Overcoming Babbling-Idiot Failures in the FlexCAN Architecture: A Simple Bus-Guardian

Giuseppe Buja and Alberto Zuccollo
University of Padova
Via Gradenigo 6/A
35131 Padova, Italy
g.buja@ieee.org, zuccollo@die.unipd.it

Juan Pimentel
Kettering University
1700 West Third Avenue
48504 Flint, Michigan
jpimente@kettering.edu

Abstract *The paper is concerned with the key issue of protecting FlexCAN networks against the babbling-idiot faults, e.g., node faults that busy the bus unduly. A full solution of the problem would enhance the dependability of these networks greatly, making them attractive for safety-critical applications. After analyzing the various modes by which a babbling-idiot fault affects the network operation, a classification of the babbling-idiot faults into hardware and software is proposed. Then it is shown that the FlexCAN architecture provides a suitable means for tolerating hardware babbling-idiot faults. Afterwards, a simple bus-guardian is proposed to cope with the software babbling-idiot faults in the FlexCAN environment. The proposed bus-guardian has been implemented and tested, and some significant results are given to demonstrate its effectiveness.*

Keywords *Babbling-Idiot Faults, Bus-Guardian, CAN networks, FlexCAN Networks.*

1. Introduction

Networking safety-critical distributed control systems poses some stringent requirements on the underlying network in terms of fault-tolerance and real-time operation. Fault-tolerance means that the network must suitably react to at least one failure in its components, with a behavior correlated to the role played by the component in assuring the desired safety level [1]. Real-time operation means that the communication tasks must be completed within an established time interval. Such interval is correlated with the dynamics of the controlled plant while hardness in complying with the requirement is dictated by the application [2]. Examples of very demanding safety-critical applications are the drive-by-wire systems, especially the steering one, where the needs of fault-safety and/or fault-operational behavior and strictly time-bounded communications are of uppermost relevance for driving safely.

The fulfillment of the above requirements is impeded by the babbling-idiot phenomenon. This typically occurs, for example, when a node monopolizes a CAN bus by transmitting high-priority messages at erroneous time instants so frequently to delay more than the necessary the communications between properly operating nodes. Under this situation, the node *is affected by a babbling-idiot fault*. The worst-case scenario happens when a node keeps the bus continuously busy, thus inhibiting every communication between the other nodes. When the occurrence of a babbling-idiot fault prevents the execution of the requested application, then the network suffers from a babbling-idiot failure, which is "*the most serious timing failure in a system with a shared communication channel such as a bus system*" [3]. This justifies the numerous efforts taken on in recent times to analyze the causes and to tolerate the occurrence of a babbling-idiot fault.

The aim of this paper is twofold. In the first part we analyze the issue of the babbling-idiot fault in CAN-based networks, whilst in the second part a bus-guardian for FlexCAN networks is developed. More specifically, in the first part we analyze the modes by which a faulty node busies the bus. Depending on the mode, hardware and software babbling-idiot faults are defined. Solutions employed to protect a network against the two kinds of babbling-idiot faults are then discussed. Component (bus and/or node) replication is commonly used to prevent hardware babbling-idiot failures. Such a replication is facilitated in time-triggered networks (e.g., TTP and FlexRay) by the availability of a synchronized global clock. However, event-triggered networks like CAN call for a different solution. One solution is represented by the FlexCAN architecture [4], which is briefly summarized in Section 4. Tolerance to software babbling-idiot fault is commonly achieved by bus-guardians for both time- and event-triggered networks. In Section 3 it is explained why the bus-guardian has found an easy implementation in the time-triggered networks while in the event-triggered ones complex or partially tolerant structures have been used.

In the second part of the paper a simple-in-structure bus-guardian is proposed for FlexCAN networks, with the purpose of inhibiting a node affected by the software babbling-idiot faults. The novel bus-guardian has been implemented and tested. Experimental results confirming the effectiveness of the solution are given. It is worthwhile to note that the proposed bus-guardian can be used in other CAN-based networks with minor modifications.

2. The CAN protocol

The CAN protocol was developed by Robert Bosch gmbh at the end of the 80's for automotive applications not related to safety [5], [6]. It has enjoyed widespread diffusion in other environments, such as in the industrial and building automation. Recently, the CAN merits in terms of low cost and efficient operation have fuelled the interest in extending its usage to safety-critical applications.

