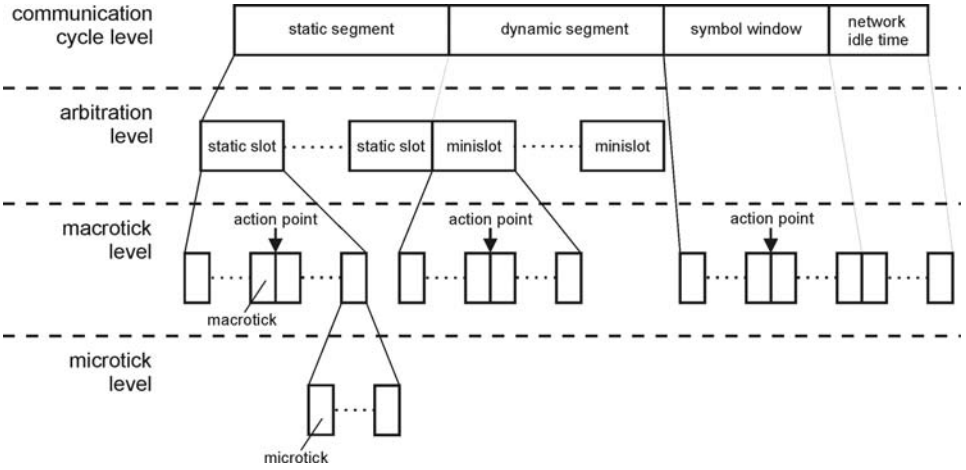


Figure 2.6: Structure of a Flexray Communication Cycle[Fle05b]

There are three possibilities how communication in a Flexray cluster can be performed:

1. The dynamic segment is not used. All communication is performed within the static segment.
2. Only one static slot is used for synchronization. The actual communication is performed within the dynamic segment.
3. Both, the static and the dynamic segment are used for communication.

It depends on the requirements of the application, which of these modes of operation is adequate.

Static Segment

The static segment is used to exchange information with a guaranteed maximum delay and jitter. Time Division Multiple Access (TDMA) is used for arbitration within the static segments. The segment is divided into time slots with the same length that is equal for both channels.

The application can decide to send the same frame on both channels. Thus, faults on one channel can be tolerated. If fault tolerance is not needed the communication speed can be doubled for particular slots by sending different frames on each channel.

Dynamic Segment

The dynamic segment is used to exchange information that has to be transmitted in varying and unknown intervals, efficiently. However, generally it is not possible to determine a maximum delay for frames that are transmitted within the dynamic segment. Mini-slotting is used for the arbitration within this segment. The segment is divided into mini-slots. Dynamic slots are superimposed on them. If no data is transmitted in a dynamic slot, it only consists of one single mini-slot. Otherwise the dynamic slot is expanded over as many mini-slots as needed for the transmission of the complete frame. The number of mini-slots depends on the length of the dynamic segment and must not be exceeded. The smaller the identifier of a frame, the smaller is the number of the slot a message is assigned to and the higher is the priority of this message.

Symbol Window

In this segment symbols (such as the Media Access Test Symbol) may be transmitted. The protocol performs no arbitration for this segment. This has to be handled by the application.

Network Idle Time

No communication is allowed during the network idle time. The length of the idle time is the total length of the communication cycle minus the length of the static segment, of the dynamic segment and of the symbol window.

Addressing Modes

The nodes of a Flexray cluster do not have an address on the protocol layer. But all frames have an identifier that enables the receiving nodes to filter incoming messages. Only those important for the application layer are stored.

Clock Synchronization

The right timing is crucial for the communication within a Flexray cluster. All nodes must agree on the number and the start of the current macrotick. This can only be guaranteed by performing a clock synchronization.

Within a Flexray cluster this is performed using the Fault-Tolerant Midpoint Algorithm[LL84]. Every node saves the arrival time of data frames that are transmitted in the static segment and have a special bit set. These frames are called Sync Frames and are transmitted from nodes that have a reliable

clock that should be used for synchronization. Then the sender's action point (the point in time when the sender started transmitting the frame) is estimated by considering propagation delay. The difference between the estimated action point of the sending nodes and the action point of the receiving node are stored in a sorted list. If a frame is transmitted an both channels, the smaller difference is stored in the list.

The k smallest and the k largest values are removed from the list, where k is calculated from the number of stored differences (n) using equation 2.30.

$$k = \begin{cases} 0 & \text{if } 1 \leq n \leq 2 \\ 1 & \text{if } 3 \leq n \leq 7 \\ 2 & \text{if } 8 \leq n \end{cases} \tag{2.30}$$

The largest and the smallest of the remaining values are averaged to calculate the midpoint value. This value interpreted as the deviation of the node's clock from the global time. Offset and rate error are calculated from the deviation and corrected every second communication cycle during the network idle time.

In a Flexray cluster with a communication speed of 10Mbit/s the clocks can be synchronized with a precision better than $1\mu s$.[Fle05b]

Node Integration

When a new node is connected to a cluster it has to adopt the current timing of the cluster to be able to join the communication. The process of integration differs for normal nodes and nodes that can initiate a cluster start-up (coldstart nodes)

Normal nodes These listen on both Flexray channels and tries to receive two valid startup frames from a coldstart node, adopts its timing and performs clock synchronization. Then it searches for startup frames from two distinct coldstart nodes and checks whether they fit in the own schedule. If this is the case for four cluster cycles and clock synchronization can be performed without errors the node enters the normal operation mode.

Coldstart nodes These perform a very similar procedure of integration, but they start normal operation after synchronizing themselves with at least one coldstart node for three cluster cycles.

Cluster Startup

If coldstart nodes do not receive any Flexray frames during the process of integration they initiate a cluster startup. The application may veto the

startup.

The startup is initiated by sending a Collision Avoidance Symbol (CAS). Then the coldstart node starts sending startup frames accordingly to its schedule. If a startup frame is received between the transmission of the CAS and the first transmission of the startup frame the node stops the startup and tries to integrate on the other coldstart node that transmitted this frame. Thus, the scheduling of the startup frame decides, which coldstart node performs the startup if more then one node initiated the cluster startup simultaneously by sending the CAS.

Fault Handling

A fatal error within the protocol engine or the product specific part causes the Flexray node to stop participating in the communication immediately. This stop can also be encompassed by the application through the Controller Host Interface (CHI) .

Non fatal errors may cause the protocol engine to only degrade the communication service that means it receives data frames from the bus but does not transmit any. However, the exact behavior in such cases depends on the configuration.

2.4.2 Additional Services

Beyond the basic data transportation features Flexray provides following additional services.

Fault Tolerance

Flexray supports the implementation of a fault-tolerant communication. The dual-channel topology provides means to implement communication systems that can tolerate the breakdown of one channel. But Flexray does not provide a complete ready-to-use solution.

Also the clock synchronization algorithm, the node integration and cluster start-up process of Flexray are designed to tolerate faults. However, a fault hypothesis is not part of the Flexray specification.

In an additional specification bus guardians for Flexray controller are described. These guardians are are used locally, directly on the Flexray node. The bus guardians protect only the static segment against timing violations.

Global Time

To synchronize the medium access of the nodes within the TDMA scheme the clocks of the communication controllers have to be synchronized as described earlier. All communication controller of a Flexray cluster agree with a limited deviation on this time. It is made available to the host application and can be used for performing actions synchronously to the communication or for time-stamping.

Membership

Flexray supports the implementation of a membership service but does not provide a ready-to-use solution.

The frame format defines a flag that specifies whether a membership vector is included within the payload or not. During a communication cycle Flexray controllers process the vector from all valid incoming frames from both channels and perform an or-operation. The result can be used by the application for implementing a membership service. It is the responsibility of the application software to calculate and to transmit an appropriate network management vector.

Cluster Wake-Up

To save energy a cluster can be set to sleep mode. Every node of the cluster can wake it up by sending the wake-up symbol. The transmission of this symbol is triggered by the host application. The bus drivers of the other nodes detect the wake-up symbol and wake the communication controller that inform the host processor. The wake-up is only sent on one of the two channels to ensure that communication is still possible, in case a faulty node sends wake-up symbols continuously.

2.4.3 Frame Format

The Flexray frame format is shown in figure 2.7.

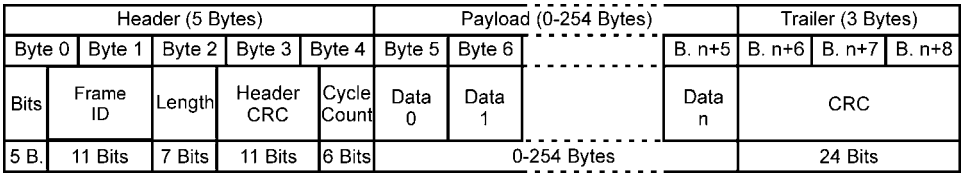


Figure 2.7: Flexray Frame Format

Header segment This segment contains important information about the frame.

Bits These bits are used to mark special frames.

Reserved Bit This bit is reserved for future use, must be transmitted as 0 and must be ignored when it is received.

Payload Preamble Bit This bit indicates that additional information is stored at the beginning of the payload segment.

Null Frame Indicator If this bit is set the data in the payload segment is invalid.

Sync Frame Indicator If this bit is set, the node should use the timing of this frame for its synchronization process.

Startup Frame Indicator This bit indicates whether the frame is a startup frame or not. Frames having this bit set are used for cluster startup and also have to be sync frames.

Frame ID The frame identifier specifies the slot, which the frame has to be transmitted in. Therefore, every identifier may only be used once per communication round and per channel. 0 is an invalid frame identifier.

Length The length of the payload segment in words (two bytes).

Header CRC Some parts of the data in the header (sync and startup frame indicator, frame identifier and payload length) is relevant for the right operation of the protocol. Hence, it is protected with this additional header CRC. The CRC code is 11 bits long and is generated using the polynomial shown in equation 2.31.

$$x^1 1 + x^9 + x^8 + x^7 + x^2 + 1 = (x+1)\cdot(x^5 + x^3 + x^1)\cdot(x^5 + x^4 + x^3 + x + 1) \tag{2.31}$$

0x01A is used as initialization vector of the header CRC.

Cycle Count This field contains the the value of the communication cycle counter of the sending node.

Payload segment In this segment the actual application data is transmitted. According to the value the state of the payload preamble bit in the header segment additional administrative information might precede the payload.

For frames transmitted within the static segment this information is a network management vector. With network management vectors that are received during a communication cycle a bitwise OR operation is performed and the result is made available to the application layer at the end of the cycle. The length of the network management vector has to be configured in the communication controller. All additional

network management functionality has to be performed by the application software.

For frames within the dynamic segment the additional information is a 16 bit long message identifier. This identifier may be used by receiving nodes for filtering certain packages. However, it is the task of the application to set the identifier and the payload preamble indicator correctly.

The maximum length of the payload is limited by the number of bits that are used for encoding it. The seven bit long length field limits the maximum size of the payload to $(2^7 - 1) \cdot 2 = 127 \cdot 2 = 254$ Bytes.

Trailer segment The trailer segment contains a 24 bit CRC code. It is 11 bits long and is generated using the polynomial shown in equation 2.32.

$$x^{24}+x^{22}+x^{20}+x^{19}+x^{18}+x^{16}+x^{14}+x^{13}+x^{11}+x^{10}+x^8+x^7+x^6+x^3+x+1 = \\ = (x+1)^2 \cdot (x^{11}+x^9+x^8+x^7+x^5+x^3+x^2+x+1) \cdot (x^{11}+x^9+x^8+x^7+x^6+x^3+1) \tag{2.32}$$

0xFEDCBA is used as initialization vector for frames sent through channel A and 0xABCDEF for those on channel B.

2.4.4 Data Encoding

The bits of the Flexray data frames are transmitted using the NRZ encoding. Before and after the actual frame data and between every two transmitted bytes special bit sequences are inserted to support synchronization between sender and receiver.

Transmission Start Sequence Each transmission is started by pulling the data line low for a predefined amount of time. This period is used for setting up active star coupler accordingly.

Frame Start Sequence Each data frame is preceded with one high bit.

Byte Start Sequence Each data byte is preceded with one high and one low bit.

Frame End Sequence One low bit and one high bit are appended to each frame.

Dynamic Trailing Sequence After the transmission of the last bit of a frame within the dynamic segment, the data line is hold low until the next minislot. Then, a single high bit is transmitted.

Three different bit sequences may be transmitted as symbols:

	point-to-point	passive bus	passive star	active star
maximum distance between two nodes	24m	24m	24m	24m to/from star
minimum number of stubs	0	4	3	2
maximum number of stubs	0	22	22	-
minimum number of splices	0	2	1	-
maximum number of splices	0	-	1	-

Table 2.2: Flexray Physical Layer Characteristics

Collision Avoidance Symbol This symbol is transmitted by pulling the data line low for a certain amount of time. It is preceded by a transmission start sequence. The symbol is used during the cold start to determine the leading cold start node. All other nodes integrate on this node and synchronize their communication on it.

Media Access Test Symbol This symbol looks like the collision avoidance symbol. However, they can be distinguished because the media access test symbol is sent within the symbol window. The transmission of this symbol is triggered via the CHI .

Wakeup Symbol This symbol consists of alternating sequences of low and high bits. The sequences have a predefined length. The number of signal changes also can be configured. The symbol is used to wake up the sleeping nodes of a cluster. It may only be transmitted on one channel at a time. Its transmission is triggered by the host.

2.4.5 Physical Layer

A passive bus, a passive star, or an active star topology can be used in a Flexray cluster. It is also possible to use a combination of the active star and the bus topology. The nodes of a Flexray cluster can be interconnected with one or two busses. If two busses are used, both have to belong to one single cluster. Bridges between different clusters can only be built using a second communication controller.

A dedicated electrical physical layer specification for Flexray systems exists [Fle05a]. It defines a differential transmission using ± 600mV.

The maximum distance between two active stars is limited to $24m$. Only two active stars are allowed to be placed on every signal path.

The information is sent over the network using the NRZ coding and a maximum speed of 10 Mbit/s.

2.4.6 Host Interface

The Flexray host interface is divided in two parts, the protocol data interface and the message data interface. The protocol data interface allows to configure the protocol parameters, to control the protocol execution (e. g., change the execution state, perform external clock synchronization), and to read the status of the protocol execution (e. g., slot counter, macro tick counter, clock rate correction).

Through the message data interface the received data can be accessed and the data that has to be transmitted, is be handed over to the communication controller. This interface is also used for configuring the handling of incoming and outgoing data (e. g., in which time slot data is stored in which buffer).

Beyond, the host interface offers additional services: Received messages can be filtered using the first two bytes of their payload. A powerful, configurable interrupt logic can provide interrupts on all important protocol events. Moreover, the network management service preprocesses all incoming network management vectors (all vectors of a communication cycle are or-ed and the result is made available to the application).

2.4.7 Efficiency

For the calculation of the efficiency, only the static segment is considered. The maximum length of the payload of a Flexray frame is 240 bytes (= 1920 bits = $n_{payload\_max}$). The overhead caused by wrapping the payload in a frame ($T_{frameing}$) is calculated according to equation 2.34.

$$T_{framing} = (n_{header} + n_{trailer}) \cdot T_{bit} = \tag{2.33}$$

$$= (5 \cdot 8 + 3 \cdot 8) \cdot T_{bit} = 64 \cdot T_{bit} \tag{2.34}$$

Another overhead (T_{coding}) is added because additional bits have to be transmitted at the start of a transmission, of a frame, and of a byte, as well as at the end of each frame.

$$T_{coding} = (n_{transmission\_start} + n_{frame\_start} + n_{byte\_start} \cdot n_{bytes\_max} + n_{frame\_end}) \cdot T_{bit} = \tag{2.35}$$

$$= (3 + 1 + 2 \cdot 240 + 2) \cdot T_{bit} = 486 \cdot T_{bit} \tag{2.36}$$

$$efficiency_{max} = \frac{T_{maxpayload}}{T_{maxpayload} + T_{frameoverhead} + T_{coding}} = \tag{2.37}$$

$$= \frac{1920 \cdot T_{bit}}{(1920 + 64 + 486) \cdot T_{bit}} = 0.7773 \tag{2.38}$$

According to equation 2.38 the maximum efficiency of the Flexray frame encoding is 77.73%. However, this calculation does not consider the padding time of the frames in the time slots. This time depends on the topology of the cluster. A lower bound for the maximum efficiency can be calculated from the maximum slot duration (T_{slot_max}). According to [Fle05b] $t_{slot\_max} = 659$ Macroticks. At 10Mbit/s a Macrotick has a length of at least $1\mu s$. Thus $T_{slot\_max} = 659\mu s$.

$$efficiency_{max\_lo} = \frac{T_{maxpayload}}{T_{slot\_max}} = \tag{2.39}$$

$$= \frac{1920 \cdot T_{bit}}{659\mu s} = \frac{1920 \cdot 10^{-7} s}{6590 \cdot 10^{-6} s} = 0.2914 \tag{2.40}$$

The lower bound for the maximum efficiency is 29.14%.

