

COMPONENTS TO ENFORCE FAIL-SILENT BEHAVIOR IN DYNAMIC MASTER-SLAVE SYSTEMS

Joaquim Ferreira ^{*,1} Luís Almeida ^{**}
Ernesto Martins ^{**} Paulo Pedreiras ^{**,2}
José A. Fonseca ^{**}

** EST - Instituto Politécnico de Castelo Branco
P-6000 Castelo Branco, Portugal
Email: jjf@est.ipcb.pt*

*** DET-IEETA, Universidade de Aveiro
P-3810-193 Aveiro, Portugal
Email: {lda, evm, pedreiras, jaf}@det.ua.pt*

Abstract: This paper considers the case in which master-slave fieldbus networks are used in safety-critical embedded applications, such as transportation systems. Traditional approaches to system design, due to fault-tolerance reasons, have considered static cyclic table-based traffic scheduling, only. However, there is a growing demand for flexibility and integration, mainly to improve efficiency in the use of system resources, with the network playing a central role to support such properties. This calls for dynamic on-line traffic scheduling techniques so that dynamic communication requirements are adequately supported. This paper considers such dynamic master-slave architectures and addresses the problem of enforcing fail silent behavior both in the master and in the slave nodes. Two different mechanisms are proposed, one based on dynamic bus guardians for the slave nodes only, to impose fail silent behavior in the time domain, and other based on internal replication and temporized agreement, to impose fail silence both in the temporal and value domains. Despite being potentially applicable to a set of master-slave networks, this paper discusses the specific implementation of the proposed mechanisms on top of the FTT-CAN protocol. *Copyright ©2001 IFAC.*

Keywords: fault tolerance; distributed systems; fieldbus; safety-critical; fail-silence.

1. INTRODUCTION

Many safety-critical embedded systems used today, e.g. in transportation systems, are distributed and rely on a fieldbus network that interconnects

sensors, actuators and controllers in a reliable and timely way. One popular network access control paradigm that is used in many of these applications is the master-slave paradigm, in which a single node controls the traffic on a bus using a cyclic traffic-dispatching table. Several examples can be pointed out, such as WorldFIP (IEC, 2000) and TCN (IEC, 1999), largely used in train control systems, as well as MIL-STD-

¹ This work was partially supported by Ministério da Educação under a grant of PRODEP III.

² This work was partially supported by FCT under a PRAXIS XXI grant BD/21679/99.

1553B (Haverty, 1986), which has been used for a long time in the USA, for distributed hard-real time applications. Foundation Fieldbus FF-H1 (IEC, 2000) is an important protocol in the domain of industrial automation that can also be categorized as master-slave. Despite using mixed transmission control combining master-slave and token-passing mechanisms, the former has priority over the latter and the whole operation in each segment (link) is still coordinated by a master node called Link Active Scheduler (LAS). Some of the interesting properties of this type of networks are the inherent global synchronization with respect to the master since this node explicitly tells each slave when to transmit, as well as the relative simplicity of supporting dynamic communication requirements because these are concentrated on a monolithic structure, the communication requirements database, in the master node, only. This simplifies the admission control of change requests and reduces the respective reaction time with respect to distributed transmission control alternatives. With this type of operational flexibility, one can turn on and off the transmission of message streams, or vary the respective transmission rates, according to the run-time needs of the system, or even change the traffic scheduling policy on-line. This results in a higher efficiency of network utilization, freeing bandwidth that can be used to serve more streams or to facilitate error recovery. On the other hand, master replication is essential to avoid the single point-of-failure. This issue is, however, out of the scope of this paper and has already been addressed in other publications (Ferreira *et al.*, 2002; Martins *et al.*, 2002). In dynamic master-slave systems based in a broadcast bus, a faulty node that sends unsolicited messages at arbitrary points in time (babbling idiot failure mode (Kopetz, 1997)) without respecting the bus access rules imposed by the master node can disable nodes with legitimate messages to access the network. In a dynamic master-slave system, the master node alone is responsible for controlling the slave nodes access to the bus. After the reception of an explicit command from the master to produce a message, a slave node has the exclusive right to transmit a specific message during a certain time interval. In this paper, mechanisms to enforce fail silent behavior both in the master and in the slave nodes are proposed, discussed and analyzed. Fail silent behavior in the time domain, for the slave nodes only, is enforced based on bus guardians, while fail silence both in the temporal and value domains, applicable to both master and slaves, is based on internal replication and temporized agreement. The concepts behind the proposed mechanisms may be generically applied to master-slave networks that support dynamic or multiple traffic dispatching tables, which are essential to support dynamic