CAN is an event-triggered protocol that solves the bus access competition by a bit-wise arbitration technique: for any two messages with different priorities sent by two distinct nodes, that one with the higher priority is transmitted without collision whilst the message with the lower priority is stopped and retransmitted as soon as the bus becomes idle. The arbitration strategy relies on the logical wired-AND functioning of the bus, with 0 as dominant state. This functioning is also exploited by the protocol to implement a powerful error notification mechanism; as soon as a node detects an error in the incoming message, it forces on the bus an error frame message formed by some dominant states that, besides destroying the incoming message, inform the other nodes of the error.

The conjunction of these features (event-triggered strategy, bit-wise arbitration technique, and error notification mechanism) facilitates the possibility that faulty nodes interfere with other non-faulty transmitting nodes whether superposing dominant states on recessive bits or delaying the communications by inserting continuously high-priority messages, thus dislocating the information exchange though the network.

In this paper, a CAN network is considered made up of at least two nodes and a channel, where the nodes have the following components:

- the host-controller: it is the component of the node where the application software is implemented;
- the communication-controller: it is the component of the node entitled to transmit and receive the messages;
- the transceiver: it is the component of the node that applies to and detects on the bus the voltage levels associated to the logical states.

3. Babbling-idiot faults

What gives rise to a babbling-idiot fault are some types of fault in the node components. The effects of a fault for the various components are as follows:

- for the host-controller: it produces data traffic on the network carrying no useful information;
- for the communication-controller: it sends either correct messages more frequently than required by the host-controller or incorrect sequences of bits;
- for the transceiver: it keeps the bus fixed at the dominant voltage level,
- for the channel: it has the dominant voltage level permanently.

The babbling-idiot fault of transceiver and channel is the direct consequence of hardware faults, for instance the short circuit of the transceiver output or the channel wires. Under these circumstances, a node is affected by a **hardware babbling-idiot fault**. Note that if the hardware faults forces on the bus the non-dominant voltage level, the network can go on with its operation, i.e. the node exhibits a fault-silent behavior.

The babbling-idiot fault of the host-controller is likely due to software. Instead, that of the communication controller probably originates from hardware; nevertheless one can look at the corresponding fault as though it were produced by a software fault. For this reason, the babbling-idiot faults of either the host- or the communication-controller are designated with **software babbling-idiot faults**.

In safety-critical systems the occurrence of a babbling-idiot fault leads to potentially catastrophic consequences. Then suitable solutions must be implemented, aimed at tolerating the babbling-idiot faults regardless of whether they are hardware or software.

3.1 Hardware babbling-idiot faults

A satisfactory level of protection against the hardware babbling-idiot faults is achieved by bus-replication. For a duplicated bus, a CAN network is formed by two channels; each node is endowed with two transceivers -one per channel- and sends the same data over both the channels. The two channels are physically separated, electrically isolated and dislocated along different spatial paths. The transceivers are also mutually separated and isolated. Therefore the occurrence of a fault in one transceiver or in one channel does not affect the companion one. In this way, a hardware babbling-idiot fault does not block the operation neither of the network nor of the node, i.e. the node exhibits a fault-operational behavior.

Time-triggered protocols like TTP/C [7], FlexRay [8] and SafeBus [9], the first two developed for drive-by-wire applications and the third one for aerospace applications, provide for bus redundancy; in the first two protocols the envisaged channels are two whilst in the third one they are four. Networks based on these protocols take advantage of the deterministic transmission of the messages on the replica to override a hardware babbling-idiot fault [2]. Bus redundancy, instead, is not straightforward with networks based on event-triggered protocols, like CAN, since the transmission on the replica is non-deterministic. In fact, with a bus access ruled by the bit-wise arbitration "*it is not possible to guarantee that the message order on two replicated channels will always be the same*" [10] and comparison of the data flow on the two channels is not significant. As a result, these networks necessitate of solutions different from the conventional bus redundancy to be tolerate a hardware babbling-idiot. One of them is the FlexCAN architecture illustrated in Section 4.

3.2 Software babbling-idiot faults

As mentioned in the introduction, a node affected by a software babbling-idiot fault typically sends high-priority messages at arbitrarily high rates on the bus. The possibility that it transmits formally correct (i.e. meeting the protocol) bit sequences without meaning must be also accounted for. As a matter of fact, some human involvement can introduce faults of this kind, either accidentally (f.i. a bug in the code) or intentionally (f.i. the exposition to a malicious attack). Whatever the reason may be, the most promising solution for protecting a network against a software babbling-idiot failure is the bus-guardian. It is an additional component of the node that interacts with the other components in the way of silencing the node itself when it begins to be affected by a software babbling-idiot fault. In order to prevent that a fault in the node propagates to the bus-guardian, the latter one is physically separated, electromagnetically shielded, supplied by an independent source and equipped with an own oscillator.