2.4.8 Availability

At the day of writing several Flexray implementations are available. The Robert Bosch GmbH[7] offers a Flexray intellectual property (IP) module that is called E-Ray and that can be synthesized either as a stand-alone device or as part of another IC.

Freescale Semiconductors[8] produces stand alone Flexray controllers called MFR4200 and MFR4300.

Currently no software implementations of the Flexray protocol are known.

The development of Flexray is managed by the Flexray consortium. A registered copy of the Flexray specification is available for free, but the use of the intellectual property is only allowed to members. Memberships are available for EUR 7.500,- and for EUR 15.000,-.[9]

2.5 TTP/C

The TTP/C is a part of the Time-Triggered Architecture (TTA) and the major member of the time triggered protocol family. It is developed by the

[7] www.bosch.com

[8] www.freescale.com

[9] according to the information on the Flexray website at 2006-08-26

TTA-Group[10] and is mainly used in the avionic industry. Class C refers to the group of applications that TTP/C is designed for, as defined by the Society of Automobile Engineers (SAE)[11]. Class C communication protocols provide high speed communication at minimum 125kbit/s that can be used for real-time control (e.g. engine control, brake by wire).

This chapter is based on the specification of the version 1.1 of the TTP/C [TTA03].

2.5.1 Mode of Operation

TTP/C uses a TDMA scheme to access the bus after a successful startup has been accomplished. In a TDMA round every node may access the bus only once. At a configured time it is allowed to transmit data for a predefined time (node slot). Thus, all nodes in the cluster have to agree on the actual time. This is achieved by synchronizing the clocks of the nodes as described later in this section. A appropriate configuration for the communication has to be created for every node. It is stored in the Message Descriptor List (MEDL), a data structure in the Communication Network Interface (CNI), the application interface of the communication controller. The communication has to be designed in a way that at no time more than one node is allowed to transmit data on the bus.

The TTP/C communication controller decides according to the MEDL when data has to be transmitted on or received from the bus, and where in the CNI the data is read from and stored to, respectively. The application can not influence the timing of the communication. The CNI forms a temporal firewall[Kop97].

TDMA rounds with the same sequence of slots are grouped forming a cluster cycle. Nodes with the same position within different TDMA rounds of the same cluster cycle are equal in length and have the same sending node. The cluster cycle is repeated continuously. Every node sends the same piece of data (same length and same semantic meaning) in its slot within a particular type of TDMA round. The content of a frame may differ from one TDMA round to another. Figure 2.8 shows the structure of a cluster cycle.

Cluster Mode

Distributed systems may have more than one functional behavior, which are called modes. Depending on the current task of the system the mode can be changed accordingly. Each mode of the distributed systems requires the exchange of a different set of data. TTP/C reflects these needs by allowing

[10] http://www.ttagroup.org
[11] www.sae.org

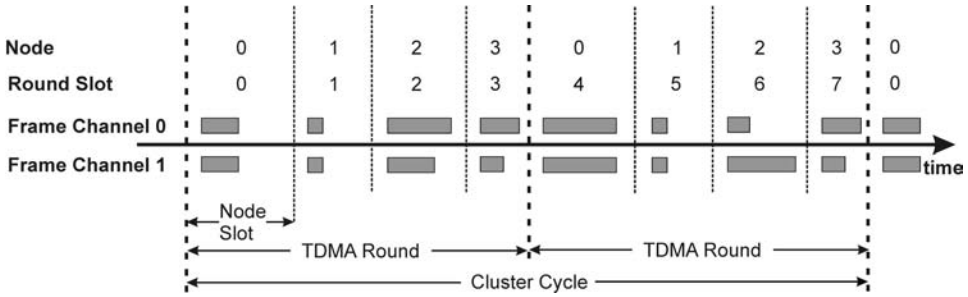


Figure 2.8: TTP/C Cluster Cycle[TTA03]

the definition of multiple cluster cycles and offering means for switching between them safely.

At least two cluster modes have to exist in every TTP/C cluster: the startup mode and one application mode. If it is required by the application additional application modes can be added. Special modes may also be designed for debugging, diagnosis or maintenance.

Virtual member nodes

In TTP/C a fixed slot in every TDMA round is allocated for every node. However, in a TTP/C cluster physical nodes may share a single node slot. The transmission of the data is multiplexed. The physical nodes access the bus alternately. They behave like a single node that is called virtual member node.

Controller State

The controller state (C-state) is composed of a set of values that reflect the actual state of the communication controller:

Global Time The time of the start of the current slot in macroticks.

Round Slot Position The number of the current slot in the cluster cycle.

Cluster Mode The current cluster mode (defines, which schedule is used).

Deferred Pending Mode Changes Pending mode changes that will be performed at the start of a new cluster cycle.

Membership Information Information about the activity of the nodes in the cluster. The membership information is consistent among all cluster nodes.

Thus, the C-state must be equal on all nodes under normal operating conditions. Otherwise, one or more nodes would violate the timing when they access the bus or they do not agree on the list of active nodes. TTP/C provides means to check the consistency of the C-state on all nodes and to recover from C-state errors (Never Give Up (NGU) strategy).

Clock Synchronization

TTP/C uses the Fault-Tolerant Average (FTA) algorithm [KO03] to synchronize the clocks of the nodes: Every node measures the time between the expected and the real arrival time of a data frame. The propagation delay is considered by subtracting a configurable amount of time from the measurement. If the received data frame is sent by a node with a precise clock (according to the MEDL of the receiving node), the measured time difference is included in the calculation of the clock correction term. It is stored on a push-down stack with a depth of four entries. At a slot that is configured in the MEDL, all nodes calculate the correction term. The smallest and the largest value are removed from the stack. The average of the two remaining values is then used as clock correction term.

The nodes correct their clocks by a configurable number of microticks everytime a definable number of macroticks has elapsed until the complete clock correction term has been applied.

TTP/C also supports the synchronization to an external clock. The host application informs the TTP/C controller about an external clock correction term, that is added to the internal one.

At minimum four nodes are needed to be able to tolerate one byzantine fault per TDMA round[KHK$^+$96]. A dependable clock synchronization is crucial for the functioning of the TDMA bus accessing strategy.

According to [KHK$^+$96] the achievable precision of the ensemble of clocks in a typical cluster with a communication speed of 10Mbit/s is better than $0.25\mu s$.

Node Integration

An integrating node has to adopt the current C-state of the cluster. It listens for frames, which include the current C-state (frames with explicit C-state, see 2.5.3). Within two TDMA rounds at least one frame with an explicit C-state has to be transmitted in order to allow integration of nodes. If such a frame is received, the node sets its own C-state accordingly and starts participating in the communication in a passive way: frames are received and handed to the host and clock synchronization is performed. After a configurable number of correct frames has been received and more correct

than incorrect frames have been received (clique avoidance, see 2.5.2) the node starts sending in its assigned slot.

Cluster Startup

A node performs a cold start, if its configuration allows a cold start and the node has not received any frames during the integration process for a certain time. The node sends cold start frames (see 2.5.3) on both channels. It then receives frames until the it reaches its own sending slot. If the node has received frames and the majority of them have the same C-state, the node starts transmitting frames according to its schedule.

In case that the majority of nodes in the cluster have another C-state, or in case that no frame has been received the node starts the integration process again. The number of cold start attempts that my be performed is limited, to detain cold starting nodes with an incoming link failure from continuously impeding the communication on the bus.

Cold start integration enabled nodes integrate on cold start frames. This process is very similar to the normal node integration. The main differences is that cold start frames may collide. If this collision is detected by all nodes, the integrity of the frame destroyed and the frame is ignored. A critical situation may occur, if two nodes send a cold start frame at almost the same time. Due to the propagation delay of the bus, these two frames may be received by the other nodes in a different order on networks with a bus topology. In networks with star topology each of the replicated star couplers may decide differently, which frame comes in first.

TTP/C solves this problem by introducing the *big bang*. Every integrating node ignores the first cold start frame. The cold starting nodes receive no response and retries the cold start after a certain time. This retry time-out is larger than the maximum propagation delay of the bus and is unique within the cluster. Therefore, there is no collision during the second attempt of the cold start and all nodes integrate on the same cold start node.

2.5.2 Additional Services

Besides the normal data transfer TTP/C offers some additional services. These are described in this section.

Fault Tolerance

One of the main design goals of TTP/C is fault tolerance. An additional specification for a fault tolerant layer (FT-COM) on top of TTP/C exists. It has the same interface as the normal CNI. Under this interface the com-

plete implementation of the fault tolerant communication is hidden. Therefore, the FT-COM CNI is completely transparent for the host application. The fault tolerant layer uses the replicated communication channels of the TTP/C bus.

TTP/C specifies central bus guardians that are displaced from the communication controllers and powered by their own supply. Thus, the likeliness of a failure of the node and the bus guardian at the same time is reduced because of the spacial distance.

Other services that are described in the next sections the membership service, the implicit acknowledgement and the clique avoidance improve the ability of detecting and handling faults additionally.

Global Time

To synchronize the medium access of the nodes within the TDMA scheme the clocks of the communication controllers have to be synchronized as described earlier. All communication controller of a TTP/C cluster agree with a limited deviation on this time. It is made available to the host application and can be used for performing actions synchronously to the communication or for time-stamping.

Membership

Every TTP/C communication controller implements a membership vector. Every bit in this vector represents a node of the cluster. If a valid frame is received from a certain node in its sending slot within the TDMA round, the according bit in the vector is set to one. Otherwise, the bit is set to zero. The membership vector is part of the C-state. Thus, all nodes of a cluster have to agree on the current membership vector.

Implicit Acknowledgement

A TTP/C controller can detect outgoing link failures using the membership information of its successors. If the membership information is included explicitly in the frame only the according bit has to be checked. Otherwise, the membership information is only part of the CRC calculation. In this case the receiving node calculates both possible CRCs and compares it to the received one. There are three possible cases:

1. The received CRC and the first calculated checksum are equal: The last transmission has been successful.

2. The received CRC and the second calculated checksum are equal: A disturbance during either the last transmission or during the reception occurred. The following frame has to be investigated to make a final decision about the acknowledgement.

3. The received CRC is and both calculated CRCs are unequal: The transmission of the successor has been disturbed. Further investigation is needed to make a final decision about the acknowledgement.

In case a further investigation is needed the frame sent by the successor of the successor is used similarly to the description above to make a decision. If again both checksums are unequal, the frame of the next successor is used, and so forth.

Clique Avoidance

All nodes of a TTP/C cluster have to agree on the cluster's current C-state. The C-state is communicated either implicitly or explicitly whenever a node transmits a frame. Groups of nodes that agree on the same C-state are called cliques. It is crucial for the proper functioning of the communication, that the developing of such cliques is avoided.

To detect cliques every node counts agreed and disagreed frames:

Agreed Frames are received frames with implicit C-state that pass the CRC test, received frames with explicit C-state that pass the CRC test and contain the same C-state as the receiving node and frames that are transmitted.

Disagreed Frames are received frames with implicit C-state that do not pass the CRC test and received frames with explicit C-state that pass the CRC test but do not contain the same C-state as the receiving node.

If the number of the disagreed frames is larger than that of the agreed ones, it is very likely, that there are at least two cliques and the node is not a member of the largest one. In such a case the node stops participating in the communication immediately. Thus, the formation of cliques is avoided.

2.5.3 Frame Format

TTP/C uses three different types of frames:

Frames with explicit C-state This type of frame consist of a frame type identifier, a possible mode change request, the C-state of the sending

communication controller and a CRC. Included in this CRC but not within the data frame is the schedule identifier of the frame. Integrating nodes can only synchronize themselves on frames with an explicit C-state.

Frames with implicit C-state These frames are very similar to those of the preceding type. However, they only include the C-state in the calculation of the CRC but not in the actual data frame. This allows the C-state of a controller to be tested for consistency without transmitting it. The disadvantage of this technique is, that communication failures and differences in the C-state can not be distinguished. In many cases communication failures are transient failures that do not require any action from the node whereas an illegal C-state is a critical error that detains the node from participating in the communication correctly.

Coldstart frames This is a special type of frames. It is only used during the startup of the cluster. These frames have no payload and no complete C-state. They only transport a part of C-state (the current global time and the current round slot) that is necessary for a node to synchronize itself during the cluster startup.

In figure 2.9 the structure of these three types of frames is shown.

In current implementations maximum frame length is limited to 240 bytes. In a frame with an implicit C-state thereof 3 bytes are used for the CRC and one byte for the header.[Kop01].

2.5.4 Data Encoding

The TTP/C specification does not define a certain bit encoding. Only the timing of the transmission of the data frames within the TDMA slots is specified in a more detailed way.

A TDMA slot starts with the transmission of a node (transmission phase - TP). After the transmission the receiving nodes process the received data frame (post-receive phase - PRP). Before a node starts sending, it has to prepare the data frame (pre-send phase - PSP). This has accomplished before a new slot starts, in order that the transmission starts exactly on the beginning of the time slot. In the time between post-receive phase and pre-send phase the communication controller is idle. The bus is only accessed during the transmission phase.

Figure 2.10 depicts the slot timing.

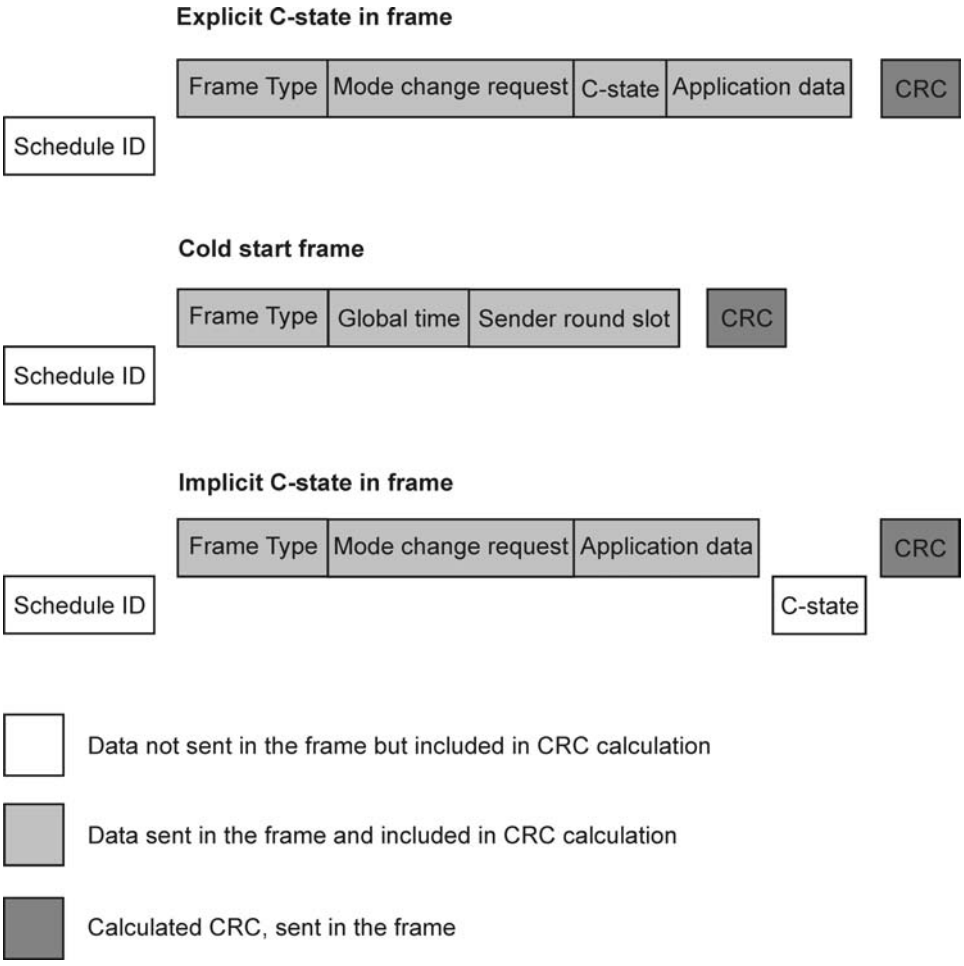


Figure 2.9: TTP/C Frame Formats[TTA03]

2.5.5 Physical Layer

The TTP/C specification does not define a physical layer. Only the requirements on the physical layer are listed:

- The physical layer must have two independent channels.
- The physical layer must support broadcasts.
- The propagation delay of the physical layer must be known.