communication requirements. In the former case, such as in FF-H1, there are already mechanisms (dynamic scheduling profile) to support on-line changes to the traffic-dispatching table. In the latter case, such as in WorldFIP, several traffic dispatching tables can co-exist and be swapped on-line. This swapping mechanism can be used to support dynamic communication requirements (Almeida *et al.*, 1999). Despite the possible use of the proposed mechanisms on several different master-slave networks, its effective implementation is dependent on the particular network. In this paper we discuss the enforcement of fail silent behavior of nodes communicating on top of Controller Area Network (CAN) using FTT-CAN (Almeida *et al.*, 2002), which can also be categorized as a master-slave communication protocol specifically developed to support dynamic communication requirements handled in a time-triggered fashion. The paper is organized as follows, in the next section the problem is clarified and existing solutions are discussed. In section 3, the target architecture is shown as well as the considered fault hypothesis. Section 4 presents the mechanisms to enforce fail silence while section 5 concludes the paper.

2. PROBLEM STATEMENT AND RELATED WORK

In dynamic master-slave networks the master holds the current communication requirements, e.g. period, relative phasing, deadline and transmission time of message streams, in a specific database. For compatibility with the nomenclature used further on in the scope of FTT-CAN, we will refer to this database as the synchronous requirements table (SRT). This table is scanned on-line by the traffic scheduler running in the master, to trigger bus transactions. Thus, the master explicitly instructs the slave nodes to produce the appropriate messages. However, it may do so either in a message-by-message basis (e.g. WorldFip and TCN), or in an elementary cycle basis (e.g. FTT-CAN). In the first case, each slave message is transmitted upon reception of the respective triggering master message. In the second case, a single master message triggers the production of several slave messages during an elementary cycle. In both situations the slave nodes are not aware of the bus schedule, that might be dynamic and which is local to the master node. Thus, the mechanisms and techniques to enforce fail silent behavior in the slave nodes must also be dynamic. In this paper we address the specific issue of enforcing fail silence by means of hardware components attached to the nodes' network interface. Such hardware components double-check the correctness of the node transmissions, and isolate the node from

the network in case of erroneous behavior. Moreover, they must be completely transparent from the point of view of the application software running in the node host processor. In our target architecture, and considering the slave nodes, we propose the use of bus guardians to enforce fail silent behavior in the time domain only, that is, inhibit faulty nodes from transmitting messages at arbitrating points in time wasting bandwidth and disturbing legitimate traffic. The proposed bus guardians are programmed at initialization time by the node host processor, with the set of messages that the node is supposed to transmit. At run time, the bus guardians police the traffic generated by the node by receiving the triggering messages transmitted by the master and verifying the correctness of the nodes' transmissions. Notice that the node host processor is unable to interfere with the bus guardian except during initialization time and that the bus guardian never transmits to the network. Concerning fail silence enforcement in both time and value domain, as required for the master and possibly for some slaves, we propose the use of specific network interface that supports internal duplication of the master node in a transparent way. Basically, this interface enforces an agreement both in the time and in the value domain between all the messages generated by both internal replicas. In case of disagreement, no message is actually sent to the network. The absence of message transmission can be detected by a backup node, which eventually replaces the faulty one.

