Introduction to FlexRay and TTA

Peter Böhm

November 21, 2005

Contents

Part I.
FlexRay

1. Introduction

The FlexRay protocol is a communication protocol for distributed systems (cf. [1]). It was developed exclusively for automotive, although it might be used somewhere else. It was developed by the FlexRay consortium which was founded in 1999. The aim was to develop a flexible and fault-tolerant communication protocol. A FlexRay system consists of several electronic control units (ECU) each with a bus interface connected to one or two communication channels (cf. [3]). Such a system is called FlexRay cluster and represents a distributed real-time system.

In Section 2, we handle the different network topologies supported by FlexRay. In Section 3, the structure of the electronic control units and the bus interface is described. After that architectural part, the protocol's timing relevant features are described. Section 4 explains the schedule and Section 5 describes how a message is processed. In Section 6, we will have a look towards the clock synchronization algorithm. In Section 7 the wakeup/startup procedure is explained. Finally we summarize in Section 8.

2. Network Topology

There are several possible network topologies. A FlexRay cluster consists of one or two channels. The former is called *single-channel* and the latter *dual-channel*. A cluster can be build as a *bus*, a *star* or a so called *hybrid network*. In a dual-channel topology, each ECU can be connected to either one or both channels. Typical examples for a bus topology and a star topology are shown in Figure 2.1. In a star topology the central connection points are called *star couplers*. A hybrid network is a combination of a bus topology and a star topology, this means that one channel is realized as a bus and the other one as a star. A typical example for such a network topology is shown in Figure 2.2. If a topology with star couplers is chosen, there is also the possibility to build *cascaded* networks by connecting two star couplers directly (c.f Figure 2.2).

Optionally so called *bus guardians* can be added, what increases the fault-tolerance. Bus guardians are small devices, which are placed between the bus interface and the channel. They prevent a bubbling-idiot failure, where a certain ECU transmits all the time. The bus guardian disconnects the ECU from the channel when it is not allowed to transmit data. In a bus topology, the guardians are placed directly between the bus controller and the communication channel. In case of a star topology, it is sufficient if each star coupler contains one bus guardian.

3. Electronic Control Units and Bus Interface

A electronic control unit (*ECU*) consists of a processor with a memory management unit and an I/O-interface. The bus controller is attached to this interface as a I/O-device with

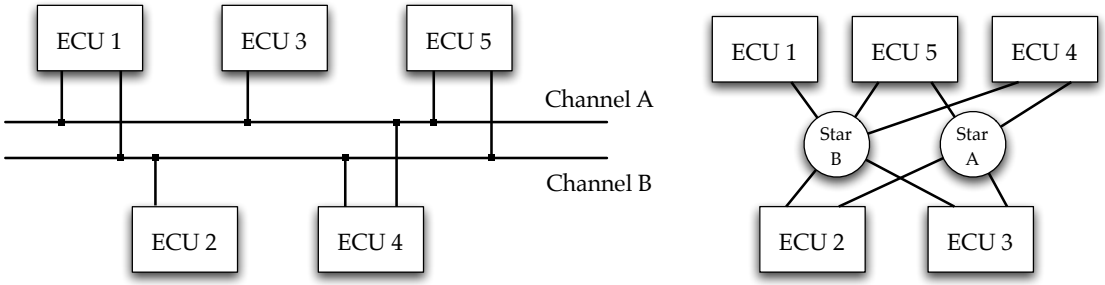

Figure 2.1: Sample bus (left) and star (right) network topologies

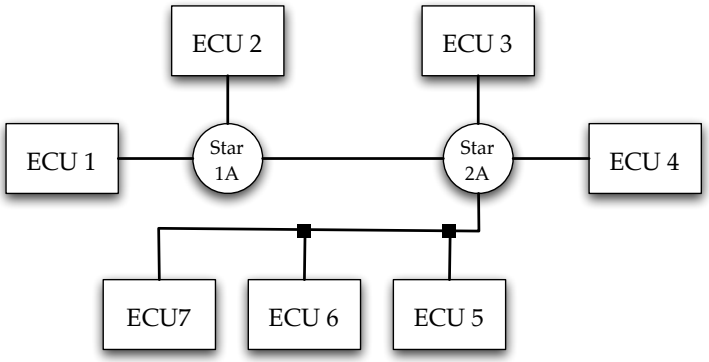

Figure 2.2: Single-channel hybrid network with a cascaded star

the following ports:

- control and status port (c/s)
- data port (data)
- configuration port (config)

The overall structure is shown in Figure 3.1. As mentioned in Section 2, bus guardians can be added to the bus controllers, optionally.

The bus controller is structured into six main components:

- controller host interface: The controller host interface provides the interface between the host and the bus controller.
- protocol operation control: The protocol operation control handles host commands and protocol conditions.
- media access control: The media access control schedules the message assembling and transmission.
- frame and symbol processing: The frame and symbol processing handles received messages and separates the header from the payload data.
- coding/decoding unit: The coding/decoding unit performs the physical write and read accesses to the bus. Furthermore It decodes a received message and encodes a message for transmission.
- clock synchronization: The clock synchronization generates a timer value which is used as a local time unit. Furthermore it synchronizes this timer.

The functionality of these components is described in the following subsections.