2.5.6 Host Interface

TTP/C 's host interface is realized by the CNI. It is accessed like normal memory and forms a temporal firewall between the host Central Processing

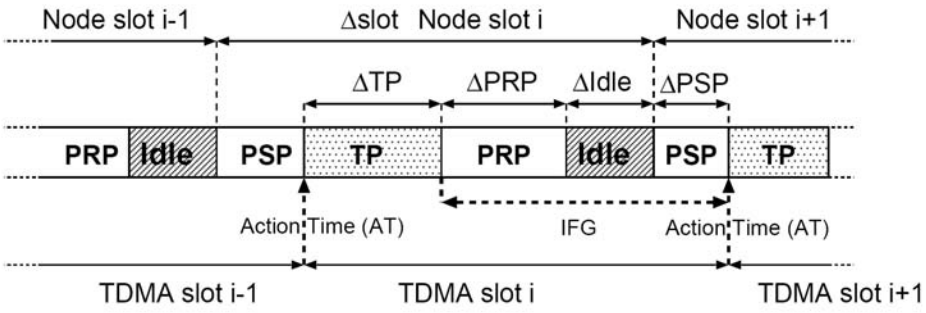


Figure 2.10: TTP/C Slot Timing[TTA03]

Unit (CPU) and the communication controller. Hence, the host CPU can not influence the timing on the bus.

The CNI is composed of three different parts:

Status area In this part the communication controller stores all variables that refer to the current protocol state (e.g. C-state, protocol execution state, current clock correction term, ...). This section is read-only for the host.

Control area This part is used by the host to govern the controller (e.g. switch it on and off, request a cluster mode change, synchronize the clock to an external clock, ...)

Message area In this section the data for every incoming and outgoing frame of every time slot and every communication channel is stored. Every block of frame data is prefixed with a frame status field that is not transmitted and contains important informations about the frame (e.g. error flags). The MEDL of the communication controller has to be configured properly so the controller puts the received data to and sends the data from the correct address in the memory.

2.5.7 Efficiency

The calculation of the efficiency of Time-Triggered Protocol (TTP)/C is difficult, because the specification neither contains a description of the frames on bit level nor a detailed description of the timing for the bus access. According to [Kop01] the minimum overhead of a TTP/C frame with implicit C-state is one byte for the header and three bytes for the checksum and the minimum length of the interframe gap is $5\mu s$ at a communication speed of $10Mbit/s$.

$$T_{payload\_max} = n_{payload\_max} \cdot 8 \cdot T_{bit} = \tag{2.41}$$

$$= 236 \cdot 8 \cdot T_{bit} = 1888 \cdot T_{bit} \tag{2.42}$$

$$T_{framing} = n_{framing} \cdot 8 \cdot T_{bit} \tag{2.43}$$

$$= 4 \cdot 8 \cdot T_{bit} = 32 \cdot T_{bit} \tag{2.44}$$

$$efficiency_{max} = \frac{T_{payload\_max}}{T_{payload\_max} + T_{framing} + T_{interframe\_gap}} = \tag{2.45}$$

$$= \frac{1888 \cdot T_{bit}}{1888 \cdot T_{bit} + 32 \cdot T_{bit} + 5\mu s} = \tag{2.46}$$

$$= \frac{1888 \cdot 0.1\mu s}{1920 \cdot 0.1\mu s + 5\mu s} = \frac{188.8\mu s}{197\mu s} = 0.9584 \tag{2.47}$$

The maximum efficiency that can be achieved in a TTP/C cluster is 95.8%.

2.5.8 Availability

TTP/C is available as IP from TTChip[12] and as complete IC (e.g. AS8202NF from Austria Micro Systems[13]).

TTP/C has been designed to be implemented in silicon. Hence, there is no software implementation available.

The development of TTP/C is managed by the TTA-Group[14]. A registered copy of the TTP/C specification is available for free. Chip designers and producers have to pay a license fee for using the intellectual property.

The TTA-Group accepts associate members (companies that develop products related to TTP/C) and affiliate members (companies that want to use TTP/C related technology). Only the committee members have an influence on the development of TTP/C .

2.6 TTP/A

TTP/A is a member of the time triggered protocol family for SAE class A applications. Class A covers low speed communication (about 10kbit/s) in the domain of convenience features in vehicles. TTP/A has been designed to be implemented in software on small off-the-shelf microcontrollers and is used for sensor actuator bus systems. TTP/A is not fault tolerant and is rarely used by the industry.

[12] `http://www.ttchip.com`
[13] `http://www.austriamicrosystems.com`
[14] `http://www.ttagroup.com`

This section is based on the version 2.0 of the TTP/A specification[EEE+01]. The specification is explained and interpreted in [EHK+02]. This document also gives some hints for implementing TTP/A.

2.6.1 Mode of Operation

TTP/A is a master slave protocol and uses a combination of polling (like LIN) and TDMA (like TTP/C and Flexray) for bus arbitration. The communication is performed in cycles (rounds) that are started by the master node. It sends a round identifier that is one byte long (fireworks byte) and different from all bytes that may be sent during normal data transmissions.

The firework byte signals the start of a new round and, which of the predefined communication schedules, the Round Definition List (RODL), should be used by the slaves. There are up to six different schedules available. They are designed offline and have to ensure, that at no time two nodes transmit at the same time. Thus, each slave has its own set of schedules. According to the entries in the schedule the nodes send or receive data. This data is stored in a special data structure, the Interface File system (IFS) where it can be accessed by the application. This type of communication cycle is called Multi partner (MP) round. Such rounds are used for real-time communication. An example for a MP round is shown in figure 2.11. At most 62 bytes of application data can be transferred within a MP rounds.

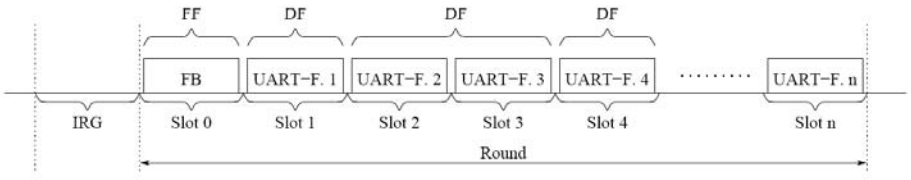


Figure 2.11: TTP/A Multi Partner Round[EHK+02]

There are two additional firework bytes that start special communication cycles that are used for maintenance and configuration. The Master slave (MS) rounds are scheduled regularly and allow to read and write arbitrary files in the cluster's IFS. First a record is addressed by the master by sending a Master Slave Address (MSA) that carries the address of the data record: the node identifier, the file number and the record number. The type of the access is also encoded in this frame. In the Master Slave Data (MSD) that is scheduled at least after one MP round, the slave node either receives data from the master and stores it in its IFS or the slave reads the record from the IFS and transmits it to the master within the MSD round. The MSA round is also used for clock synchronization. It carries the epoch counter, the number of the current communication round.

As the description above shows, the proper reception of the fireworks byte is crucial for a slave to participate in the communication correctly. Thus, the fireworks bytes are protected by a code with a Hamming distance of at least four[HH00]. The fireworks byte of the MSA round is 0x55. This bit pattern allows clock synchronization and an automatic baud rate detection. The lower three bits of the fireworks bytes specify the number of the communication schedule used for this round. Thus, there are eight different rounds (0–7). Schedule number five (fireworks byte 0x55) is reserved for the MSA round and number one is reserved for the MSD (fireworks byte 0x49).

The execution of task can be triggered internally when a execution of a file is scheduled in the RODL or externally by the master, when the MSA contains an execution command. This feature is used to synchronize the actions of a TTP/A node to the communication.

2.6.2 Additional Services

Beyond the basic functionality for transferring data TTP/A offers additional services for synchronizing the cluster, for network management and for accessing the data stored in the IFS of a cluster. The features membership service, cluster wake-up, and baptizing are optional and need not be part of every TTP/A implementation.

Global Time

TTP/A provides a simple clock synchronization that is not fault tolerant. The slaves synchronize their clocks on the start of the fireworks bytes that are sent by the master. Additionally, the slaves adopt the epoch counter, which is included in every MSA round.

However, the protocol is designed to be implemented on very cheap hardware and not every node is expected to have a precise crystal oscillator. To guarantee that even in long MP rounds the clock precision of such nodes is good enough to be able to participate in the communication without violating the timing constraints of the TDMA scheme, TTP/A slave nodes can be resynchronized on arbitrary bytes in this round. The time slots that are used for synchronization are marked with a special flag in the schedule of the slaves.

The clock synchronization has to be precise enough to ensure that the time difference between the expected and the real start of a transmission never exceeds the duration of one bit.

Membership Service

TTP/A provides a simple membership service that is not fault-tolerant. The master of the cluster maintains two membership vectors. In one a flag is set for every slave if it participates in the communication in MP rounds as expected. The other vector provides information, if the slave nodes have responded in the last MSD round correctly after they had been addressed for a read access in the previous MSA round.

Sleep mode

A cluster can be put in sleep mode by the master by sending a special execution command during the MSA round. The cluster wakes up again as soon as traffic is detected on the bus.

Baptizing

For configuration and debugging the slaves of a TTP/A cluster have to be addressed in the MSA round. This address can be assigned statically at compile time or dynamically when the node is connected to the bus. A baptizing algorithm as proposed in [EHPS02] is used for the dynamic assignment. The problem is that all unbaptized that only all unbaptized nodes can addressed at once.

First the unique physical name of an unbaptized node has to be found:

1. The master write a physical name in a special record on all unbaptized nodes.
2. The master triggers a comparison of this physical name with the unique name of the unbaptized nodes.
3. In the following MSD round only unbaptized nodes respond with a physical name identifier that is greater than or equal the value written in step 1.
4. By repeating this algorithm a binary search is implemented.

Then the logical name has to be assigned:

1. The new logical name is written to a special record on all unbaptized nodes.
2. The master triggers the execution of a task that copies the logical name to the intended location in the IFS if the physical name of the node is equal to the comparison value.

2.6.3 Frame Format

The TTP/A protocol transfers one byte per frame. The meaning of the bytes is defined only by the time, when they are sent.

In figure 2.12 a piece of a TTP/A communication with a MP round is shown.

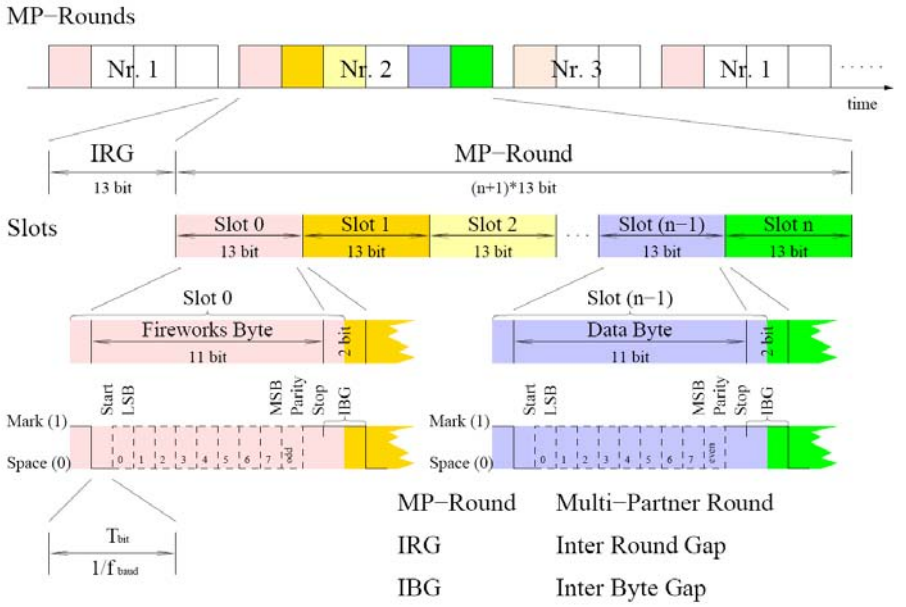


Figure 2.12: Structure of a TTP/A Communication Cycle[EHK+02]

Every round starts with a fireworks byte. In a MP round every node sends and receives bytes accordingly to its schedule. In contrast in the MSA round only the master node transmits data. The data of a MSD round is either sent by the master or a single slave node. Both, the MSA and the MSD round have the same structure (see figure 2.13): the fireworks byte, four bytes of data and a checksum. The checksum is calculated from the first five bytes of the MS round using the exclusive OR operation.

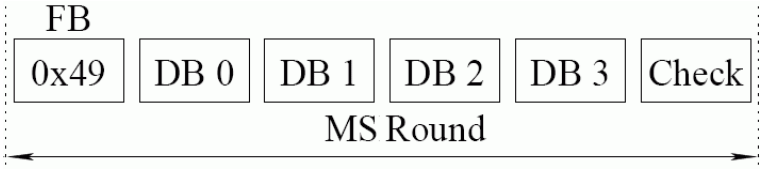


Figure 2.13: Structure of a TTP/A Master – Slave Round[EHK+02]

2.6.4 Data Encoding

TTP/A uses the NRZ encoding. The data is transmitted in standard UART frames with a start bit, eight data bits, a parity bit, and a stop bit.

A bus idle time of one bit time before and after the UART frame is added. These idle times form a two bit Inter Byte Gap (IBG) when the frames are sent consecutively. Figure 2.14 shows a TTP/A byte frame.

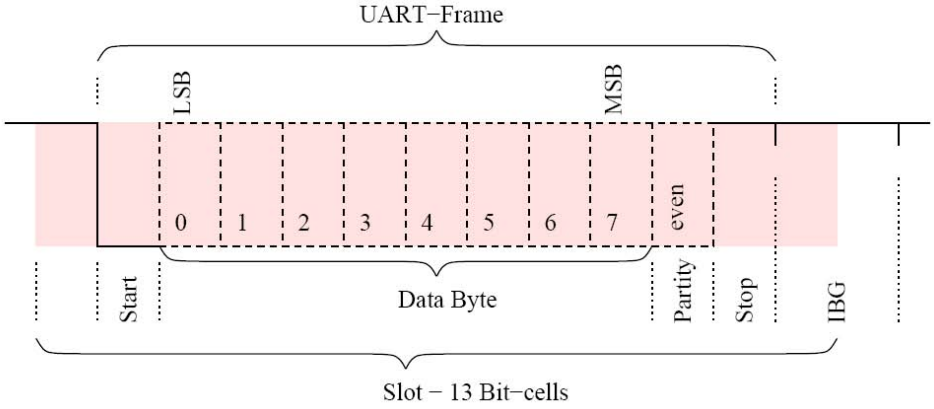


Figure 2.14: Structure of a TTP/A Frame[EHK+02]

Two communication rounds are divided by a Inter Round Gap (IRG) of at least 13 bit times. During this time the bus is idle.

2.6.5 Physical Layer

The definition of the physical layer is not within the scope of the TTP/A specification. The specification only defines the requirements on the physical layer:

- The physical layer has to support the transportation of UART messages.
- If the baptizing feature of TTP/A is used, the physical layer also has to support concurrent transmissions. A collision may destroy the transmitted frames, but the master has to be able to detect that there is ongoing communication on the bus.