2.1 Related work

In the scope of safety-critical embedded systems based in fieldbuses (e.g. TTP/C (TTTech, 2002) and FlexRay (Belschner, R. *et al*, 2002)), fail silence enforcing relies ultimately in bus guardians. Bus guardians are necessary to disable network access from faulty nodes that otherwise could occupy scheduled time of other nodes. In a limit situation, the babbling idiot fault mode where a faulty interface transmits constantly, all legitimate network traffic can be disrupted. To control this failure mode the bus guardian filters all interface transmissions to the network. To be effective in the filtering, the bus guardian must be fail-independent with respect to the interfaces it monitors, i.e. it must belong to a separate FCU (Fault Confinement Unit), needs to have its own copy of the schedule and an independent knowledge of the time (Rushby, 2001). In SAFEbus, the network interface is fully duplicated (the bus itself is also duplicated) and each replica (Bus Interface Unit - BIU) acts as a guardian of the other. Furthermore, each BIU makes its own clock synchronization and has its own copy of the schedule. In TTP/C

each node has access to two bus lines and thus has two network interfaces each with its own bus guardian. Each bus guardian (Temple, 1998) has an independent copy of the schedule and their own clock oscillator, however they do not synchronize independently and share the same power supply and the same encapsulation as their controllers. Consequently common mode failures are possible. Recently, the bus guardianship has been moved to a central hub, which is capable of performing its own clock synchronization. This new version is a fully independent FCU, but is at the same time a single point of failure. The duplication of the central star hub overcomes this issue. FlexRay uses an approach that is very similar to TTP/C but just for time triggered traffic. FlexRay bus guardians do perform independent clock synchronization. Notice however that SAFEbus, TTP/C and FlexRay all are based on static offline scheduling, i.e. the scheduling tables located at each bus guardian are fixed even though they may support the coexistence of different operational modes, as in TTP/C.

3. TARGET ARCHITECTURE

As referred before, we tested our approach on an FTT-CAN system (Almeida *et al.*, 2002). In this protocol, bus time is broken in consecutive fixed duration cycles called Elementary Cycles (ECs). Each EC is triggered by a special message called Trigger Message (TM) that not only triggers the start of a new EC but also carries the so-called EC-schedule, which is the list of messages that must be produced (transmitted) in the respective EC. The slave nodes that are producers of the specified messages transmit them. The native MAC of CAN sorts out collisions at bus access within the EC. By sending only one control message per EC, the bandwidth efficiency of the protocol is substantially improved with respect to conventional master-slave access control.

The protocol also supports synchronous (periodic) and asynchronous (aperiodic) traffic within two disjoint windows in the EC (Fig. 1). The master schedules the synchronous traffic, only. Finally, the synchronous communication requirements are stored in a table called Synchronous Requirements Table (SRT). On-line changes to the SRT go through an admission control that accepts them only if the overall traffic schedulability is guaranteed.

Beyond the enforcement of fail silence both in the master and in slave nodes, which is the focus of this work, there are other issues related to fault-tolerance aspects in FTT-CAN that were already dealt with in recent work. Particularly, two such aspects are the mechanism for master failure de-

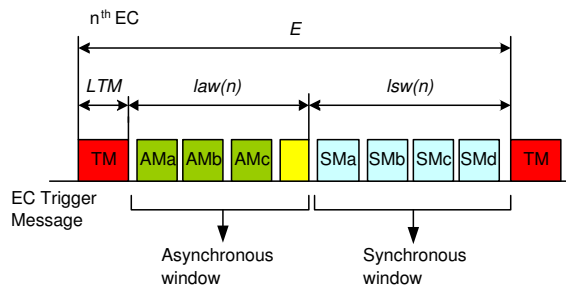


Fig. 1. Double phase elementary cycle with synchronous and asynchronous windows.

tection and replacement (Ferreira *et al.*, 2002) and the fault-tolerant scheduling of synchronous messages (Ferreira *et al.*, 2001).

3.1 Fault hypothesis

The fault hypothesis upon which this work is based considers:

- Physical faults such as those caused by electromagnetic interference, defects or damage (external faults).
- No support for spatial proximity faults, as common in any bus-based topology, because the destruction of a single node may also, with high probability, disrupt the network interface and the network connectivity itself.
- As common in all bus-based systems, physical bus partition is not tolerated unless the bus is replicated and the partition affects just one of the busses.
- Slave nodes are fail-silent in the time domain, although they can also be fail silent in the value domain, while master nodes are always fail silent both in the time and in the value domain.