In time-triggered networks, each node is entitled to access the bus only during predetermined time slots and the task of the bus-guardian is eased. For instance, a straightforward technique consists in silencing the node out from its own time slots, irrespectively of its safe or faulty operation. A bus-guardian for TTP/C is presented in [11]: it is designed as a state machine, built up around a logic device and connected by four wires to the host-controller and by one wire to the transceiver.

In event-triggered networks silencing a software babbling-idiot is less easy. The time-free access to the bus from the nodes and the delay in transmitting the low-priority colliding messages entail that any implementation of a bus-guardian must reckon with the worst-case transmission time of a message. A working approach relies on the definition of a minimum inhibition time and the arrangement of the communication tasks as follows: a node has the right to access the bus at any time but is not allowed to transmit within the inhibition time. The rationale underlying this approach is that even if the node is affected by a software babbling-idiot fault, during the inhibition time it is disabled and the other nodes can communicate. The approach has been applied in [12] to CAN networks. The bus-guardian is built up with a communication-controller and a transceiver, and acts as an auxiliary node able to read (but not to write) the messages on the bus, including those of the guarded node. When, by inspecting the identifier of the messages, the bus-guardian detects that the guarded node transmits too frequently, it disables the transceiver of the node. This solution is very efficient but has two shortcomings. The first one is the complexity of the bus-guardian since it has embedded the network protocol. The second shortcoming is related to the fault detection capabilities: they do not include the case that a node babbles by transmitting messages with an incorrect identifier or error frame messages. Such a shortcoming has been eliminated with the solution presented in [13]. Here the communication-controller of the bus-guardian is connected to the output of the node instead of to the bus and hence the bus-guardian is directly aware of the rate of recurrence of the node transmission.

4. The FlexCAN architecture

As with other proposals [14], [15] the FlexCAN architecture represents an effort of enhancing the dependable characteristics of the native CAN protocol with the purpose of making it suitable for safety-critical systems. A noticeable characteristic is that the modification is made just by means of application software, using COTS (Commercial Off The Shelf) hardware components. The only particular hardware requirement is that the host-controller is able to drive two or more independent communication-controllers. At any rate, this is not a stringent requirement: at the moment, there are in the market several COTS microcontrollers that drive even more than two CAN communication-controllers (e.g., the HCS12 microcontroller of Motorola [16]).

Basically, FlexCAN relies on the replication of nodes ("replicas") and the channel. As a whole, the replicas constitute a "Fault Tolerant Unit" (FTU). There exists a hierarchical order among the replicas of a FTU: the primary node, the secondary node, and so on. In a running system, only the primary node is entitled to send messages, and the other replicas monitor its behavior. If the secondary node does not receive any message in any channel within a predetermined time-out interval, then it assumes that the primary node is failed and starts sending messages behaving as the primary node. If any replica (besides the primary) receives a message via any of the replicated channels, then the primary node is considered non-faulty. In this way, the bus replication is easily supported. On-line algorithms are provided for the nodes to agree with the hierarchical order of the replicas.

Nodes within an FTU interact among themselves transmitting and receiving "inter-group messages", which carry on application-specific data and other protocol information (like the time-synchronization information). Apart from the initialization stage, the communication period is the same for all the FlexCAN nodes, and within a period a single message identifier is not repeated. From the viewpoint of message transmission, the FTUs are organized in chains: the transmission of a FTU triggers the processing of its data and the transmission of the subsequent message by the following FTU. Since FlexCAN has been developed for control applications, the most important types of FTUs are: the sensor FTU, the controller FTU, and the actuator FTU. The sensor FTU is the input device of the system; it is configured as the first software link in the chain. The second software link is the controller FTU, the task of which is to receive the messages transmitted by the sensor, to perform some computations and to transmit the results. The actuator FTU is the output device of the system and the third software link in the chain. It receives the reference data from the controller FTU, actuates them, and sends a feedback message to the controller FTU. Much emphasis is placed in the design stage on the maximum number of the nodes that can access the bus, for avoiding problems of network traffic and non replica determinism. A FlexCAN network for a safety-critical system always has to be characterized by a small number of nodes, basically transmitting safety-related messages.