The timing of the TTP/A protocol is defined on bit level. One node sends after the other and the gaps between two frames on the bus is fixed to 2 bit times. Although this gap allows some jitter, this fact puts a restriction on the length of the TTP/A bus.

2.6.6 Application Interface

The interface to the TTP/A protocol is IFS. The IFS consists of up to 64 files with up to 256 records. Records are four bytes long. The first record of a file is the header record and stores the attributes of the file.

There are special files that are used to configure the TTP/A protocol:

RODL File This file contains the communication schedules.

Configuration File The configuration file contains the node identifier (logical name) of the node and is used by the baptizing algorithm.

Membership File The membership file is only available on the master node and contains the membership vectors.

Round Sequence File The Round Sequence (ROSE) file is also only available on the master node. The master schedules the communication rounds according to this file.

Documentation File In the documentation file the unique, 64 bit long physical name of the node is stored.

All other files can be used by the application. Callbacks can be registered for every file. It is called whenever the the current slot of the schedule contains an execute command for that file.

Through the IFS on the master node also the content of the MSA and the MSD can be assigned. Thus, it is possible to implement a gateway on the master that handles incoming requests over another interface and accesses the IFS of the cluster accordingly. Such a gateway can be used with appropriate software [KS04b] to visualize and modify the IFS conveniently on a Personal Computer (PC).

2.6.7 Efficiency

The maximum efficiency of TTP/A round can be calculated from the maximum payload of 62 Bytes and the overhead that is added by the framing of each byte ($T_{framing} = 3 \cdot T_{bit}$), the IBG for every byte ($T_{IBG} \geq 2 \cdot T_{bit}$),

the IRG ($T_{IRG} \geq 13 \cdot T_{bit}$), and the fireworks byte.

$$T_{max\_payload} = N_{max\_payload} \cdot 8 \cdot T_{bit} = \tag{2.48}$$

$$= 62 \cdot 8 \cdot T_{bit} = 496 \cdot T_{bit} \tag{2.49}$$

$$T_{max\_round} = (N_{max\_payload} + 1) \cdot (8 \cdot T_{bit} + T_{framing} + T_{IBG}) + T_{IRG} = \tag{2.50}$$

$$= (63 \cdot 13 + 13) \cdot T_{bit} = 832 \cdot T_{bit} \tag{2.51}$$

$$efficiency_{max} = \frac{T_{max\_payload}}{T_{max\_round}} = \tag{2.52}$$

$$= \frac{496 \cdot T_{bit}}{832 \cdot T_{bit}} = 0.5962 \tag{2.53}$$

The upper bound for the efficiency of TTP/A is 59.62%.

2.6.8 Availability

The TTP/A protocol has been standardized in the Smart Transducer Specification[OMG02] of the Object Modelling Group[15]. The specification of the TTP/A protocol is available for free. Moreover, the source code of a C-implementation for the eight bit microcontrollers from Atmel[16] is available under a modified version of the BSD license[17].

Currently no pure hardware implementation of TTP/A exists, but an enhanced UART supporting synchronization and time-stamping in hardware has been implemented[DEE03].

2.7 Other Real-Time Protocols

2.7.1 Real-time Ethernet

Ethernet is not suitable for real-time application because of its medium access strategy. The CSMA/CD technique just detects collision and tries to retransmit the frame after a random time. Thus, CSMA/CD is not predictable and frames may even get lost. However, due to the enormous success of Ethernet within the last decade, Ethernet components have become very cheap. Meanwhile industrial Ethernet is even used in plant automatization.

[15] `http://www.omg.org`
[16] `http://www.atmel.com`
[17] available at `http://www.vmars.tuwien.ac.at/ttpa`

To make Ethernet suitable for real-time applications, the use of CSMA/CD has to be avoided, that means that collisions have to be avoided. Mainly, there are three approaches for solving this problem:

Topology based Switched Ethernet avoids collision by establishing a point-to-point connection between sender and receiver. Thus no collisions can occur. The problem of this solution is that switches create an additional, unpredictable delay.

Software based Examples for software based solutions are Powerlink and the IEEE 1588 Precise Clock Synchronization Protocol. A low-level driver is used for synchronizing the clocks of the nodes and to establish a TDMA scheme on top of Ethernet to allow a collision free communication. The precision that can be reached using these techniques is coarse.

Hardware based Examples hardware based solution are PROFInet V3, EtherCAT and Time-Triggered Ethernet. PROFInet enforces a TDMA based access scheme through the hardware. EtherCAT uses a kind of master-slave concept, where the frames from the master are piped through one node after the other. The major problem of these methods is, that new, proprietary hardware is needed. Time-Triggered Ethernet also uses special switches but allows standard Ethernet controllers to participate in the communication that is not time critical.

A detailed discussion of these approaches can be found in [KS04a].

The components of real-time Ethernet are too large as they could be used on small robots.

2.7.2 Universal Serial Bus

Another very wide spread protocol is the Universal Serial Bus (USB) protocol[USB00]. The USB protocol is a master-slave protocol. The host controller, the master, polls data from the slaves. The USB specification defines various transfer modes that fit needs of different applications. Two of them are interesting for real-time communication:

Isochronous Transfer For every connection with this transfer type a configurable bandwidth is assigned. In case of an error no retransmission is scheduled.

Interrupt Transfer This transfer mode allows fast polling of the slave in regular intervals (up to one access every millisecond in high speed mode). In case of a transmission error the slave is polled once more.

More and more microcontroller with included USB slave controllers are brought on the market . There are also stand-alone ICs available. It is much more difficult to find USB host controllers. They are integrated in powerful microprocessors or are designed as Peripheral Component Interconnect (PCI) devices.

One of the major problems of using USB for real-time communication is, that the scheduling of the messages is performed by the driver of the host controller. This task and the control of the USB host controller make the driver complex. Moreover, important features like the clock synchronization are missing.

2.8 Comparison

In table 2.3 the five protocols described above are compared.

The appearance of the specifications of Flexray and TTP/C shows the different design philosophies: Flexray is defined very precisely using Specification and Description Language (SDL) diagrams, but leaves the implementation of many higher level services (e. g., the membership service) to the application designer. On the contrary the TTP/C specification defines a lot of sophisticated features as part of the protocol, but does not give any detailed information for the implementation. A comparison of a previous version of Flexray with TTP/C is presented in [Kop01].

Flexray allows event-triggered communication in the dynamic segment. In the static segment all time slots have the same length, but a node is allowed to send in various of these slots. TTP/C schedules slots with varying lengths within a TDMA cycle, but every node is allowed to transmit data at most once in a cycle.

Most of the algorithms used in the TTP/C protocol and some used in the Flexray protocol are verified formally. Another difference is that TTP/C also supports the use of central bus guardians. Flexray uses local bus guardians.

TTP/A provides real-time communication, but no fault tolerance. It is the only protocol that is only available as software implementation. LIN can be also used for real-time communication, but lacks the functionality of a global time. These two protocols are compared in detail in [EK04].

CAN is the only protocol that is not suitable for real-time communication. But a real-time communication capable protocol (TT-CAN) can be built with adapted hardware and an additional software layer.

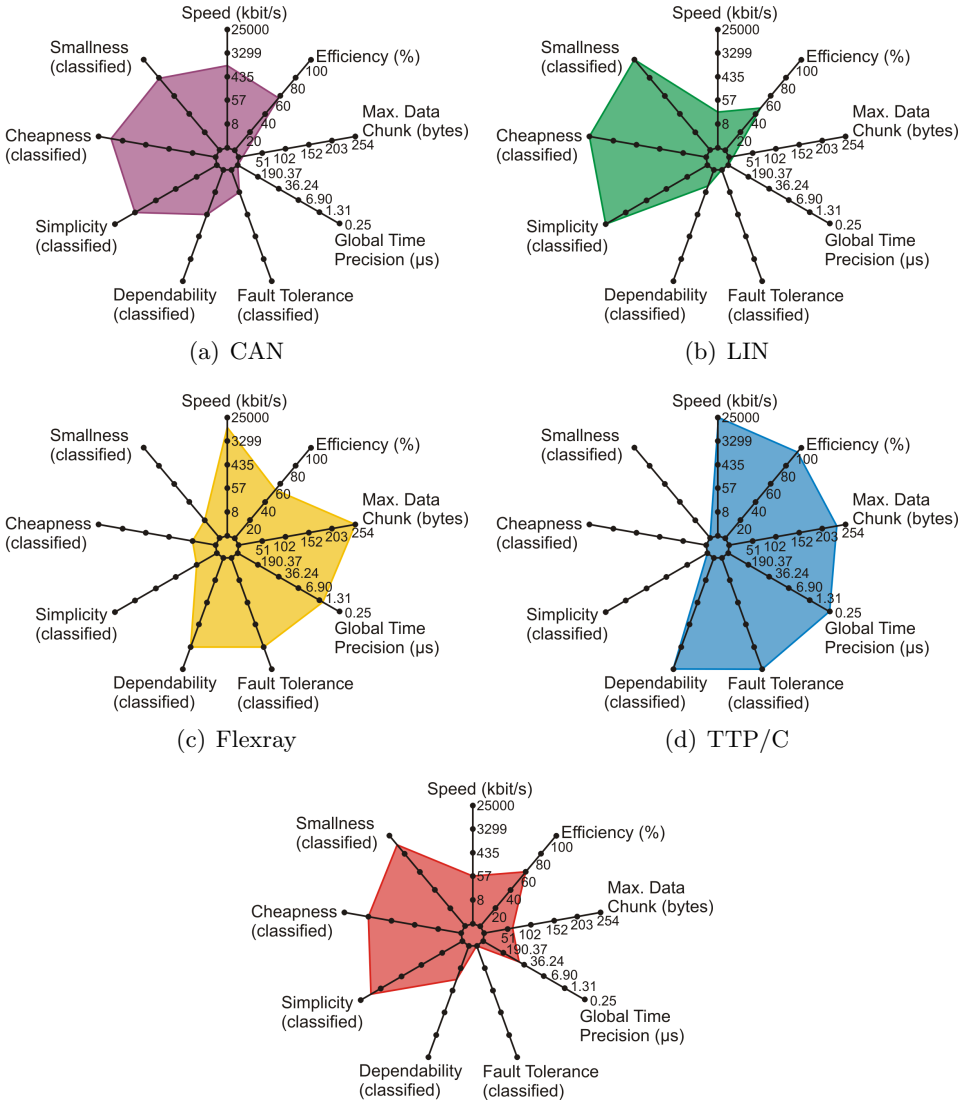
	CAN	**LIN**	**Flexray**	**TTP/C**	**TTP/A**
gross comm. speed	1 Mbit/s	20 kbit/s	10 Mbit/s	typical 25 Mbit/s [18]	typical 19,2 kbit/s [18]
medium	twisted pair	ISO k-line	twisted pair	typical 100Mbit-Ethernet [18]	typical ISO k-line [18]
topology	bus	bus	bus, star (a/p), mixed	bus, star (active), mixed	bus
max. number of nodes	> 100	16	64	depends on physical layer	depends on physical layer (max. 255)
medium access	CSMA/CA	Master/Slave	TDMA, Minislotting	TDMA	Master/Slave, TDMA
acknowledgement	explicit	no	no	implicit	no
fault tolerance	no	no	yes	yes	no
bus guardians	no	no	local	local or central	no
membership	no	no	supported	fault-tolerant	simple
clock sync.	no	no	$\leq 0.25\mu s$ @10Mbit/s	$\leq 1\mu s$ @10Mbit/s	$\leq 52\mu s$ @19.2kbit/s
determinism	no	yes	yes (static seg.)	yes	yes
max. payload	8 bytes	8 bytes	254 bytes	236 bytes	1 byte
frame protection			CRC-24 (+CRC-11)	CRC-24	1 parity bit/byte
efficiency	$< 57.66\%$	$< 46.08\%$	$> 29.14\%$, $< 77.73\%$	$< 95.8\%$	$< 59.62\%$
availability	HW, IP	HW, IP, SW	HW, IP	HW, IP	SW
size	integrated	integrated, 3.1 kbyte ROM / 648 byte RAM[Atm05]	LQFP64[19]	LQFP80[19]	2784 bytes ROM / 63 bytes RAM[Trö02]
cost	IC or IP licence	IC or licence	IC or IP licence	IC or IP licence	free

[18]The physical layer is not in the scope of the protocol specification.

[19]Low Profile Quad Flat Package with 64 and 80 pins respectively.

Table 2.3: Comparison of Protocol Characteristics

Figure 2.15 shows a diagram that compares the most important attributes of the protocols. The attributes have been selected accordingly to the requirements that are put on a communication system when it is used on a small autonomous system.



(a) CAN (b) LIN

(c) Flexray (d) TTP/C

(e) TTP/A

Figure 2.15: Comparison of the Protocols

Speed The comparison of the gross speed of the protocols shows that Flexray and TTP/C are high speed protocols whereas LIN and TTP/A are used for low speed applications. CAN also allows high speed communication

but far not as fast as Flexray and TTP/C. Because of the large differences of the speed of the protocols a logarithmic scale has been used for speed in the diagrams in figure 2.15.

Efficiency The most efficient protocol is TTP/C, which includes important protocol data only in the calculation of the CRC. Thus, the current state of the nodes can checked without transmitting any extra bytes. The efficiency of CAN and TTP/A is comparable to the average efficiency of Flexray. But in many application the efficiency of Flexray exceeds that of CAN and TTP/A. The polling technique reduces the efficiency of LIN.

Max. Data Chunks The maximum length of data chunks that can be transmitted without fragmentation is limited to eight bytes in CAN and LIN. TTP/A solely uses frames with the length of a single byte. However, up to 62 bytes can be transmitted consecutively, because no additional protocol information is stuffed in the frames. Flexray and TTP/C allow the transmission of up to 254 and 236 bytes respectively in a single data frame.

Global Time CAN and LIN do not provide a global time for synchronizing action in the cluster and for creating timestamps that are valid on all nodes of the cluster. The precision of the global time in a TTP/A cluster is not as accurate as that of the global time in a Flexray or TTP/C cluster. The two latter protocols allow a clock synchronization with a precision better than one microsecond. Because of the large differences of the precision of the global time in Flexray, TTP/C, and TTP/A clusters the logarithm of the reciprocal precision has been used for the diagram in figure 2.15.

Fault Tolerance Flexray and TTP/C are the only two communication systems that are fault-tolerant. They have replicated communication channels, fault-tolerant start-up, and clock synchronization algorithms and bus guardians. A specification of a transparent fault tolerance layer is available for TTP/C. The Flexray protocol leaves the design of such a layer to the developer of the application.

Dependability Flexray and TTP/C have been designed to provide a high dependability. Especially the algorithms used by TTP/C have undergone numerous tests and have been verified formally. CAN has been approved for many years and has be integrated in billions of products. TTP/A has not been thoroughly tested in everyday life, but its design addresses the issue of dependability. LIN is used for convenience functions in the automotive industry and dependability is not a main design criteria of LIN.

Simplicity LIN is a very simple protocol. Especially a slave implementation can be realized with very little effort. The implementation of TTP/A is more sophisticated, but the effort is still in the same magnitude. The medium access technique of CAN makes an implementation more elaborate. Flexray and TTP/C are far more complex than the three other protocols.

Cheapness The cost of using the protocols in an application is comparable to their simplicity. The simpler the protocol the cheaper is its use. However, apart from mass production this is not true for CAN. Because of the high number of applications that use this protocol, CAN controllers are very common and cheap.

Smallness LIN and TTP/A can be implemented in software occupying some hundred bytes of FLASH memory. CAN controllers are integrated on many microcontrollers and no additional space on the Printed Circuit Board (PCB) is needed. In contrast, Flexray and TTP/C are only available as separate communication controllers. Some implementations are even realized in large Field Programmable Gate Arrays (FPGA).

When I'm working on a problem, I never think about beauty. I think only how to solve the problem.
But when I have finished, if the solution is not beautiful, I know it is wrong.

R. Buckminster Fuller (1895 - 1983)
US Architect and Engineer

Chapter 3

Target System – Tinyphoon Robot

The Tinyphoon[NM05] is a small autonomous mobile robot in the shape of a cube with a side length of about seven centimeters. It has been developed at the Center of Excellence for Autonomous Systems of the Vienna University of Technology by a team led by Gregor Novak and Stefan Mahlknecht as a research platform for small autonomous robots. New concepts can be developed and approved in the challenging field of robot soccer. Its design is modular, various modules can be stacked one over another (see figure 3).