4. ENFORCING FAIL SILENCE IN FTT-CAN

Two different approaches are proposed in this paper to enforce fail silent behavior both in the master and in the slave nodes. Fail silence in the slave nodes can be enforced either using dynamic bus guardians or internal replication and temporized agreement. The latter mechanism enforce fail silent both in the time and in the value domain and it was designed to be used primarily on the master nodes, given its higher cost. Notice however that internal replication and temporized agreement can also be adopted at the slave nodes whenever needed. Dynamic bus guardians in their turn are to be used on the slave nodes only, since they cannot be adopted in the master nodes.

4.1 Brief overview of can error detection capabilities and fault confinement mechanisms

In CAN each station that detects an error, sends an error flag composed of six consecutive dominant bits enabling all stations on the bus to be aware of a transmission error. The frame affected by the error automatically re-enters into the next arbitration phase. The error recovery time (the time from detecting an error until the possible start of a new frame) varies from 17 to 31 bit times. To prevent an erroneous node from disrupting the functioning of the whole system, e.g. by repetitively sending error frames, the CAN protocol includes fault confinement mechanisms that are able to detect permanent hardware malfunctioning and to remove defective nodes from the network. To do this a CAN controller has two error counters; the transmit error (TEC) and the receive error (REC) counters which are incremented/decremented according to a set of rules (BOSCH, 1991). Despite CAN efficient implementation of error detection capabilities and automatic fail-silence enforcement, referred above, a CAN node only reaches the bus off state (fail silence) after a relatively long period of time (when the TEC reaches 255). For example, in the case of an erratic transmitter in a 32 node CAN network at 1 Mbps, the worst-case time to bus off is 2.48 ms (Rufino and Veríssimo, 1997). Moreover, a CAN node running an erroneous application can also compromise most of the legitimate traffic scheduled according a higher layer protocol implemented in software in a standard CAN controller, simply by accessing the network at arbitrary points in time. Notice that a faulty application running in a node with a CAN controller may transmit at any time without causing any network errors, and consequently unable of leading the CAN controller to the bus-off state, simply by using the highest message priorities. An uncontrolled application transmitting at arbitrary points in time via a non-faulty CAN controller is a much severe situation than a faulty CAN controller also transmitting at arbitrary points in time because, in the first case, a non faulty CAN controller has no means to detect an erroneous application. In the second case the CAN controller would enter bus-off state after a while.

4.2 Dynamic bus guardians

Implementing a bus guardian in CAN is not a trivial task. CAN protocol works in a bit by bit basis and at any instant any node may transmit (e.g. start an error frame) depending of its local view of the bus. That is, the transmission and reception signals are closely related at the bit level and not on the frame level as in other

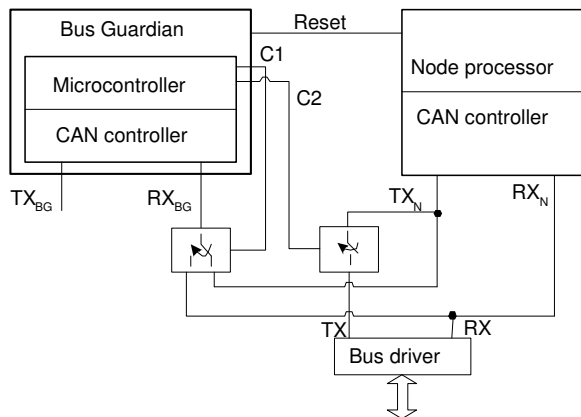


Fig. 2. Implementing the bus guardian based on an off-the-shelf CAN controller.

fieldbuses, e.g. WorldFIP, TTP/C and FlexRay. Thus the typical technique used in bus guardians for the last two networks, i.e. isolating the respective node from the network outside its allowed transmission/reception windows, is not possible in CAN because, even outside such windows, CAN nodes must answer to every frame with the acknowledge bit and must participate in the resynchronization process upon errors by transmitting error frames. In our approach the CAN controller of a non-faulty node participates in every network transaction. It is isolated from the network, by the bus guardian, only if it attempts to corrupt the bus schedule. The bus guardians proposed for the slave nodes are initially programmed, at reset time, by the node host processor with the set of messages that the node is responsible to transmit. At run time the bus guardian is programmed by the network master, for every elementary cycle, by means of the trigger message. Both the bus guardian and the node CAN controller (connected to the node host processor) receive the master trigger message in parallel. Notice that the node host processor is unable to interfere with the bus guardian except during initialization time. Two design options were considered for implementing the bus guardians. One is based on an off-the-shelf CAN controller and the other one is based on specialized hardware that implements a subset of the CAN protocol, only.