3.1. Controller Host Interface

The *controller host interface (CHI)* provides the interface between the bus controller and the ECU. As shown in Figure 3.2, the CHI consists of three main parts: 1) the message interface for the data port, 2) the configuration interface for the config port and 3) the control and status interface for the c/s port.

Within the message interface, two buffers are maintained: a *send buffer (sb)* and a *receive buffer (rb)*. In the first one, the host stores the payload for messages meant to be sent. The host fills the buffer by write accesses to the data port. Together with the send buffer, a address pointer *sbp* is maintained. A address pointer is a counter which value is used as address when the buffer is accessed. So, the the value of sbp is used as the write address for the send buffer. When the host writes data to the data port, the data is stored to the address sbp. Thereafter the value of sbp is increased by one. Hence successive writes to the data port fills the send buffer. From the second buffer, the host reads received data. Analogously, the CHI maintains a address pointer *rbp* for the receive buffer. On data port reads, the host receives the data from the receive buffer at address rbp and the pointer is increased. Again, successive reads from the data port read out the receive buffer.

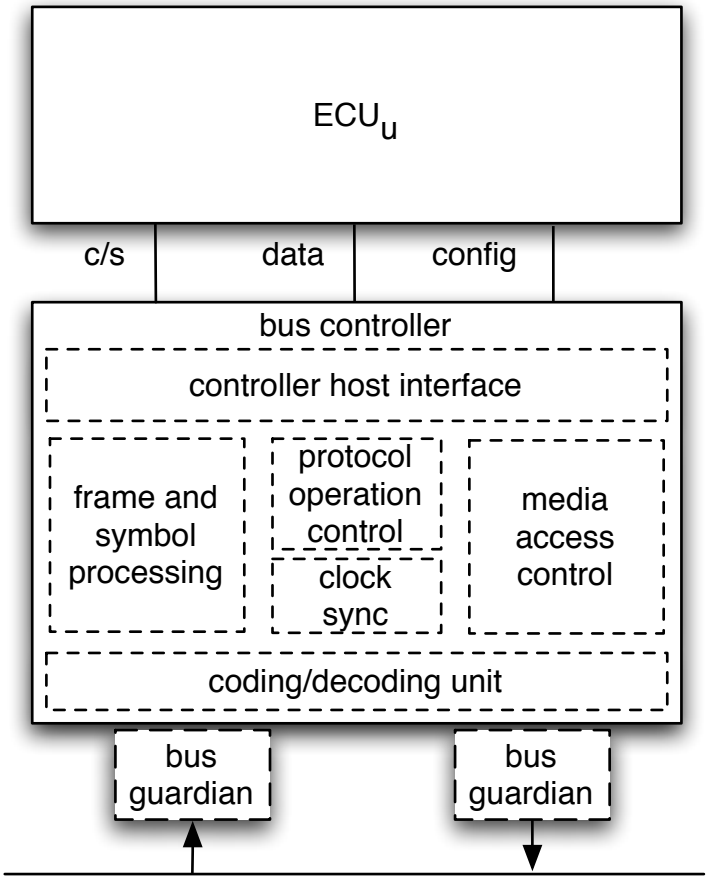

Figure 3.1: Overall structure of the ECU and the bus controller

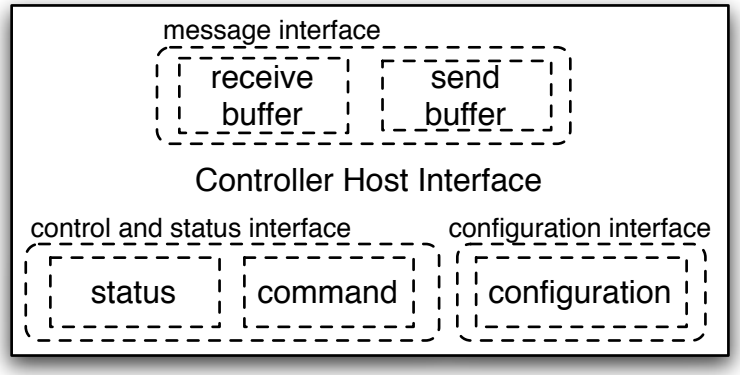

Figure 3.2: Controller host interface

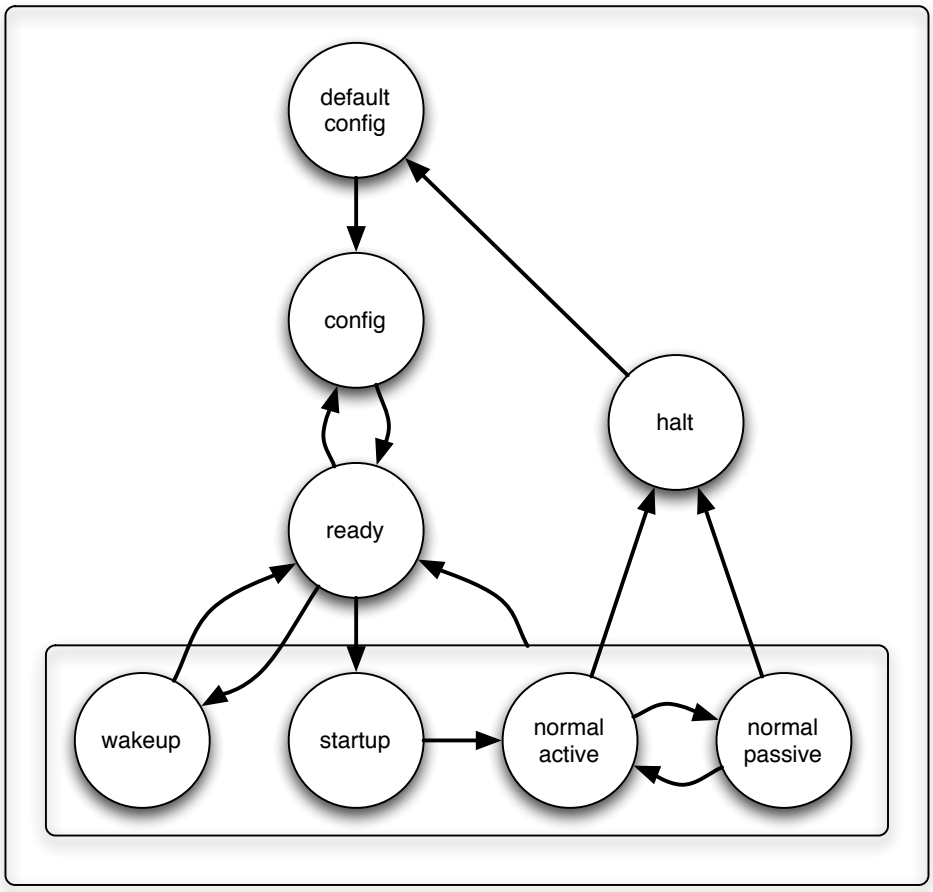

Figure 3.3: Operation overview of the protocol operation control

The configuration interface maintains several configuration parameters. These configuration parameters are only written during the startup phase via the config port. During normal operation the configuration is not changed. Most of the parameters concern the schedule, which is described in detail in Section 4.

During normal operation, the control and status interface provides status data about the frame reception to the host. This status data can be read from the c/s port. To interact with the bus controller, the host can send some control commands to the bus interface, such as a standby command. To send a command to the controller, the host writes the command to the c/s port.

3.2. Protocol Operation Control

The main purpose of the *protocol operation control (POC)* is to react to host commands or to protocol conditions such as errors. The POC sets the other components to the appropriate operation mode and its operation state represents the different controller states. An operation overview is shown in Figure 3.3. After power-on, the POC starts in the *default config* state, before it proceeds to the *config* state. Only in these two states, the controller

is configurable via the config port. After configuration, the POC goes to the *ready* state. There it depends on the configuration and the communication channel activity whether it proceeds to *wakeup* or *startup*. If a connected communication channel is idle and the ECU is allowed to wakeup a channel, the control goes to wakeup until the idle channel is ready to work. Then, in the startup state, the controller integrates into the cluster. As long as no error occurs, the controller stays in *normal active*. In case of an error, the POC falls to *normal passive* and tries to reintegrate. On a fatal error, the bus controller stops operation in the *halt* state.