FlexCAN tolerates single hardware babbling-idiot faults, with no distinction between babbling-idiot faults due to the channel or the transceivers. A hardware babbling-idiot fault is simply decoded as a sequence of error frames that destroy the communication on the channel. However, the hypothesis of non-propagation of the faults ensures a hardware babbling-idiot fault does not affect the replicated channels, which continue to carry on the correct information both in the case that the fault is transitory or permanent.

A FlexCAN prototype network has been built at Kettering University, Flint, USA, for development and testing purposes. A network with three types of devices, namely a controller node, a sensor node, and an actuator node, was built up, where the controller FTU consists of one controller and two replicas (i.e., it is triplicated). All the nodes were instantiated by Elektronikladen HCS12 T-Boards, endowed with Motorola HCS12 microprocessors having five independent CAN channels.

The role of hardware babbling-idiot simulator was played by an additional device, formed by a transceiver connected to the bus and with the T_x pin entered by an independent digital signal source. The babbling-idiot fault was obtained by injecting continuously a 0 value in the simulator, destroying all the transmissions in the channel. The other nodes read sequences with error frames and, in turn, transmitted error frames. Even if the information carried on by that channel was destroyed, the system continued to work properly, and the actuator node never suffered from lack of information.

Despite this behavior against hardware babbling-idiot faults, FlexCAN does not cope with software babbling-idiot ones at all. As every node is physically connected to both the channels, it is possible that a host-controller continuously transmits high priority messages on all the channels thus denying any network activity. Hence the simple bus replication or triplication has no effect in this case.

5. A Simple Bus-Guardian

In this section, a simple but effective bus-guardian is presented. It has been developed, implemented, and tested in the framework of the FlexCAN architecture. Nevertheless the bus-guardian is generic enough so that it can be used with other CAN-based networks.

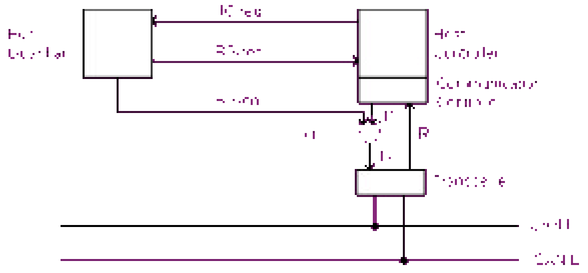


Fig.1. Bus-guardian implementation.

Bus-Guardian: BG-en	Host-Controller: T'x	OR Port: Tx	Transceiver
0 en	0	0	Dominant (0)
0 en	1	1	Recessive (1)
1 inh	0	1	Recessive (1)
1 inh	1	1	Recessive (1)

Table 1. The Truth table of the Bus-Guardian

5.1 Bus-guardian description

The bus-guardian is a simple logic device, connected to the host-controller by two wires (Fig.1). The output of the communication-controller (T'_x) and that of the bus-guardian (BG-en) are connected to an OR gate. In turn, the output of the OR gate (T_x) is connected to the input of the transceiver. The bus-guardian inhibits and enables the transmissions of the host-controller by driving BG-en as depicted in Table 1. With this configuration, the following situations occur:

- the communication-controller transmits and the bus-guardian enables the transmission: the transmission is considered correct in the time domain;
- the communication-controller tries to transmit but the bus-guardian inhibits the transmission: in this case, one of these two components is faulty (it is not possible to know a priori which one); in the FlexCAN architecture, the secondary node replaces the primary one and the time error is completely masked;
- while the communication-controller is transmitting its message, the bus-guardian changes from enabling to inhibiting the transmission: in this case, all other nodes detect the occurrence of an error (e.g., either a CRC error, a stuff error ,or other error types) and error frames are transmitted on the network.

The connection between bus-guardian and host-controller consists of two wires: one is used for sending signals from the host-controller to the bus-guardian (HC-req), and one in the other direction (BG-res). The main idea is that a host-controller has to request the bus-guardian the permission to transmit; the bus-guardian enables the transmission only if enough time has elapsed from the previous transmission (Fig.2). The bus-guardian and the host-controller use independent clocks since their operation does not necessitate being synchronous. After a given time interval, the bus-guardian inhibits the communication. In the laboratory test implementation, HC-req and BG-res were physically connected to two pins of Port A of the Motorola HCS12 microcontroller. Later on the relevant bits are used as state variables. Decoded as binary number, Port A is hence composed by HC-req as the least significant bit and by BG-res as the second least significant bit. That is

Port A = [BG-res, HC-req]

The bus-guardian activates two timers for pacing the time intervals mentioned above:
- inhibit_time (t_i): during this interval the bus-guardian inhibits the node transmissions;
- write_time (t_w): during this interval the bus-guardian enables the node transmissions.