4.2.1. Bus guardian using an off-the-shelf CAN controller A possible solution to implement the FTT-CAN bus guardian is to use a second standard CAN controller, that supports silent mode operation, attached to a dedicated microcontroller. This ensemble is connected in parallel with the node host processor CAN controller, hereafter also called main controller, and before the bus driver, to monitor and control the operation of the node (Fig. 2).

Initially, the bus guardian receives the network traffic, i.e. $RX_{BG}=RX_N=RX$, in parallel with

the main CAN controller. After receiving and decoding a trigger message the bus guardian waits for the beginning of the synchronous window to enter policing mode. In this mode, the reception of its CAN controller is switched to the output of the node main CAN controller, i.e. $RX_{BG}=TX_N$, and starts policing the network traffic generated by the node main CAN controller until the end of the synchronous window. After that it goes back to the initial state and the whole process is repeated every cycle. One of the limitations of this approach comes from the fact that the bus guardian microcontroller only becomes aware of any message after it has been fully received by the CAN controller. Therefore, the error detection latency (Fig. 3) is at least as long as the transmission time of one message plus the time taken by the microcontroller to check the message validity (receive interrupt latency and processing - RX ISR). If the RX ISR duration is shorter than the CAN inter-frame space, i.e. 3 bit times, the node will be isolated before any subsequent transmission can start. Thus, the worst-case fail silence enforcement latency will equal 136 bit times corresponding to 133 bit times for a maximum CAN 2.0A message plus 3 bit times for the inter-frame space. However, this latency can be longer if the RX ISR duration extends beyond the CAN inter-frame space. In this case, the node may start another transmission before the bus guardian takes action and isolates it from the network. Then, the bus guardian aborts the on-going transmission immediately and an error is generated in the network causing interference from error frames between 17 and 31 bit times. The worst-case fail silence enforcement latency will thus be 164 bit times (133 + 31) plus the RX ISR duration.

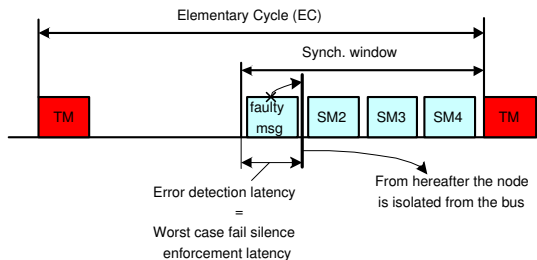


Fig. 3. Fail silence enforcement using a bus guardian based on a standard CAN controller.

For small sized synchronous windows and low bus transmission rate, a faulty node can corrupt a significant part of, or even all, the synchronous window. However, it only corrupts a single synchronous window since the node is then isolated from the network by the bus guardian. After disabling the bus access, the bus guardian may also cause a hard reset the node host processor and its respective CAN controller, if programmed to do

so at initialization time. This might be helpful for the case of a non-permanent failure, allowing to node to recover autonomously.

4.2.2. Specialized hardware Another approach is to use specialized hardware that implements a subset of the CAN protocol and is able to examine the validity of an ongoing message transmission, right after the transmission of the respective ID and DLC (Fig. 4). Notice that a message may also be erroneous due to a wrong size (DLC error). In this approach, since the bus guardian can detect an erroneous message earlier, it can also take action sooner and thus, its worst-case fail silence enforcement latency is shorter.