Furthermore, the host has the possibility to change the controller's state. If the bus controller receives such a command, it has to decide, whether the command is allowed in the current state and when it has to be applied, e. g. a halt command send by the host is only allowed in the normal active or normal passive state and, it is processed at the end of one communication round. A communication round is one execution of the whole schedule.

3.3. Media Access Control

The *media access control (MAC)* is the component that handles the data transmission. It schedules the bus write accesses and composes the message out of the payload data from the send buffer in the controller host interface. In FlexRay, each ECU is only allowed to transmit a message within certain time intervals which are called *slots*. In each slot the MAC checks whether the attached ECU is allowed to transmit a message. If this is the case, the media access control imports the payload data from the controller host interface and generates the message header. At the time point within the slot, when the transmission can start, the coding/decoding unit is notified that it can write the assembled message to the bus.

3.4. Frame and Symbol Processing

The *frame and symbol processing (FSP)* handles the received data. The FSP is notified by the coding/decoding unit if a valid frame was received. After frame reception, the message is separated into the payload and the header. The payload is copied to the receive buffer in the CHI. Additionally, the frame and symbol processing provides status data to the host interface after each slot. This status data contains the local time and information about frame reception, such as whether a valid frame was received in the last slot.

3.5. Coding/Decoding Unit

The *coding/decoding unit (CODEC)* is the component of the bus controller that physically accesses the bus. When the MAC signals the CODEC to transmit a message, it appends a CRC checksum to the message. Thereafter the coding part encodes the message for transmission. For frame reception, the coding/decoding unit listens to the bus. If a message is detected, the decoding part reassembles the message. After that, the CRC check is performed and if the message passes, the FSP is notified that a valid frame was received. The coding/decoding unit is explained detailed by Gerke [2].

3.6. Clock Synchronization

The *clock synchronization* component performs two ways of time synchronization and generates a local time unit, the so-called *macroticks*. They represent the local time base for the scheduling of message transmission and interrupts for the host. Macroticks are synchronized time units, which means that their length relative to the local oscillator clock might vary from macrotick to macrotick. The aim is that a macrotick has approximately the same length on any bus controller within a cluster relative to real time. That is a non-trivial problem in distributed systems because the local oscillator clocks may drift due to fabrication tolerances or environment differences.

For any kind of correction a time difference between the local view of time and the view of other bus controllers is needed. This is achieved by so called *sync messages*. If a bus controller receives a sync message, the difference between expected and observed arrival time is calculated and stored. During a communication round up to 15 sync messages are received. The clock synchronization in FlexRay performs two different corrections: the offset correction and the rate correction.