Figure 3.1: The Tinyphoon Robot

These modules are interconnected with a bus and are carried by a me-

chanical system that has been designed for playing robot soccer. This mechatronic part is about three centimeters high and is equipped with two strong DC motors. Moreover, the rechargeable battery for powering the Tinyphoon is stored here.

Three electronic modules have been designed up to now: a Vision Unit, a Decision Making Unit and a Motion Unit controlling the locomotion. Additional modules are planned to host extra sensors or provide computing power. The current architecture of the Tinyphoon robot is shown in figure 3.

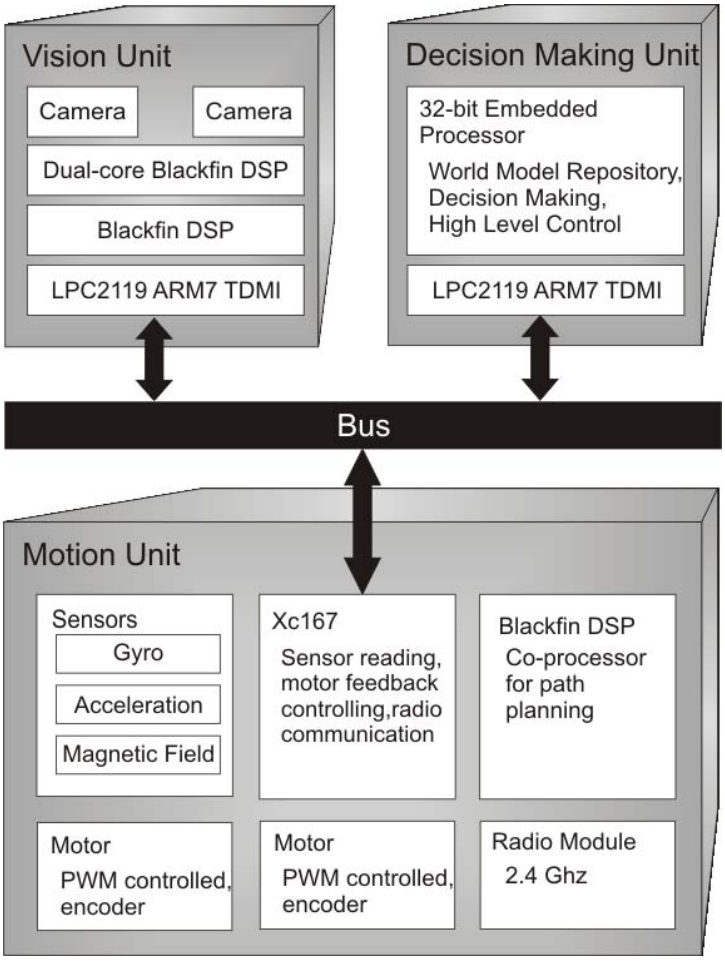


Figure 3.2: System Architecture of the Tinyphoon Robot

3.1 Subsystems

In order to play soccer a robot has to collect data about its environment, to make strategic decisions upon this information and to act accordingly. Each of these capabilities is a complex task. To handle this complexity modularity has been one of the most important principles for the design of the Tinyphoon. There is a separate and exchangeable module for each capability, namely the Vision Unit module for performing image recognition and visual self localization, the Decision Unit module for making strategic decisions and the Motion Unit module for path planning and for controlling the motors of the wheels. The hardware of the modules differs heavily.

3.1.1 Motion Unit

In figure 3.1.1 the Motion Unit of the Tinyphoon is shown. The main task of this unit is the feedback control of the motors. A XC 167 processor[Sie06](marked in the figure with letter a) from Infineon[1] is used for this task because of its outstanding peripherals. It also processes the signals from the encoders (512 impulses per rotation) of every motor, monitors the supply voltage from the battery pack and reads the sensors mounted on the Motion Unit: an analog gyro sensor with built-in thermometer (letter b), an analog magnetic field sensor (letter c) and two two-axis acceleration sensors with Pules Width Modulation (PWM) output (letter d).

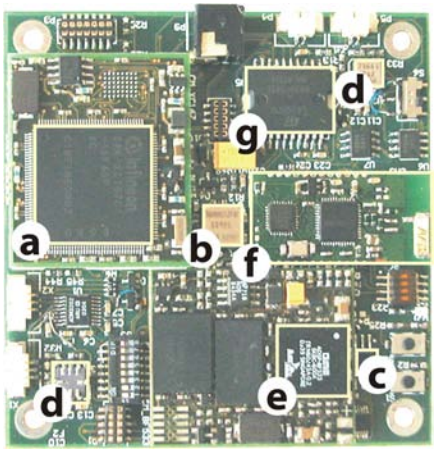


Figure 3.3: The Motion Unit

A Blackfin 531 Digital Signal Processor (DSP) (letter e) acts as coprocessor. It delivers the computing power needed for path planning [SJ05][SJN05]. In earlier versions of the Tinyphoon the DSP was also used for running a ball

[1] `http://www.infineon.com`

detection algorithm[MON05]. It is connected with the XC 167 via a fast SPI connection.

Moreover, the Motion Unit is equipped with a radio module (letter f) that allows exchanging information with other Tinyphoon robots and receiving commands from a personal computer.

The motor driver (letter g) controls the supply voltage of the motors accordingly to the PWM signals from the XC 176.

The communication interface to other modules will be implemented on the XC 167, there is no dedicated communication controller.

XC 167CS The XC 167CS is a 16-bit automotive microcontroller with 5 pipeline stages and runs at 40 Mhz. It has multiple register banks allowing fast context switches. It has several powerful peripherals that support a kind of Direct Memory Access (DMA) mode. Some of the main features of the XC 167CS are listed below:

- 6 kbyteon-chip Static Random Access Memory (SRAM), 128 kbyteon-chip Flash Program Memory, 512 kbyteexternal Random Access Memory (RAM)
- One cycle 16bit multiplication and one cycle Multiply and Accumulate (MAC) instruction
- 16 channel Analog Digital Converter (ADC) (10-bit or 8-bit, conversion Time down to 2.55μs)
- 16 programmable Interrupt priorities for 77 sources
- Five multifunctional timers/counters (with the functionality of a two channel decoder)
- Two 16-channel capture/compare units and flexible PWM signal generation
- Two synchronous/asynchronous serial communication channels (USART), two high-speed synchronous serial communication channels
- Dual CAN Interface
- I^2C bus interface

Blackfin coprocessor The Blackfin processor is connected to the XC 167 via a Serial Peripheral Interface (SPI) interface. It runs at 600 MHz and works as coprocessor. The Blackfin processor from Analog Devices [2] is a

[2] www.analogdevices.com

mixture of a microprocessor and a DSP. Therefore, it is perfectly suited for number crunching applications like path planning.

Sensors A gyro sensor detects the rotation of the robot and two orthogonally mounted acceleration sensors sense the change of velocity in each direction. The magnetic field of the earth is also measured, but the results of this sensor are disturbed because of the strong magnetic fields of the electro motors. 2 magnetic encoders on the wheels with 512 pulses per rotation

3.1.2 Vision Unit

The Vision Unit subsystem is shown in figure 3.1.2. The Vision Unit enables the robot to get an overview about its environment. Two CMOS cameras with a resolution of 640 x 480 take pictures simultaneously. Each of them is connected to a separate core of a dual-core Blackfin 561[Dev06] DSP from Analog Devices[3] (marked with letter h). The two cores perform edge and color blob detection. The gathered data is then sent to a single-core Blackfin 537 DSP (letter i) via SPI[4], where the two-dimensional edges are combined and three-dimensional lines are calculated. Based on the blobs and the lines object recognition can be performed. More details about the vision system can be found [BAS+06].

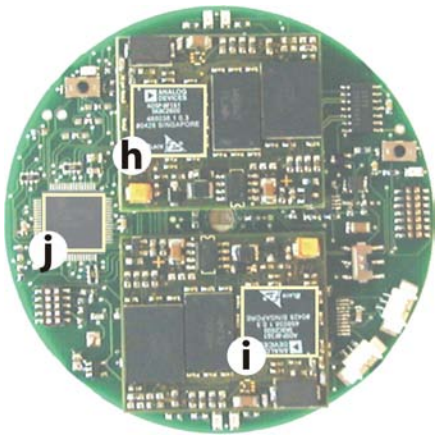


Figure 3.4: The TinyVision subsystem

The complete subsystem is mounted on a head that can be rotated by a stepper motor. This motor is controlled by a LPC 2119 ARM7-TDMI processor[Phi06] (letter j) that also functions as a communication interface to the other subsystems. It is connected to the single-core DSP via SPI.

[3] `http://www.analog.com`
[4] Serial Peripheral Interface

3.1.3 Decision Making Unit

An embedded 32-bit processor provides the necessary computation power for decision making. A built-in floating point unit supports algorithms based on fuzzy logic or neuronal networks and various filter algorithms. Linux or Windows CE can be used as operating system on this platform. The processors of the MCP family with Power architecture from Freescale[5] are currently evaluated regarding its use on the Decision Making Unit. S fuzzy logic framework for the Decision Unit has already been implemented [Web06].

Two tasks are assigned to the Decision Making Unit:

1. It merges the data received from the sensors on the Motion Unit, from the vision unit and from other robots to create a world model.

2. Decisions are made based upon the interpretation of this world model and a predefined strategy.

Integrating a communication system with high real-time requirements as part of such a complex software system is only possible using special operating systems (e.g. Real-Time Linux). Therefore, we decided to integrate a LPC 2119 ARM7-TDMI processor for handling the communication.

3.2 Communication Requirements

This section analyzes the data to be exchanged and additional attributes of the communication system.

3.2.1 Data Provided/Needed by the Subunits

Tables 3.1, 3.2, and 3.3 show the data that needs to be communicated by each module with a description of the data, its length, the transmission direction (input or output) and the update interval.

Table 3.1 lists the data that is sent and received by the Vision Unit. The Vision Unit has to be configured with the description of the objects that have to be recognized. The more different types of objects are to be searched for in the image the slower runs the detection algorithm. A maximum of eight object descriptions can be processed at a time. But these descriptions can be changed for every new image. A list of the recognized objects is then transmitted by the Vision Unit. Moreover, the to-be and the current position of the rotatable head and and the result of the visual self-localization has to be communicated.

[5] `http://www.freescale.com`

Description	Len	Dir	Interval
object descriptions	$8 \cdot 32$	I	20ms
detected objects	$16 \cdot 8$	O	20ms
to-be head position	2	I	20ms
last head position	2	O	20ms
self-localization	6	O	20ms

Table 3.1: Input/Output Data of the Vision Unit

The communication requirements of the Motion Unit are shown in table 3.2. It receives a command specifying the action to be performed with the necessary parameters. The Motion Unit provides the data collected from the sensors as well as commands and data from other robots that is received via the wireless link.

Description	Len	Dir	Interval
command	12	I	20ms
position delta	6	O	20ms
gyro sensor	2	O	20ms
acceleration sensor	2*2*2	O	20ms
magnetic field	2	O	20ms
temperature sensor	1	O	1000ms
supply voltage	1	O	1000ms
remote command	32	O	20ms
remote response	32	I	20ms
remote robot data	3*64	O	20ms

Table 3.2: Input/Output Data of the Motion Unit

The communication requirements of the Decision Unit are shown in table 3.3. The results of the object recognition of the Vision Unit, data received via the serial link and the information from the sensors on the Motion Unit are combined with the knowledge about the environment (e.g. the football ground, the soccer game) to a more reliable world model.

3.2.2 Real-Time Requirements

The main sensor of the Tinyphoon is the camera system. The maximum update frequency of the Vision Unit is 50 Hz. This frequency is also used for scheduling communication cycles. A trigger for the communication may be received via the wireless link to synchronize the robots. This is important,

Description	Len	Dir	Interval
current position	6	O	20ms
teammates	$3 \cdot 6$	O	20ms
opponents	$4 \cdot 6$	O	20ms
ball	4	O	20ms
goals	$2 \cdot 6$	O	20ms
boards	$4 \cdot 6$	O	20ms

Table 3.3: Input/Output Data of the Decision Unit

because information about recognized objects is exchanged among the robots and this is much more useful if the robots rate exactly the same situation.

The maximum speed of the Tinyphoon robot is 4 m/s. Therefore, an object may move up to $2 \cdot 8$ cm from one image to another (e.g. two Tinyphoon robots driving with full speed in opposite directions). State estimation can be used to calculate the likely position of objects in the near future. However, this requires that the point in time when the picture is taken is known exactly.

If an error of ≤ 1 mm is considered as negligible, the overall jitter (including the jitter of the communication) has to be smaller than 125 μs.

3.2.3 Fault Tolerance / Dependability Requirements

Every functional unit exists only once on the Tinyphoon. Therefore, there is no fault tolerance on level of functional units. This is not a serious problem because a fail safe state exists, the robot can stop all motors at any time.

The communication system is partly integrated on functional units (e.g. the XC 167 microcontroller on the Motion Unit) and can not be replicated to provide redundancy.

Moreover, all subsystems depend on each other. The Tinyphoon can not continue operation when a subsystem fails. However, the failure of a subsystem or of the communication system must be detectable for every other unit to change to a safe state.

The length of the communication wires amounts only in a few centimeters and using the physical layer with differential signal transmission protects the data on the bus against electromagnetic disturbance (e.g. from the motors).

The small size of the Tinyphoon robot guarantees that no human and no machine could be harmed in case of a failure. Hence, features related to fault tolerance need not be considered when a communication protocol for the Tinyphoon robot is selected.

3.2.4 Data Throughput

Table 3.4 shows the number of bytes that need to be transferred for every subsystem.

Subsystem	**Length**	**Interval**
Vision Unit	394	20ms
Motion Unit	288	20ms
Decision Unit	88	20ms
Total	**770**	20ms

Table 3.4: Tinyphoon Communication Requirements

770 bytes need to be transferred every 20ms.

$$\frac{770\ byte}{20\ ms} = 15400\ byte/s \tag{3.1}$$

TTP/A needs thirteen bits (one bit inter-byte gap, one start bit, eight data bits, one parity bit, one stop bit and again one bit inter-byte gap) to encode one byte on the bus.

$$15400\ byte/s \equiv 204100\ bit/s \tag{3.2}$$

Considering the protocol overhead and future expansions, a communication speed of at least 0.25 Mbit is required.

3.2.5 Maintainability

Maintainability is one of the main issues for the communication system of the Tinyphoon robot. New subsystems will be created and the old ones are modified continuously. Hence, the communication system has to be adaptable to new hardware platforms and new fields of application. The same code has to be reused on many platforms to confine the effort of changing the protocol.

3.2.6 Debugging and Monitoring

As the Tinyphoon is a research project many people work on various topics around the robot. Hence, debugging facilities are one of the crucial features the communication system is required to provide. On the one hand the communication itself has to be debugged on the other hand the interfaces and the functioning of the subsystems have to be tested using the communication system's debugging facilities.

For software development a Joint Test Action Group (JTAG) interface[IEE90] exists for every microcontroller or processor. However, JTAG debugging interfaces are not compatible for devices from different vendors. Thus, a mechanism is needed to access the interface of every subsystem.

In the future we also plan to simulate parts of the Tinyphoon in software for testing purposes. Hence, a fast real-time communication between the simulation host (typically a standard personal computer) and the respective Tinyphoon hardware will be required.

3.2.7 Cost

The latest technologies are used to build the Tinyphoon, which is very cost-intensive. However, the cost for particular development tools turned out to be much more expensive than the actual hardware. Therefore, the availability of free or low priced development tools and documentation is vitally important for future extensions.

3.2.8 Implementation Effort

Currently various platforms are used on the Tinyphoon. Therefore, the communication system has to be integrated on each of this platforms. A small implementation effort allows the fast adaption of the communication system to new hardware platforms.

3.2.9 Comprehensibility of Interfaces

On large microcontrollers or processors a communication layer is introduced. So only a few developer get in touch with the interface of the communication system directly. Application programmers can use an API to access the functionality of the communication system. As the Tinyphoon is a very heterogeneous system a comprehensible interface makes it easier to port this API to other platforms.