Upon trigger message reception, the bus guardian decodes it, extracts the node schedule for that EC and the synchronous window length (Table 1). From the EC-schedule, the bus guardian also derives the number of messages to transmit (N_{msg}) in that EC. The operation of the bus guardian during the synchronous window is based on its capacity to observe the network state (Rx_{bus}) and the node transmission state (Tx_{node}), independently. Together with the knowledge of N_{msg} , the bus guardian drives the bus transmission line (Tx_{bus}) according to the following rules:

- (1) If $N_{msg} > 0$ and the bus is idle than $Tx_{bus} = Tx_{node}$. This allows the node to start a message transmission. N_{msg} is decremented for each successful transmission.
- (2) If $N_{msg} = 0$ and the bus is idle than $Tx_{bus} = recessive$ (the node cannot start a message transmission).
- (3) If a transmission from other node is in progress than $Tx_{bus} = Tx_{node}$, but only to transmit acknowledge bits, error frames or overload frames. At all other times $Tx_{bus} = recessive$.

Whenever, the node starts a transmission during the synchronous window with $N_{msg} > 0$, the bus guardian policies the transmission bit by bit until the end of the message ID and the DLC fields. If one of these fields is incorrect, the bus guardian immediately aborts the on-going transmission, inducing an error in the bus and isolating the node from the network. Similarly to the previous case, a hard reset to the host node processor and CAN controller may also be generated. In the prototype implementation, the bus guardians have independent voltage and clock sources. The FTT-CAN bus guardian shares most of building blocks of a standard CAN controller, except the ones directly related with message transmission since it does not transmit. It is based on an IP core developed at University of Aveiro (Arqueiro and Oliveira, 2003). In this approach, the worst-case fail silence enforcement latency (WCFSL)

corresponds to the error detection latency, i.e. 22 bit times (SOF to DLC plus stuff bits) plus the hardware error detection delay ($\Delta_{Hardware}$), plus the time taken by the transmission of error frames induced by the bus guardian when the on-going transmission is aborted, i.e. between 17 and 31 bit times (Fig. 5). The resulting value can be obtained by expression (1).

$$WCFSL = \lceil (22 + 31 + \Delta_{Hardware}) \rceil \times \delta_{bit}(1)$$

If $\Delta_{Hardware}$ is of the order of magnitude of a few bit times, then, WCFSL will be close to 52 bit times, which is the transmission time of a minimum-sized CAN frame. Thus, the interference caused by aborting an erroneous transmission is always shorter or equal to the time allocated by the master to a correct message. Since a node issuing an erroneous message must have $N_{msg} > 0$, this means that the interference uses the bus time allocated to that node, with practically no interference on the bus schedule.

4.3 Internal replication and temporized agreement

For the case of the master node, and possibly some slave nodes, it is necessary to enforce fail-silence both in the time and value domains. This is mandatory in the master, to guarantee the correctness of the EC-schedule that is broadcast to the network. This approach is based on a specific network interface that supports internal duplication of the node in a transparent way. Basically, this interface enforces an agreement both in the time and in the value domain between all the messages generated by both internal replicas. In case of disagreement, no message is actually sent to the network. Fig. 6 depicts such a network interface designed for an FTT-CAN system called Dual Processor- CAN controller Interface (DPCI). The CPUs run from independent clocks and each of them is connected to a dedicated DPCI port (on the left side) through which it sees the CAN controller programming interface as if it was physically connected to it. Synchronization between CPUs is enforced by semi-synchronous port interfaces.

The CAN controller is connected on the right side port. This port is customized according to the interfacing requirements of the specific controller being used. Apart from that, the datapath shown in Fig. 6 is standard for any CAN controller type. The bus transactions initiated by the CPUs are translated by the DPCI in read or write CAN controller accesses. All CPU writes directed from each port to any controller register, go through a separate Write FIFO memory. These memories decouple the two CPUs, allowing them to run at their own pace. The Comparator unit compares at