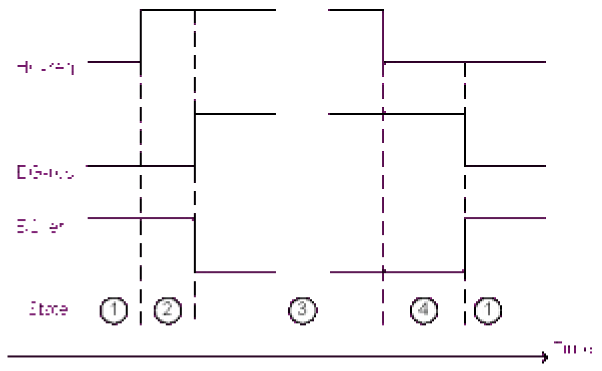


Fig. 2. Bus-guardian timing.

The algorithm established for the bus-guardian can be described as a four-state machine (Fig.3), where the states are given by the Port A value. The following text describes the state evolution within the algorithm:

State 1) The bus-guardian inhibits the transmission but the host-controller has no messages to send. In this situation, Port A = 0, HC-req = 0, BG-res = 0 and BG-en = 1.

State 2) The host-controller requests the communication by setting HC-req (Port A = 1). If the inhibit_time has already expired, the bus-guardian enables the communication (BG-en = 0).

State 3) The bus-guardian enables the communication by clearing BG-en and reports the enabling action to the host-controller by setting BG-res (port A = 3). The write_time interval starts and, during this period, the host-controller is enabled to transmit messages.

State 4) After transmission, the host-controller clears HC-req (port A = 2).

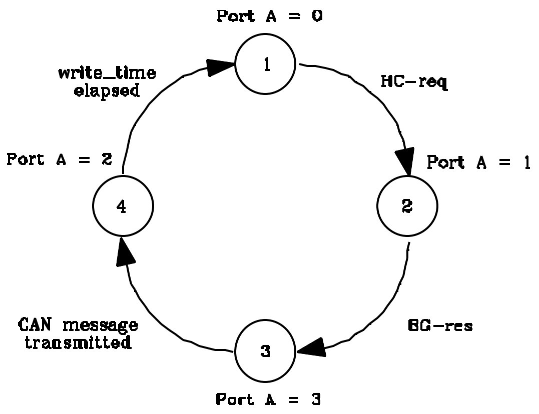


Fig.3. Bus-guardian state machine.

To illustrate the algorithm, Fig.4 lists the transmission routine for a typical host-controller with a single thread; at any rate, this issue is not important as the actual bus-guardian is a simple device that does not involve any operating system. The transmit function is invoked whenever a message has to be transmitted. It requires as an input argument a pointer to a variable of type TxMessage. TxMessage is a data structure that describes the characteristics of the messages. The CANTransmit function invokes the transmission of the message to the communication-controller. Before the transmission, the transmission_request function is invoked for setting HC-req and waiting for BG-res. When BG-res is found to be 1, then the transmission is granted to the communication-controller. After the invocation of the transmission_end function, HC-req is cleared. Although the bus-guardian was implemented on a separate HCS12, simpler devices (e.g. a general-purpose timer) can be used.

```
void transmission_request (void)  {
 PORTA = 0x01;              // state 2
 while (PORTA != 0x03) { } // wait for the BG-res
}
void transmission_end (void)  {
   PORTA = 0x02;            // state 4
}
void transmit (struct TxMessage *message)  {
     transmission_request();
     CANTransmit (message);
     transmission_end();
 }
```

Fig.4. Transmission routine of the host-controller.

Fig.5 lists the C code of the main segment of the bus-guardian routine. The timer drivers were programmed so that the associated variables (inhibit_time and write_time) are equal to 0 within the intervals and different from 0 after the intervals are completed.