It requires a very unusual mind
to undertake the analysis of the obvious.

Alfred North Whitehead (1861 - 1947)
English Mathematician and Philosopher

Chapter 4

Problem Analysis

This section analyses existing implementation of the communication on the Tinyphoon robot and discusses the suitability of the real-time protocols described in chapter 2 for the use on this autonomous system.

4.1 Current Communication

Currently, two modules for the Tinyphoon are available: the Motion Unit and the Vision Unit module. The two modules originally communicated using the CAN protocol. The communication was performed by the integrated CAN controllers of the XC 167 on the Motion Unit and of the LPC 2119 on the Vision Unit. The LPC 2119 has been added to the schematics of the Vision Unit to perform the rotation of the head and all communication related tasks, whereas the XC 167 has to perform many other, partly time critical tasks: reading the sensors and the encoders, generating the PWM signals for the motors, sending and receiving data through the wireless link and communicate with its co-processor.

The hardware of the modules has been designed in a way that allows connecting a CAN transceiver either to the I/O pins of the integrated CAN controller or to the I/O pins of the UART of the LPC 2119 and the XC 176, respectively. This guarantees flexibility and allows the software implementation of a communication protocol. Thus, a communication protocol that can be implemented in software and that uses the serial ports can be integrated on the Tinyphoon without having to modify the hardware.

For the FIRA World Championship in Summer 2006 a stripped-down version of the Tinyphoon has been created. It consists only of a motion and a vision unit. These two modules communicate with a very simple protocol, based on

standard UART frames. There is no mechanism for bus arbitration required because the connection is bidirectional. However, the communication system turned out to be one of the most error-prone subsystems. Problems with the interpretation of the semantic and the structure could be solved by rendering the specification of the protocol more precisely. But the reception of messages asynchronously to the program execution caused unexpected problems that were hard to track down and solutions were quite complex. Moreover, this approach is not applicable for systems with more than two communication partners.

4.2 Features Versus Complexity

Flexray and TTP/C offer by far the most features related to real-time and fault tolerance. The communication controllers have interfaces with several communication parameters that have to be configured extensively to establish a communication. This can hardly be accomplished without using special tools, like cluster designers. These tools are very expensive. The fault-tolerant features of Flexray and TTP/C would lead to an unnecessary overhead since the intended application does not require fault-tolerance. Moreover, with a small number of nodes as on the Tinyphoon both communication systems only run in a degraded mode that is not really fault-tolerant.

CAN, LIN, and TTP/A provide less features but the configuration and the use of the communication interfaces are much simpler. Anyway, TTP/A even provides a simple membership service and a global time that is required to synchronize tasks in different subsystems.

4.3 Hardware Versus Software Implementation

Small systems like the Tinyphoon put strong restrictions on the design of the PCB. An external communication controller needs additional space on the PCB. It may not be possible on all systems to provide this space. However, on the Vision Unit the LPC 2119 microcontroller is only used for the generation of a PWM signal for rotating the head and for the communication. The PWM signal could also be generated on one of the Blackfin DSP. The MF4300 Flexray controller has about the same size as the LPC 2119. Thus, the communication controller could be used instead of the microcontroller.

On the Motion Unit the XC 167 performs many tasks and can not be replaced by a communication controller. Because of the high number of sensors and the necessary analog circuitry there is not enough space for an additional communication controller without a complete redesign of the

Motion Unit.

Software implementations of communication systems require additional calculation time and peripherals. On the LPC 2119 of the Vision Unit enough resources are available. The XC 167 on the Motion Unit is also equipped with enough peripherals and plenty of calculation time is available, because calculation intensive tasks are delegated to the coprocessor. However, the XC 167 already has to perform some time-critical tasks. These have to be combined with the requirements of the communication system. One possibility is that the scheduler of the communication also triggers all the other time critical tasks.

4.4 Real-time Communication in Software

This section discusses the feasibility of the software implementation of time-triggered CAN, LIN and TTP/A on the Tinyphoon hardware.

4.4.1 Time-Triggered CAN

The most obvious idea is to use a software layer above CAN that makes the communication predictable and fits the needs of real-time applications by bypassing the CSMA/CA bus access schema. TDMA and clock synchronization can be used to avoid collisions on the bus completely. TT-CAN follows this strategy.

This approach uses the integrated CAN controllers for handling the communication. Only the timing is controlled by the software layer, which reduces the effort of the implementation. However, TT-CAN implementations need an adapted CAN controller that supports the handling of timing information in hardware. Each node has to synchronize itself on the reference message of the master node. The controller has to measure the start of the message exactly and supply the software layer with this information[HMFH00]. Usually, CAN controllers only inform the software when a message has been received. This technique does not allow a precise clock synchronization. Thus, the communication is not efficient because long breaks between two messages have to be scheduled and the temporal coordination of actions in the subsystems is hardly possible.

Another critical point is the implementation effort on the LPC 2119 and the XC 176 platform respectively. To our knowledge, currently no such implementation is available, least of all a free one.

4.4.2 LIN

LIN is designed to be implemented on small microcontrollers with a standard UART. Both microcontrollers a perfectly adequate for implementing LIN. Moreover, a lot of source code for various LIN implementations is available on the Internet. Moreover, the compatibility of development tools is guaranteed by the LIN specification that defines file formats for the description of complete LIN clusters and single LIN nodes. Thus, application specific tools can be developed that interface with off-the-shelf LIN configuration tools.

The major disadvantage of LIN is the lacking of a global time. Thus, LIN can be used for real-time communication but not for synchronizing actions within the cluster. As argued above, this is a crucial feature of a communication system that is used on an autonomous system like the Tinyphoon.

The slow nominal communication speed is an additional, but minor problem.

4.4.3 TTP/A

TTP/A is very efficient (see table 2.3) and the source code of an open source implementation in C[1] exists, which can be used for free, even for commercial applications. TTP/A offers clock synchronization and a comprehensible application interface, the IFS. The global time allows the the generation of time stamps that are valid in the whole cluster.

Hence, TTP/A fulfils all major requirements for the communication on the Tinyphoon. However, there are some issues that require a closer investigation:

- The performance has to be boosted. The transmission is byte-oriented with a synchronization event between every two bytes, limiting the performance and the efficiency.
- In the current implementation, the error handling of messages has to be done by the application, increasing the complexity of the application software.
- The existing implementation is designed for eight bit microcontroller from Atmel[2]. The effort for porting the code to other platforms (16- and 32-bit platforms) has to be evaluated.
- The C implementation of TTP/A uses a software UART, but only

[1] available at `http://www.vmars.tuwien.ac.at/ttpa`
[2] `http://www.atmel.com`

pins with hardware UART functionality are connected to the bus transceivers on the modules of the Tinyphoon.

4.5 Results

The discussion in this chapter has shown that the lack of space on the PCB does not allow the use of an external communication controller and that the multitude of features of Flexray and TTP/C makes their interfaces very complex. Thus, a software implementation of a simpler protocol is suggested.

The implementation of time-triggered CAN would require extra hardware support that currently is not available on the Tinyphoon robot. LIN can be implemented with small effort, but lacks features like a global time. Moreover, the frame length of CAN and LIN is limited to eight bytes. Thus, long pieces of data have to be fragmented into several frames. The fragmentation and defragmentation produces an additional overhead in the complexity of the software implementation and the execution of the communication.

TTP/A is more efficient than CAN and LIN. It features a global time and the byte-wise transmission of the payload allows the exchange of messages up to 62 bytes without any fragmentation.

Due to these reasons we decided to evaluate TTP/A on the hardware of the Tinyphoon.

There are three principal means of acquiring knowledge ...
observation of nature, reflection, and experimentation.
Observation collects facts; reflection combines them;
experimentation verifies the result of that combination.

DENIS DIDEROT
French Author and Philosopher (1713 - 1784)

Chapter 5

TTP/A on the Tinyphoon

This chapter describes the techniques used to make the existing implementation of TTP/A portable and to adapt it for the Tinyphoon platform. Finally, the result of the implementation are evaluated and discussed.

5.1 Existing TTP/A implementation

The existing implementation of TTP/A was written by Christian Trödhandl[1]. In contrast to the first version that is realized in assembler and therefore is extremely hardware dependent, the current version is written in C. The TTP/A is targeted to the the AVR platform from the Atmel[2].

5.1.1 Source Code

The source was developed for the AVR enabled version of the GNU C Compiler (GCC). It uses some features of the GCC that are not compliant with the American National Standards Institute (ANSI) C standard[ANS89].

The source code is organized three subdirectories:

/src This directory contains the actual source code of the TTP/A protocol.

/include In this directory the header files are store. It has to be added to the application that wants to use the TTP/A protocol.

/ldscripts The linker scripts for the special memory sections of the protocol code are stored here.

The code is divided in six logical parts:

[1] available at `http://www.vmars.tuwien.ac.at/ttpa`
[2] `http://www.atmel.com`

Main File The main file `main.c` implements the entry point and initializes the protocol and the user application.

Scheduler In `schedule.c` a small scheduler is implemented that supports the prioritization of tasks. The scheduler can be deactivated.

Protocol Core The core of the protocol is implemented in `ttpa.c`. The files `ttpa_*.c` contain some core related functions.

MS round The handling of MSA and MSD rounds is implemented in the files `ms.c` and `ms_*.c`. The implementations of MS rounds for master and slave differ strongly. Thus, role specific parts are implemented in separate files.

IFS Functions for managing and accessing the IFS are provided in the file `ifs.c`. Helper functions are implemented in `ifs_*.c`. Two assembler files are used for the definition of the file look-up table (`ifs_tab.S`) and weak symbols for the files (`ifs_weak.S`).

Bus The bus related functions are implemented in the files `bus_*.c`. The original version of TTP/A provides a software UART only. Additional code is needed for accessing the bus in the slave implementation (`bus_*_slave.c`).

5.1.2 Architecture

Figure 5.1 shows a simplified Unified Modelling Language (UML) activity chart of the TTP/A slave implementation.

After the initialization the slave waits for a MSA round. It uses the periodical pattern of the MSA fireworks byte for synchronization (`bus_sync`). When the bus operation succeeds the received fireworks byte is handled (`ttpa_recvfb`). After bus synchronization the only possible firework byte is 0x55, the identifier of a MSA round. The new communication round is started (`ttpa_newround`) by selecting the RODL that is assigned to the round. Then the first frame is marked as current frame (`ttpa_next_frame`) and the scheduled action is read from the RODL (`ttpa_next_rodlentry`). The code execution is then continued depending on the current schedule entry:

Bus Receive Sync One or more bytes have to be received from the bus. The start of the transmission is used to resynchronize the node. The reception is prepared in the function `ttpa_recvsyncframe` by configuring the bus subsystem for the next receive operation (`bus_receivebyte_init`). The completion of the bus operation is signaled by the bus subsystem by calling the function, which the pointer

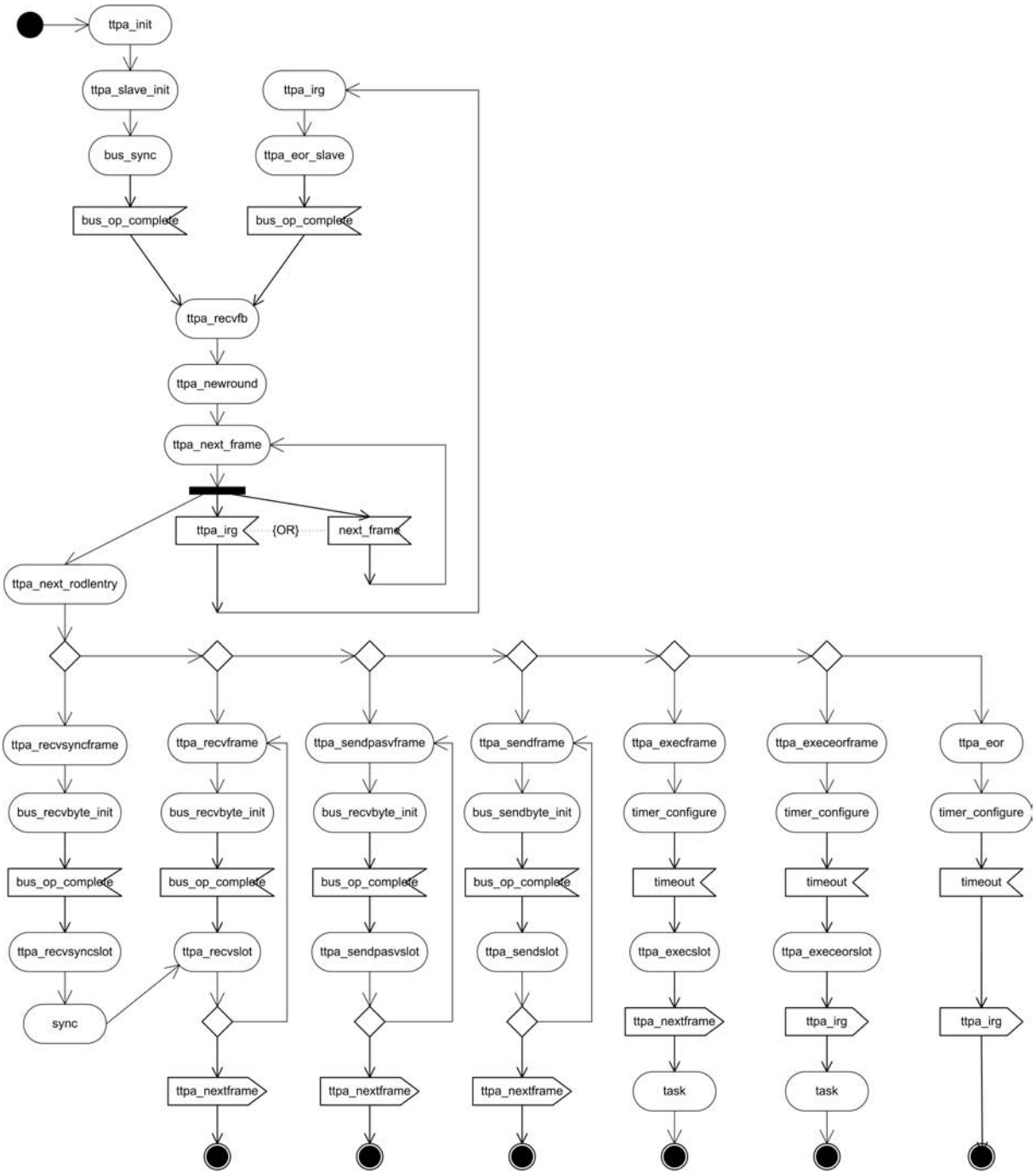


Figure 5.1: UML Activity Chart of the TTP/A Implementation

`bus_op_complete` points at. It has been set to `ttpa_recvsyncslot`. In this function the clock synchronization is performed and then the

handler for received bytes, `ttpa_recvslot`, is called. If there are more bytes that have to be received, the normal receive operation (see next paragraph) is performed. Otherwise, a new frame is started.

Bus Receive This operation is the same as the one described above, but no clock synchronization is performed.

Bus Send Passive This operation is performed in the send slots of the node when the node tries to integrate itself in the cluster. It is used at start-up or after an error. The node tries to receive something in its send slots to ensure that no other node uses these slots for transmissions. Thus, collisions due to wrong configurations can be avoided. First `ttpa_sendpasvframe` configures the bus subsystem (`bus_recvbyte_init`) for receiving and waits for the completion of this operation. Then `ttpa_sendpasvslot` is called. If the receive operation was not successful, the slot is empty and the node can use it for transmissions. If there are more slots to be tested, this operation is repeated, otherwise the handling of the next frame is started.

Bus Send Once the node has verified, that its send slot is free, it uses this operation to transmit data in the slot. In `ttpa_sendbyte_init` the bus subsystem is configured for a transmission (`bus_send_byte_init`). After the completion of the bus operation, `ttpa_sendslot` is called. If more bytes have to be transmitted, the send operation is repeated. Otherwise, otherwise the handling of the next frame is started.