The offset correction is the most intuitive way to correct the local time. A fault-tolerant midpoint algorithm computes an average over the time differences from one communication round and the next schedule execution is delayed or starts earlier, such that all bus controller start the next communication round at approximately the same time.

The rate correction tries to minimize the difference between the time views by adjusting the length of a macrotick. Hence the clock synchronization computes the difference between the expected and observed arrival time of a sync message in two consecutive communication rounds. Then the macrotick length is adjusted accordingly.

4. Schedule

This section will handle the schedule of FlexRay. The FlexRay protocol uses a time-triggered schedule. In contrast to an event-triggered system, a time-triggered schedule is controlled by time points. The bus write access is scheduled by a time-division multiple access (TDMA) scheme, which means that bus write accesses are only allowed in exactly specified time intervals. These intervals are called *slots*. Each ECU has a configurable amount of slots in which it is allowed to transmit a message. In FlexRay, every bus controller just knows its own transmission slots and does not know anything about the other controller's slots. Hence there is no common knowledge in FlexRay about the slot assignment to the ECUs. In each slot exactly one message is transmitted. The TDMA schedule provides a static and deterministic timing.

The whole specified schedule is called a *communication round*. It contains a *static segment* with fixed number of slots, a *symbol window* and a *network idle time*. The number of slots is a configuration parameter and denoted with ns. The symbol window is used to transmit special messages such as a wakeup symbol, which is needed during a cluster wakeup [2]. More interesting within this document is the network idle time. During this time interval, the clock synchronization calculates the clock correction values and performs the clock correction. Such a communication round is shown in Figure 4.1.

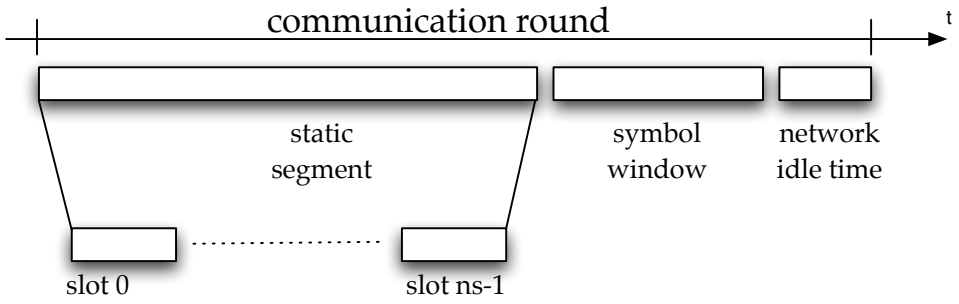

Figure 4.1: Structure of a FlexRay communication round

5. Message Processing

In this section, the message processing is described. The procedure is shown in Figure 5.1. Let ECU A be the sending ECU and ECU B any receiving ECU. First, ECU A has to write the message payload to the send buffer within the controller host interface of its bus controller. Then, the media access control assembles the correct header for the payload and hands both to the coding/decoding unit. This unit computes the CRC and appends it to the header and the payload. At this point, the frame composition is finished. Finally the coding part of the coding/decoding unit encodes the message for transmission. Then, the data is transmitted bitwise on the physical bus.

On the receiver side the procedure is performed vice versa. First, the decoding part of the coding/decoding unit decodes the received data and the message consisting of header, payload and the CRC is reassembled. Thereafter, the CRC is checked. If the message passes the test, the header and payload proceeds to the frame and symbol processing where other error checks are performed, such as right header syntax. Finally, the frame and symbol processing writes the payload data to the receive buffer in the controller host interface where ECU B can read it out.

6. Clock Synchronization

Section 6 deals with the clock synchronization algorithm used in the FlexRay protocol. Clock synchronization is obligatory in a FlexRay cluster because of the time-triggered schedule. In order to prevent overlapping of transmission times, each bus controller needs approximately the same view of time.

As described before, there are 2 different correction methods: offset and rate correction. The clock synchronization calculates the correction value for the offset correction after the static segment at the end of each communication round. Because the rate correction value depends on the time drift during two consecutive communication rounds, the correction value for the rate correction is only computed at the end of odd rounds.

The offset correction is performed at the end of odd cycles within the network idle time after both calculations are finished. The offset correction adjusts the length of a macrotick within the network idle time accordingly to the correction value. The rate correction adjusts