5.2 Time parameter choice

Some guidelines for choosing the parameters t_w, and t_I are given below for a generic message *m*. The write_time t_w must be greater than the worst-case message latency R_m, that is

$$\tau_w > R_m \quad (1)$$

where [17]

$$R_m = C_m + J_m + I_m \quad (2)$$

In (2) C_m is the transmission time of the message, J_m is the queuing jitter, and I_m is the queuing delay referred to as interference time. I_m is the sum of the longest time that all the messages with priority higher than *m* can occupy the bus plus B_m, which is the longest transmission time taken by a message with priority lower than *m*.

```
while (TRUE) {
  if (inhibit_time > 0) {
   if (PORTA == 0x01) {
      BGEN = 0                // enable transmission
      PORTA = 0x03;          // state 3
      write_time = 0;        // start the timer
      while (write_time == 0) { } // wait for timer
      PORTA = 0x00           // state 1
      BGEN = 1;             // disable transmission
      inhibit_time = 0;       // start the timer
   }                          // end if PORTA
  }                           // end if inhibit_time
}                             // end while
```

Fig.5. Main segment of the bus-guardian routine.

In the FlexCAN architecture we assume that the data sources send only one message per period (denoted by T_s). Moreover, by assuming that all the periods are of the same length, it is

$$I_m = NC_{max} + B_m \quad (3)$$

where N is the number of higher priority messages and C_{max} is the maximum transmission time of the higher priority messages. For CAN 2.0A, the transmission time of a message C_m is given by

$$C_m = [\lfloor (34 + 8d_m)/5 \rfloor + 47 + 8d_m]\tau_b \quad (4)$$

where d_m is the message size (length of the message data field in bytes), t_b is the time to transmit one bit on the bus (for a transmission rate of 1 Mbit/s it is 1 μs) and the symbols + and + stand for truncation of the decimal value.

One approach is to have the write_time plus the inhibit_time equal to the period of the data sources, i.e.

$$\tau_w + \tau_i = T_s \quad (5)$$

However, because of some small variability in the transmission times due to the error frames, the following position is preferred

$$\tau_w < \tau_i < T_s - \tau_w \quad (6)$$

6. Error modes

In the case of a babbling-idiot fault of the host-controller, the latter one starts transmitting messages without taking into account the signals of Port A. The following error modes can be considered for a faulty host-controller:

Mode 1) If HC-req=0, then the bus-guardian never enables the communication. The error is tolerated in the FlexCAN environment.

Mode 2) If HC-req=1, then the bus-guardian alternately enables and inhibits the communication on the basis of the elapsing of the inhibit_time and write_time intervals. This means that the buses will be busy during the write_time by the messages sent by the faulty host-controller and free during the inhibit_time.

```
void transmission_request (void) {
   PORTA = 0x01;
   while (PORTA != 0x03) { }
}
void transmission_end (void) {
   PORTA = 0x02;
}
void transmit (struct TxMessage *message) {
    transmission_request();
    CANTransmit (message);
    transmission_end();
 }
 void main (void) {
    while (TRUE) {
     transmit (message);
    }
 }
```

Fig.6. A babbling-idiot simulator code for mode 3).

Mode 3) Another possibility is that the host-controller takes into account Port A signals but tries to transmit a great deal of messages overwhelming the network. The code listed in Fig.6 exemplifies this error mode. Note that under error mode 3), the behavior of the network is exactly the same as mode 2.

A faulty bus-guardian can lead to the following error modes:

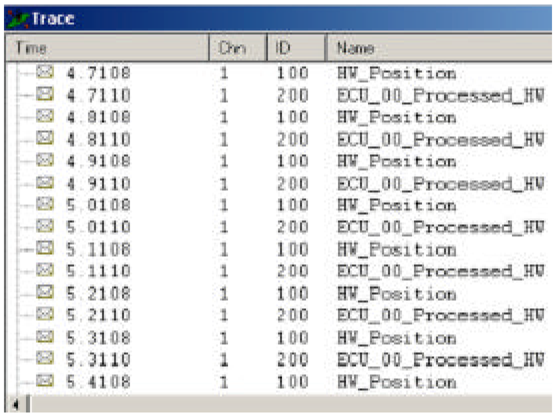
Mode 4) The bus-guardian inhibits permanently the communication by setting BG-en; in this case, the host-controller is kept silent. This mode is tolerated in a FlexCAN architecture, because the secondary node takes the place of the silent primary node.

Mode 5) The bus-guardian enables permanently the communication and the host-controller can transmit messages at any time. The working hypothesis is the occurrence of one fault at a time. If both bus-guardian and host-controller become faulty, then the hypothesis is not fulfilled.

Mode 6) While the host-controller is transmitting, the bus-guardian suddenly inhibits the communications. This is due to an error in the timing of the bus-guardian. Some error frames in the network appear and the message is transmitted as soon as the bus-guardian allows again the communications, provided that the bus is idle.