Execute This operation executes a task that has either been defined by the application or by the protocol (e. g., for handling MS rounds). In the function `ttpa_execframe` a timer is configured to generate a time-out event at the start of the time slot. After the time-out the function `ttpa_execslot` is called. It triggers the preparation of the next frame and then performs the configured task.

Execute at the End of the Round Also the IRG can be used for executing tasks. This operation is almost the same as the normal execute operation but it signals the end of a communication round instead of preparing the next frame.

End of the Round If the communication round is finished and no task execution is scheduled for the IRG, tis operation is performed. In `ttpa_eor` a timer is configured to trigger the start of a new round at the appropriate point of time.

After the processing of the frame either a new frame or a new communication round is started. In the first case `ttpa_next_frame` is called and

the next scheduled action is read from the RODL (`ttpa_next_rodlentry`). In the second case the slave node executes the function `ttpa_eor_slave` that configures the bus subsystem for the reception of a new fireworks byte. As soon as the bus operation is finished, the fireworks byte is interpreted (`ttpa_fbrecv`) and the according communication round is started (`ttpa_newround`).

5.1.3 Bus Subsystem

The read and write operations on the bus are performed through the bus subsystem. This design decision makes it easy to integrate new implementations of the bus access. Listing 5.1 shows the interface of the bus subsystem that is defined in `bus.h`.

```
int bus_init(void);
void bus_sendbyte_init(bus_iobuf_t *param);
void bus_recvbyte_init(bus_iobuf_t *param);
void bus_sync(void);
void (* volatile bus_op_complete) (bus_iobuf_t *);
```

Listing 5.1: Interface of the Bus Subsystem

The interface defines function for initializing the bus subsystem and for initiating a transmission or a reception of a byte. On slave nodes an additional function for synchronizing with the bus has to be implemented. A pointer `bus_op_complete` has to be set to a function that is called after the bus operation has been finished. The configuration and the result of a bus operation is stored in the structure `bus_iobuf_t`. Listing 5.2 shows the fields of this structure.

```
typedef struct bus_iobuf_t {
  uint8_t buf;
  parity_t par;
  uint16_t slotstart;
  union status_t {
    struct fields_t {
      unsigned spdup   : 3;
      unsigned timeout : 1;
    } fields;
    bus_io_err_t error;
  } status;
} bus_iobuf_t;
```

Listing 5.2: Structure for Bus Operations

The field `uint8_t buf` stores the byte that has to be transmitted or that has been received. The field `parity_t par` is an enumeration type that specifies the parity. The number of the microtick when a byte has to be transmitted or has been received is stored in `uint16_t slot_start`. The union `status` stores a speed-up factor and a timeout flag, when the structure is used for the configuration of a bus operation, or an error field when the structure is used for returning the result of an operation.

5.2 Hardware UART Implementation

The TTP/A implementation originally uses a UART that is implemented in software. This ensures a very precise synchronization because an input capture logic is used to detect the start of a transmission with the precision of one internal micro tick. However, the software implementation needs one interrupt for every bit. This leads to a high interrupt load even at small bit rates and influences coexisting applications on the same microcontroller. Thus, the bus interface has also been implemented using a hardware UART. Due to the schematics of the Tinyphoon's modules only this implementation can be used for the communication.

The source code shown in the next sections 5.2.1 and 5.2.2 has been designed to work with the Hardware Abstraction Layer (HAL) proposed in section 5.3.2. All hardware specific parts are replaced by the call of a macro.

5.2.1 Receiving

The code for receiving a byte from the bus is divided in four functions and a interrupt service routine.

The first function is part of the bus interface and is called by the TTP/A core code to initiate the reception of a byte in the next time slot. The function `bus_recvbyte_init` has a parameter of the type `bus_iobuf_t *` (see listing 5.3). It stores the time-out of this parameter in its own data structure, sets the global pointer for the timer callback to the setup function and configures the timer to raise an interrupt on the start of the next slot. `HAL_HW_UART_RECV_SU_CORR` can be used to correct a possible delay of the hardware UART. Then, the function `bus_transcvr_recv` switches a possible transceiver to receive mode, if needed. The UART receive interrupt is still disabled.

```
void bus_recvbyte_init(bus_iobuf_t *param)
{
  bus_hwuart_buf.timeout =
```

```
        param->status.fields.timeout;

  ttpa_sig_oc = (void (*)()) bus_recvbyte_setup;

  hal_set_timer_compare_value(param->slotstart +
       HAL_HW_UART_RECV_SU_CORR + ttpa_bitlength);
  hal_delete_compare_match_interrupt();
  hal_enable_compare_match_interrupt();

  bus_transcvr_recv();

  hal_uart_rx_disable();
}
```

Listing 5.3: HW UART Receive Initialization

On the start of the time slot `bus_recvbyte_setup` is called (see listing 5.4). The expected transmission starts one bit time after the call of this function. First the reception of bytes is enabled and the output compare logic is configured to generate a match event 12.5 bit times after the start of the slot. Following, the call back that is called, when a byte is received by the UART, is set to `bus_recvbyte`.

The bus subsystem has to implement two modes of byte reception, one with time-out and another without a time-out. In the case that a time-out is used, the pointer for the callback of the compare match interrupt is set to `bus_recvbyte_to` and the interrupt is enabled. Otherwise, the interrupt is disabled.

At the end of the function pending UART receive interrupts are cleared and the interrupt is enabled.

```
void bus_recvbyte_setup(void)
  hal_uart_rx_enable();
  hal_inc_timer_compare_value(12 * ttpa_bitlength +
                              ttpa_bitlength>>1);
  bus_uart_recv = bus_recvbyte;

  if(bus_hwuart_buf.timeout) {
    ttpa_sig_oc = (void (*)()) bus_recvbyte_to;
    hal_delete_compare_match_interrupt();
    hal_enable_compare_match_interrupt();
  } else {
    hal_disable_compare_match_interrupt();
  }
  hal_delete_uart_receive_interrupt();
```

```
  hal_enable_uart_receive_interrupt();
}
```

Listing 5.4: HW UART Receive Setup

If a byte has been received successfully, then `bus_recvbyte` is called to handle it (see listing 5.5). The UART receive interrupt is disabled and a structure `bus_iobuf_t` is prepared to be passed to the `bus_op_complete` callback. In this structure the time of the start of the transmission, a possible error during the transmission, the received byte and its parity are stored. Afterwards, the reception of bytes is disabled. This has to be done after the received byte and its parity have been read, because this information may be lost, when the UART is disabled. Finally, the `bus_op_complete` callback is called.

```
void bus_recvbyte(void)
{
  bus_iobuf_t buf;
  hal_disable_uart_receive_interrupt();
  buf.slotstart = recv_time - (12*ttpa_bitlength);
  if (hal_uart_check_error()) {
    buf.status.error = BUS_IO_FE;
  }
  else {
    buf.status.error = BUS_IO_OK;
  }
  buf.par = hal_uart_get_parity();
  buf.buf = hal_uart_get_value();

  hal_uart_rx_disable();

  (*bus_op_complete)(&buf);
}
```

Listing 5.5: HW UART Handle Received Byte

In case that a time-out occurs, the function `bus_recvbyte_to` (see listing 5.6) is called by the compare match interrupt service routine. Again a `bus_iobuf_t` is prepared. The error field is set accordingly to signal a time-out. Then both, the compare match and the UART receive interrupts are disabled and the callback `bus_op_complete` is called.

```
void bus_recvbyte_to(void)
{
  bus_iobuf_t buf;
```

```
  buf.status.error = BUS_IO_TIMEOUT;

  hal_delete_compare_match_interrupt();
  hal_enable_compare_match_interrupt();
  hal_disable_uart_receive_interrupt();

  (*bus_op_complete)(&buf);
}
```

Listing 5.6: HW UART Receive Time-Out

The last piece of the hardware UART implementation for the reception of bytes is the interrupt service routine for the UART receive interrupt (see listing 5.7). It stores the time of the reception of the byte and adds a correction value to consider the different delays of different hardware UART. Then the callback, which the pointer `bus_op_complete` points at is called.

```
hal_uart_receive_interrupt()
{
  recv_time = hal_get_timer();
  recv_time += HAL_HW_UART_RECV_CORR;
  (*bus_uart_recv)();
}
```

Listing 5.7: HW UART Receive Interrupt

5.2.2 Sending

The implementation for sending bytes is realized in two functions. The first one is called by the TTP/A core to configure the transmission of a byte for the next time slot. The second is used internally as callback for the timer, when the transmission starts.

The function `bus_sendbyte_init` (see listing 5.8) is part of the bus interface. It initiates the transmission of a byte at the beginning of the next time slot. The byte to be transmitted, its parity and the start of the next time slot is passed to the function in a `bus_iobuf_t` structure.

The parameters for the transmission are copied to local data structures and the callback for the compare match interrupt service routine is set to `bus_sendbyte`. Then, the compare value is set accordingly, so that the compare match interrupt is raised exactly when the transmission has to be started. Possibly pending compare match interrupts are deleted and the interrupt is enabled. Finally, a possible transceiver is switched to transmit mode.

```
void bus_sendbyte_init(bus_iobuf_t *param)
{
  bus_eff_bitlen = ttpa_bitlength >>
                   param->status.fields.spdup;

  bus_hwuart_buf.buf = param->buf;
  bus_hwuart_buf.par = param->par;

  ttpa_sig_oc = (void (*)()) bus_sendbyte;

  hal_set_timer_compare_value(param->slotstart +
                              HAL_HW_UART_SEND_CORR);

  hal_delete_compare_match_interrupt();
  hal_enable_compare_match_interrupt();

  hal_transceiver_send();
}
```

Listing 5.8: HW UART Initialize Transmission

The function `bus_sendbyte` (see listing 5.9) is called when the transmission has to be started. The parity mode of the hardware UART is configured and the data byte is transmitted. Then a `bus_iobuf_t` structure is prepared. The start of the next slot is set to 13 bit times after the start of the last slot. The error field is set accordingly to signal a successful transmission. Finally, the `bus_op_complete` callback is called and the structure is passed as parameter.

```
void bus_sendbyte(void)
{
    bus_iobuf_t buf;

    hal_uart_set_parity(bus_hwuart_buf.par);
    hal_uart_send_byte(bus_hwuart_buf.buf);

    buf.slotstart = hal_get_timer_compare_value() +
                    bus_eff_bitlen * 13;
    buf.status.error = BUS_IO_OK;

    (*bus_op_complete)(&buf);
}
```

Listing 5.9: HW UART Perform Transmission

5.3 Portable TTP/A Implementation

Based on the description of the TTP/A implementation above, an architecture and modifications of the source code are proposed in this section, to make the main part of the C implementation independent from the target platform. The adoptions are minimized and pooled in two files, one for the processor architecture and one for the concrete hardware of the node (including processor model and I/O configuration).

5.3.1 Compiler Independence

Many hardware platforms are only supported by a single compiler and there is no universal compiler. Therefore, a portable source code has to be written in a way, that as many compilers as possible are able to compile it correctly. These compilers are often very specialized, limited and/or proprietary. However, every modern compiler is at least compatible to the C standard of ANSI. Source code that does not use any other C constructs than defined by this standard is understood by most of the compilers. Thus, the first step of creating a portable version of TTP/A is replacing constructs that are not compliant with the ANSI C standard. Especially constructs for the definition and initialization of complex data types (e. g., anonymous structures, initialization of selected fields of a structure, ...) that are supported by the GCC are not part of the ANSI C standard.

5.3.2 Hardware Abstraction Layer

To make the source code independent from the hardware the implementation of a HAL is proposed. The HAL defines a standardized interface to the hardware at a very low level of abstraction. The level has to be that low, because portability has not been considered consequently during design phase of the implementation. One the one hand there is an interface to the bus subsystem that allows a high level of abstraction. It would allow driver like implementation of the bus subsystem. On the other hand the timer used by TTP/A is read, written and configured in several protocol states. Moreover, in the current source code the timer hardware is accessed directly, there is no abstract software interface.

Figure 5.2 shows the proposed architecture for a portable version of a TTP/A implementation.

The HAL is implemented using macros. The advantage of macros is that it is guaranteed, that no overhead is added to the code. Almost all compilers understand macros. Another possibility is the use of inline functions. A compiler can check the syntax of a call of an inline function in contrast to

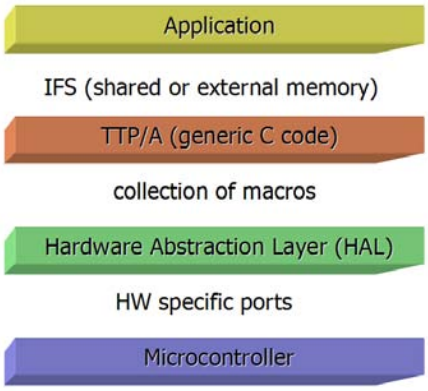


Figure 5.2: Layers of the Portable TTP/A implementation

the call of a macro. However, inline functions are not part of the ANSI C standard. Moreover, if a compiler supports inline functions, it is up to the compiler to decide how the call of an inline function is realized and whether an overhead is added or not.

The HAL is divided in three parts: timer related functionality, UART-related functionality and other functionality that does not fit in one of the two other groups. Because of the low level of abstraction many macros have to be defined. However, tests on the LPC platform have shown, that most of the macros can be kept simple and implemented with one line of code.

Timer

The macros of the HAL that are related to the timer are listed and described in table 5.1. If the implementation of the software UART is used, another seven macros listed in table 5.2 are necessary.

Additionally, for every used timer interrupt an according macro has to be defined that is used as entry point for the interrupt service routine (`hal_overflow_interrupt()`, `hal_compare_match_interrupt()`, and `hal_input capture_interrupt()`. Another macro (`hal_timer_t`) is used to define the data type of the timer (e. g., `uint16_t` for a 16 bit wide timer register).

UART

The macros of the HAL that are related to the timer are listed and described in table 5.3.

Macro	Description
`hal_configure_timer()`	Configures timer for use with TTP/A
`hal_set_timer(value)`	Sets TTP/A timer/counter to `value`
`hal_get_timer()`	Sets TTP/A timer/counter to `value`
`hal_set_timer_compare_value(value)`	Sets TTP/A timer/counter compare register to `value`
`hal_inc_timer_compare_value(value)`	Sets TTP/A timer/counter compare register to `value`
`hal_get_timer_compare_value()`	Sets TTP/A timer/counter compare register to `value`
`hal_delete_compare_match_interrupt()`	Deletes eventually pending timer overflow interrupt
`hal_enable_compare_match_interrupt()`	Enables timer overflow interrupt
`hal_disable_compare_match_interrupt()`	Disables timer overflow interrupt
`hal_delete_overflow_interrupt()`	Deletes eventually pending timer overflow interrupt
`hal_enable_overflow_interrupt()`	Enables timer overflow interrupt
`hal_disable_overflow_interrupt()`	Disables timer overflow interrupt
`hal_delete_input_capture_interrupt()`	Deletes eventually pending timer overflow interrupt

Table 5.1: Timer HAL Macros

Other Issues

Some more macros are defined in the HAL for accessing to Flash memory and Electrically Erasable Programmable Read-Only Memory (EEPROM), for enabling and disabling interrupts, for configuring the node, for reading the I/O pin for the software UART and for controlling a possible, external transceiver. These macros are described in table 5.4, 5.5, 5.6, and 5.7.

The macros for accessing the EEPROM may be defined as constant numbers if on a particular platform no EEPROM is available. The parts of the code for accessing the EEPROM is never executed, if no files of the IFS are configured to be stored in the EEPROM.

The macros related to the transceiver may be empty if no transceivers are used that need to be switched between transmit and receive mode.

Macro	Description
`hal_enable_input_capture_interrupt()`	Enables timer input capture interrupt
`hal_disable_input_capture_interrupt()`	Disables timer input capture interrupt
`hal_set_input_capture_falling_edge()`	Configures input capture mechanism to trigger on a falling edge
`hal_set_input_capture_raising_edge()`	Configures input capture mechanism to trigger on a raising edge
`hal_set_input_capture_toggle_edge()`	Configures input capture mechanism to trigger on any edge
`hal_get_input_capture_value()`	Reads value from input capture register
`hal_set_pin_on_compare_match(value)`	Configures compare match logic to set pin state to `value` on next match event

Table 5.2: Additional Timer HAL Macros for the Software UART

5.3.3 Linker Script

A special linker script is used for generating special sections for the IFS and a table of tasks that is executed during the initialization. The linker script strongly depends on the used platform and of the linker. Thus it has to be changed accordingly. Unfortunately, at the time of writing there is no documentation about the layout of these sections available.