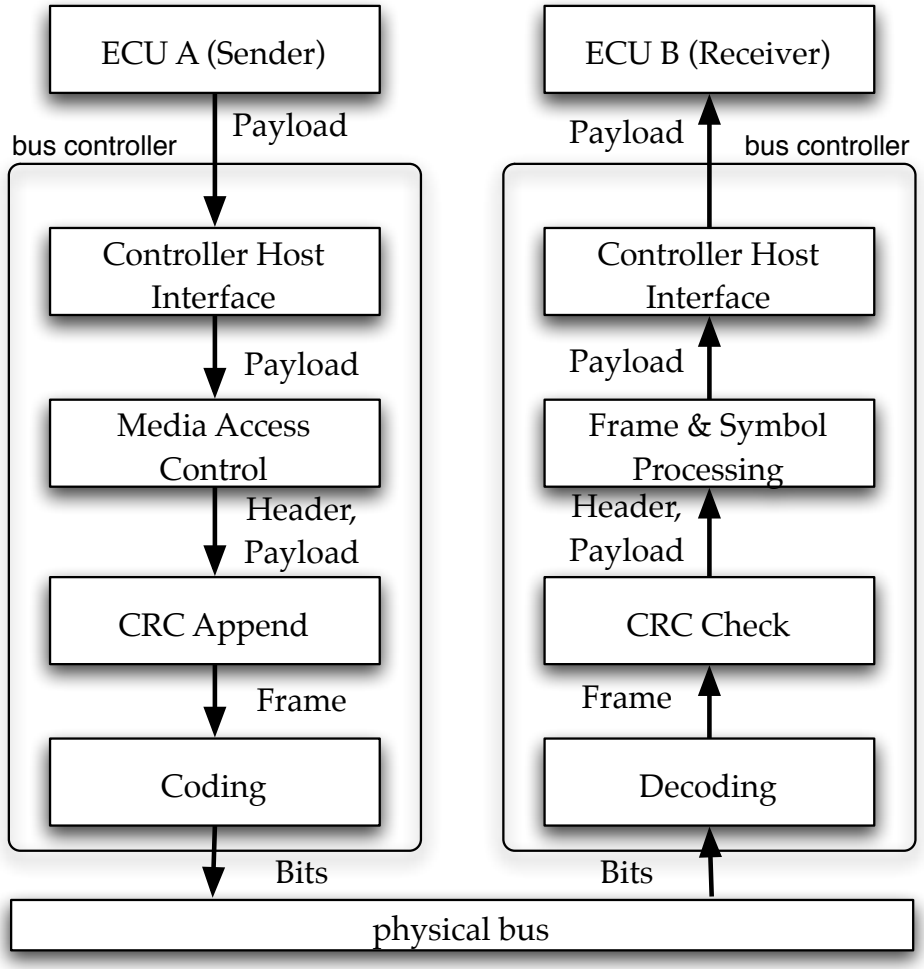

Figure 5.1: Message processing in a FlexRay system

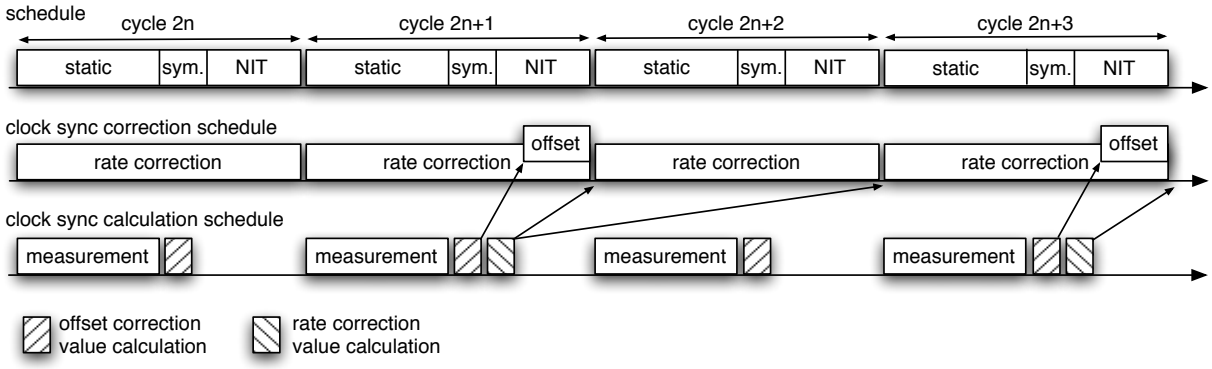

Figure 6.1: Overview of the clock synchronization timing

the average length of a macrotick within a communication round. A calculated correction value is used for two communication rounds. Because the clock synchronization calculates the correction value at the end of odd cycles, the value is used for the next even and odd cycle. A timing overview of the clock synchronization is shown in Figure 6.1.

Within the calculation of the correction values, a fault-tolerant midpoint algorithm is used. Depending on the amount of available measurement values, the k biggest and smallest values are discarded, where $k \in \{0, 1, 2\}$. This provides a certain degree of fault-tolerance: up to two bus controllers, which send sync messages, are allowed to operate incorrectly, such that the measured time difference values on an correct controller are not used for calculation.

The most important fact about the clock synchronization is of course, that with clock synchronization the local time drift is bound to a certain factor even if there are some faulty bus controllers. This is a non-trivial property if some bus controllers operate faulty. In [3] a simpler clock synchronization algorithm in a fault-free FlexRay like environment is described. There, the bound of the local clock drift is proven. The proof of the clock synchronization algorithm described here is still an open problem.

7. Wakeup/ Startup

In this Section the wakeup and startup procedure is introduced. In order to understand, why startup/wakeup is an important feature in FlexRay, we first need to look at the error model. In FlexRay, a three-level error model is used with the following states:

- *active*: In the active state, the bus controller operates normal. There was no error detected.
- *passive*: The passive state is entered on error detection if the detected error can possibly be fixed, such as if the clock correction values grew to large. In this state, a bus controller is not allowed to transmit any data. It just listens to the bus and tries to reintegrate itself.
- *halt*: If a bus controller enters the halt state due to a error condition, the detected error is fatal. A fatal error is an error that cannot be fixed by listening to the bus in passive mode. In the halt state, the bus controller completely stops operation.

This state can also be entered on host command and can only be left through the default configuration state (cf. Section 3). So, there is no possibility to enter a normal operation state directly from the halt state.