7. Laboratory Tests

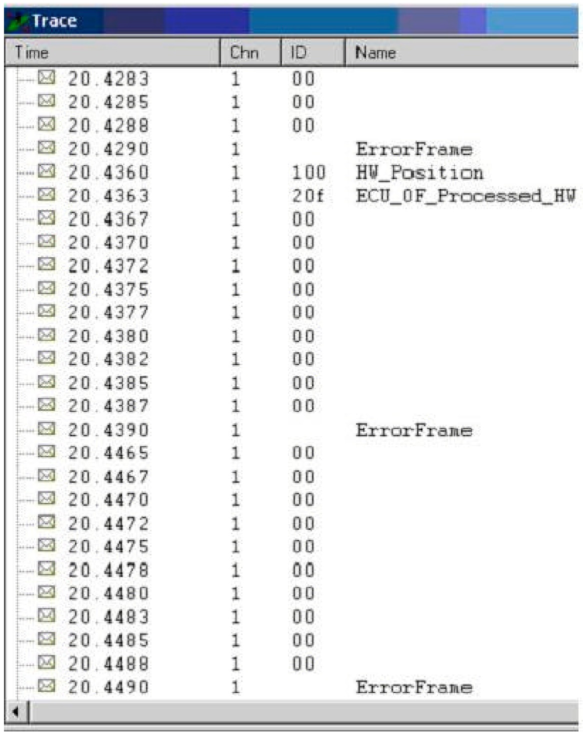
Laboratory tests were carried out on the FlexCAN prototype described in Section 4. As a CAN analyzer, CANoe, a tool marketed by CANTech Inc., has been used. In the experiment, the sensor node transmits its messages every T_s=100ms (identifier 0x100). The three nodes of the triplicated controller FTU read the message and respond appropriately. The primary node (HC0) sends its message with identifier 0x200, whilst the secondary node (HC1) sends its message (0x20F) when HC0 fails. The behavior of the third node (HC2) is not described here as it is immaterial for the experiment. The primary node embeds a bus-guardian with an inhibit_time of 8ms and a write_time of 2ms. Fig.7 shows a CANoe trace containing the messages sent during correct network operation (the time resolution of the trace program is 0.1ms).


Trace

Time	Chn	ID	Name
4.7108	1	100	HW_Position
4.7110	1	200	ECU_00_Processed_HW
4.8108	1	100	HW_Position
4.8110	1	200	ECU_00_Processed_HW
4.9108	1	100	HW_Position
4.9110	1	200	ECU_00_Processed_HW
5.0108	1	100	HW_Position
5.0110	1	200	ECU_00_Processed_HW
5.1108	1	100	HW_Position
5.1110	1	200	ECU_00_Processed_HW
5.2108	1	100	HW_Position
5.2110	1	200	ECU_00_Processed_HW
5.3108	1	100	HW_Position
5.3110	1	200	ECU_00_Processed_HW
5.4108	1	100	HW_Position

Fig.7. Messages transmitted during correct network operation.

In the following the situation of a primary node affected by a babbling-idiot fault is considered (error mode 2). Even if the faulty node starts sending messages, the behavior of the network is not impaired as one can see from Fig.8. The effects of the babbling-idiot fault cover just the 2ms time interval of the write_time. During this interval (e.g., Time 20.4367 to 20. 4387), the channel is taken by the babbling-idiot faulty node sending high priority messages (ID=00) and no other node accesses to the bus. During the 8ms time interval of the inhibit_time interval (e.g., Time 20.4387 to 20. 4465), the inhibition of the node produced by the bus-guardian permits the other nodes to send their messages. Thus, the sensor node and HC1 can send their messages although HC0 is affected by a babbling-idiot fault. Being that some messages sent by the faulty node are cut during their transmission, error frames are found at the end of every write_time interval. It is worth to note that the babbling-idiot fault is not removed but masked; at any rate, the system works properly.


Trace

Time	Chn	ID	Name
20.4283	1	00	
20.4285	1	00	
20.4288	1	00	
20.4290	1		ErrorFrame
20.4360	1	100	HW_Position
20.4363	1	20f	ECU_0F_Processed_HW
20.4367	1	00	
20.4370	1	00	
20.4372	1	00	
20.4375	1	00	
20.4377	1	00	
20.4380	1	00	
20.4382	1	00	
20.4385	1	00	
20.4387	1	00	
20.4390	1		ErrorFrame
20.4465	1	00	
20.4467	1	00	
20.4470	1	00	
20.4472	1	00	
20.4475	1	00	
20.4478	1	00	
20.4480	1	00	
20.4483	1	00	
20.4485	1	00	
20.4488	1	00	
20.4490	1		ErrorFrame

Fig.8. Messages transmitted under a babbling-idiot fault.