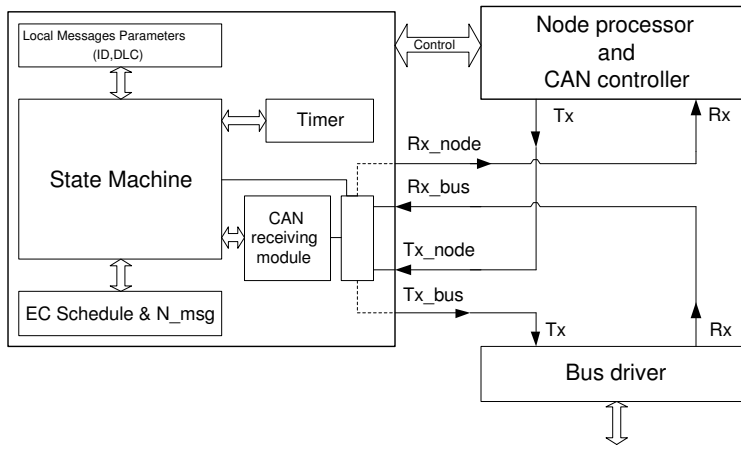


Fig. 4. Bus guardian integration in the node.

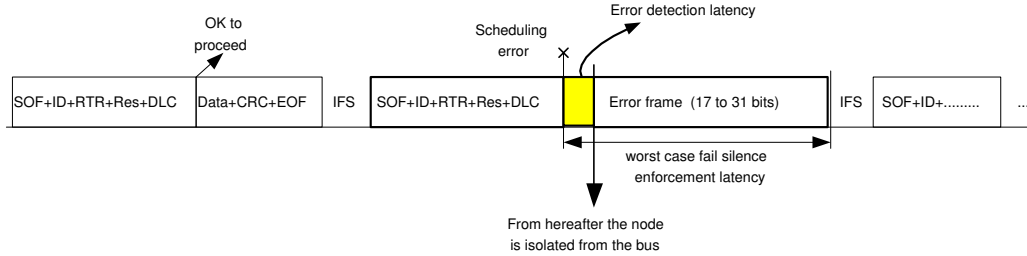


Fig. 5. Enforcing fail silence with bus guardians based on specialized hardware.

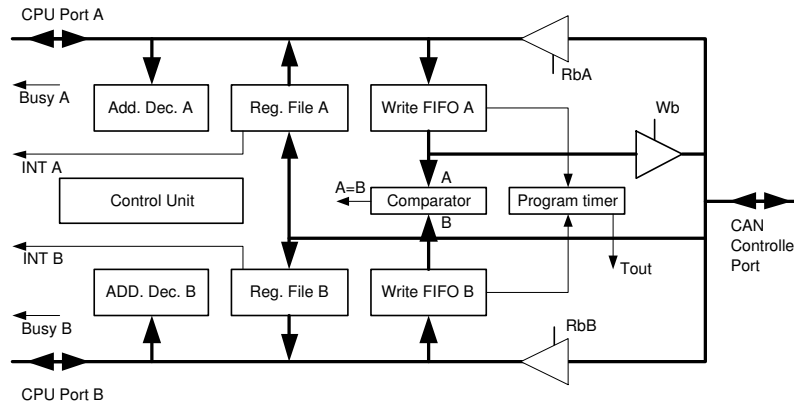


Fig. 6. Interfacing a pair of processors with a CAN controller in a master node to enforce fail-silent behavior.

all times the contents at the head of the two Write FIFOs. A write access to the CAN controller is generated only if a match is detected (correct output in the value domain). The Programmable Timer (PT) allows this validation to be extended to the time domain as required for example with the generation of FTT-CAN trigger messages. In this case, when a CPU writes the transmission request on its Write FIFO, the PT is started. The other CPU must issue its request before the programmed time-out interval (correct output in the time domain) if the transmission request is to be effectively passed to the CAN controller. If a time-out is reached or a mismatch is detected on the Comparator, an external line signals the fault, allowing the node to be disabled. The temporal validation is selected on a per transaction basis

according to hardwired DPCI configuration. Associated with each CPU port there is also a Register File, which replicates most CAN controller registers. These include the interrupt, status, control and command registers, as well as the receive buffer. All the registers have the same functionalities as their original counterparts within the CAN controller. This replication of the controller registers inside the DPCI is necessary for two reasons. Firstly, because in most controllers, some registers change after being read (e.g. interrupt and status registers). Secondly, because this allows the CPUs to run decoupled (for example, when they both read the receive buffer). The DPCI maintains both register files up to date according to the internal state of the CAN controller. This is achieved by a combination of polling periodically the controller

registers and using interrupts to signal changing conditions in the controller. For example when the interrupt line from the CAN controller is asserted indicating a message reception, the control unit copies the contents of the controller interrupt register, status register and read buffer to their respective positions in both Register Files inside the DPCI. Both interrupt lines are then activated and the CPUs now respond to these interrupts as if they were servicing directly the CAN controller.