For obvious reasons, a startup/strategy is needed after power-on of a bus controller. At this time its local time is not synchronized at all. A startup/wakeup routine is also required after a fatal error, because the halt state of the failure model can only be left through the default config state. At last, there is also a requirement for a reintegration strategy after entering the passive mode.

Within a FlexRay cluster, there is a configurable amount of ECUs which are allowed to broadcast their view of time. Such a ECU with its bus controller is called a *coldstart node*. After power-on, a leader election algorithm between these coldstart nodes chooses a leader among them. Thereafter, this leader broadcasts its view of time. The other bus controllers can integrate into the cluster by synchronizing their local clock against this broadcasted time view. This integration into the cluster is also performed after a bus controller has entered the halt state and wants to proceed to normal operation again.

After entering the passive state, the bus controller needs to reintegrate itself. This reintegration is done by synchronizing the local clock while listening to the bus. When the bus controller decides that it operates normally again, the normal state is entered.

8. Summary

In Section 2, we have seen that FlexRay supports a very flexible network topology. Because of this flexibility regarding the network topology, the protocol provides a scalability of the fault-tolerance. The fault-tolerance could be increased by using a dual-channel topology instead of a single-channel in order to provide redundancy.

Furthermore, we have seen a time-triggered schedule with a TDMA bus access scheme with no common knowledge about the transmission times of the different bus controllers (cf. Section 4) and a fault-tolerant message transmission with error checks on reception (cf. Section 5).

Additional fault-tolerance is achieved by the clock synchronization algorithm in Section 6. Moreover, the error model from Section 7 provides the ability of a self-diagnostic error mechanism with the possibility of reintegration.

So we can say, that the FlexRay protocol is a flexible as well as fault-tolerant communication system for an automotive context.

Part II.
TTA

9. Introduction

In this second part of the document, a different protocol for communication within an automotive context is presented: *The time-triggered architecture, TTA*. The deterministic protocol behind TTA is called the*time-triggered protocol, TTP*.

Like FlexRay, TTA is an architecture for fault-tolerant, safety-critical real-time systems. It was developed by Prof. Kopetz at the Technical University of Vienna. He started in 1979, published it first in 1993 and launched it in 1998 ([6]). TTA specifies two different versions of the protocol TTP:

1. *TTP/C:* TTP/C is the full version of TTP for real-time busses in fault-tolerant distributed systems

2. *TTP/A:* TTP/A is a low cost version of TTP with reduced functionality for so called field busses. Field busses are busses to connect sensors or actuators.

This document will only handle the first, complete version of TTP. So, we refer with TTP to TTP/C and with TTA to TTA with the full protocol version.

This document will introduce how TTA works and will show where the differences to FlexRay are. Section 10 describes the different network topologies, which are supported by TTA. Afterwards, in Section 11, the schedule is introduced. Section 12 handles the frame format and the different operation modes of a TTA bus controller. Thereafter, the group membership (cf. Section 13) and the clock synchronization (cf. Section 14) in TTA are introduced. In Section 15, we will se what a controller state is, before we summarize in Section 16.

10. Network Topology

The network topology in TTA is less flexible as the one of FlexRay. TTA only supports *either* a bus or a star topology and no hybrid topology. Furthermore, a TTA cluster can only be build with a dual-channel bus or star topology. A single-channel topology is only available tin the reduced protocol version TTP/A. For fault-tolerance reasons, the bus guardians are obligatory in a TTA system. Two typical examples are shown in Figure 10.1. As in FlexRay, in case of a bus topology, the bus guardians are placed directly between the nodes and the channel. In case of a star network, the guardians are placed within the star couplers.

11. Schedule

In this section the schedule of TTA respectively TTP is described. The structure is quite similar to the one of FlexRay but with one more hierarchical level (cf. Figure 11.1). The schedule is also based on a TDMA scheme for bus write accesses ([4]).