The according linker file for a target platform have to be created using the linker file of the available implementation as example. For many linker graphical tools exist, which allow a convenient adaptation of the linker files. In many cases linker files have to be created for new projects anyway.

Macro	Description
`hal_uart_configure()`	Configures the UART
`hal_uart_set_parity(value)`	Sets parity mode to even or to odd
`hal_uart_rx_disable()`	Disables reception of bytes (often also the received data is cleared)
`hal_uart_rx_enable()`	Enables reception of bytes
`hal_uart_send_byte(value)`	Transmits the byte `value`
`hal_uart_get_value()`	Returns last byte received
`hal_uart_check_error()`	Checks whether an error occurred during reception of the last byte
`hal_uart_get_parity()`	Returns parity of the last byte received
`hal_uart_receive_interrupt()`	Header of UART receive interrupt
`hal_delete_uart_receive_interrupt()`	Deletes possibly pending UART receive interrupt
`hal_enable_uart_receive_interrupt()`	Enables UART receive interrupt
`hal_disable_uart_receive_interrupt()`	Disables UART receive interrupt

Table 5.3: UART HAL Macros

Macro	Description
`hal_progmem_read_byte()`	Reads a byte from the program memory (usually flash ROM)
`hal_progmem_read_word()`	Reads a word from the program memory (usually flash ROM)
`hal_eeprom_is_ready()`	Returns true if the EEPROM is ready for a read or write operation
`hal_eeprom_read_byte()`	Reads a byte from the EEPROM

Table 5.4: HAL Macros for Memory Access

Macro	Description
`hal_enable_interrupts()`	Globally enables interrupts
`hal_disable_interrupts()`	Globally disables interrupts

Table 5.5: HAL Macros for En-/Disabling Interrupts

Macro	Description
`hal_configure_swuart_io()`	Configures the I/O pins used by the software UART
`hal_configure_node_swuart()`	Performs node specific configuration that is related to the software UART
`hal_configure_hwuart()`	Performs general configuration that is related to the hardware UART
`hal_configure_node_hwuart()`	Performs node specific configuration that is related to the hardware UART
`hal_sw_uart_rxpin_is_high()`	Tests whether the receive pin of the software UART is at a high level

Table 5.6: HAL Macros for the Node and I/O Configuration

Macro	Description
`hal_init_transceiver()`	Initializes the transceiver
`hal_transceiver_recv()`	Switches the transceiver to receive mode
`hal_transceiver_send()`	Switches the transceiver to transmit mode

Table 5.7: HAL Macros for Controlling an External Transceiver

5.4 Evaluation

To evaluate the proposed HAL architecture on the one hand and the applicability of TTP/A on the current Tinyphoon hardware TTP/A has been ported to the LPC 2119 from Philips Semiconductors[3]. The experiences gained during porting have been used to improve the HAL architecture and to suggest improvements of the TTP/A protocol that make it even more suitable for the use on small autonomous robots.

5.4.1 TTP/A on the LPC 2119

The LPC 2119[Phi06] microcontroller bases upon the ARM7 core. It has been selected to be used on the Tinyphoon because of its powerful architecture despite its small footprint. It features the standard peripherals of a typical microcontroller (UART, timers with input capture and compare match logic, I/O ports, I^2C and CAN bus, ...).

A test cluster consisting of a master and a slave node has been build. The WinARM version of the ARM GCC 4.0.2 has been used for the development. The original TTP/A implementation also has been written for a version of the GCC. Thus, all compiler extensions that are not supported by the ANSI C standard have been disabled to emulate a compiler that understands ANSI C constructs only.

During the implementation of the HAL some problems had to be solved:

- The AVR platform supports at most 16-bit wide timers whereas the timer registers of the LPC 2119 are 32 bit wide. A special data type has been defined that has the same bit width as the timer registers. Whenever the value of a timer is stored in a variable, this data type has to be used.

- The timers of the LPC 2119 can not raise an interrupt when the timer register overflows. Thus, this behavior has to be emulated using another compare match unit.

- The interrupt concept of the LPC 2119 is much more complex than that of the AVR microcontrollers. The macros have been designed in a way to support both of them. However, on platforms with a complex interrupt logic some additional steps might be necessary till the interrupt service routine is called correctly (e. g., entries in linker and/or start-up files, modifications of interrupt vector table, ...

[3]now NXP, `http://www.nxp.com`

- The timings might vary from one platform to another (e. g., the UART receive interrupt is raised earlier or later after the reception of a byte). To solve this problem correction values are used in the critical parts of the source code.

- The adaption of the linker file turned out to be a very cumbersome job.

- The implementation of the hardware UART has not improved the maximum performance as expected. The reason for this behavior is, that the action for the next frame is prepared in the inter frame gap after the previous bus operation has been finished. The implementation of the software UART stops the reception of a byte after the raising edge of the stop bit has been detected. In contrast the hardware UART samples mid of the stop bit and then raises the receive interrupt. Therefore, the preparation time for the next slot is even shorter and the maximum baud rate is smaller than that of the implementation using a software UART.

5.4.2 Suggested Improvements of TTP/A

The suggestions are based on the issues that came up during porting and testing TTP/A on the hardware of the Tinyphoon. The problems are grouped in three categories: enhancing performance, improving the handling of large chunks of data and avoiding the use of linker scripts.

Enhancing Performance

A hardware UART has been implemented in order to improve the performance of TTP/A while the load of the microcontroller is reduced. However, tests have shown, that on the AVR platform the hardware UART configuration (up to 16000 bit/s) does not even reach the performance of the version using the software UART (up to 19200 bit/s).

The different way of stop bit detection in combination with the way of preparing the next byte frame causes this problem. The hardware UART generates a receive interrupt after the stop bit has been sampled in the middle of the bit, whereas the software UART stops receiving as soon the raising edge of the stop bit has been detected. The preparation of the next frame slot is started as soon as the reception or transmission of the last slot has been completed. Thus, the preparation time is shorter when the hardware UART is used.

To solve this problem the preparation of next frame slot can be started while the preceding frame is still being received or transmitted. In this

moment the communication is completely handled by the hardware UART. Therefore, the computing power of the CPU can be used for the preparation.

The frames of TTP/A are 13 bits long. but the maximum length of a transmission with a hardware UART is twelve bits (one start bit, eight data bits, one parity bit and two stop bits). Thus, a continuous transmission using a standard hardware UART is impossible. This restriction impedes the exploitation of the powerful features of more enhanced hardware UART that allow the transmission of multiple bytes without the interaction of the CPU.

Moreover, the transmission of every byte has to be triggered by a timer. One the one hand this increases the load on the CPU(e. g., timer configuration, interrupt handling) on the other hand not only the start of a transmission but the complete transmission is closely related to the timing, which makes a separate implementation of the UART and the timer subsystems hardly possible. Hence, the code lacks portability.

Handling of Large Data Packets

TTP/A has been designed as communication protocol for smart transducers. The protocol is adequate for exchanging small amounts of data. It causes only a little overhead but only supports frames with the length of one byte. Though many frames can be scheduled consecutively to avoid fragmentation of the payload, TTP/A provides no means to check the integrity of the complete payload. Thus, the application has to ensure, that all bytes of the payload have been communicated successfully. A checksum mechanism as provided by LIN would reduce the effort for the application design. Additionally, the encapsulation of an integrity check in the protocol ensures the compatibility thereof among all TTP/A nodes.

Linker Scripts

Linker scripts are highly hardware and linker dependent. The process of porting TTP/A also includes the complex adaption of the linker scripts. This effort only can be reduced by avoiding linker scripts. The current version of the TTP/A implementation uses them heavily for providing a comfortable programming interface while keeping memory requirements low.

In the future configuration tools will be available that can configure the IFS and the RODL of TTP/A nodes. Hence, there programming interface will not be used by the programmer directly and can be designed in a way that no linker scripts are needed.

A conclusion is the place
where you got tired thinking.

MARTIN H. FISCHER
American (German-born) Physician and Author (1879 - 1962)

Chapter 6

Conclusion

This chapter summarizes the contribution of this thesis and gives an outlook on future work in the covered domain.

6.1 Contribution of This Thesis

This thesis has described the five real-time protocols CAN, LIN, Flexray, TTP/C, and TTP/A in detail. The comparison of the protocols also considers the special needs of small autonomous systems. All important characteristics are listed in table form and the main attributes are also compared in a diagram.

The communication requirements of the subsystems of the Tinyphoon robot have been analyzed. Each subsystem and the data it needs or provides has been described. This analysis of the communication on the Tinyphoon has shown, that real-time features (e. g., predictable communication, small jitter, global time) improve the interaction of the subsystems and allow new approaches of application design. Fault tolerance is not an issue on small robots but a high data throughput and the maintainability of the communication system are vitally important.

CAN and LIN lack some of the important real-time features whereas Flexray and TTP/C are very complex. TTP/A combines the basic real-time features with a small implementation footprint. Its standardized debugging and configuration interface allows the observation of the communication and the platform-independent tuning of the configuration of each subsystem. TTP/A is implemented in software and coexists with other applications, which saves valuable space on the PCB. Moreover, the source code of TTP/A is available for free. Because of these features TTP/A has

been evaluated on the Tinyphoon robot.

Based on an analysis of the mode of operation of the current TTP/A implementation an architecture for making TTP/A portable is proposed. A HAL has been designed and implemented on the original platform of the TTP/A implementation. A case study on the hardware of the Tinyphoon robot has proved the capability of this concept and that TTP/A can be used for the communication on small autonomous systems.

Based on the evaluation of this case study a proposal for further improvements of TTP/A have been derived. Methods have been proposed to make TTP/A easier to port to new hardware platforms and to make it even more suitable for the use on small autonomous systems like the Tinyphoon robot.

6.2 Outlook to Future Work

Currently a new protocol based on TTP/A is being developed. Its performance will be boosted by supporting the application of the advanced features of modern, powerful UART hardware units. The protection of data frames with a checksum will ease the handling of large chunks of data. This protocol will suit the needs of small autonomous systems even better than TTP/A.

The communication architecture of the Tinyphoon robot has to be revised in the future. The current implementations of the subsystems are designed according to the event-triggered paradigm. With the introduction of a global time base the complete system can be synchronized. Hence, all measurements on the robot can be triggered at the same point in time, which makes the fusion of the result considerably easier. Therefore, we plan to integrate real-time communication on the Tinyphoon robot and to use a time-triggered software design.

In the future the wireless communication system of the Tinyphoon robot can be used to synchronize measurements within a group of robots. This simplifies the exchange and the interpretation of data about their environment. Thus, a distributed world model of all robots can be established.

The separation of the communication protocol from the application reduces the effort of maintaining the source code. The communication system can be added to the software of new subsystems easily.

Bibliography

[ANS89] ANSI – American National Standard Institute. Ansi standard x3.159-1989, December 1989.

[Atm05] Atmel Corporation. Application note AVR322: LIN v1.3 protocol implementation on atmel avr microcontrollers, 05 2005.

[BAS+06] M. Bader, M. Albero, R. Sablatnig, J. E. Simó, G. Benet, G. Novak, and F. Blanes. "embedded real-time ball detection unit for the yabiro biped robot". In *"Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES'06)"*, Vienna, Austria, 2006.

[DEE03] Martin Delvai, Ulrike Eisenmann, and Wilfried Elmenreich. A generic architecture for integrated smart transducers. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, Lecture Notes in Computer Science, Lisboa, Portugal, September 2003. Springer Verlag.

[Dev06] Analog Devices. *Datasheet Blackfin 561*, 2006.

[EEE+01] Stephan Eberle, Christian Ebner, Wilfried Elmenreich, Georg Färber, Peter Göhner, Wolfgang Haidinger, Michael Holzmann, Robert Huber, Ralf Schlatterbeck, Hermann Kopetz, and Alec Stothert. Specification of the ttp/a protocol. Research Report 61/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[EHK+02] Wilfried Elmenreich, Wolfgang Haidinger, Raimund Kirner, Thomas Losert, Roman Obermaisser, and Christian Trödhandl. TTP/A smart transducer programming - a beginner's guide. Research Report 33/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[EHPS02] Wilfried Elmenreich, Wolfgang Haidinger, Philipp Peti, and Lukas Schneider. New node integration for master-slave fieldbus

networks. In *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI 2002)*, pages 173–178, Feb. 2002.

[EK04] Wilfried Elmenreich and Stefan Krywult. Comparison of fieldbus protocols LIN 1.0, LIN 2.0, and TTP/A. Research Report 3/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[Fle05a] FlexRay Consortium. FlexRay communication system, electrical physical layer specification, version 2.0, 2005.

[Fle05b] FlexRay Consortium. FlexRay communication system, protocol specification, version 2.0, 2005.

[FMD+00] Thomas Fuhrer, Bernd Muller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther, and Robert Bosch GmbH. Time triggered communication on CAN. In *Proceedings of the 7th International CAN Conference*, Amsterdam, 2000.

[HH00] Wolfgang Haidinger and Robert Huber. Generation and analysis of the codes for TTP/A fireworks bytes. Research Report 5/2000, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2000.

[HMFH00] Florian Hartwich, Bernd Müller, Thomas Führer, and Robert Hugel. CAN network with time triggered communication. In *Proceedings of the 7th International CAN Conference*, Amsterdam, 2000.

[IEE90] IEEE – Institute of Electrical and Electronics Engineers. Ieee standard 1149.1: Standard test access port and boundary-scan architecture, 1990.

[ISO04] ISO – International Organization for Standardization. Road vehicles – controller area network (CAN) – Part 4: Time-triggered communication, 2004.

[KHK+96] H. Kopetz, R. Hexel, A. Krüger, D. Millinger, and A. Schedl. A synchronization strategy for a TTP/C controller. In *Application of Multiplexing Technology (SP-1137)*, Detroid, MI, USA, February 1996. Society of Automotive Engineers, SAE Press.

[KO03] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. In *IEEE Transactions on Computers*, pages 933–940, August 2003.

[Kop97] Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[Kop01] Hermann Kopetz. A comparison of TTP/C and FlexRay. Research Report 10/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[KS04a] Stefan Krywult and Christian Steiner. Survey on present real-time ethernet solutions. Research Report 12/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[KS04b] Stefan Krywult and Christian Steiner. TTP/A gateway. Research Report 37/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[LIN03] LIN Consortium. LIN Specification Package, Revision 2.0, September 2003.

[LL84] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of PODC '84*, New York, NY, USA, 1984. ACM Press.

[MON05] S. Mahlknecht, R. Oberhammer, and G. Novak. "real-time image recognition system for tiny autonomous mobile robots". *Real-Time Systems*, 29, 2005.

[NM05] G. Novak and S. Mahlknecht. Tinyphoon - a tiny autonomous mobile robot. Technical report, Institute for Computer Technology, Vienna Universitys of Technology, Vienna, Austria, 2005.

[OMG02] OMG – Object Management Group. Smart transducers interface specification v1.0 – ptc/2002-10-02, 08 2002.

[Phi06] Philips. *Datasheet LPC2119*, 2006.

[Sie06] Siemens. *Datasheet XC167*, 2006.

[SJ05] M. Seyr and S. Jakubek. "mobile robot predictive trajectory tracking". In *Proceedings of the 2005 ICINCO*, ICINCO, 2005.

[SJN05] M. Seyr, S. Jakubek, and G. Novak. Neural network predictive trajectory tracking of an autonomous two-wheeled mobile robot. In *Proceedings of the 16th IFAC World Congress*, Elsevier, Prag, 2005.

[Trö02] Christian Trödhandl. Architectural Requirements for TTP/A Nodes. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2002.

[TTA03] TTAGroup. Time-triggered protocol TTP/C high-level specification document, Protocol Version 1.1, edition 1.4.3, 2003.

[USB00] USB Implementers Forum. Universal serial bus specification, revision 2.0, 04 2000.

[Web06] Daniel Weber. Decision making in the robot soccer domain. Master's thesis, Vienna University of Technology, 2006.