5. CONCLUSIONS

This paper considered the general problem of imposing fail silent behavior in dynamic master-slave architectures. Two solutions are addressed, both based on hardware components that are attached to the node's network interface. One solution relies on bus guardians that allow enforcing fail-silence in the time domain, i.e. they inhibit faulty nodes from transmitting messages at arbitrating points in time wasting bandwidth and disturbing legitimate traffic. These bus guardians are adapted to support dynamic traffic scheduling and are fit for use in the slave nodes, only. The other solution relies on a special network interface, with duplicated microprocessor interface, that supports internal replication of the node, transparently. In this case, fail-silence can be assured both in the time and value domain since transmissions are carried out only if both internal nodes agree on the transmission instant and message contents. This solution is adapted for use in the master node but can also be used, if desired, in slave nodes. The concepts behind both solutions are applicable to master-slave networks in general. However, the discussion and implementation were focused on the FTT-CAN protocol. Two possible implementations for the bus guardians are presented and discussed, which have different costs and performance in terms of latency to enforce fail silence upon detection of an erroneous transmission. One is based on the COTS components while the other one relies on specialized hardware. Both this latter solution as well as the special network interface with direct support for internal replication are being currently implemented using FPGA technology.

REFERENCES

- Almeida, L., P. Pedreiras and J. A. Fonseca (2002). The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*.
- Almeida, L., R. Pasadas and J. A. Fonseca (1999). Using a planning scheduler to improve flexibility in real-time fieldbus networks. *Control Engineering Practice* **7**, 101–108.
- Arqueiro, N. and A. Oliveira (2003). Design, Implementation and Test of an FPGA based CAN Controller. *Technical Report, Universidade de Aveiro/IEETA*.
- Belschner, R. *et al* (2002). FlexRay Requirements Specification, version 2.0.2. *FlexRay Consortium*, <http://www.flexray-group.com>.
- BOSCH, Robert (1991). *CAN Specification Version 2.0*. Postfach 300240, D-7000 Stuttgart 30.
- Ferreira, J., P. Pedreiras, L. Almeida and J. Fonseca (2001). FTT-CAN error confinement. *Proceedings of FeT'2001 4th FeT IFAC Conference on Fieldbus Technology* pp. 8–15.
- Ferreira, J., P. Pedreiras, L. Almeida and J. Fonseca (2002). Achieving fault tolerance in FTT-CAN. *Proceedings of the 4th Workshop on Factory Communication Systems (WFCS 2002)*.
- Haverty, N. (1986). MIL-STD 1553 - a standard for data communications. *Communication & Broadcasting* **10**(1), 29–33.
- IEC (1999). IEC International Standard 61375-1 Electric railway equipment - Train bus - Part 1: Train Communication network. *International Electrotechnical Committee*.
- IEC (2000). IEC International Standard 61158: Fieldbus standard for use in industrial control systems - Type 1: Foundation Fieldbus H1; Type 7: WorldFIP. *International Electrotechnical Committee*.
- Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Press.
- Martins, E., J. Ferreira, L. Almeida, P. Pedreiras and J. Fonseca (2002). An Approach to the Synchronization of Backup Masters in Dynamic Master-Slave Systems. *Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS)*.
- Rufino, J. and P. Verissimo (1997). Hard real-time operation of CAN. *CSTC Technical Report RT-97-02*.
- Rushby, J. (2001). Bus Architectures For Safety-Critical Embedded Systems. *Proceedings of the First Workshop on Embedded Software (EMSOFT 2001)* **2211**, 306–323.
- Temple, C. (1998). Avoiding the babbling-idiot failure in a time-triggered communication system. *Fault Tolerant Computing Symposium* pp. 218–227.
- TTTech (2002). Time-Triggered Protocol TTP/C High-Level Specification Document (edition 1.0). <http://www.ttagroup.org>.