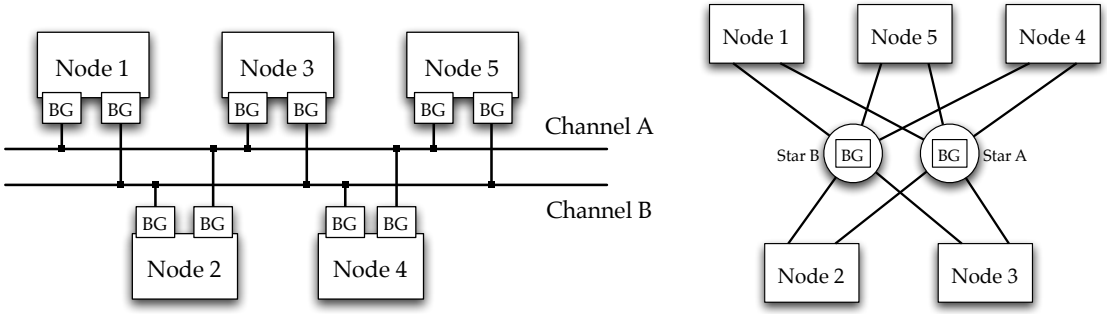

Figure 10.1: Sample bus (left) and star (right) topology

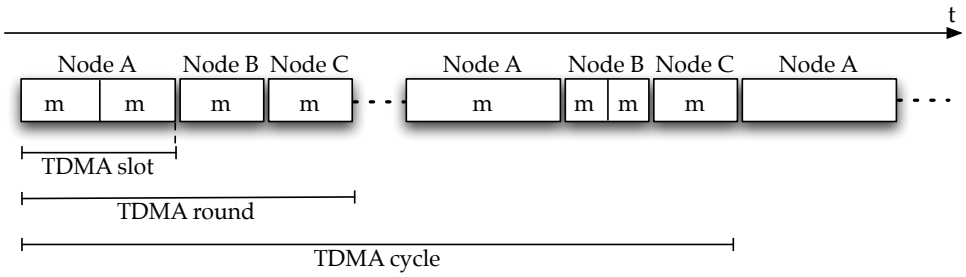

Figure 11.1: Structure of the TTA schedule

The smallest communication unit is a *TDMA slot*. In contrast to FlexRay, such a TDMA slot may contain more than one message but only one node is allowed to transmit within one slot. Furthermore, the slot length might differ, as, e.g., in Figure 11.1, node A has a larger slot than node B.

The next level in the timing hierarchy is a *TDMA round*. Within one round, each node has at least one transmission slot. Moreover, in each TDMA round of a cycle, the sequence of the sending nodes and the slot length is the same, i.e. the first slot in each TDMA round has the same length and the same node is allowed to send in this slot. Only the message length and the amount of messages within one slot may differ from round to round. This is illustrated in Figure 11.1, where node A send two messages in the first slot of the first TDMA round. In a later round, it transmits only one message in the first slot of the round.

The highest level in the timing hierarchy is a *TDMA cycle*. A TDMA cycle is the periodically, recurring time unit in TTP such as the communication round in FlexRay. A cycle consists of a configurable amount of TDMA rounds.

Since TTP uses a different approach to startup, reintegration and clock synchronization, as we will see in Section 12 and Section 14, the schedule needs no symbol window or network idle time as in FlexRay.

Regarding the schedule, the main difference to FlexRay is a global knowledge of all nodes. In a TTA system, each node knows the complete schedule in a so called *message descriptor list (MEDL)*. The MEDL specifies one whole TDMA cycle for each operation mode (cf. Section 12) and assigns all the slots to the sending nodes. In FlexRay, each bus controller just knows its own transmission slot, whereas in TTA each node knows exactly when which node is allowed to send.

In the message descriptor list, some nodes are marked as so called *sync nodes* with a *SYF-flag*, and some TDMA slots are marked as *synchronization slots* (*CS-flag*). Both flags will be explained in Section 14 within the clock synchronization.

At last, the MEDL also specifies time points within each TDMA cycle, where mode changes are allowed. Operation modes and mode changes will be explained in the next section.

12. Frame Format and Operation Modes

In contrast to FlexRay, TTA uses two different frame formats: *normal frames* and *initialization frames*. Moreover, TTA supports different operation modes: for normal operation, different modes can be configured, each with its own schedule. If a node changes its operation mode, it is called *mode change*.

Normal frames are used during normal operation and contain application data in their payload. Besides the data transport, they provide two more important functions. The first one is that normal frames contain information about mode changes in their header, i.e. they are used to request cluster wide mode changes by a host. The other important function is, that the headers also contain so called *acknowledgment bits*. With these bits, the message sender provides information to the cluster about the reception of the messages from its predecessor and from its pre-predecessor.

The second frame format, the initialization frames, contains the controller state of the sender in their payload segment. The controller state is a set of status information, which other nodes need to reintegrate themselves into the cluster by adopting the information.

Hence, initialization frames are needed during the startup phase as well as during normal operation. During normal operation the message descriptor list specifies when a node has to send an initialization frame.

The different operation modes supported by TTA are:

- *join mode*
- *application modes*
- *blackout mode*

The join mode is entered during startup of a bus controller. In this mode, a node only transmits initialization frames and on reception of one, it adopts the controller state from the message. Hence, a very fast synchronization of all nodes within a cluster after power-on is provided. After the join mode, the bus controller proceeds to an application mode.

Application modes are the ones during normal operation. TTP supports more than one operation mode and as mentioned in Section 11, for each of them a separate schedule. To change between application modes, a host requests a mode change with a normal frame. The change is performed at the time point, that is defined in the MEDL. The schedule of each application mode must contain a certain amount of initialization frames.

The third mode is the blackout mode. A bus controller enters the blackout mode on an error condition. In the blackout mode, the bus controller listens on the bus for initialization frames and tries to reintegrate itself.

13. Group Membership

The group membership algorithm provides the ability to identify faulty nodes and exclude them such that their messages are no longer considered. The set of all bus controllers that a bus controller identifies as working correctly is called the bus controller's *group*. Hence, each bus controller maintains its own private *membership list*, which records all non-faulty nodes of its group including the node itself.

The fault hypothesis behind the algorithm consists of two parts ([5]):

1. faults are two or more rounds apart
2. all or exactly one bus controller fail to receive a message. If all bus controller fail to receive a message, the failure is called a *send fault*. The case if exactly one node fails to receive a message is called a *receive fault*.

This fault hypothesis is necessary for the reliability of the group membership algorithm which is characterized by:

1. *agreement:* the membership lists of all non-faulty bus controllers are the same.
2. *validity:* the membership lists of all non-faulty bus controllers contain all non-faulty controllers and at most one faulty one.