8. Conclusions

Two types of babbling-idiot faults have been analyzed: hardware and software. Replicated components are effective to overcome hardware babbling-idiot faults while bus-guardians are used to tolerate software babbling-idiot faults. The specific implementation of bus guardians depends on the network architecture, its protocol, and the kind of protection desired (i.e. time domain and/or value domain). A simple bus guardian suitable for the

FlexCAN architecture, providing protection in the time-domain, has been presented,. The bus guardian has been implemented and tested yielding satisfactory results.

Acknowledgements

The authors wish to thank Vector CANTech Inc. for supplying equipment, to Mark Wicks for his continued support of the FlexCAN architecture, to Douglas Melton for his help with the bus-guardian design, and to J. Kozlowsky for his help with the hardware prototypes.

References

[1] B.W.Johnson, “Design and Analysis of Fault Tolerant Digital Systems”, Addison Wesley Publishing, 1989.

[2] H.Kopetz, “Real-Time Systems: Design Principles for Distributed Embedded Applications”, 1997. Boston: Kluwer Academic Publishers.

[3] G.Heiner and T.Thurner, "Time-triggered architecture for safety-related distributed real-time systems in transportation systems", Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 23-25 June 1998, pp. 402-407.

[4] J.R.Pimentel and J. Kaniarz, “A CAN-Based Application Level Error-Detection and Fault-Containment Protocol”, Proc. of 11th IFAC Symp. on Information Control Problems in Manufacturing (INCOM), Salvador, Brazil, 2004.

[5] “CAN Specifications version 2.0”, Robert Bosch gmbh, available at http://www.can.bosch.com/docu/can2spec.pdf.

[6] K.Etschberger, “Controller Area Network, Basics, Protocols, Chips and Applications”, IXXAT Press, Germany, 2001.

[7] “Time-Triggered Protocol TTP/C, High-Level Specification, Document Protocol Version 1.1”, available at http://www.ttagroup.org/technology/specification.htm.

[8] “FlexRay Communication System, Protocol Specification 2.0”, available at http://www.flexray.com/specification_request.php.

[9] K.Hoyme and K.Driscoll, “SAFEBus”, IEEE/AIAA. Proc. of 11th Digital Avionics Systems Conference, 1992.

[10] H.Kopetz, “A Comparison on CAN and TTP”, available at the web page: http://www.tttech.com/technology/docs/protocol_comparisons/HK_1998-99 Comparison-TTP.pdf

[11] C.Temple, “Avoiding the babbling-idiot failure in a time-triggered communication system”, Fault-Tolerant Computing, Twenty-Eighth Annual International Symposium, 23-25 June 1998, pp. 218 - 227.

[12] I.Broster and A.Burns, “An analysable bus-guardian for event-triggered communication”, 24th IEEE Real-Time Systems Symposium, 3-5 Dec. 2003, pp. 410–419.

[13] J.Ferreira, E.Martins, P.Pedreiras, J.Fonseca and L.Almeida, “Components to Enforce Fail-Silent Behavior in Dynamic Master-Slave Systems”, 5th IFAC, International Symposium on Intelligent Components and Instruments for Control Applications, Aveiro, Portugal, July 2003.

[14] J.Ferreira, P.Pedreiras, L.Almeida, and J.Fonseca, “Achieving fault tolerance in FTT-CAN”, Factory Communication Systems, 2002. 4th IEEE International Workshop on Factory Communication Systems, Vasteras, Sweden, 28-30 Aug. 2002, pp. 125-132.

[15] T.Fuehrer, B.Mueller, W.Dieterle, F.Hartwich, R.Hugel, and M.Walther, “Time-Triggered Communication on CAN”, R.Bosch gmbh, available at http://www.can-cia.org/can/ttcan/fuehrer.pdf .

[16] http://www.freescale.com/webapp/sps/site/taxonomy.jsp?nodeId=0162468636bJwn.

[17] K.Tindell, A.Burns, and A.J.Wellings, “Calculating Controller Area Network (CAN) Message Response Time”, Control Engineering Practice, 3(8), pp. 1163-1169, 1995.