The membership algorithm is based on the system-wide knowledge of the schedule. If a node does not send a message when it should, all other nodes will exclude it from their membership list. Hence, if a message does not arrive at a node, the node first blames the sender. This succeeds as long as the receiver has no failure. So the critical part of the algorithm is the self-diagnostic. There, two cases must be distinguished: send and receive faults.

The self-diagnostic of a send fault is done via the acknowledgment bits of the normal frames. Each bus controller listens to these bits of its first and second successor. If both successors exclude the listening node and the second successor includes the first one in its membership, the current node knows, that it had a send fault.

A receive fault could be recognized in a similar way, but the specification of TTP uses a different approach. The receive fault identification is based on the CRC check of received messages. The CRC is generated on the sending bus controller with the help of its membership list. Hence, if the message passes the CRC check on the receiver's side, it is likely that the sender has the same membership list as the receiver. Now, each bus controller counts the CRC check passes and fails after it has send a message. If the fail rate is greater than the pass rate before the next transmission, it is likely the case that the node had a send failure during the last transmission. This method succeeds because each controller blames the message sender first if it does not receive the message correctly. So, if the majority excludes a sending node from their lists, the node assumes that it has a send failure. The counters are checked before each transmission and the message is only send if the fail rate is less than the pass rate. After the transmission, the counters are reset.

The group membership algorithm is an important difference to the FlexRay protocol because FlexRay does not support a group membership as the one presented here.

14. Clock Synchronization

The clock synchronization of TTP is done is a quite similar way as the offset correction in FlexRay. The deviation value for a message is computed like in FlexRay. A controller knows when a message should arrive and can observe, when the message arrives. In contrast to FlexRay, there are not the messages marked as sync messages but the MEDL defines some sync nodes with the SYF-flag. Each node stores the two largest and smallest deviation values since the last synchronization. A new value is only stored if the sender has the same group membership and the SYF-flag of the sender is set in the message descriptor list.

Out of the four stored values, the correction value is calculated as a fault-tolerant average by discarding the smallest and biggest value and average over the remaining two ([5]). Finally, the local clock is adjusted if the CS-flag of the current slot is set. Therefor, this algorithm does not need anything like a network idle time as the algorithm in FlexRay.

The algorithm itself is not as fault-tolerant as the one in FlexRay, because the clock synchronization algorithm tolerates only one faulty node's deviation value. The algorithm used in FlexRay can tolerate more faulty values if the amount of received sync messages in total is large enough. But in combination with the group membership, the fault-tolerance in TTA is increased. Under its fault hypothesis, the group membership minimizes the amount of deviation values of faulty nodes to a maximum of one (cf. Section 13).

15. Controller State

This section introduces the concept of a *controller state*. The problems in TTA systems is that the bus controllers need an agreement on three parameters:

1. *operation mode:* A node can only interpret received data correctly if its operation mode is equal to the one of the sender.

2. *time view:* The whole communication is based on a correct view of time. So, the nodes have to agree on the current slot number within a certain tolerance to prevent overlapping of transmission times.

3. *group membership:* The group membership algorithm and the clock synchronization are based on the assumption that all non-faulty nodes have the same membership lists.

Now, the aim behind the controller state, also called *C-state*, is that only nodes with the same C-state can communicate. Therefor, the controller state of a bus controller covers the three mentioned parameters. It is used to generate the CRC of normal frames before transmission. Hence, a receiver likely uncovers different C-states with the CRC check and can drop the message.

16. Summary

In this second part of the document we have seen a different approach to provide communication in a distributed, safety-critical real-time system. In contrast to FlexRay, TTA provides a less flexible network topology. The schedule is based on a system wide-wide common knowledge and is more complex then the one of FlexRay. By supporting different application modes, the TTP is a bit more flexible than the one of FlexRay.

The approach to the startup procedure, the reintegration strategy and the clock synchronization is also different. For startup, TTA provides an extra operation mode and a special frame format, which is also used for reintegration. Due to the strategy to correct the clock during a normal slot, the network idle time is dropped. This may improve the throughput a little bit.

The system-wide knowledge of the schedule provides features which cannot be found in FlexRay such as the group membership.

As a conclusion, we could say that TTA gives more priority to fault-tolerance and functionality than to flexibility.

References

[1] FlexRay Consortium. *FlexRay Communications System Specifications Version 2.1*, 2005.

[2] M. Gerke. Coding and decoding, media access control, frame and symbol processing. *FlexRay Seminar*, Universität des Saarlandes, 2005.

[3] S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. I. der Rieden, S. Knapp, D. Leinenbach, and W. J. Paul. Towards the formal verification of lower system layers in automotive systems. In *International Conference on Computer Design (ICCD 2005)*, 2005.

[4] H. Kopetz, R. Hexel, A. Krüger, D. Millinger, R. Nossal, A. Steininger, C. Temple, T. Führer, R. Pallierer, and M. Krug. A prototype implementation of a ttp/c controller. In *SAE Congress and Exhibition, February 1997, Detroit, MI, USA SAE Paper No. 970296*, 1997.

[5] J. Rushby. An overview of formal verification for the time-triggered architecture. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469, pages 83–105, Oldenburg, Germany, Sept. 2002. Springer-Verlag.

[6] K. Weyerhäuser. TTA (Time Triggered Architecture) und TTP (Time Triggered Protocol). *Seminar "Mobile Systeme"*. Universität Koblenz-Landau, 2005.